Workgroup:	RFT Working Group				
RFC:	draft-01				
Published:	12 May 2023				
Intended Status:	Informational				
Expires:	13 November 2023				
Authors:	T. Blum	H. Cech	M. Engelbart	L. Nagel	R. Nai
	TUM	*TUM*	*TUM*	*TUM*	*TUM*

Robust File Transfer

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 13 November 2023.

Copyright Notice

Table of Contents

1. Abstract

This document describes the simple file transport protocol which enables fast and robust file transfer via UDP.

2. Status

Current Version 0.2 (Work in Progress)

3. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Other terms used in this document are defined as follows:

- Byte: A byte is defined as a sequence of 8 bits.
- Chunk: A chunk is defined as a sequence of 1024 bytes.

4. Open Todos

- Find protocol name
- Benchmark an implementation and adjust magic numbers like timeouts accordingly
- See how robust the UDP checksum is

5. Introduction

The XXX Protocol is an application layer file transfer protocol to transmit one or multiple files from a server to a host. It is based on a client-server architecture, where the server stores the files and the client can request files. The focus of the protocol is on performance, robustness and on simplicity resulting in a small server state. The transport protocol used by XXX is UDP which allows high performance. To mitigate the issues when using UDP over TCP like data integrity and congestion control, XXX uses specific mechanisms to guarantee proper data transport also under difficult circumstances. Encryption or authorization mechanisms are currently not supported, can however be added later due to the highly flexible option fields. Data integrity is provided via UDP checksum as well as file checksums.

6. Basic Overview

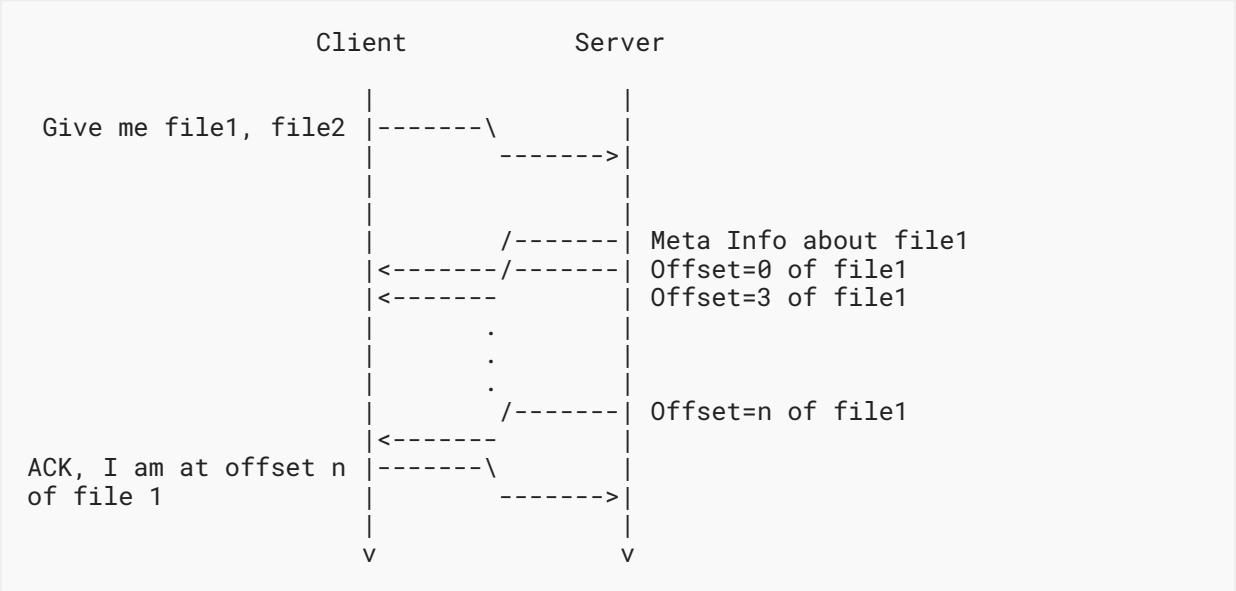

Figure 1: Very basic data flow of the protocol

6.1. Dataflow from client perspective

To initiate the file transfer, the client starts with a `Client Request` message containing all files it wants to download. It will then wait for the first server response messages. If it receives any server messages, it will start sending ACK messages periodically to tell the server what data it has received. Besides the last received chunk of data, the ACK message also contains resend entries of packets, which might have been lost as well as different flags and data to get information about the RTT and flow control.

6.2. Dataflow from server perspective

Once the server receives a request message from the Client, it will start sending meta data followed by the content of each file sequentially. To minimize congestion, it will start slow and become faster, if the required ACK messages from the client are received. If the ACK messages contain many resend entries, however, the transmission speed will slow down and will resend those packets. If after a certain period of time no ACK message is received, it will drop the connection and remove the connection state.

7. Messages

In the following the different message layouts are explained.

7.1. Message Header

Each message contains a global header to enable server and clients to quickly parse and handle messages. This header has a length of at least three bytes. The global header looks as follows: It starts with the version of the protocol which takes the first four bits of the first byte.

The details of the versioning are explained in section 3.2.

The second four bits of the first byte contain a message ID. The message ID indicates which type of message the packet contains. All valid message types are described in the following sections together with a unique identifier, which MUST be used as the message ID in the header.

If a participant receives a message with an invalid request ID, the message MUST be dropped.

An 8 bit `ACK number` MUST be sent by clients to the server in the second byte. This allows the server to tell the client the last ACK received within a small window, so the client can calculate the RTT. The client MUST increment the ACK number by 1 after each packet it sends and wrap around back to 0 when it overflows. The ACK number in messages sent by the server MUST be equal to the ACK number in the last received message from the client.

The third byte of the header contains the number of options following the header.

The data integrity of a single packet is guaranteed by the UDP checksum which MUST be enabled.

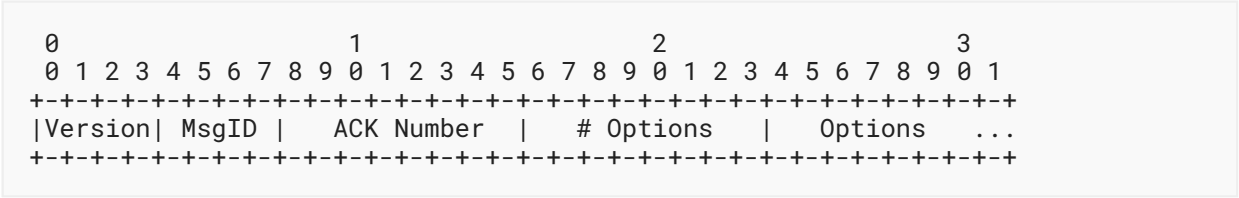

Figure 2: Global header of all messages

- `Version` - 4 bit unsigned integer specifying the version used for this message.
- `MsgID` - 4 bit unsigned integer specifying the message ID.
- `ACK Number` - Unsigned 8 bit integer that distinguishes this packet from packets sent within a small window before and after this payload. The server will then use its last received ACK number in its next messages, so that the client can use that to determine the RTT.
- `# Options` - 8 bit unsigned integer specifying the number of options in the `Options` field.
- `Options` - TLV-encoded options (see section *Options* below).

7.1.1. Versioning

The version of a message is based on the protocol version implemented by the client/server software. If the client sends a message with a version not supported by the server, then the server MUST send a `Close Connection` message to the client containing the reason code

Unsupported Version (`0x2`). The version of the `Close Connection` message MUST be the lowest version the server supports in case the version of the client's message is lower than any of the supported versions of the server and the highest version the server supports otherwise.

If the version of the client message is supported, then the server MUST act accordingly to that version. During the transmission the client and the server MUST NOT change the used version.

7.1.2. Options

A packet header may contain zero or more, but at maximum 255, type-length-value (TLV) encoded options. The number of options MUST be given in the second byte of the header. Each option MUST contain three fields: the first byte holds the type of the option, the second byte contains the length of the option value in bytes. Afterwards follow as many bytes as indicated by the second field to hold the actual option value. Option types MUST be registered in this specification or a future version of this specification. Option types MAY be different in different versions. Currently there are no option types specified and implementations SHOULD be able to operate without using any options.

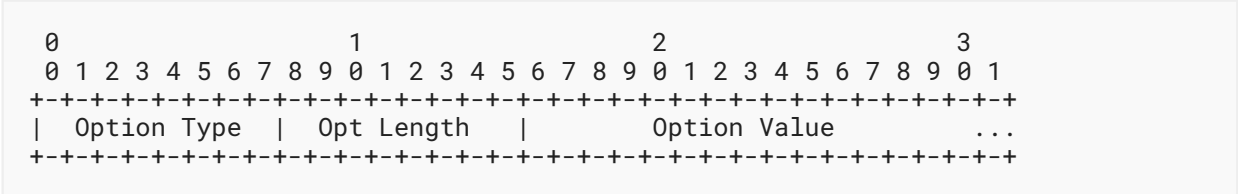

Figure 3: TLV-encoding of header options

7.2. Client Request

To initiate a download, the client MUST send a client request message to the server. The message ID for a client request MUST be set to `0x0`. In a client request, the first field after the header contains four bytes for the maximum transmission rate, at which the client is willing to operate. The maximum transmission rate MUST be given in messages/second (which is roughly equivalent to KiB/s). See section 5.1 for more information about the maximum transmission rate.

The protocol also supports the download of multiple files. Thus a client needs to indicate how many and which files it wants to download from a server. The first two bytes after the `Maximum Transmission Rate` contain the number of files requested. The number of files is then succeeded by exactly that number of file descriptors to the requested files.

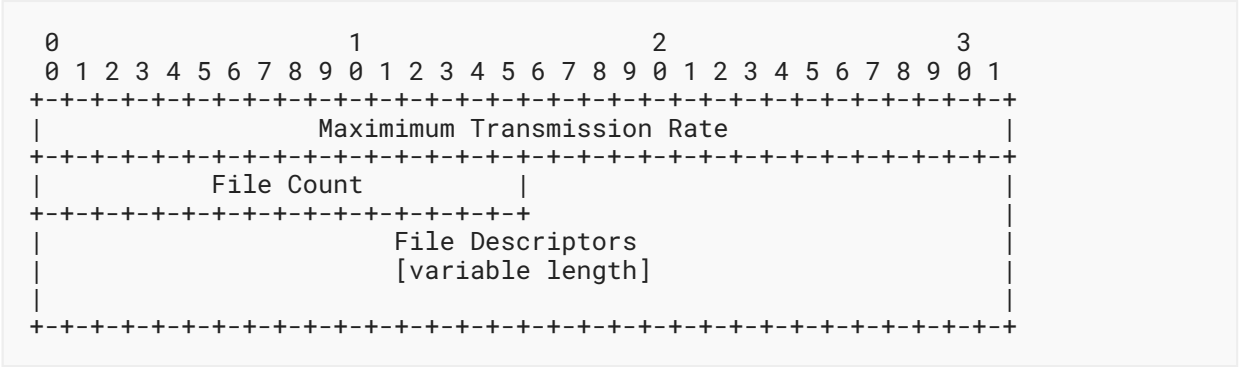

Figure 4: Client Request Message

- `MsgID` : `0x0`
- `Maximum Transmission Rate` - Maximum transmission rate a client is willing to operate at given in messages/second. Transmission rates may be updated in following Acknowledgement packets.
- `File Count` - 16 bit unsigned integer specifying number of file descriptions in the request. This specifies the total number of files being requested.
- `File Descriptions` - Array of File Descriptors with the length FileCount. Each File Descriptor describes one requested file.

7.2.1. File Descriptor

The File Description describes one file being fetched and is part of the `Client Request` message. Random access of files by this protocol is supported, so an offset is defined. A full file path is then provided, along with the length of the string describing the path.

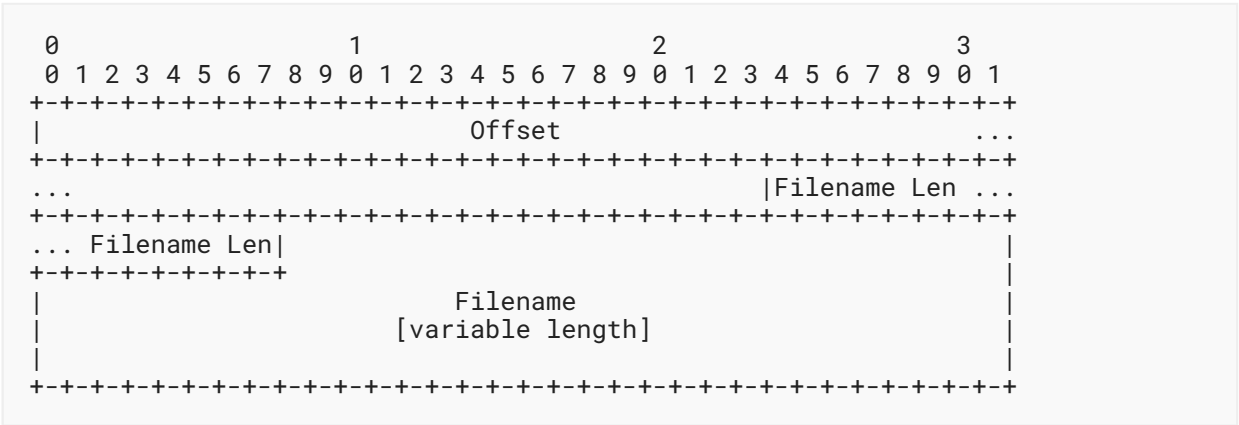

Figure 5: File Descriptor frame

- `Offset` - Unsigned 56 bit integer describing the offset into the file in chunks.
- `Filename Length` - Unsigned 16 bit integer containing the length of the filename string in bytes.
- `Filename` - UTF-8 encoded string identifying a file to be downloaded.

7.3. Server Metadata

For every file requested by the client, the server is going to send one message containing metadata about the file. The message ID for metadata messages MUST be set to `0x1`. If the server can not serve a file for any reason, the reason MUST be set in the status byte of this message. If status is set to `0x0`, the metadata serves as a confirmation to the client that the data for the requested file is about to be sent. Special error status codes are specified below. All nonzero status messages imply the file will be skipped by the server.

Additionally, the last field of the metadata message contains a checksum, which can be used to verify the correct transmission of a file and to check for changes after a resumed download.

TODO: Which checksum algorithm will be used and thus how big this field needs to be, will be defined in a future version of this document.

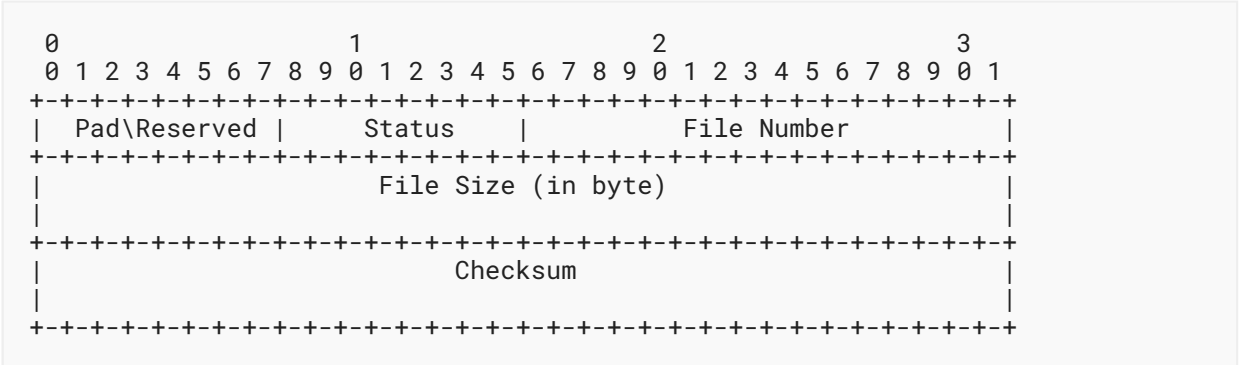

Figure 6: Server Metadata Message

- `MsgID` : `0x1`
- `Status`
  - `0x0` - Download will proceed as normal
  - `0x1` - File does not exist
  - `0x2` - File is empty
  - `0x3` - Access denied
  - `0x4` - Offset bigger than filesize
- `File Number` - Unsigned 16 bit integer. This index corresponds to the index in the `client request file descriptors`. For example, if the `file descriptors` were `[a, b]`, then file *a* would have `file number` *0* and file *b* would have `file number` *1*.
- `File Size` - 64 bit unsigned integer specifying the length of the file.
- `Checksum` - A checksum to validate contents of the file. Which kind of checksum will be used will be defined in a future version of this draft.

7.4. Server Payload

The `Server Payload` contains a chunk of information about a file being downloaded. The server MUST save client session information so it can uniquely identify a file based on the file number and connection port and IP of the client. The message ID MUST be set to `0x2`. An 8 bit ACK Number can be used for the client to calculate the RTT. If no ACK message has been received, it MUST be set to 0. If an ACK message has been received, this field MUST be the same as the ACK number of the last received ACK message. The offset is in the granularity of chunks, thus is a 56 bit unsigned integer.

The server MUST initially send payloads in the order in which they were requested from the client. In other words, payload at offset `N+1` of a file MUST be sent after payload at offset `N` and all payload packets of file `N+1` MUST be sent after all payload packets of file `N`.

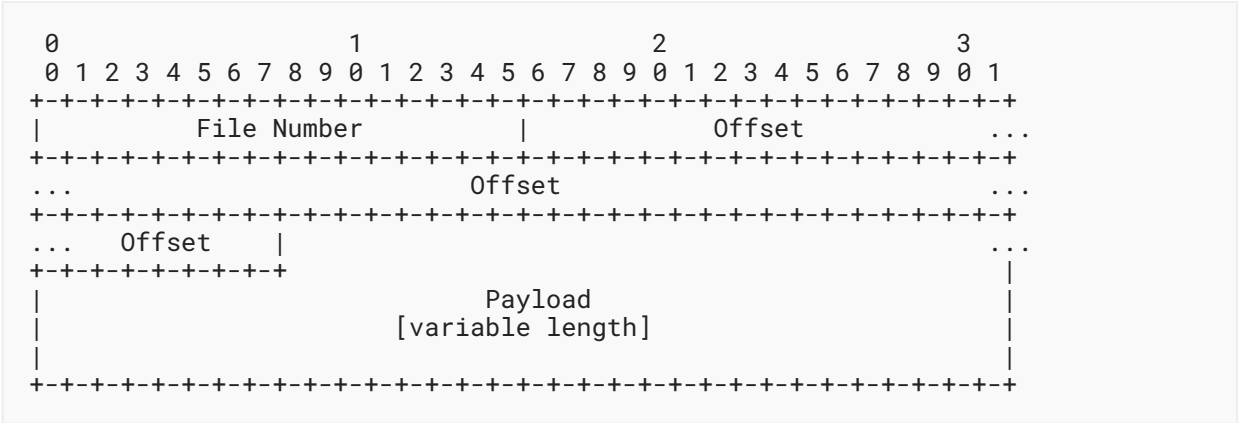

Figure 7: Server Payload Message

- `MsgID` : `0x2`
- `File Number` - Unsigned 16 bit integer, same as in the `Server Metadata` message.
- `ACK Number` - Unsigned 8 bit integer that is from the last received `Client ACK`.
- `Offset` - Unsigned 56 bit integer describing the offset into the file in chunks.
- `Payload` - Binary data of the file being downloaded. Note here that the payload length can be inferred from the UDP packet length

7.5. Client ACK

The client MUST send a client `Client ACK` message every `RTT/4` seconds. The message ID for the client ACK is `0x3`. An 8 bit `ACK number` MUST be sent to the server. This allows the server to tell the client the last ACK received within a small window, so the client can calculate RTT. The client MUST increment the ACK number by 1 after each ACK is sent and wrap around back to 0 when it overflows. The `Offset` is in the granularity of chunks, thus is 56 bits and MUST point to the highest offset of the file with the highest File Number, for which a payload packet has been

received. The `Status` MUST be set if the special conditions described below are met. The `Maximum Transmission Rate` specifies, in messages/second, the maximum transmission rate the client can handle from the server. More information about this can be found in section `5.1 Flow Control`.

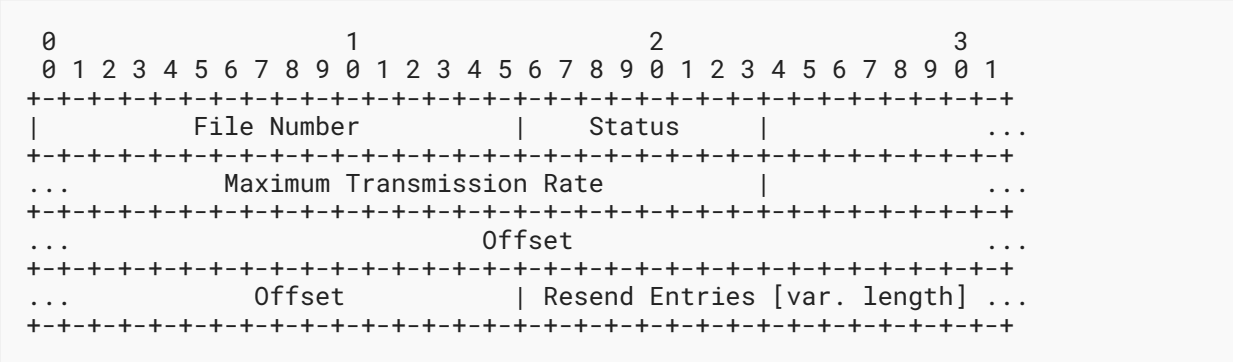

Figure 8: Client ACK Message

Note that the number of resend entries in the packet can be calculated from the UDP packet size.

- `MsgID` : `0x3`
- `File Number` - Biggest File Number for which a payload packet has been received. Unsigned 16 bit integer, same as in the `Server Metadata` message.
- `Status`
  - `0x00` - Nothing special
  - `0x01` - No metadata received
- `Maximum Transmission Rate` - unsigned 32 bit integer describing the maximum transmission rate, same as in `Client Request`.
- `Offset` - Unsigned 56 bit integer describing the offset of the chunk, with the highest offset, in the file identified by `File Number`.
- `Resend Entries` - Array of `Resend Entry` structures that describe missing data in the received file.

7.5.1. Resend Entry

A resend entry describes a contiguous chunk of missing data. It consists of a `File Number`, in case the missing data is from a previous file where the server already reached the EOF in file transmission. The offset specifies the position in the file at which the missing data starts. Length specifies how many contiguous chunks are missing.

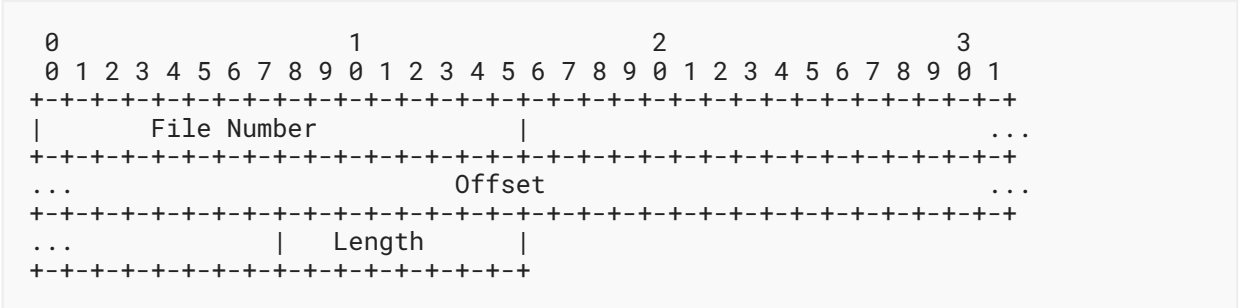

Figure 9: Resend Entry

- `File Number` - Unsigned 16 bit integer, same as in the `Server Metadata` message.
- `Offset` - Unsigned 56 bit integer describing the offset into the file in chunks.
- `Length` - Number of contiguous missing chunks. If more than 255 chunks are missing, multiple resend entries are needed.

7.6. Close Connection

For politeness and convenience, servers and clients SHOULD send close connection messages to notify the other party to free all session related state immediately upon receival, rather than wait for the connection to timeout. The server SHOULD also send close connection messages to clients with improper or expired requests. All close connection requests are optional and are not guaranteed to be received.

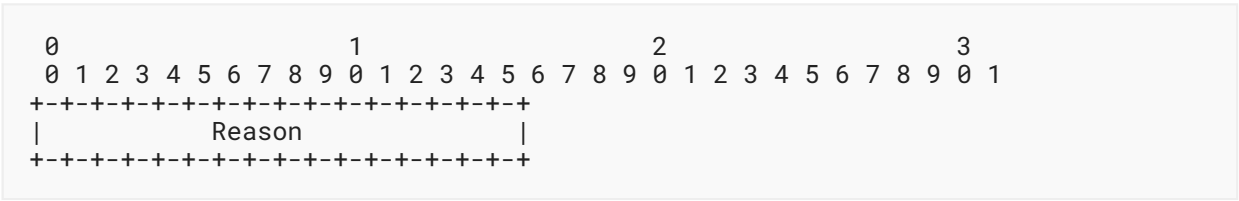

Figure 10: Close Connection Message

- Reason
  - `0x0` - Unspecified/No Reason Provided
  - `0x1` - Application closed (ctrl^c)
  - `0x2` - Unsupported Version
  - `0x3` - Unknown RequestID
  - `0x4` - Wrong Checksum (client to server)
  - `0x5` - Download Finished (client to server)
  - `0x6` - Timeout

8. Exception handling

8.1. Unknown Exception

If the server or the client is shut down in a controlled manner due for reasons that do not fit into the reasons below, it SHOULD close all currently established connections by sending a `Close Connection Message`. The reason field in this message MUST be *Unspecified/No Reason Provided* (`0x0`).

8.2. Application closed

If the server or the client application is shut down during a transmission, it SHOULD close all currently established connections by sending a `Close Connection Message` with the reason *Application closed* (`0x1`).

8.3. Unsupported Version

If the server does not support the version of the client specified in the main header, it SHOULD close the connection by sending a `Close Connection Message` with the reason *Unsupported Version* (`0x2`). This does not hold for the options which can be appended to the global header. Options are designed as optional features. Thus the server can ignore the option if it does not support the option type.

8.4. Unknown Client

If the server receives a message from a client (identified by the IP address and port) it does not know it SHOULD respond with a `Close Connection Message` with the reason *Unknown RequestID* (`0x3`). This is mostly due to timeouts in which case the server dropped all connection state.

8.5. Wrong checksum

If a connection got dropped, the client can try to establish a new connection with a new request ID with the offsets of the half downloaded files. The first message of the server will then be the metadata of the file. If the checksum is different than the one of the previous request ID, the file has changed in between both requests. The client SHOULD therefore drop the connection by sending a `Close Connection Message` with the reason *Wrong Checksum* (`0x4`) and start a new request.

8.6. Download finished

If the client receives all requested files with no data loss, the client SHOULD notify the server by sending a `Close Connection Message` with the reason *Download Finished* (`0x5`). Once the server's file offset reaches the end of file, it SHOULD wait for ACKs with resend request until the

standard timeout period has expired. Thus, if a client has the end of a file but has holes in the middle, the client MUST send a request with the file offset set to the end of the file and send ACKs with resend entries describing the location of the holes.

8.7. Timeout

Every 0.25 seconds, the client MUST send a `Client ACK Message` to the server, updating the server with the client's current position in the file, and if there are any holes. The server SHOULD wait for a period of at least 3 seconds for new `Client ACK Messages` before closing the connection and dropping the client state. The server SHOULD also send a close connection message with the reason *Timeout* (`0x6`), notifying the client of a timeout.

In addition, once a connection is established, the server must continually send binary data of the requested files to the client. The client MUST wait a period of at least 3 seconds before closing the connection and attempting to start a new connection with adjusted offsets. The client SHOULD also send a close connection message with a reason of `0x6`, notifying the server of a timeout.

If either the client or server wants to cancel a connection early, they SHOULD send a close connection message so the other side can free up the associated session memory. However, due to the 3 ACK period timeout, if the close connection message is lost, the other side will eventually time out.

8.8. File too small

If the client requests a file at an offset that is larger than the file, the server MUST send the metadata for this file with the flag *File too small* (`0x04`).

8.9. Client Request retransmission

The client expects the server to start the data transmission after if has sent a client request message. The request message can however get lost, in which case the client won't receive any messages from the server. The client should wait for a server response for a reasonable time. If no message is received, the client SHOULD stop listening on its current port, start listening on another port, and resend the client request message. If the server does again not respond, the client MAY increase the time until the request is retransmitted exponentially or otherwise.

9. Flow and Congestion control

XXX has mechanisms to prevent overloading the network, the receiver, or the sender. It consists of two parts: The client-controlled flow control and the congestion control. Both values are calculated independently of each other and the minimum of both values will be used by the server as its maximum transmission rate: `server sending rate = min(flow control, congestion control)`. The server MUST NOT send more packets per second than the minimum of both values. In the following two sections both mechanisms are specified.

9.1. Flow control

Flow control guards the clients from being overloaded. The client specifies a maximum transmission rate in both the `Client Request` and each `Client ACK` message. The unit of this field is number of messages/second. This rate SHOULD be an estimate of the maximum data rate that the client is able to process. The server MUST NOT transmit more messages (and therefore `Server Payload` messages) per second than this field of the last received message specifies. An exception to this rule is the special value 0: The client can set the maximum transmission rate to 0 to signal that it does not want to limit the server. In this case, the server sending rate is solely determined by congestion control.

The client is free to adapt this value throughout the protocol execution, e.g., to fit its memory resources. The server MUST update the maximum transmission rate with each received packet.

9.2. Congestion control

The purpose of congestion control is to prevent overloading the network. This protocol uses packet loss indicated by the resend entries in the ACK messages as an indicator that the network's performance ceiling is reached.

The server has to maintain a sending rate. That sending rate MAY be small at the start of a connection but SHOULD increase with every `Client ACK` that does not contain a resend entry. If the server receives a `Client ACK` with one or more resend entries, it MUST reduce its sending rate.

10. Security considerations

The security of a software system is defined on the basis of certain security goals. Those are typically confidentiality, integrity, availability, authenticity, accountability, and non-repudiation.

The XXX protocol takes insufficient measures to realize any of those properties. Any of them is violated in at least one way:

- Confidentiality: The data is sent in plaintext over the network and can be read by any observer.
- Integrity: An XXX client does not check the integrity of the received data. An on-path attacker could intercept all payload packets of a particular file and replace the payload with a different file. The file checksum that is sent in the metadata message does not protect from this attack as the attacker can change the checksum as well.
- Availability: A client could send many request messages until the server is overloaded. The client is in a stronger position as it could simply discard the incoming traffic while the server will send data messages until it is limited by flow or congestion control.
- Authenticity: The identity of the server is not checked. An on-path attacker could reply to client messages and act as the server. IP address spoofing could also be used to attack the connection.

- Accountability: The server is deleting the whole client state after a download has finished.
- Non-repudiation: The client does not authenticate itself during a transfer. Source IP addresses can be faked, non-repudiation is therefore not given.

Confidentiality, integrity, and authenticity could be offered by incorporating an encryption and authentication scheme. An obvious option would be to utilize TLS to setup a secured channel; Datagram Transport Layer Security (DTLS) [RFC6347] adapts TLS to UDP.

While an encryption scheme would also provide a mitigation for some availability and denial-of-service (DoS) attacks, further modifications to the XXX protocol would be needed. Two easy to exploit attacks along with a potential mitigation are described in the following.

- DoS amplification attack: Upon receiving a request message, the server will start transmitting the first requested file. The data is sent to the address from which the request message was sent. However, an attacker can potentially spoof the source IP address and set it to a victim's IP address. The server will then send multiple payload messages to the victim before the connection times out due to missing ACK messages. Because of the high amplification factor the XXX-protocol offers, public XXX servers are an attractive target to be abused for such attacks by malicious clients, similar to DNS-DoS attacks. A possible mitigation is a handshake that needs to be completed before the server starts sending payload messages.
- DoS attack: The client could leave the resend entries field empty irrespective of the actual packet loss to ramp up the server's sending rate to its limit. This could impair the server's availability. One potential mitigation would be to force the client to put some information of an acknowledged packet into its ACK message. This could be a agreed-upon part of the payload or a hash of the payload message.

11. Informative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

Authors' Addresses

Thomas Blum
TUM
Email: thomas.blum@tum.de

Hendrik Cech
TUM
Email: hendrik.cech@tum.de

Mathis Engelbart
TUM
Email: mathis.engelbart@tum.de

Lennard Nagel
TUM
Email: lennard.nagel@tum.de

Richard Nai
TUM
Email: richard.nai@tum.de