A-K-H-FTP

CS544-Protocol Design Document

Group 1
Ni An
Jae Hoon Kim
Junbang Huang

Table of Contents

Section 1. Service description

1. Overview

The designed protocol in this project is a secure FTP protocol operates on top of UDP. The objective is to provide an more efficient file transferring protocol compared with the traditional FTP, but at the same time guarantee the reliability of the transfer and enhance the security by encrypting data and authenticating both sides of the communication.

2. Global Timer

Our protocol uses a global timmer widely in implementation. The uses of global timer will be discussed in the following service description.

3. Connection-Oriented Service

The protocol is using UDP to improve efficiency of data transition and UDP provides connectionless services so our protocol handles connection. The connection oriented service of our protocol provides logical connection between two end user and it keeps connection until it finishes file transmission. The connection oriented service guarantees that all data is transmitted. To create a connection, the receiver will first send a request to the sender. After receives the request, sender will send back a response to the receiver. If the receiver does not receive any response after the request, it will send the request again when the timer goes off. When the receiver gets the response from sender, it will again send a response. After getting the response, a connection between receiver and sender will be established. A timer also exist in sender, when the sender doesn’t get anything after sending response, it will send it again after the timer goes off.

4. Reliable Service

Out protocol has similar functionality as TCP acknowledgment but the concept is different. In TCP, acknowledgment is the signal for sender side and sender keeps track the acknowledgment sent by receiver then frees segments in outstanding area. The sender has responsibility to send data but in our protocol, receiver has responsibility to request data. During the transmission, sender sends what it has. Retransmission happens when sender sends all the data and request to close and receiver notices that there are certain data segment is missing. At this time, receiver informs sender missing segments. After that if it still doesn’t get anything, it will resend that again. The transmission is done when sender sends a closing request and get confirm from receiver. Any missing message will trigger resend when timer goes off.

5. Flow Control

The main different part from TCP is that sender does not have buffer and window, and receiver has buffer but no window. So the flow control signal is from receiver to sender. When the buffer is full, receiver sends flow control signal and moves to ‘clean’ state and sender moves to ‘block’ state. Any new message received during ‘clean’ state will be discarded. In ‘clean’ state, receiver organizes and cleans buffer then requests segments which receiver needs. Before it requests missing segments, the receiver will be ‘ready’ state again so that it can receive missing segment. The sender stays ‘block’ state until it receives new request. New request comes after the receiver receives all the missing segment and asks sender to start sending again.

6. Error Control

Since UDP has only checksum mechanism for error control, our protocol should deal with other error control mechanism. Receiver has responsibility to request missing segments and discards duplicated segments. Receiver should also organize correct order of data. Unless it receives unordered segments, it writes file from received segments then free the segments. If it receives unordered segments, it keeps until it receives correct order segment then use unordered segments. Every segment has a sequence header so that the receiver can track the order. If buffer is full, it is handled by flow control.

7. Congestion Control

Since our protocol does not have window size, therefore we can’t control the congestion. When congestion happens, we will just leave it. From certain perspective, it seems that our protocol is not stable since we don’t deal with congestion, which might slow the transmission down. However, when congestion doesn’t happen, we our protocol does well. More importantly, sacrifice congestion control enables us to implement no-window transmission mechanism.

Section 2. Message definitions (PDUs)

1. Variables in PDU

A. MsgType (2 bytes)

The Header contains message type and there are eight different types of message types: control messages, error messages, and data message.

MsgType	Value	Description
Request Download	RD(0x01)	When client request download, it uses RD message type
Request Upload	RU(0x02)	When client request upload, it uses RU message type
Accept Download	AD(0x03)	When sender receives RD, it responds with AD message type
Request Segments	RS(0x04)	When receiver requests segments, it uses RS message type. RS message type is used in follow scenario: - After sender responds AD or RU - Receiver does not receive any segment in certain time (time-out) - When receiver check missing segments in ‘Check’ state. - After receiver cleans its buffer then get out from ‘Clean’ state
Send Segment	SS(0x05)	When sender transfers data, it uses SS message type
Request Blocking	RB(0x06)	When receiver does not have enough buffer, it sends blocking signal and it is RB message type
Request Close	RC(0x07)	When sender transfers all segments, it requests close connection by RC message type
Accept Close	AC(0x08)	When receiver gets RC, it checks whether file is completed or not. If it is completed, it accepts close connection by AC message type
Deny Request	DR(0x09)	The message type is used by server. When server receives request download or upload, it can deny the request.

Error Alert	EA(0x0A)	This message can trigger an immediate termination of current connection to prevent further damage caused by dangerous actions.

B. Version (2 bytes)

contains the highest version of AKHFTP protocol that can be supported by the host; first byte is the first number, second byte is the second number (e.g. For version 1.0, the first byte is 1, the second byte is 0)

C. SeqNumber (4 bytes)

contains the sequence number of the packet. In order to prevent sequence number proofing attack, the initial sequence number in the upload/download request will be randomly generated. Even segment sent by sender will be assigned continuous sequence number. Receiver will send missing segments' sequence number to request retransmission of missing segments from the sender.

For the "SeqNumber" field of the Request Download (RD), Request Upload (RU), Accept Download (AD), Request Blocking (RB), Request Close (RC), Accept Close (AC), Deny Request (DR) messages, we put a Token with 4 Bytes length. This Token should be a pseudorandom number. For example, when a user A send a RD (or RU) message to a server B, it will generate a pseudorandom number (i.e. Token) with 4 Bytes length as a value for the "SeqNumber" field. When the server B send response (i.e. AD) to the RD (or RU) message, the "SeqNumber" field in the response should be the same value of the Token in that RU (or RU) message.

D. MsgLength (4 bytes)

MsgLength indicates the true length of the data length in byte

E. FileSize (4 bytes)

FileSize indicates the size of the requesting file in unit of byte. The maximum file size is 4GB (= 2^32 bytes)

F. FileName

When receiver requests file, it sends file name in body.

G. Data

It is used when sender transfers data in body

2. PDUs

A. MsgType - Request Download (RD 0x01)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType = RD (0x01)
	4	SeqNumber = [Token]	
	8	MsgLength	
BODY	12	FileName	
	...		

B. MsgType - Request Upload (RU 0x02)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType = RU (0x02)
	4	SeqNumber = [Token]	
	8	MsgLength	
BODY	12	FileSize	
	16	FileName	
	...		

C. MsgType - Accept Download (AD 0x03)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType = AD (0x03)

	4	SeqNumber = [Token]
	8	MsgLength = 0x0004
BODY	12	FileSize

D. MsgType - Request Segments (RS 0x04)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType = RS (0x04)
	4	SeqNumber = 0x0000	
	8	MsgLength = 0x0004	
BODY	12	SegmentSize	
	16	Number of missing segments	
	20 . . .	A list of SeqNumbers for missing segments	

E. MsgType - Send Segment (SS 0x05)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType = SS (0x05)
	4	SeqNumber	
	8	MsgLength	
BODY	12	Data	
	...		

F. MsgType - Request Blocking (RB 0x06)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType=RB (0x06)
	4	SeqNumber = [Token]	
	8	MsgLength = 0x0000	

G. MsgType - Request Close (RC 0x07)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType=RC (0x07)
	4	SeqNumber = [Token]	
	8	MsgLength = 0x0000	

H. MsgType - Accept Close (AC 0x08)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType=AC (0x08)
	4	SeqNumber = [Token]	
	8	MsgLength = 0	

I. MsgType - Deny Request (AC 0x09)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType=DR (0x09)
	4	SeqNumber = [Token]	
	8	MsgLength = 0	

J. MsgType - Error Alert (EA 0x0A)

	Offset (octet)	2 Bytes	2 Bytes
HEADER	0	Version	MsgType = EA (0x0A)
	4	SeqNumber = 0x0000	
	8	MsgLength = 0x0004	
Body	12	SeverityLevel = 1	ErrorDescription

- SeverityLevel: this field indicates the severity of a certain error message to instruct the receiver of this error message to take actions. There are two levels: 1) Low (0): which means it is only an warning/notification message, the receiver does not have to terminate the current connection. 2) High (1): which indicates the error is very severe and both endpoints must terminate the connection immediately. Note, current EA messages all have “High” SeceverityLevel, and in the future version, some alert messages at “Low” SeverityLevel will be added.
- ErrorDescription: this field indicates the type of errors.
  - 0x01: File not found. When a client sends download request of a certain file to a server, if the server does not find this file, then it will send a “File not found” error message to the client. Both the client and server will terminate current connection.
  - 0x02: File oversize. This message may be sent after 1) an endpoint receives a upload request with the value of “FileSize” field larger than the available storage space on this endpoint; 2) an endpoint receives the “Accept download” message from the server, and finds that the value of “FileSize” is larger than the available storage space on this endpoint. Both parties in the connection will terminate the current connection once they receive this error message.

Section 3. DFA

1. Download - Receiver (Client)

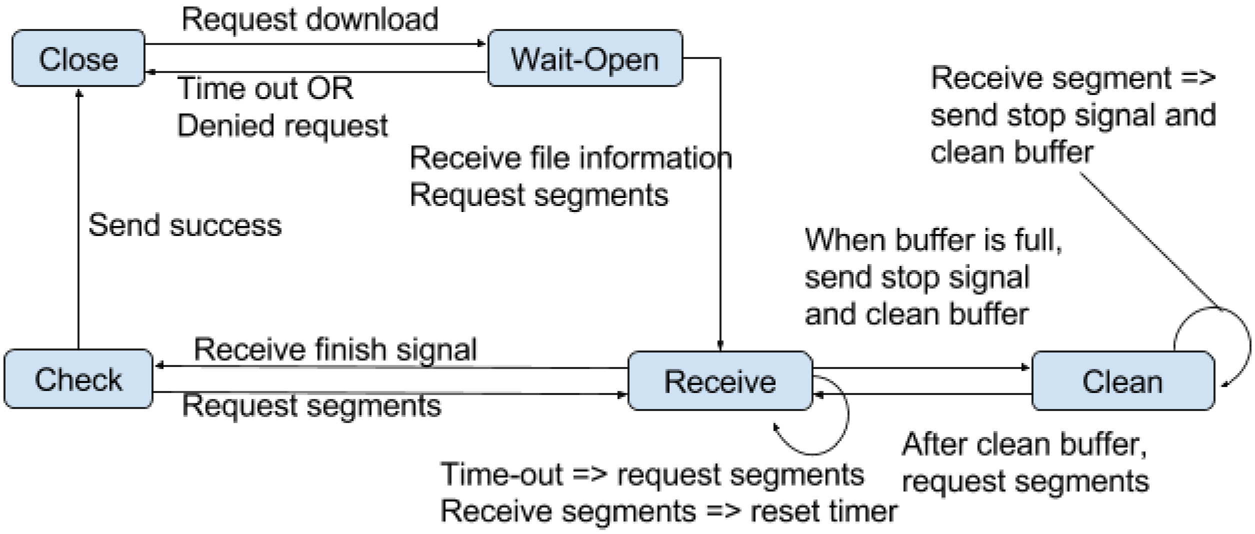

Figure - DFA: Download on reciever

First receiver sends download request then wait until sender sends file information which includes file size. If the receiver does not get file information before time out, it goes back to close. Based on file size, receiver requests segments. If receiver does not get any data in certain time (time-out), it keeps request. However, if buffer in receiver is full, receiver sends flow control signal (stop signal). In clean state, receiver organizes and cleans buffer then requests segments again and moves to receive state. If it receives finish request from sender, it checks whether file transition is completed or not then accepts finish request and close. Otherwise, it sends segments request again

2. Download - Sender (Server)

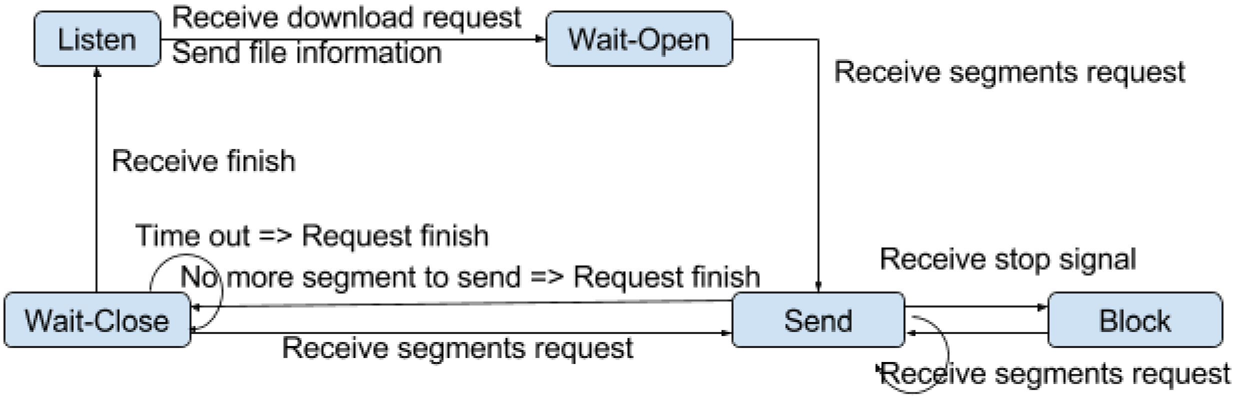

Figure - DFA: Download on sender

When server receives download request, it sends file information then wait until receiver sends back request. The sender transmits segments until it receives stop signal or sends all requested segments. When it is blocked by receiver, it stays in block state until it gets back request. When it finishes sending data, it request termination of connection. If receiver does not respond, it keeps sending terminate signal. If receiver verifies terminal signal, close connection. Otherwise, it sends requested data.

3. Download - Example

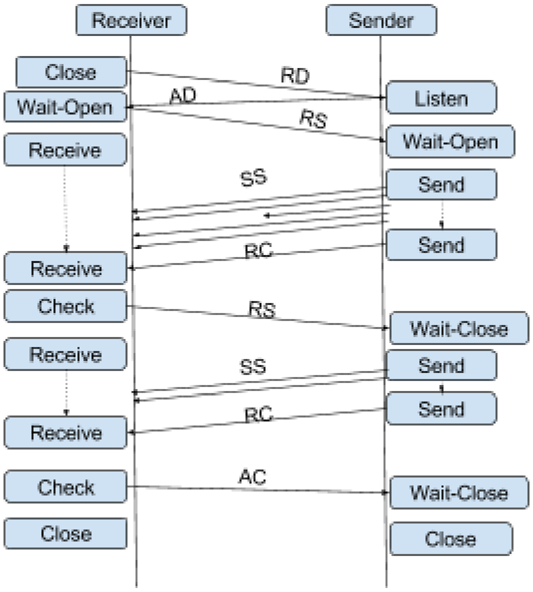

Figure - Download example

4. Upload - Sender (Server)

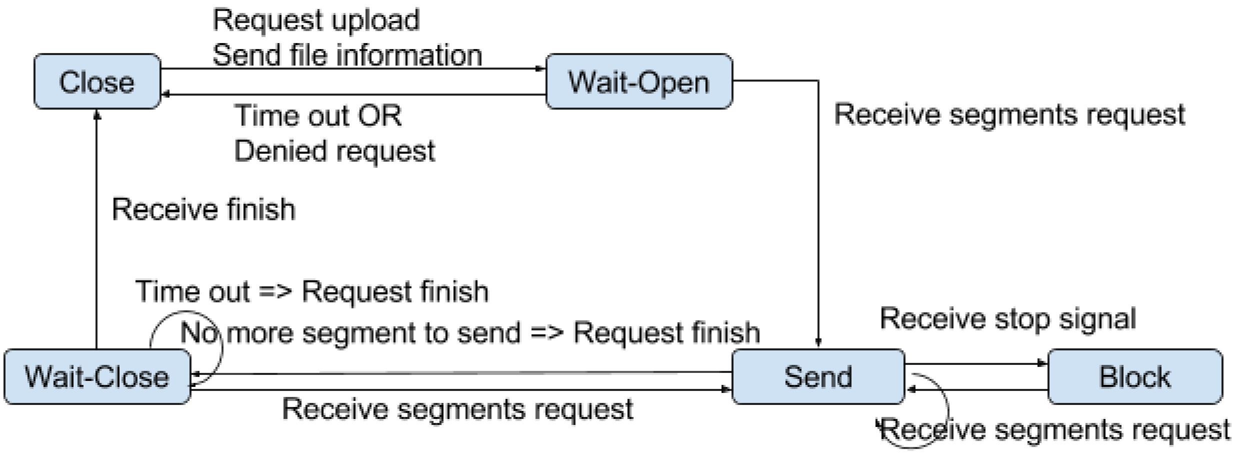

Figure - DFA: Upload on sender

When client wants to upload, it sends upload request with file information which include filename and size of the file. After receiver responds it with segment request, sender starts to transit data. It keeps sending until it receives stop signal or finishes transition. When it is blocked, it waits segment request. When it finishes transition, request finish connection. In wait-close state, it keeps sending finish request until receiver verifies finish connection or request more segments.

5. Upload - Receiver (Client)

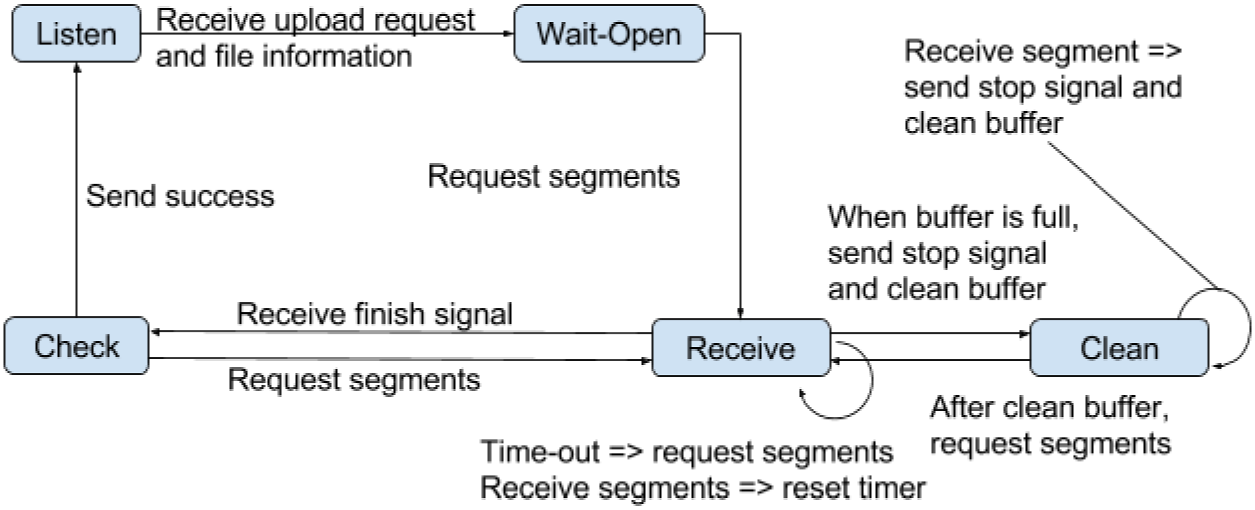
Listen
Receive upload request and file information
Wait-Open
Receive segment => send stop signal and clean buffer
Send success
Request segments
When buffer is full, send stop signal and clean buffer
Receive finish signal
Check
Receive
Clean
Request segments
After clean buffer, request segments
Time-out => request segments
Receive segments => reset timer

Figure - DFA: Upload on receiver

When server receives upload request with file information, it request segments. If it does not receive in certain time (time-out), it request segments again. If the buffer is full, receiver sends stop signal. If receiver gets more data in clean state, it assumes that stop request is missing so it sends stop signal again. After it cleans buffer, it request segment again.When it receives finish signal, it checks whether file transition is complete or not. If it is done, it accepts terminate signal and goes back to listen state. Otherwise, it requests missing segments and moves to receive state.

6. Upload - Example

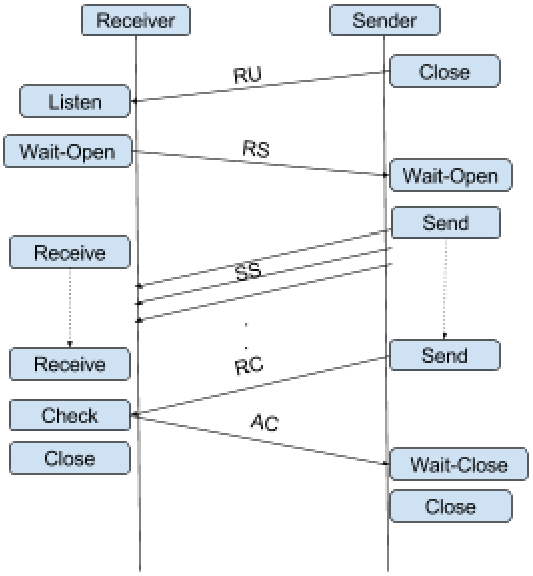

FIgure - Upload example

Section 4. Extensibility

The first two bytes of protocol header of each PDU is the “Version” field, which indicates the version of the current AKHFTP protocol. For example, our AKHFTP protocol starts from version 1.0, then the first byte of the “Version” field represent digit 1, the second byte represents digit 0. Endpoint with higher version of AKHFTP can go back to lower AKHFTP version to accommodate with endpoint with a lower protocol version. The protocol version used in the communication are decided by the two endpoints at the stage of negotiating the connection. First, one endpoint A sends an upload/download request in which the “Version” field indicates the highest AKHFTP version that can be supported by this endpoint. Then the other endpoint B sends an accept upload/download message (as a response to the request message) which contains the highest version lower than the version in the request sent by A and can be supported by B. Then A receives this message and it can finalize the AKHFTP version between A and B. All of the following communication will use this negotiated AKHFTP version.
The common AKHFTP PDU header is designed so that new command types and functions can be added to the protocol in the future. The “MsgType” field is 2 bytes, and new messages can be defined in the future. Currently there are only two types of alert messages, however, in the future more alerts can be defined and added in order to prove the reliability of this protocol.

Section 5. Security

The AKHFTP protocol is implemented on top of Datagram Transport Layer Security (DTLS)[1] protocol which provides endpoint authentication, message encryption to strengthen security of our protocol. The traditional TLS protocol is designed for protecting network traffic over TCP connection. However, AKHFTP protocol in this project uses UDP packets, thus TLS does not apply. DTLS is a protocol designed to secure communication over unreliable UDP traffic.
DTLS is designed so that it can have most of TLS's security functionalities, however since DTLS is implemented over an unreliable, connectionless transport layer protocol, it provides several mechanisms to deal with packet loss, packet reordering, etc.
Implementing DTLS, the AKHFTP protocol can 1) build private communication between the sender and receiver through symmetric encryption of the data packet; 2) ensure payload integrity by message authentication code (MAC), which means any tampering or change of the payload can be detected using the MAC; 3) authenticate the identities of sender and receiver using certificates. As for the design of AKHFTP itself, it has two types of error messages to protect the sender/receiver from inappropriate requests and messages.

Section 6. Proof of Stability

In this section, we will cover all the scenario to prove that our protocol is stable.

1. Request upload and download

When nothing bad happen:

Client sends upload or download request	
	Server receives request and response
Client receive response and send back another response	
	Connection establish

When upload and download request is lost or any of the responses are lost:

Client sends upload or download request	

[1] RFC 6347: https://tools.ietf.org/html/rfc6347

The request is lost	
Timer goes off, client sends again	
	Server receives request and response
	Response is lost
	Server send response again
Client timer goes off and sends request again	
Client receives response and send another response	
	Server receives the same request and discard it
	Server receives response from server
	Connection estabilish

2. Buffer is full:

Client receives a segment and there is no room for this segment	
A stop message is sent to server and at the same time, client write sequential segment into disk. Client is moved to 'clean' state	
	Server moves into block state
Client sends missing segment sequence number to request the segments.	
	Server sends segment to client.
Client gets all the segment and store them into the disk. Now the buffer is empty. Client set a restart message to server.	
	Server receives restart message and starts sending data again.

3. Segment is missing

	Sender sends a closing request
Receiver receives the closing and realizes that there are missing segments	
Receiver sends request for missing segments	
	Sender sends requested segments
Receiver receives missing segments	
	Sender sends closing request again

4. Retransmission request is missing

When the sender sends a closing request:

	Sender sends a closing request
Receiver receives the closing and realizes that there are missing segments	
Receiver sends request for missing segments	
Request is lost	
	Sender timer goes off and sends a closing request again
Receiver sends request for missing segments again	
	Sender receives request

When the sender is in ‘block’ state:

Receiver’s buffer is full and send a block request to the sender	
	Sender receives a block request and moves to block state
Receiver is moved to clean states and store buffer into disk	

Receiver sends request for missing segments	
Request is lost	
Receiver sends again	
	Sender gets the request and resume from block state and start sending data

5. Disconnection

When the closing request is lost:

	Sender sends a closing request to receiver
	Sender sends again
Receivers receives the request and notice that there are no missing segment	
Receiver sends an acknowledgement response	
The response is lost	
Receiver sends response again	
	Sender receives the response and close itself
Receiver timer goes off and close itself	