Protocol Design (IN2333) Group 7 and Group 1337
INTERNET-DRAFT 10001 TUM
August 2022

B^2RFT

Abstract

This document specifies a two layer file transfer protocol, Best Robust File Transfer Protocol (BRFT), on top of UDP. BRFT uses a two layer connection, enabling large file transfer with file change detection, re-transmission and chunk based compression.

Contents

1. Introduction

The Best Robust File Transfer Protocol (BRFTP) is an application- and transport level request/response protocol with minimal state. The protocol enables file transfer over an unreliable network. A server thereby offers files for download. A client may request one or more files for download. BRFTP enables the client to robustly retrieve files from the server, transfer in a compressed manner. Additionally, BRFTP enables file change detection, download resumption and multi-file downloads.

2. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [4].

3. Design Considerations

The Best Robust File Transfer Protocol (BRFTP) has been designed with reliability as its main consideration. The protocol must ensure this reliability in face of being deployed on top of UDP RFC768 [1]. As UDP is a connection-less protocol, offering besides a packet checksum no reliability, BRFTP must ensure this property itself. In particular, BRFT must handle flow- and congestion control as well as connection drops.

3.1. Layer Structure

The Best Transport Protocol layer (BTP) and the Best Robust File Transfer Protocol layer (BRFT) compose the structure of B^2RFTP. This layered structure ensures a strict separation of concerns. This structure MUST be implemented for full protocol adherence. Figure 1 displays this structure. The protocol is in between the transport layer protocol, UDP, and an arbitrary application layer. The application layer and BRFT exchange files. Moreover, BRFT and BTP exchange packets containing optionally compressed chunks in their payload. Further, BTP and UDP reciprocate the packets which the client or server is to send over the wire.

BTP is the layer on top of UDP. This layer ensures reliable communication over an unreliable network. In particular, the BTP layer modifies state internal to the protocol. This modification of state includes e.g. the congestion and flow control, which modify protocol state to ensure a robust file transfer.

BRFT is the layer on top of BTP. The BRFT layer handles all file operations. These file operations consist of the requisition and serving of files, as well as the negotiation of connection specific state for file transfer. Moreover, this layer modifies state on objects external to the protocol. This state modification contains the writing and reading of files to the file system.

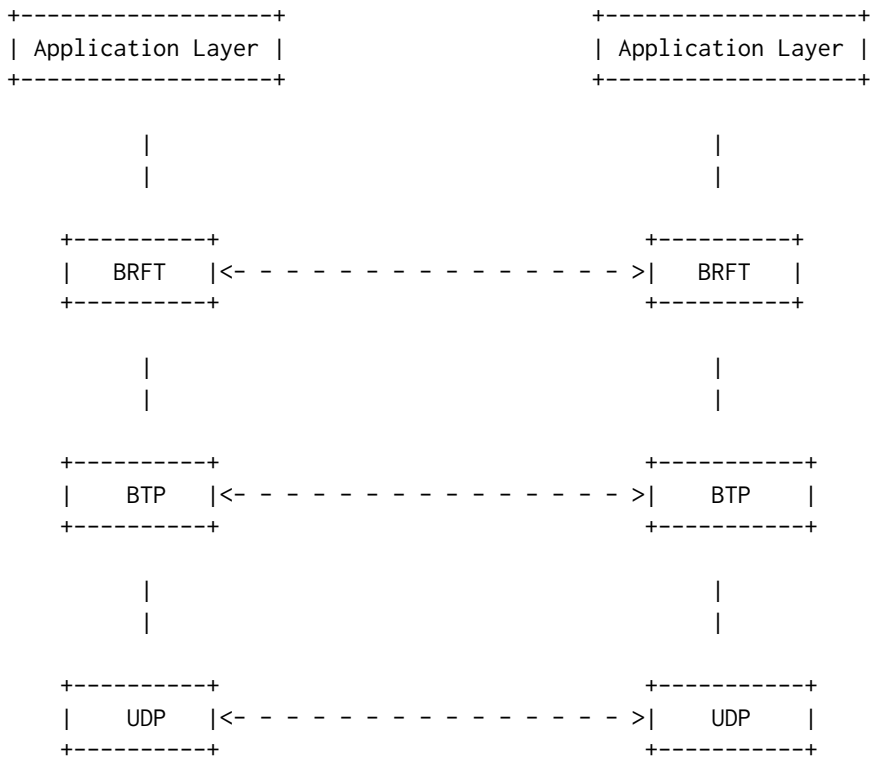

Table 1: BRFTP layered structure

4. Best Transport Protocol

Since the successful transfer of files implies that a transferred file can be reassembled in full at the receiver's side, a mechanism for the re-sending of data packets is required. As the transport layer protocol UDP does not provide this assurances, the Best Transport Protocol (BTP) layer is designed to provide a reliable and ordered transport layer. Additionally, this layer handles congestion- as well as flow control, as described in the respective sections. BTP offers a full duplex connection, as described at length in Section 4.2.

4.1. Server and Client

Every BTP communication MUST consist of two parties, a server and a client. Every interaction is initiated by the client via a connection process, as described at length in the "Establishing a Connection" section 4.10. It should be noted that a client is exactly the party that initiates a connection and the server is the party that accepts it. Data can be sent both ways and the connection can be terminated by either party.

4.2. Connection

A server-client association, that has completed the connection process and is not closed or interrupted, is referred to as a connection. Every connection MUST be uniquely identifiable by a combination of IP and Port. The server MUST listen for incoming connection requests on port 1344 while the client can use any port for reception. The server MUST use the source address and port of incoming requests as the return address to the client in further communications.

4.3. Discovery Mechanism

BTP does not offer an integrated server discovery mechanism, and therefore the client MUST know the server IP before initiating a connection.

4.4. Encoding

All packet fields MUST be encoded in big-endian byte-order.

4.5. Best Transport Protocol Generic Header Format

The BTP Generic header is shared by all BTP packets. Since the header is included in all packets, its size is a key concern. The full header is 32 bits long (BtpHeaderSize) and is therefore byte aligned to fit into a single double word. This header MUST be included at the very start of each message. This header MUST be included at the very start of each packet. The combination of Version and Type SHOULD be used by client and server to distinguish BTP packets from unknown traffic on the UDP connection. Unknown Version and Type combinations MUST be ignored by all parties. The full header is depicted in Fig. 1:

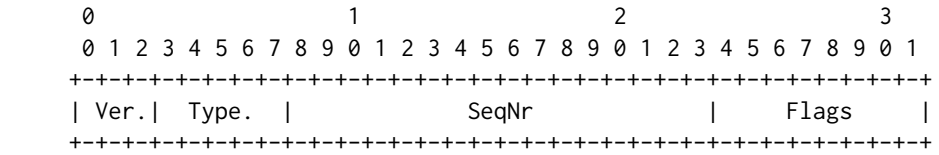

Figure 1: BTP Header Format

Version: 3 bits

The Version field indicates the protocol version of BTP. All packets described in this RFC are considered V0 and MUST start with BTPv0 as defined in the table below. The value 000 is considered invalid and SHOULD never be used. Other values are reserved for future versions and MUST be linearly ordered, as is illustrated with the future version BTPv1.

Name	Bit Encoding
BTPv0	001
BTPv1	010

Table 2: Version values

Type: 5 bits

The Type field denotes the semantic meaning of a given packet and SHOULD be used to assert the packet size for fixed size packets. The field is composed out of 5 bits, allowing for a maximum of 32 distinct message types. Currently, five message types are defined as part of BTPv0. The full list is presented in Table 3. The meaning and structure of all packets will be discussed at length in Section 4.9.

Name	Bit Encoding
Conn	00001
ConnAck	00010
Ack	00011
Data	00100
Close	00101

Table 3: Full list of packet type values

Sequence Number: 16 bits

The sequence number is a 16-bit field that MUST be used for congestion control and flow control mechanisms.

Flags: 8 bit

Flags header is used to transfer meta information about the packet. Currently only one flag is available, the ‘RETRANSMISSION‘ flag.

Name	Bit Encoding
RETRANSMISSION	00000001
RESERVED	00000010
RESERVED	11111111

Table 4: List of all currently supported flags

4.6. Sequence Numbers

BTP Sequence numbers are 16-bit unsigned integers. A two byte number was chosen to balance out BTP header size and offer sufficient sequence length without frequent integer overflow. Every server and client MUST select independent initial sequence numbers (ISNs) at the beginning of every connection process. It is RECOMMENDED to sample the initial values at random from the 0 to 2^{16} number space to mitigate packet injection attacks without packet monitoring by an external party. Sequence numbers must be linearly ordered and incremented by one for every packet send. The only exception being if the congestion window includes an integer overflow, in this

even all values larger or equal 0 are considered larger 2^{16}. Only data ordered via sequence numbers SHOULD ever be forwarded to the upper BRTF layer.

4.7. Congestion Control

BTP uses the Elastic-TCP [2] algorithm for Congestion Control to achieve a high average throughput with low sensitivity regarding the loss ratio. Elastic-TCP operates in two distinct phases. Upon the start of a new data connection it uses a standard slow start procedure to quickly grasp the maximum capacity of the link. The initial congestion window size is set to the InitialCwndSize value suggested by the Conn packet (see 7) of the 3-way handshake. After the first packet loss - signaled either by a timeout or by duplicate ACK packets - it switches into the stage of congestion avoidance. During the initial slow start phase, BTP increases its congestion window exponentially over time by incrementing the window size with every ACK packet received:

$$cwnd_{i+1} = cwnd_i + 1$$

In the congestion avoidance stage, the congestion window size for the next timestamp i+1 is computed as follows:

$$cwnd_{i+1} = cwnd_i + \frac{WWF}{cwnd_i}$$

with the Window-correlated Weighting Function (WWF) being computed as

$$WWF = \sqrt{\frac{RTT_{max}}{RTT_{current}} \times cwnd_i}$$

As shown in the preceding formula, the congestion window size computation of Elastic-TCP during the congestion avoidance stage requires the RTT of the link to be known. The RTT of the connection is therefore calculated as described in 4.13.

If BTP encounters duplicate ACK packets or packet loss, a multiplicative decrease is applied to the congestion window size:

$$cwnd_{i+1} = \beta \times cwnd_i$$

where β is the constant multiplicative decrease factor, which SHOULD be set to 0.7. It leads to a more stable congestion window size compared to TCP Reno [7, Sec. 2.3.1].

4.8. Flow Control

BTP uses a Selective Repeat [3] sliding window for flow control to enable reliable and efficient delivery of Data packets. The sending system can send up to n Data packets at once, where n is the size of the sliding window. The size of the sliding window is determinded by BTP's Congestion Control algorithm described in 4.7. Once the receiving system receives a Data packet, it answers wih an ACK packet that has the same Header Sequence Number.

Once the sending system has received an ACK packet for the Data packet with the lowest Header Sequence Number (see 4.6) currently within the sliding window, it will move the window further, stoping at the Data packet with the next lowest Header Sequence Number that has not been acknowledged yet.

If the sending system does not receive an ACK packet for a Data packet sent, the same packet is sent again after a timeout as defined in 4.14.

When the receive buffer of the receiving system is full, it SHOULD intentionally send duplicate ACK packets to decrease the window size of the sending system. This follows the Congestion Control algorithm used by BTP.

4.9. Packet Formats

This section outlines all packet types for the server and client. The communication is asymmetric and as such only some packets may be sent by client or server. All packets supported by either party are listed in Tab. 5. All packets except for Data have a fixed length the respective sizes in bytes are listed in Tab. 6.

Packet Name	Server	Client
Conn	No	Yes
ConnAck	Yes	No
Ack	Yes	Yes
Data	Yes	Yes
Close	Yes	Yes

Table 5: Packets to sender matrix.

Packet Name	Size with Header	Size without Header
Conn	8 bytes	4 bytes
ConnAck	10 bytes	6 bytes
Ack	4 bytes	0 bytes
Data	>6 bytes	>2 bytes
Close	5 bytes	1 byte

Table 6: BTP Packets sizes in bytes.

4.9.1. Conn Packet

The Conn Packet is the first packet sent in every connection. By transmitting this packet, the client begins the connection process. The sequence number within the header MUST be selected by the client in accordance to Section 4.6.

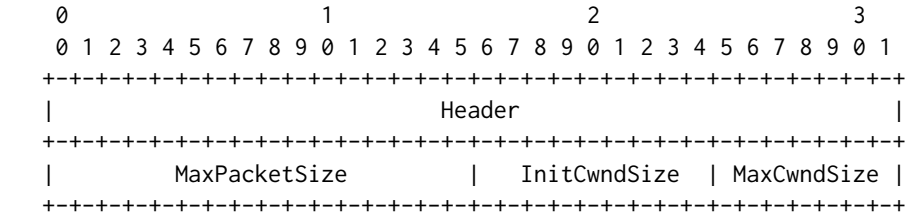

Table 7: Conn Packet fields

Header: 32 bits
Header as described in Section 4.5.

MaxPacketSize: 16 bits
This field is used to inform the server about the maximum possible packet size in bytes the client can process. This field is included to allow clients with limited processing power, networking capabilities or small path MTUs to negotiate reasonable configurations with the server. The server MUST respect this value and can only propose values smaller than the value suggested by the client. After a connection is established, both parties MUST respect this value when transmitting data packets. Data packets larger than the negotiated value SHOULD close the connection, requiring the client to reconnect. The ‘MaxPacketSize‘ is REQUIRED to be at least ‘BtpHeaderSize + Payload Length (2 byte) + 1 byte‘ long. Any value

below would transmit no payload and as such is considered invalid and MUST be rejected by the server in the form of a ‘Close(BadRequest)‘ packet.

InitialCwndSize: 8 bits
Initial congestion window size as proposed by the client.

MaxCwndSize: 8 bits
Maximum congestion window size as proposed by the client.

4.9.2. ConnAck Packet

The ConnAck packet is used by the server to announce the actual maximum packet size to the client and acknowledge the incoming connection. The sequence number within the header MUST be selected by the server in compliance with Section 4.6. This packet will only be sent following a response if the proposed ‘MaxPacketSize‘ and CC options are supported by the server, otherwise a ‘Close(BadRequest)‘ packet MUST be sent instead.

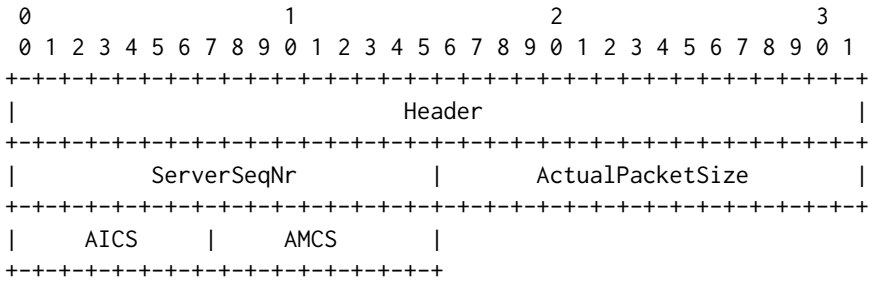

Table 8: ConnAck Packet fields, A.InitCwnd=ActualInitCwndSize

Header: 32 bits
Header as described in Section 4.5.

ServerSeqNr: 16 bits
This value informs the client about the ISN packet by the server. This field MUST match the value send by the client in the preceding Header Sequence Number of the Conn packet.

ActualPacketSize: 16 bits
This field represents the actual maximum packet size in bytes as selected by the server. This value MUST be smaller than Conn MaxPacketSize and also be at least HeaderSize + Payload Length (2 bytes) + 1 in size. It is NOT RECOMMENDED to exceed 1500 byte values to avoid IP layer fragmentation due to MTU restrictions.

ActualInitCwndSize (AICS): 8 bits
The actual initial cwnd size used for the connection as selected by the server. This value MUST be equal or smaller than InitCwndSize proposed in the Conn packet.

ActualMaxCwndSize (AMCS): 8 bits
The actual maximum cwnd size as selected by the server. This value MUST be equal or smaller than MaxCwndSize proposed in the Conn packet.

4.9.3. Ack Packet

The Ack Packet is used by servers and clients to confirm reception of the previously transmitted packet by the other party. As such, the Header Sequence Number of an Ack is not the next sequence number of the server or client, but MUST be the sequence number of the last received packet. The Ack contains no other fields and MUST NOT be counted towards the current sequence number of the transmitting party.

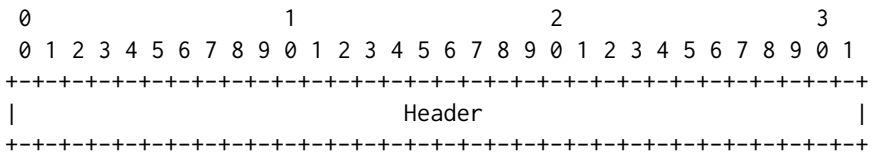

Table 9: Ack Packet fields

Header: 32 bits
Header as described in 4.5.

4.9.4. Data Packet

The Data packet is used to transmit payload between client and server. It is the only BTP packet of variable length. The maximum Data packet size MUST not exceed ‘ActualPacketSize‘ as negotiated during the initial connection handshake.

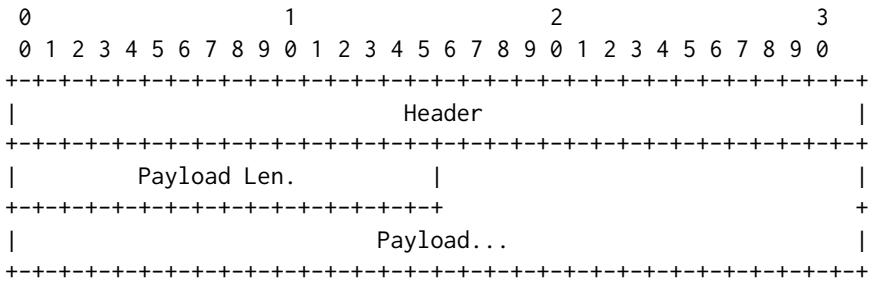

Table 10: Data Packet fields

Header: 32 bits
Header as described in 4.5.

Payload Length: 16 bits
This field is the actual size of the payload included in the Data Packet. This value MUST be at most be ‘ActualPacketSize - BtpHeaderSize + Payload Len. (2 bytes)‘ bytes in size. Any value below is considered valid.

Payload: max. ‘ActualPacketSize - BtpHeaderSize + Payload Len. (2 bytes)‘
Payload is a variable length field that MUST match ‘Payload Length‘ value in its length.

4.9.5. Close Packet

The Close Packet indicates a connection termination, either gracefully or due to an error. This packet may be sent at any time during an ongoing connection by either party. Upon receiving a Close Packet, the receiving party MUST stop all transmissions and SHOULD clear all state associated with the connection.

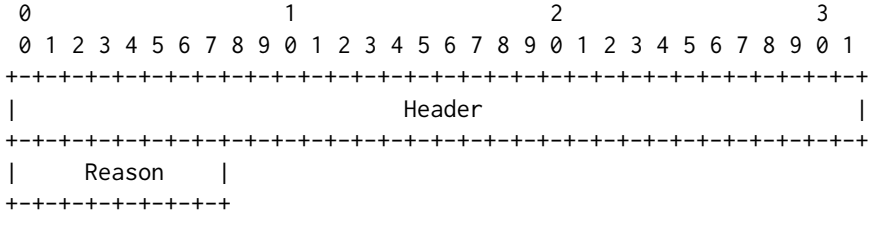

Table 11: Close Packet fields

Header: 32 bits
Header as described in 4.5.

Reason: 8 bits
Reason for closing the connection. This field is included to allow for implementation specific error handling not covered by BTP directly. Meaning of the error codes is covered in Fig. 2

Reason Code	Value
Disconnect	0x01
BadRequest	0x02
Timeout	0x03

Table 12: Close Reason codes

4.10. Establishing a Connection

Before any data can be transmitted over the BTP protocol, a server client association or connection must be established. New connections are established via a 3-way handshake similar to RFC 793 [1]. Both parties sample initial sequence number (ISN) in accordance to Section 4.6 and advertise their ISN to the opposing party in the Conn and ConnAck packets respectively. This handshake is also used to negotiate the ActualMaxPacketSize. Each party MUST store ActualMaxPacketSize for later use during data transmission. Additionally, congestion control configurations are exchanged in this step. The role of ‘InitCwndSize‘, MaxCwndSize‘ is discussed at length in Section 4.7. Fig. 13 illustrates the full connection establishment process. In general, a BTP connection can be in one of three distinct states: Initialization, Open and Closed.

Initialization:
The connection has not completed the initial 3-way connection handshake.

Open:
The connection has completed the initialization step successfully and no errors or Close Packets were encountered.

Closed:
The connection has encountered a critical error or was gracefully closed by one of the parties.

4.11. Data Transfer

Data transfer is only possible when a connection is in the open state as defined in Section 4.10. Data Packets MUST NOT be written into a closed or uninitialized connection, and the opposing party MUST ignore any packet received without a preexisting open connection. BTP allows for data transmission of virtually unlimited data sizes, this payload MUST be encapsulated into Data Packets. If the payload does not fit into a single Data Packet, it must be split into multiple data packets with ‘Payload Length‘ of each packet not exceeding ‘ActualMaxPacketSize - DataPacketSize (without header and payload)‘.

4.12. Closing a Connection

BTP connections SHOULD be closed whenever possible to minimize the number of unused or partially closed connections. Each party can initiate the closing process by transmitting a Close Packet. The receiving party of a Close Packet MUST stop all

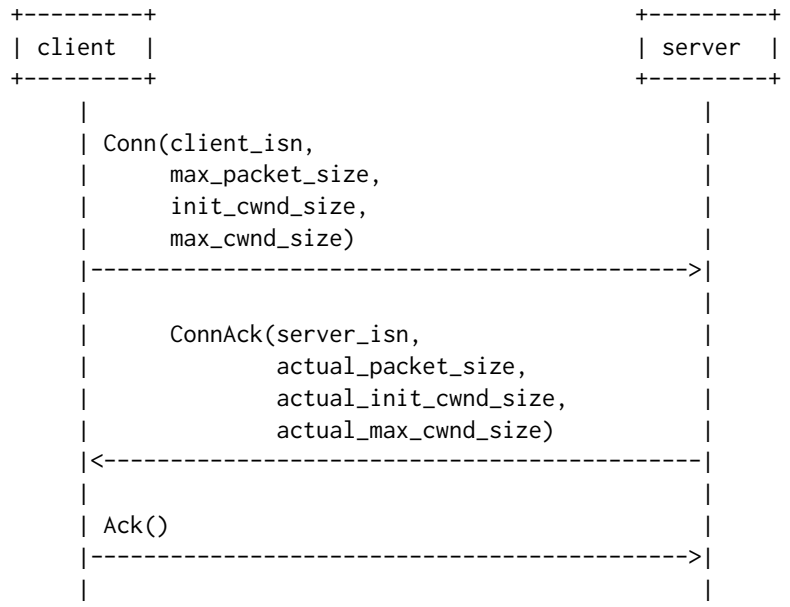

Table 13: Client-Server connection establishment sequence diagram

transmissions and perform implementation specific cleanup. Whenever possible, the receiver of a Close Packet SHOULD transmit an Ack Packet, indicating the Close Packet was received. The sender of the Close Packet MUST close the connection independent of acknowledgement status of the Close packet, when a Close Packet is transmitted the connection SHOULD be assumed to be unreliable.
The reason codes are defined as follows:

Disconnect:
Client or server announces a graceful connection termination. This reason may be sent at any time during an ongoing connection.

BadRequest:
Server or client could not agree upon a compatible maximum packet size. This reason can only be sent during the initial connection handshake.

Timeout
Server or client has not received a valid packet acknowledgement with the Acknowledgement Timeout, will stop re-transmitting and will close the connection. Only the acknowledging party SHOULD send out this reason code.

Figure 2: Client-Server connection closing sequence diagram

4.13. Round Trip Time Computation

Round Trip Time (RTT) is primarily used to estimate values for re-transmission timeouts. Raw packet RTT is computed by the delay between an Ack immediating packet and the associated Ack. ConnAck is treated as a regular Ack in this case and is also used for RTT calculation. Karn's algorithm MUST be used for taking RTT samples [5]. Since Acks for normally transmitted and re-transmitted packets carry the same sequence number, they cannot be distinguished by the receiver and therefore MUST be ignored in RTT measurements.

4.14. Re-transmission Timeouts

The Re-transmission Timeout (RTO) is used to detect lost packets and MUST be started by the sending party after a non-Ack Packet is transmitted. RTO timeouts are handled on a per connection basis. The timeout duration MUST be initially set to a conservative value of two seconds. After a RTT measurement is present RTO duration MUST be adapted. To avoid high variability in the RTT measurement a smoothed RTT (SRTT) MUST be computed. The SRTT computation is analogous to RFC-6298 [6]. Additionally, the RTO MUST use a Back-Off mechanism to prevent re-transmission loops, where the receiver cannot acknowledge the packet within the current RTO duration. The Back-Off mechanism is further specified in RFC-6298 [6].

If the RTO elapses without a matching Ack Packet, the following actions must be performed:

- A new RTO must be computed including the Back-Off component in accordance with RFC-6298 [6].
- The lost packet must be resend with the ‘RETRANSMISSION‘ flag set. The implementation MUST never re-transmit before the RTO has elapsed.

If an Ack Packet is received before the RTO elapses:

- Back-Off MUST be reset in accordance with RFC-6298 [6].
- The RTO MUST be cleared.

4.15. Connection Timeouts

The Connection Timeout is used to detect long-lasting open connections without traffic. Since no data was transmitted over an extended time, the connection could have been interrupted or closed without notice. To prevent such long-lasting idle connections, the Connection Timeout SHOULD be set after each transmitted and acknowledged or received Packet. The Connection Timeout SHOULD be set to 10 minutes. If no new traffic is detected on the connections, the connection SHOULD be closed with Reason Code: Timeout.

4.16. Packet Re-transmission

BTP ensures a reliable connection for all non Ack packets. Lost packets MUST be re-transmitted if no acknowledgement is received within the Acknowledgement Timeout defined in Section 4.14. When the timeout is exceeded, the packet is considered lost and MUST be resent with the original sequence number. The ‘RETRANSMISSION‘ flag MUST be set in the BTP heder. After five subsequent re-transmission timeouts, a connection is assumed to be broken and a close event MUST be sent by the sending party.

5. Best Robust File Transfer Protocol

The Best Robust File Transfer (BRFT) protocol utilizes the BTP layer as a reliable communication protocol. BRFT in turn defines file transfer specific capabilities. In order to allow for a more efficient transfer of files, clients and servers MIGHT implement compression.

5.1. Encoding

All fields MUST be encoded in big-endian byte-order. Strings like file names MUST be UTF-8 encoded.

5.2. BRFT Header Format

Similarly to the BTP Header described in Section 4.5, BRFT also has a header which is prefixed to every packet. However, since packet ordering is already provided by the BTP layer, a sequence number is not needed. The format of the header can be seen in Fig. 5.2.

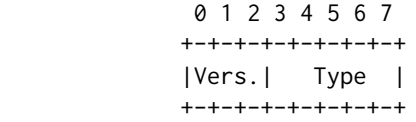

Table 14: BRFT Header Version and Message Type

Version: 3 bit
Used to provide a version control for future updates. The different versions and corresponding encoding can be seen in Tab. 15.

Name	Bit Encoding
BRFTv0	001

Table 15: Currently defined BRFT Versions

Type: 5 bit
Used to determine the different packet types. The different versions and corresponding encoding can be seen in Tab. 16.

Name	Bit Encoding
FileReq	00001
FileResp	00010
Data	00011
StartTransmission	00100
Cancel	00101

Table 16: Currently defined BRFT Packet Types

5.3. Finding a file

Before requesting a file, a client MUST acquire the file name, i.e., the server's identifier of the file. The client MAY also need the file's metadata to decide whether to request the corresponding file. To this end, the BRFT includes the MetaDataReq and MetaDataResp packets of Section 5.5.1 and 5.5.2, respectively.

Using a MetaDataReq, a client MAY either request the server to advertise the names of the files it offers for download, or it may request the metadata of a specific file by setting a valid file name in the request. A server MUST then respond with a MetaDataResp consisting of MetaDataItems. If the server received a MetaDataReq containg a single file name and the file name corresponds to a file advertised by the server, the server MUST return a MetaDataResp with one extended MetaDataItem containing the file's name, size, and checksum. The client MAY use this metadata to determine if the file is suitable for download. Further, if the specified file name is invalid or does not correspond to a file offered by the server, the server returns a MetaDataResp with 0 MetaDataItems. If instead the server receives a MetaDataReq with no file name set, the server returns a MetaDataResp consisting of MetaDataItems corresponding to the files the server offers for download. These MetaDataItems only contain the file's names. If a MetaDataResp contains the maximum number MetaDataItems, i.e., 255 MetaDataItems, the client MUST assume that the server will send a following MetaDataResp containing additional items. A server MUST only stop sending MetaDataResp packets once it sends a MetaDataResp with less than 255 MetaDataItems. As such, a server may advertise an unlimited number of files for download. If the server does not offer any files for download, it returns a MetaDataResp with 0 MetaDataItems. Figure 17 displays this process.

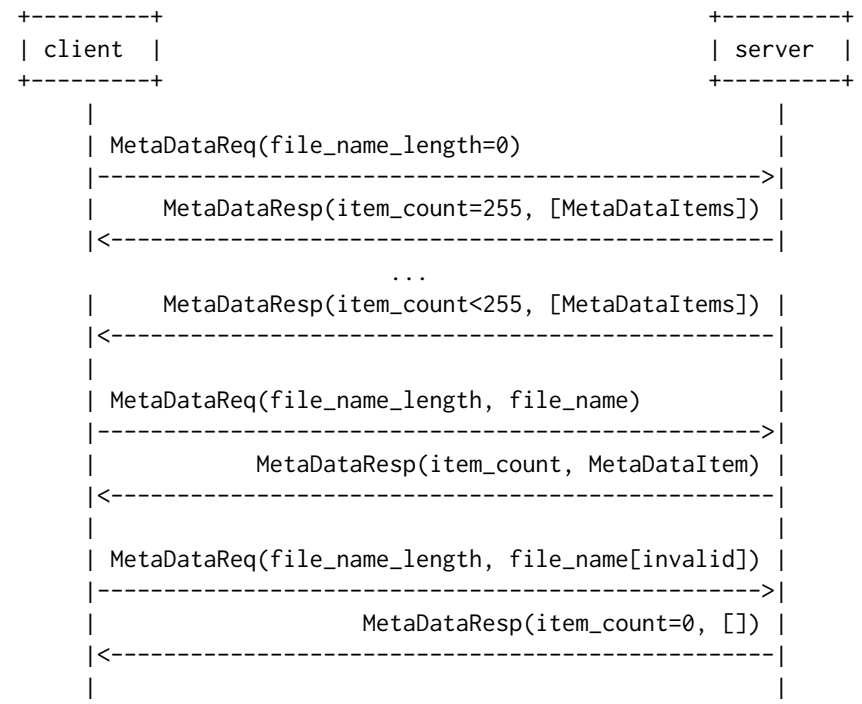

Table 17: Sequence Diagram of the two types of MetaDataReqs

5.4. Requesting a file

At the core of BRFT, a client has to be able to request a certain file that should be sent to it by the server. Intuitively, the client first has to select the file it wants to download in the FileReq packet. The file name is assumed to be known to the client beforehand or by finding it as described in Sec. 5.3. The server then responds whether it can provide the requested file. Subsequently, the client acknowledges the start of the actual data transmission. After the file has been transferred, the server sends a final packet, indicating that the transfer is complete. This typical sequence can be seen in Fig. 18.

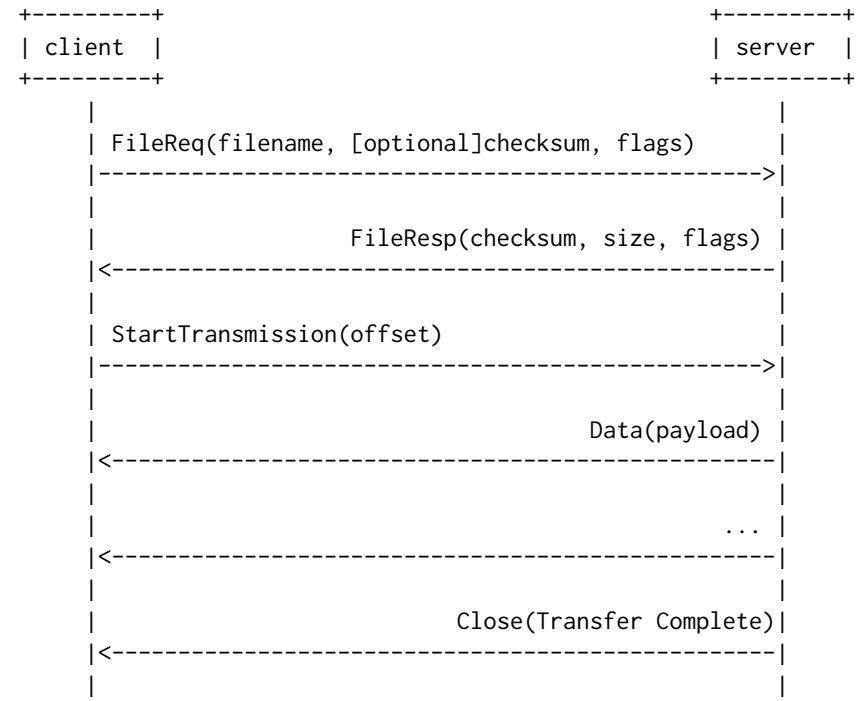

Table 18: Sequence Diagram of File Request and Data Transmission

5.5. Packet Formats

This section defines all packet types for the server and client.

5.5.1. MetaDataReq Packet

Clients may discover server contents via this MetaDataReq packet, shown in Figure 19. A corresponding server SHOULD thereby implement rate limits for the MetaDataReq per client, as otherwise a client may easily perform a DoS attack by sending excess MetaDataReq packets.

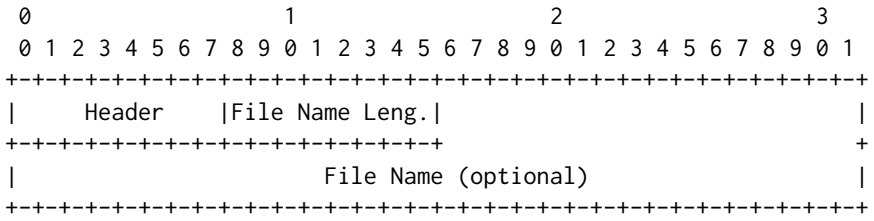

Table 19: MetaDataReq Packet Format

Header: 8 bits
Header as described in Section 5.2

File Name Length: 8 bits
Length of the supplied file name field.

File Name: max. 255 bytes
Optional name of the file name without any path prefix using UTF-8 encoding. If this field is empty, this requests demands the server to list all available files. If this field is non-empty, this requests demands the corresponding file metadata. The filename MUST correspond to the following regex: ^([\w-])+([\.]([\w-])+)?$.

5.5.2. MetaDataResp

Upon receiving a MetaDataRequest packet of Section 5.5.1, a server MUST respond with a MetaDataResp packet, shown in Figure 20. A MetaDataResp packet consists of the subpackets MetaDataItems, described in this section.

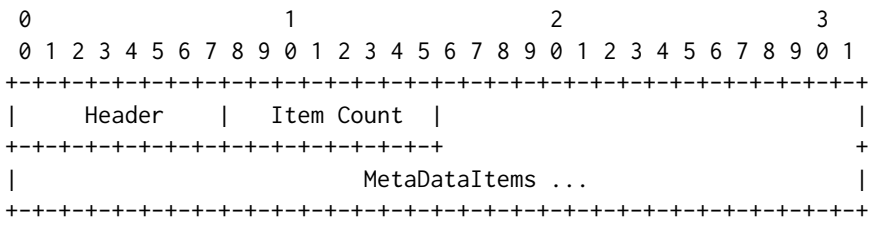

Table 20: MetaDataResp Packet Format

Header: 8 bits
Header as described in Section 5.2

Item Count: 8 bits
The number of MetaDataItems contained in a MetaDataResp. At most, a server may respond with 255 MetaDataItems. If a server responds with 255 MetaDataItems, the client assumes that the server offers more files for download. Hence, the server MUST send another MetaDataResp packet with additional MetaDataItems. If the server advertises no more additional files, it returns a MetaDataResp with 0 MetaDataItems. If a server either offers no files for download or the client specified a non-existent or invalid file name, the server returns a MetaDataItem count of 0, and further includes no MetaDataItems in the MetaDataResp.

MetaDataItems: 8 bits
The MetaDataItems composing the payload of the MetaDataResp packets. These packets contain the server-advertised file metadata, shown in Figure 21. Every MetaDataItem MUST include the file name length and the corresponding file name. If the corresponding MetaDataReq specifies a file name, the server MUST extend the MetaDataItem with the file size and checksum corresponding to the requested file.

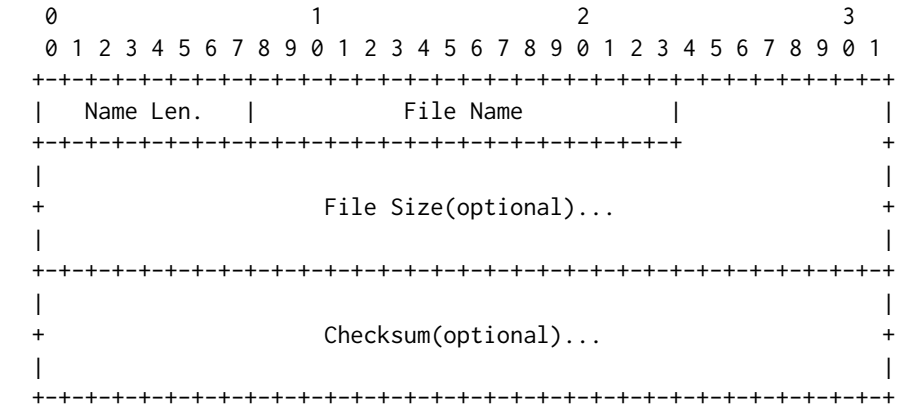

Table 21: MetaDataResp Packet Format

MetaDataItem Name Length: 8 bits
Length of the supplied file name field.

MetaDataItem File Name: max. 255 bytes
Name of the file name without any path prefix using UTF-8 encoding. The filename MUST correspond to the following regex: ^([\w-])+([\.]([\w-])+)?$.

MetaDataItem File Size: 64 bits
Optionally, the size of the target file for which the client is making a targeted MetaDataReq. The server MUST supply this field if the client supplied a single valid file name in the MetaDataReq packet. Further, the server MUST NOT supply this field if the client did not specify a file name in the MetaDataReq packet. The client SHOULD ensure there is enough storage space left on one of its devices and potentially allocate other resources if it intends to download the corresponding file.

MetaDataItem Checksum: 32 bytes
Optionally, the SHA-256 hash of the complete file of which the client made a targeted MetaDataReq. The server MUST supply this field if the client supplied a single valid file name in the MetaDataReq packet. Further, the server MUST NOT supply this field if the client did not specify a file name in the MetaDataReq packet. The server MIGHT store these checksums in order to remove redundant computation.

5.5.3. FileReq Packet

The format of the FileReq packet can be seen in Fig. 22. While it would be possible to replace the file name length and file name with a fixed-size, unique identifier, like a SHA-1 hash of the filename, this would force the server to hash all known file names every time a file request arrives or, alternatively, create a lookup table.

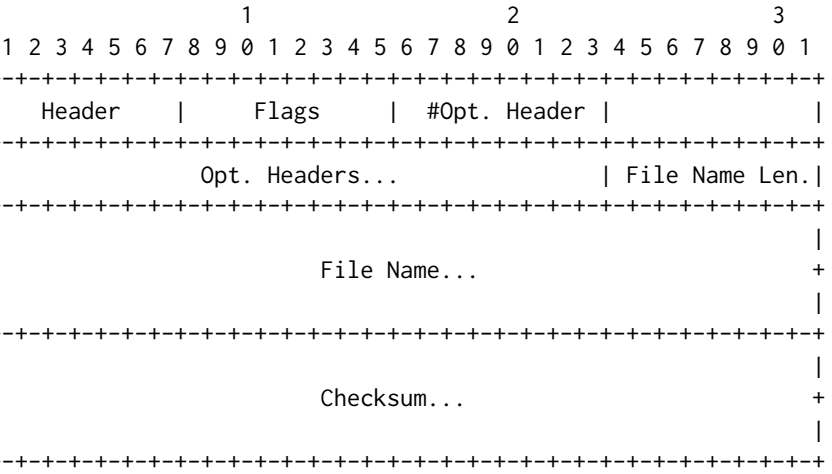

Table 22: FileReq Packet Format

Header: 8 bits
Header as described in Section 5.2

Flags: 8 bits
OR concatenated flags for the server. In this protocol version, only a flag for resumption is actually used. If a client wants to resume a download and therefore sets the Resumption flag, the checksum field MUST be set to a non-zero value. The encoding can be seen in Tab. 23.

#Optional Header: 8 bits
Number of optional headers followed after this field.

Optional Headers:
Optional headers of the FileReq packet as described in section **??**

Name	Bit Encoding
Resumption	00000001
Reserved	00000010
Reserved	00000100
Reserved	...
Reserved	10000000

Table 23: Currently defined FileReq Packet Flags

File Name Length: 8 bits
Length of the filename. Using an 8 bit field allows the filename to be at most 255 bytes long. This is the same length most unix-like file systems allow. The length does not include the length field itself.

File Name: max. 255 bytes
Actual human-readable name of the file name without any path prefix using UTF-8 encoding. The filename MUST correspond to the following regex: ^([\w-])+([\.]([\w-])+)?$.

Checksum: 32 bytes
SHA-256 hash of the complete file. If the client does not know a checksum for the file in question, the checksum MUST be set to zero. However, if the client already knows the a valid checksum of the file, he might request this version of the file by setting the checksum. Similarly, if the client already has a partial download of a file and wants to resume the transfer (i.e. Resumption flag set) it MUST set the checksum to the one advertised by the server during the previous session. A more detailed look at the resumption can be found in Section 5.8. Additionally, the considerations about file storage as described in Section 5.12 should be taken into account for servers providing multiple file versions.

5.5.4. FileResp Packet

After receiving a FileReq, the server answers the client with a file response (FileResp) packet. The server MUST provide the SHA-256 checksum of the requested file, if the file is present. The format of which can be seen in Fig. 5.5.4.

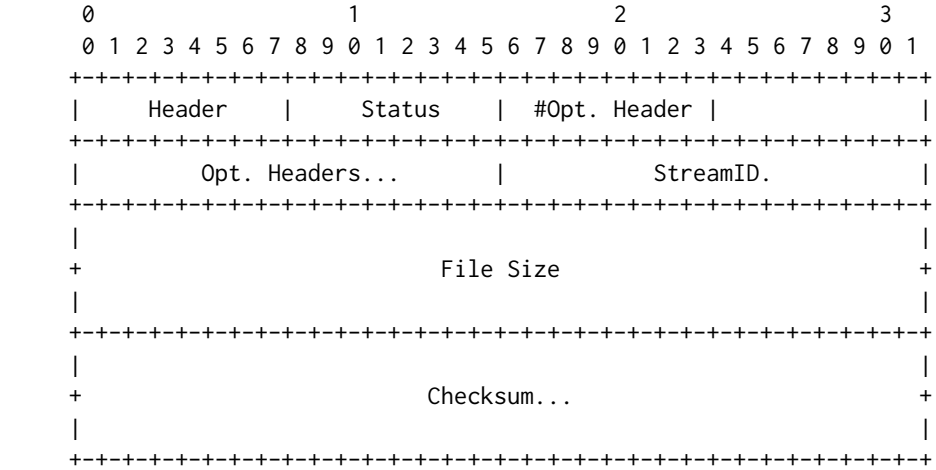

Table 24: FileResp Packet Format

Header: 8 bits
Header as described in Section 5.2.

Status: 8 bits
Status flags indicating non connection terminating information for the client. The encoding of all defined statuses can be seen in Tab. 25.

Name	Bit Encoding	Hex Encoding
Reserved	00000000	0x00
OK	00000001	0x01
FileChanged	00000010	0x02
UnsupportedOptionalHeader	00000011	0x03
UnexpectedOptionalHeader	00000100	0x04
Undefined	...	
Undefined	11111111	0xFF

Table 25: Currently defined FileReq Packet Flags

OK:
Indicates to the client that there are no complications.

FileChanged:
Set if the file the server presents to the client has a checksum different to the one requested by the client. This flag MUST only be set if server does not possess a matching version of the file with the checksum of the FileReq. If the file does not exist at all, the server MUST terminate the connection using the Close packet as described in Section 5.5.7.

UnsupportedOptionalHeader:
Set if the server does not know or support at least one of the optional headers sent by the clients. The server SHOULD close the connection after sending the FileResp. This status has a lower precedence than the FileChanged and UnexpectedOptionalHeader status.

UnexpectedOptionalHeader:
Might for example be set, if the client sent an optional header that does not belong in a FileReq packet. The server SHOULD close the connection after sending the FileResp. This status has a lower precedence than the FileChanged status.

#Optional Header: 8 bits
Number of optional headers followed after this field.

Optional Headers:
Optional headers of the FileResp packet as described in Section 5.6.

StreamID: 16 bits
Unique ID for a client download session. Used in order to allow the client to concurrently download multiple files. The server MIGHT either choose an ID that is unique per client or for all clients. The latter removing the need to combine the clients IP address and StreamID to get a globally unique identifier.

File Size: 64 bits
Size of the file that the server intends to send to the client. This SHOULD be used by the client to make sure there is enough storage space left on one of its devices and potentially allocate other resources. However, because there might be other processes reducing the amount of storage, the client SHOULD also be able to gracefully stop the download in case remaining storage is running low.

Checksum: 32 bytes
SHA-256 hash of the complete file which the server intends to send the client. The server MIGHT store these checksums in order to remove redundant computation

5.5.5. StartTransmission Packet

As the server might advertise a version to the client that does not match the checksum initially requested by the client, there is one more packet that the client has to send in order to start the transmission - the StartTransmission packet. In this packet, the client acknowledges that it wants to begin the transmission as previously negotiated.

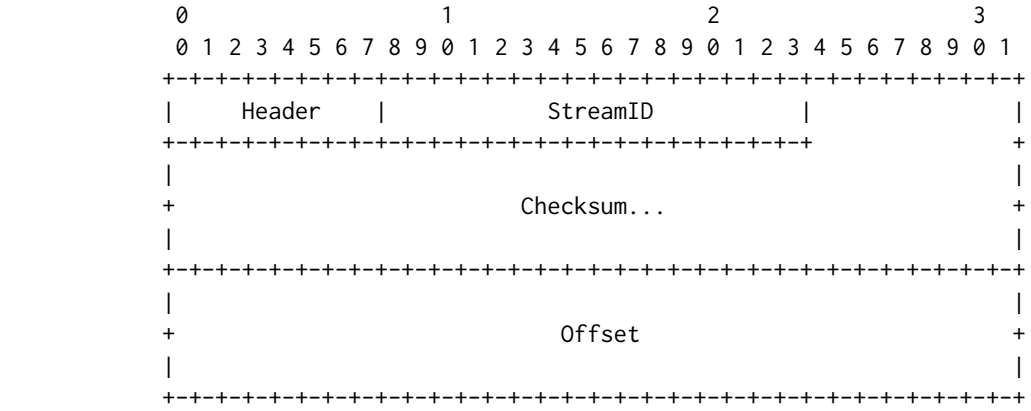

Table 26: StartTransmission Packet Format

Header: 8 bits
Header as described in Section 5.2.

StreamID: 16 bit
StreamID as defined by the servers FileResp.

Checksum: 32 bytes
SHA-256 hash of the complete file as advertised by the server in FileResp. The server MUST ensure that the client provides the same checksum as in FileResp and that the associated file has not changed. If either is not the case, the server SHOULD terminate the connection using a Close packet described in Section 5.5.7.

Offset: 64 bit
The (uncompressed) offset in bytes. MUST be set if the client wants to resume a previous partial download. Otherwise, set to zero. Since the client is only capable of decompressing complete chunks, at this time, the offset will always be a multiple of the chunk size.

5.5.6. Data Packet

Since the data packet will be the packet sent most often, it should be kept small. Therefore, the overhead by control information is limited to 48 bits. The Format can be seen in Fig. 27.

Header: 8 bits
Header as described in Section 5.2.

StreamID: 16 bit
StreamID as defined by the servers FileResp.

Payload Length: 24 bit
Length of the payload field in bytes. The length does not include the length field itself.

Payload: max. 16 MB
Actual payload consisting of one chunk of compressed file content as described in Section 5.7.

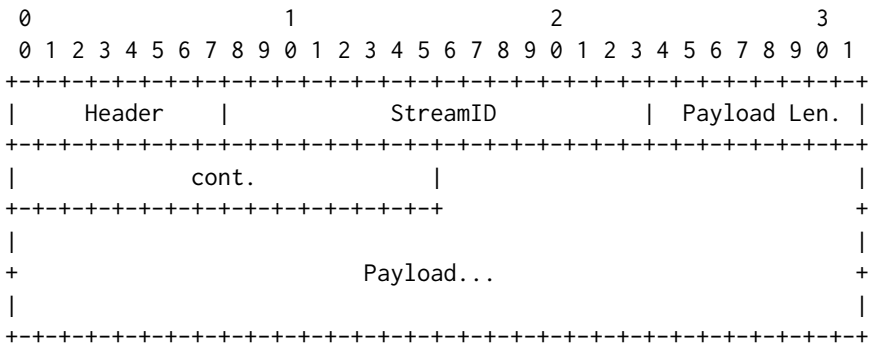

Table 27: StartTransmission Packet Format

5.5.7. Close Packet

Either the client or the server can terminate the transfer of a file. To signal to the opposing party that the connection is being closed as well as to provide a reason for doing so, a Close packet SHOULD be sent before termination. The format of this packet can be seen in Fig. 28.

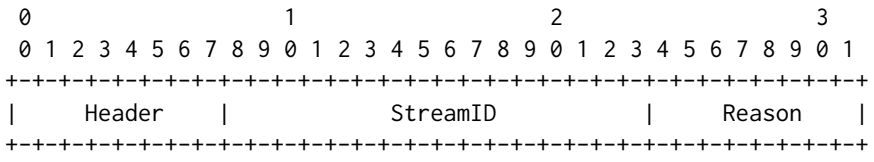

Table 28: Close Packet Format

Header: 8 bits
Header as described in Section 5.2.

StreamID: 16 bit
StreamID as defined by the servers FileResp.

Reason: 8 bits
Reason for closing the connection. The different reasons and their encoding can be seen in Tab. 29.

Name	Bit Encoding	Hex Encoding
Undefined	00000000	0x00
TransferComplete	00000001	0x01
ChecksumInvalid	00000010	0x02
InvalidOffset	00000011	0x03
NotEnoughSpace	00000100	0x04
InvalidFlags	00000101	0x05
ResumeNoChecksum	00000110	0x06
FileNotFound	00000111	0x07
UnsupportedOptionalHeader	00001000	0x08
UnexpectedOptionalHeader	00001001	0x09
Reserved	00001010	0x0A
Reserved	...	
Reserved	11111111	0xFF

Table 29: Currently defined Close Reasons

Undefined:
Catch all fallback.

TransferComplete:
Sent by the server in order to indicate that the transfer of the file has been completed.

ChecksumInvalid:
Either sent by the client if the server advertises a checksum in the FileResp that is not desired by the client, or by the server if the client sends an incorrect checksum in the StartTransmission packet.

InvalidOffset:
Sent by the server if the client sends an invalid offset in the StartTransmission packet. One example would be that the offset exceeds the length of the requested file.

NotEnoughSpace:
Sent by the client if the remaining storage space is not sufficient (anymore).

InvalidFlags:
Sent by the server if the client is sending Flags in the FileReq packet that are not defined yet or – in future versions – cannot be used in conjunction with one another.

ResumeNoChecksum:
Sent by the server if the client requests a resumption in the FileReq, i.e. setting the Resumption flag, but the checksum field of the same packet is zero. More information can be found in Sec. 5.8

FileNotFound:
Sent by the server if no file associated with the requested file name exists or the checksum does not match. More information can be found in Sec. 5.8

UnsupportedOptionalHeader:
This reason SHOULD be set by the server when the connection is closed, after the client has sent an optional header that is not know to the server. Currently, all optional header negotiations are initiated by the client and only acknowledged by the server. Therefore, the client should never come in the situation of encountering an unknown optional header in the FileResp.

UnexpectedOptionalHeader:
This option can be used by either the server or the client. One case would be if the server, or client, receives an optional header that is not supposed to be in a FileReq, or FileResp respectively. For example, if the client sends a CompressionResp header within a FileReq. Another scenario would be if the server sends a FileResp with a CompressionResp after the client did not send a corresponding CompressionReq in the FileReq.

5.6. Optional Headers

In order to configure options that require additional parameters instead of a simple flag, optional headers are introduced for the FileReq and FileResp packets. This section describes the general format of optional headers as well as the currently defined optional headers.

5.6.1. Optional Header Format

Optional headers are set by the client and confirmed by the server. Any future optional headers SHOULD also implement this pattern since a server might not provide a feature. Therefore, without confirmation by the server, the client might assume a certain feature is active that the server does not actually apply.

Both the optional headers of the FileReq as well as FileResp packet need a length and type as can be seen in figure 30

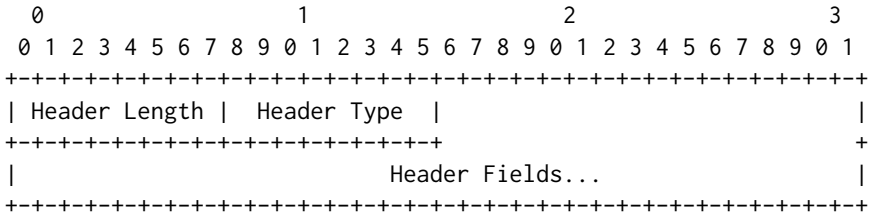

Table 30: Optional Header Format

Header Length: 8 bits
Length of this optional header in bytes; not including the length field itself. Because the length field allows to skip the optional header, it enables compatibility between clients and servers where one does not know a concrete optional header. Additionally, variable length optional headers might be implemented. Both, the server and the client, MAY introduce plausibility checks on the length of optional headers in order to mitigate the risk of being sent unusually long headers.

Header Type: 8 bits
The type of the optional header. The currently defined types and their encoding can be seen in Tab. 31.

Name	Bit Encoding	Hex Encoding
Reserved	00000000	0x00
CompressionReq	00000001	0x01
CompressionResp	00000010	0x02
Undefined	00000011	0x03
Undefined	...	
Undefined	11111111	0xFF

Table 31: Currently defined Optional Header Types

Header Fields:
Additional header fields of the concrete optional header defined by the type

5.6.2. CompressionReq

In order for the client to utilize compression of the requested file during transport, a CompressionReq optional header MUST be added to the FileReq packet. The format of such an optional header can be seen in Tab. 32.

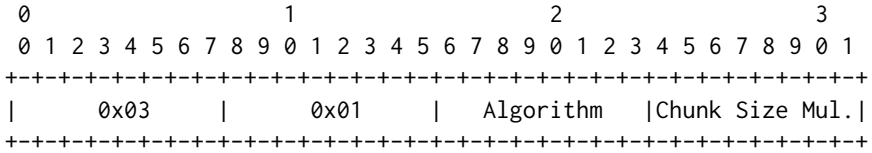

Table 32: CompressionReq Format

0x03: 8 bits
Header Length as described in 5.6.1

0x01: 8 bits
Header Type as described in 5.6.1

Algorithm: 8 bits
Algorithm used to compress the file for transport. Currently, only the gzip compression algorithm [6] is support, as can be seen in Tab. 33

Name	Bit Encoding	Hex Encoding
Reserved	00000000	0x00
gzip	00000001	0x01
Undefined	...	
Undefined	11111111	0xFF

Table 33: Currently defined Compression Algorithms

Chunk Size Multiplier: 8 bits
Chunk size multiplier used to compress the file. The actual chunk size is calculated by:

$$chunk\_size = (chunk\_size\_multiplier + 1) * 64kB.$$

This way chunk sizes smaller than 64kB are impossible. Such small chunk sizes might result in overhead that would render the compression useless.

5.6.3. CompressionResp

Once a server supporting file compression receives a CompressionReq from the client he SHOULD respond using a CompressionResp in the FileResp. The format of such an optional header can be seen in 34.

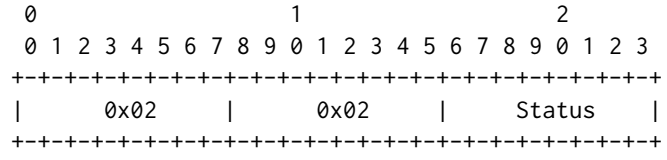

Table 34: CompressionResp Format

0x02: 8 bits
Header Length as described in 5.6.1

0x02: 8 bits
Header Type as described in 5.6.1

Status: 8 bits
Status allows the server to indicate whether compression will be applied or the reason why no compression will be applied. The different statuses and their encodings can be seen in Tab. 35.

In case the server sends an OK status, the upcoming file transfer must be compressed according to section 5.7

If the server deems the requested file to be too small for compression, the FileTooSmall status SHOULD be returned. While the server can set an arbitrary limit on the minimum file size that will be compressed, the server SHOULD require the file size to be at least bigger than the minimum chunk size (i.e. 64kB) in order to account for compression overhead. Similarly, if the server does not support the requested compression algorithm, the UnknownAlgorithm status SHOULD be returned. Furthermore, the NoCompression status gives the server an option to back out of compression because of any other reason.

If the server returns an NoCompression, UnknownAlgorithm or FileTooSmall status, the file MUST NOT be compressed in the following file transfer. In turn, if the

client does not want to continue without compression, a new connection can be established by sending another FileReq using a different CompressionReq header instead of the StartTransmission packet.

Name	Bit Encoding	Hex Encoding
Reserved	00000000	0x00
OK	00000001	0x01
NoCompression	00000010	0x02
UnknownAlgorithm	00000011	0x03
FileTooSmall	00000100	0x04
Undefined	...	
Undefined	11111111	0xFF

Table 35: Currently defined CompressionResp Statuses

5.7. Compressed Chunked File Coding

As has been already touched upon in Sec. 5.6.2 and 5.6.3, the BRFT layer MIGHT apply a file chunk encoding transformation on the transferred files. Before transferring files, the layer splits the file into chunks and compresses said chunks. Upon receiving the compressed bytes from the BTP layer, the BRFT layer parses the bytes into chunks and decompresses them. After the BRFT layer received all chunks, the layer assembles the decompressed chunks into the original file.

By compressing the individual chunks of a file before passing them to the BTP layer underneath, the BTP layer must not send as much data as a decompressed file. Hence, the BTP layer may perform less flow and congestion control and requires less time overall to send the file, resulting in an increase in performance.

Because resumption, as described in Section 5.8, requires the decompressed byte-offset, the client must decompress the incoming data on-the-fly. Decompressing a byte stream instead of chunks on-the-fly leads to various difficulties. Gzip, the only compression algorithm currently supported by BRFT, uses length-checks, checksums, and internal chunks. Moreover, in case of a transfer interruption, the BRFT layer must decompress a fragment of the compressed file to obtain the decompressed byte-offset. Hence, decompressing an incomplete gzip-compressed file conflicts with gzip's functions, and is not supported by all gzip implementations. As such, the BRFT layer decompresses chunk-wise, instead of decompressing bytes upon reception.

To prepare files for file transfer, the BRFT layer MUST split the to be transferred file into chunks. The to be compressed chunks MUST have a size of

$$chunk\_size = (chunk\_size\_multiplier + 1) * 64kB,$$

where the $chunk\_size\_multiplier$ is the multiplier defined in the CompressionReq. Further, the layer MUST compress the chunks using the algorithm that has been specified in the CompressionReq as well. Once the client receives a compressed chunk, the BRFT layer decompresses the gzip-encoded chunk. The BRFT layer can then compute the total uncompressed byte-offset, required for initiating a file transfer resumption. Finally, upon receiving all chunks, BRFT layer MUST reassemble the original file using the decompressed chunks.

5.8. Resumption

If the transmission of a file has been prematurely stopped, the client MAY attempt to resume the download of this file. For a potential resumption, the client must be able to identify the version of a file. Therefore, the client MUST store the checksum advertised by the server during the initial transfer until the download of

the file is complete. Some information on how this can be accomplished is discussed in Section 5.12.

A FileReq utilizing resumption MUST have the Resumption flag as well as the initial checksum set. If only the Resumption flag is set, the server SHOULD end the session by sending a Close packet with ResumeNoChecksum. A non-zero checksum field without the Resumption flag set MAY be used by a server advertising multiple versions of a file.

There are multiple scenarios for the server when receiving a FileReq packet:

1. no file for this file name exists: The server SHOULD send a Close packet with the FileNotFound reason.

2. a file with a matching checksum exists: The server MUST return a FileResp with the same checksum as in the FileReq to the client.

3. a file for the file name exists, but the checksums do not match: The server MAY advertise the checksum of this file instead in the FileResp. Note that this can be either the case if the server holds multiple versions of one file, or if the server only holds one version of a file and the file has been altered since the initial transfer. In either case, if the checksum in the FileResp differs from the one in the FileReq, the File Changed status flag MUST be set in the FileResp.

The client MUST then compare the checksum of the FileReq with the one of the FileResp. If both match a StartTransmission packet with the same checksum and the uncompressed byte offset, i.e. the number of bytes that have already been received and decompressed during the initial transfer, MUST be sent.

If the checksums do not match, the client can decide to either

1. not proceed the with the download: The client SHOULD send a Close packet with the ChecksumInvalid flag set.

2. start the download of the file version advertised by the server. The client SHOULD send a StartTransmission packet with the checksum of the FileResp and an offset of zero.

Finally, the server MUST ensure that it can still provide the version of the file in question. If not, or if the client has sent a checksum differing from the one in FileResp, the server SHOULD terminate the connection by sending a Close packet with the ChecksumInvalid reason set. Otherwise, the actual file transfer can resume at the given offset.

5.9. Multiple Concurrent Transfers

A server MUST be able to transfer multiple files concurrently. A client can request multiple files from a single server by sending multiple FileReqs. Per received valid FileReq, the server initiates a new BRFT layer connection. The client and server differentiate between the connections by a unique Stream ID set in the server's FileResp. These connections MAY use the single BTP connection. However, no new ports SHOULD be allocated by server to listen for incoming packets.

5.10. Timeouts

The BTP layer already handles timeouts for the connection, as described in 4.15. Therefore, the BRFT layer is not concerned about timeouts. It will simply rely on the BTP layer to close any stall connection.

5.11. Ending transfer

After the server sent the final Data packet, it MUST send a final Close packet with the Transfer Complete reason. After decompressing the final chunk, the client SHOULD then validate the file integrity using the checksum. Subsequently, any state that has been hold by the client and server MAY be removed.

5.12. File Storage

While the concrete storage of files is up to the implementation, this section lays out some requirements and guidance.

Even though a file update to a server is not part of the current specification of the BRFT protocol, one should note that the server must ensure that a file that is currently transferred (i.e. StartTransmission packet has been received) MUST NOT be altered. This might be accomplished by creating a lock file or creating a copy of the file in question.

In order to increase the likelihood of a server actually being able to provide a file matching the checksum of a client request, the server MAY store multiple different versions of one file. While one could simply append a unique string to the filename, the maximum file length must be kept in mind. Since most file systems only allow file names with a maximum of 255 characters, as does BRFT, such a simple approach is most likely not suitable. A more viable approach would be to create a directory for every file and use an encoded version of the checksum as the filename. Also note that the server still has to ensure that a file is not altered while it is transferred to some client. However, a server storing multiple versions MIGHT decide to not delete any files.

Independent of whether the server holds multiple file versions, an implementation SHOULD consider caching the checksum of the different files. Otherwise, a malicious actor could cause a significant load on a server by sending multiple file requests.

A client MUST persist the checksum of a file until the transfer of which is complete. This is because during resumption, this checksum must be sent to the server in order to identify the correct file version. This MIGHT be achieved by creating a temporary file type containing the checksum in question as well as the actual file content. However, in case of a resumption, the client MUST send the server the already received file length without this additional information.

5.13. Connection Migration

A client and server handle connection migration implicitly as the connection times-out. As in case of a connection migration the IP of the client changes, the server Data packets no longer reach the client. Further, due to the server no longer receiving Acks from the client, it stops sending Data packets. Subsequently, as a reaction to the connection time-out, the client resumes the file download, as described in Section 5.8.

6. Security Considerations

In this section, we do not consider one specific attack vector. As such, attackers could for example obtain a valid packet from a client-server connection, alter it and send it to the original recipient. In such a case, ensuring the integrity of the packets would be sufficient. However, this still allows the attacker to create new packets. Such attacks can be mitigated, ensuring the authenticity.

6.1. BTP Spoofing

A malicious party can falsify a L1 connection by spoofing the IP address of the underlying UDP packets. As such, the BRFTP server would respond to a client determined by the attacker. For this reason, the BTP protocol specification recommends using a sufficiently random initial sequence number generation to make such connection establishment more difficult. This mechanism only works if the attacker can guess the server sequence number or eavesdrop on ongoing traffic. Assuming the attacker can determine the initial sequence number, the attacker can complete the initial three-way handshake, as defined in Section 4.10 and by that open a spoofed BTP connection. Effective BTP spoofing would require L1 packets to not be protected against alteration (integrity) or the attacker must not be able to create valid L1 packets for other parties (authentication). This applies specifically to the sequence number.

6.2. Amplification Attacks

The current BRFT design may be susceptible to amplification attacks. A potential attacker may utilize a BRFT server to overload a different system. There are two distinct amplification vectors possible within the BRFT part of the protocol. As BRFT requires a valid BTP connection, the attacker must first establish a spoofed BTP connection for the respective target 6.1. Given a spoofed BTP connection was successfully established, the attacker can now perform a number of amplification attacks:

MetaData amplification
: The attacker can ask the server to provide metadata on available files, as outlined in Section 5.5.1. As the connection is maliciously constructed, the resulting information will be sent to the previously selected victim. With a minimum request size of 2 bytes, the attacker can amplify the traffic to a worst case factor of 32641 (excluding BTP wrapping). Achieving this value is only possible if the server offers 255 files and each of them has a 255 byte long filename, resulting in an 65282 byte long MetaDataResp packet. Furthermore, this amount of data will only be transmitted if the BTP layer has a sufficiently large congestion window or the attacker can transmit BTP layer Acks.

Partial File Request
: A malicious connection could be used to transmit requested file contents to a victim. To achieve this, an attacker must successfully complete the file request process and transmit a StartTransmission packet, as outlined in Section 5.4. After this, the server would transmit at least the current congestion window amount of packets to the victim. As in the previous attack, this process can be extended by the attacker via the transmission of valid Ack packets.

Both amplification attacks could be mitigated by introducing an authentication mechanism to the server. Furthermore, a Proof-of-Work approach could be introduced to increase the cost of the attack. In the case that an attacker cannot send valid Ack packets, it should be noted, that the effect of this attack can be further limited by using conservative congestion window sizes.

6.3. Replay Attacks

Since the BRFTP layer does not contain any mechanism to detect old messages, all packet types can theoretically be replayed. However, the initial concept of the BTP layer was to provide a reliable "baseline" communication layer, as such, it offers a convenient point for the mitigation of replay attacks.

For this task, integrity and authenticity of the sequence number of all BTP packets as well as the payload of the BTP Data packet should be enforced. Additionally, the client and server should reject any packets that have a sequence number that is not contained within the currently expected range.

The mitigation of BRFTP layer replay attacks is delegated to the BTP layer as well. This also guarantees the integrity and authenticity of the Data packet payload. Regarding replay attacks of packets of the BRFTP layer, a spoofed BTP connection is needed as described in Section 6.1. At the current state of BRFTP, packets without the StreamID field - namely MetaDataReq, MetaDataResp and FileReq - may be susceptible to replay attacks. However, we do not consider this information to be worth protecting, as it is publicly available. For the other packet types that do contain the StreamID field, a specific ID can be tied to the IP address of the receiving system, since the receiving system would get assigned a new ID on connection resumption. This way, replayed packets from a third party can be discarded based on the IP address.

Regarding the StartTransmission packet, the server needs to ensure that such a packet with the same StreamID is only accepted once. Note, that resumed connections are not affected by this, because a resumption entails a new StreamID. Furthermore, if the StartTransmission packet (especially its StreamID) is not authenticated, it is also sufficient for the attacker to sniff for a single data packet, extract the StreamID and create a new StartTransmission packet.

6.4. Limiting Transmission Rate

An attacker can intentionally decrease the size of the congestion window by excessively replaying ACK packets. As Elastic-TCP enters Congestion Avoidance mode and applies Multiplicative Decrease as soon as duplicate ACK packets are detected, this could reduce the window size all the way down to a single frame, effectively turning the Congestion Control into lock-step. As with other forms of ACK flood attacks, distinguishing intentionally replayed ACK packets from legitimate duplicates caused by a congestion is difficult. The sequence number of ACK packets can be used to discard any packets that are not within the currently expected range. In addition, a timestamp could be used to limit the range of valid ACK packets. However, in both cases the range cannot be too narrow to avoid unnecessary re-transmissions. Therefore, this cannot fully eliminate the risk of limited transmission rate due to maliciously replayed ACK packets.

6.5. Closing Connections

An attacker may forge a close packet for either the BTP or BRFTP layer to prematurely close the corresponding layer's connection. Using this method, an attacker may specifically deny individual clients service to the file server.

6.6. Closing BTP Connections

To execute this attack in the BTP layer, an adversary must initially eavesdrop the sequence number of the last packet received by the target recipient. This target recipient may be either the client or the server. Subsequently, the attacker must forge the close packet. To this end, the attacker creates the close packet with any close reason which the attacker pleases, and then insert the next recipient-expected sequence number into the packet using the eavesdropped sequence number. Further,

after encapsulating the close packet in a UDP and IP datagram packet, the attacker must spoof the recipient-expected sender IP source address of the IP datagram packet. This adversary may have obtained this source address beforehand or may have extracted it from the eavesdropped packet. The attacker then sends this packet to the receiver, thus closing the BTP connection. Hence, this packet closes any BRTFP connections on top of this BTP connection. Moreover, the attacker may use the close reason to fool the recipient. For instance, the attacker can set the reason to Disconnect, such that the recipient removes connection-specific resources.

6.7. Closing BRFTP Connections

An attacker may also close an individual BRFTP connection while preserving the BTP connection, although this attack requires extra steps. In addition to eavesdropping the previously described information, an attacker must also eavesdrop the StreamID of the BRFTP connection it wishes to close. The adversary must then execute the previously described steps to forge a BTP data packet, described in Section 4.9.4. Within this data packet, the attacker inserts the forged BRFTP close packet using the eavesdropped StreamID. Further, the attacker may therein again fool the recipient with using a specific close reason. For instance, the attacker may cause the client to assume that a specific file is no longer served by the server using the FileNotFound reason. Moreover, the attacker may specify the ChecksumInvalid close reason, to cause the client to assume that the desired file version is no longer available.

To mitigate these closing connection attacks which an attacker executes using forged close packets, an implementation MUST authenticate the BTP packet's sequence numbers. Using this authentication, the recipient can ensure that the received packet originates from the expected sender. As such, an adversary can no longer forge completely new packets. To ensure that no BRFTP close packet closes a different BRFTP connection, as an adversary may inject a differing StreamID in the close packet, this field must also by authenticated.

6.8. Man in the Middle Attacks

A potential Man in the Middle attacker could manipulate both data and metadata transmitted on the BTP and BRFTP layers, thus interfering with the communication and potentially injecting unwanted or even malicious data. Furthermore, an attacker interfering in the communication on file re-transmission could change specific parameters and influence the process on both the client and the server side. See the following paragraphs for a more detailed explanation of the potential range of MitM attacks.

6.8.1. MitM attacks on Data

A MitM attacker could swap out transmitted data, by interfering in the exchange of FileReq and FileResp packets on the BRFTP layer (see 5.5.3 and 5.5.4), and later using the Data packet to send custom data chunks. The client has no means of verifying that the received file and checksum were not replaced in transmission by a MitM attacker, thus receiving unwanted and potentially malicious data chunks (ie. files) in the process.

6.8.2. MitM attacks on Metadata

Certain metadata fields can also be maliciously forged or changed by a MitM attacker, in order to induce confusion and incorrect functionality on both client and server side. Examples of Metadata fields manipulation could include (but are not limited to):

- Changing the Version field in the BTP header to an unsupported value (e.g. BTPv3): This action would lead to an unexpected connection close on either side of the communication
- Changing the MaxCwndSize field on the Conn packet of the BTP layer to an unjustifiable low value: This action could lead to unwanted behavior of the Congestion Control algorithm
- Changing the FileNameLength field on the MetadataReq packet of the BRFTP layer to a value lower than the actual length of the FileName field: This action could result in transmitting an incomplete file name to the server, and thus the client could receive an incorrect MetadataResp packet

6.8.3. MitM attacks on Transmission Resumption

When a client tries to resume the previous (interrupted) file transfer, using the Resumption flag, a MitM attacker could interfere, changing the following parameters:

- Changing the Checksum field in FileReq packet: This action could lead to the server not recognizing the requested version, and thus marking it as unavailable to the client
- Changing the Checksum field in FileResp packet: This action could lead to the client believing the requested file version is not available anymore
- Changing the FileSize field in FileResp packet: This action could lead to the client falsely estimating the size of the requested file. Thus, the client can end up under/over-estimating the size of the file, in regard to the available local storage

6.8.4. MitM attacks on Transmission Close

When the transmission closes, due to any reason specified in the Close packet (see 5.5.7), a potential MitM attacker could interfere and forge the Reason field, thus creating confusion on the client side (eg. ChecksumInvalid: client ends up believing file version no longer exists, FileNotFound: client ends up believing file no longer exists, etc.)

6.8.5. Mitigation of MitM Attacks

These attacks and other similar ones could be prevented by authenticating the sequence numbers and StreamID fields before the connection is established, as previously explained in the 6.7 paragraph. This technique would ensure the authenticity of both the client and the server, and thus ensure the data integrity of the information exchanged between these peers.

6.9. Postface

Since the initial concept of the BTP layer was to provide a reliable "baseline" communication layer for any application layer sitting on top, it offers a convenient point for the mitigation of a multitude of attacks. While only protecting needed fields of the BRFTP layer might yield a better performance, it would also be more complex (as can be seen by the length of this document) and as such prone to errors. Therefore, the BTP layer is arguably the most reasonable point to introduce protection against replay, amplification and other attacks. For this task, integrity and authenticity of the sequence number of all BTP packets as well as the payload of the BTP Data packet should be enforced. Additionally, the client and server should reject any packets that have a sequence number that is not contained within the currently expected range.

References

[1] User Datagram Protocol. RFC 768, August 1980.

[2] Mohamed A. Alrshah, Mohamed A. Al-Moqri, and Mohamed Othman. Elastic-tcp: Flexible congestion control algorithm to adapt for high-bdp networks. CoRR, abs/1904.13105, 2019.

[3] M. Anagnostou and E. Protonotarios. Performance analysis of the selective repeat arq protocol. IEEE Transactions on Communications, 34(2):127--135, 1986.

[4] Scott O. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

[5] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. ACM Transactions on Computer Systems, 9, 02 2001.

[6] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.

[7] Subir Varma. Chapter 2 - analytic modeling of congestion control. In Subir Varma, editor, Internet Congestion Control, pages 27--63. Morgan Kaufmann, Boston, 2015.