Workgroup: TUM Protocol Design Meta Group 23
Internet-Draft: draft-group23-rft-00
Published: 13 November 2024
Intended Status: Informational
Expires: 17 May 2025
Authors: S. A. Gierens, Ed.
Technical University of Munich
N. Stangl, Ed.
Technical University of Munich
J. Pfannschmidt, Ed.
Technical University of Munich
D. Rentz, Ed.
Technical University of Munich
Y. E. Nacar, Ed.
Technical University of Munich
I. Gustafsson, Ed.
Technical University of Munich

Robust File Transfer

Abstract

Robust File Transfer (RFT) is a file-transfer protocol on top of UDP. It is connection-oriented, stream-parallel and stateful, supporting connection migration based on connection IDs similar to QUIC. RFT provides point-to-point operation between a client and a server, enabling IP address migration, flow control, congestion control, and partial or resumed file transfers using offsets and lengths.

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 17 May 2025.

Copyright Notice

Table of Contents

1. Introduction

The Protocol Design WG is tasked with standardizing an Application Protocol for a robust file transfer protocol, RFT. This protocol is intended to provide point-to-point operation between a client and a server built upon UDP [RFC0768]. It supports connection migration based on connection IDs, in spirit similar to QUIC [RFC9000], although a bit easier.

RFT is based on UDP, connection-oriented, stateful and uses streams for each file transfer allowing for parallelization. A point-to-point connection supports IP address migration, flow control, congestion control and allows to transfers of a specific length and offset, which can be useful to resume interrupted transfers or partial transfers. The protocol guarantees in-order delivery for all packets belonging to a stream. There is no such guarantee for messages belonging to different streams.

RFT *messages* always consist of a single *Packet Header* and zero or multiple *Frames* appended continuously on the wire after the packet header without padding. Frames are either *data frames*, *error frames* or various types of control frames used for the connection initialization and negotiation, flow control, congestion control, acknowledgement or handling of commands.

1.1. Keywords

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Terms

The following terms are used throughout this document:

Client:
: The endpoint of a connection that initiated it and issues commands over it.

Server:
: The endpoint of a connection that listens for and accepts connections from clients and answers their commands.

Connection:
: A communication channel between a client and server identified by a single connection ID unique on both ends.

Packet:
: An RFT datagram send as UDP SDU over a connection containing zero or multiple frames.

Frame:
: A typed and sized information unit making up (possible with others) the payload of an RFT packet and usually belonging to a particular stream.

Empty Packet:
: A packet without frames.

Command:
: A typed request initiated by the client to the server, e.g. to initiate a file transfer, usually opening up a new stream.

Stream:
: A logical channel within a connection that carries frames encapsulating a particular request i.e. file transfer.

Sender:
: The endpoint sending a packet or frame.

Receiver:
: The endpoint receiving a packet or frame.

1.3. Notation

This document defines `U4`, `U8`, `U16`, `U32`, `U64` as unsigned 4-, 8-, 16-, 32-, or 64-bit integers. A `string` is a UTF-8 [RFC3629] encoded zero-terminated string.

Messages are represented in a C struct-like notation. They may be annotated by C-style comments. All members are laid out continuously on wire, any padding will be made explicit. Constant values are assigned with a "=".

```
StructName1 (Length) {
    TypeName1     FieldName1,
    TypeName2     FieldName2 = 0x123,
    TypeName3[4]  FieldName3,
    String        FieldName4,
    StructName2   FieldName5,
}
```

Figure 1: Message format notation

The only scalar types are integer denoted with "U" for unsigned and "I" for signed integers. Strings are a composite type consisting of the size as "U16" followed by ASCII-characters. Padding is made explicit via the field name "Padding" and constant values are assigned with a "=".

To exemplify a actual message example we use the following notation omitting type names and again making use of "=":

```
StructName1 {
    FieldName1  = 0,
    FieldName2  = 0x123,
    FieldName3  = [1, 2, 3, 4],
    FieldName4  = "Hello",
    FieldName5  = { ... },
    Padding     = 0,
}
```

Figure 2: Filled message format notation

To visualize protocol runs we use the following sequence diagram notation:

```
Client                                                          Server
   |                                                              |
   |------[CID:1337, PID: 2][ACK, PID:3][FLOW, SIZE:1000]----->|
   |                                                              |
   v                                                              v
```

Figure 3: Sequence diagram notation

The individual parts of the packets are enclosed by brackets and only the relevant values are shown. First we always have the RFT packet header, followed by zero or multiple frames. See below for more details on the packet structure.

We use the following abbreviations mostly in diagrams:

Abbreviation	Meaning
VERS	Version

Abbreviation	Meaning
CID	Connection ID
PID	Packet ID
CRC	Packet checksum
SID	Stream ID
CMD	Command frame
DATA	Data frame
ERR	Error frame
ANSW	Answer frame
ACK	Acknowledgement frame
FLOW	Flow control frame
CHCID	Connection ID change frame
EXIT	Exit frame
READ	Read command
WRITE	Write command
CHK	Checksum command
LIST	List command
STAT	Stat command
LEN	Length
OFF	Offset
WIN	Flow window size
OLD	Old connection ID
NEW	New connection ID
MSG	Message

Table 1: Common abbreviations.

2. Overview

This section gives a rough overview over the protocol and provides basic information necessary to follow the detailed description in the following sections.

The RFT protocol is a simple layer 7 protocol for Robust File Transfer. It sits on-top of layer 4 with a single RFT packet send as a UDP SDU. The packet structure is shown in the following figure:

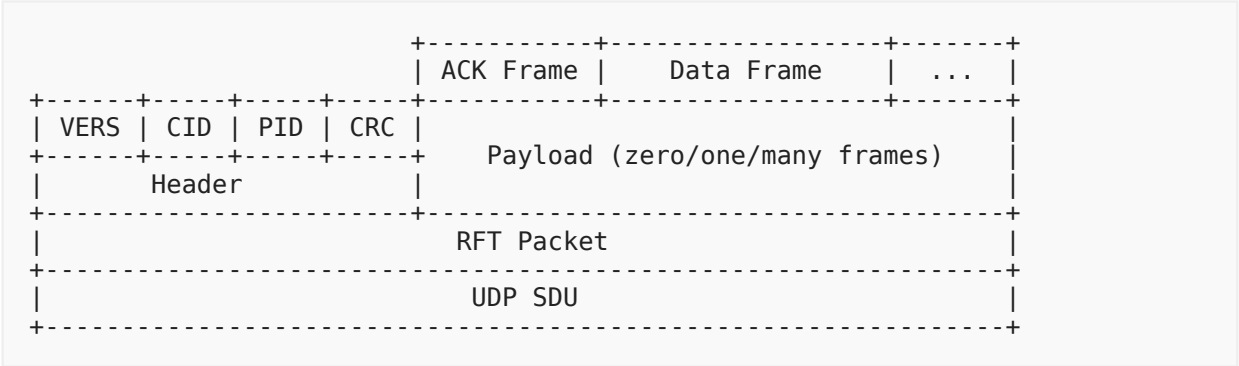


Figure 4: General packet structure

The header contains a version field (VER) for evolvability, as connection ID (CID) uniquely identifying the connection on both ends, a packet ID (PID) identifying the packet within the connection and a cyclic-redundancy-check (CRC) checksum to validate the packet integrity.

After the header follows the payload which holds one or more RFT frames inspired by [RFC9000]. These serve both for data transfer as well as any additional logic besides version matching, connection identification, and packet integrity validation. The most important types are AckFrames for acknowledging packets based on their packet ID (PID), command frames to issue commands on the server, and DataFrames to transport data for the commands to read or write a file. File data in the ReadFrame and WriteFrame as well as in DataFrames is indexed by byte offset and length making both transfer recovery and parallel transfers even of different parts of the same file possible. Each transfer is encapsulated in a stream identified by a stream ID (SID) allowing for multiplexing multiple transfers over a single connection.

The next sections provides detailed information about connection-related topics, e.g. establishment, streams, reliability, congestion control and more. The sections after that explain the message format and framing in more detail, and lists all the different frame and command types.

3. Packet

The RFT packet is the basic transport unit in the protocol. A single packet takes up the entire UDP payload and is composed of a header and a its own payload. The packet header is structured as follows:

```
PacketHeader (64) {
  U8   Version = 1,
  U32  ConnectionId,
  U32  PacketId,
  U24  PacketChecksum,
}
```

Figure 5: Packet header wire format

3.1. Version

To ensure evolvability the packet header contains a 8-bit version field. Most network protocols never hit a two-digit version number, therefore 8 bit is deemed sufficient.

The version field identifies the protocol version used by the sender of the packet. Upon connection establishment server **MUST** validate that the clients version is compatible with its own before responding to a handshake request. A peer **SHALL NOT** change the protocol version during the lifetime of the connection, and peers **MAY** re-validate the version at any time.

As long as RFT is in draft stage with rapid breaking changes the peers **SHOULD** strictly match the version number.

3.2. Connection ID

The 32-bit connection ID uniquely identifies the connection on both ends. 32 bit allows for up to roughly 4 billion connections per UDP port. While this is not as extensive as QUIC [RFC9000], it is deemed sufficient for a file transfer protocol. Deployments that require more client connections on a single server can obviously run multiple protocol instances on different server ports.

The connection ID is negotiated during connection establishment, which is discussed in more detail in Section 5.1. The connection ID furthermore allows for connection migration, which is discussed in Section 5.3.

3.3. Packet ID

The 32-bit packet ID uniquely identifies a packet in one direction of the connection. Each peer maintains a counter starting at 1 that is incremented for each packet they send to the other side. The uniqueness only holds up to wrap-around, and implementations are **REQUIRED** to handle this properly. With a 32-bit ID and up to 1500 Byte packets, this means roughly 6 TByte of data can be transferred before a wrap-around occurs. Even with a state-of-the-art 1 Tbit/s link, this would take 48 seconds which is deemed sufficient in comparison to timeouts and other protocol mechanisms.

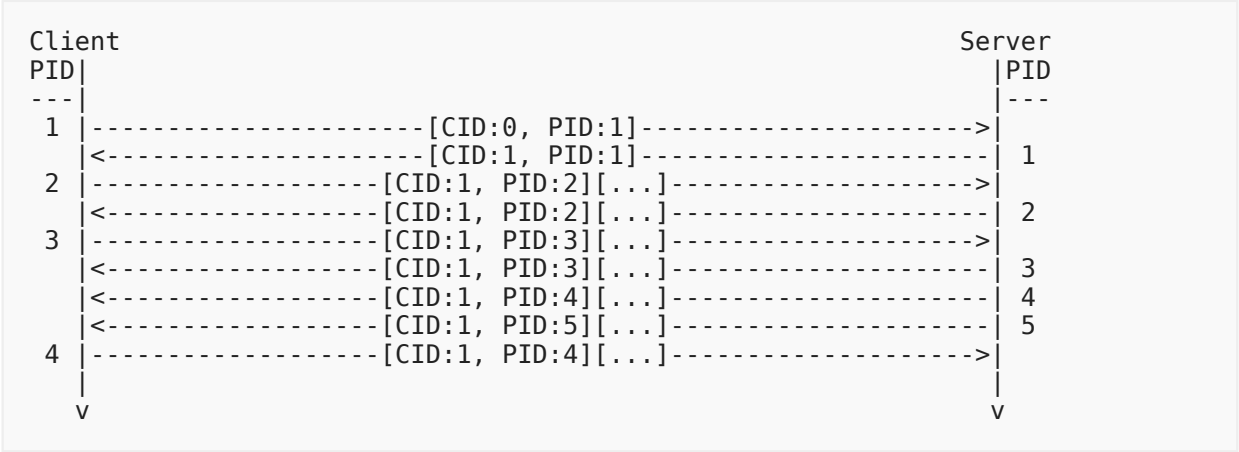


Figure 6: Sequence diagram of packet ID incrementation

3.4. Packet Checksum

The packet checksum is a redundancy check to validate the integrity of packet. It contains the first 24-bit of the 32-bit cyclic redundancy check (CRC32) [RFC3385] of the entire packet, with the packet checksum itself set to 0.

The length of the checksum is chosen for alignment reasons. Since CRC32 has a good entropy, "chopping off" 8 bit should not impede its effectiveness, and also in general make it a suitable choice.

3.5. Payload

The payload has a variable size but **SHOULD** be chosen such, that it does not produce IP packet fragmentation. So in a typical 1500 Byte MTU network with a minimal 20 Byte IP and 8 Byte UDP header, followed by the 64 Byte RFT header, up to 1408 Bytes can be used for the payload.

The payload consists of zero, one, or multiple frames, that build a second level of packetization in the protocol. The come in different flavors allowing for flexible state exchange, and also provide the means for multistreaming, both discussed in detail in the following section.

4. Communication Structure

TCP's rigid header structure has made it difficult to extend the protocol to more modern requirements. RFT as a file transfer protocol may be more specialized but still requires flexibility to fit different scenarios. For the transfer of a single large file not much is needed, but synchronizing directories consisting of many small files calls for parallelization which can introduce new challenges like head-of-line blocking or state complexity.

In general, the ideal way to handle the transfers depend on the specific nature of the application. Therefore RFT tries to provide a flexible framework for issuing file transfers, related actions and handling their responses. The application layer can then decide how to use these mechanisms and how much parallelization and complexity it requires.

RFT achieves said flexibility by adopting two structural ideas from QUIC [RFC9000], frames and streams. How these are used in RFT is discussed in the following subsections.

4.1. Frames

Frames subdivide packets into the atomic units of state exchange of the protocol. All frames have in common that they start with an 8 bit type ID. The ExitFrame for closing the connection is the only frame, that consists of nothing but the type ID 1:

```
ExitFrame (8) {
  U8  TypeId = 1,
}
```

Figure 7: Exit frame wire format

For frames with only header fields the length is implicitly given. Frames with variably sized path, message, or payload fields have a length field right in front of them. File data for example is carried in DataFrames:

```
DataFrame (72 + len(Payload)) {
  U8     TypeId = 6,
  U16    StreamId,
  U48    Offset,
  Bytes  Payload,
}
```

Figure 8: Data frame wire format

The order of frames in a packet is only relevant within the same stream, the concept discussed in the following.

4.2. Streams

Streams are logical channels within the connection encapsulating the frames belonging to a particular operation. This enables the parallel execution of multiple file transfers for example. Such frames carry a 16-bit stream ID (SID) to identify the stream they belong to. At any time there CANNOT be more than one stream with the same ID within the connection. Stream IDs **MAY** however be reused after completion of a previous stream with that ID.

A stream is created by the client when sending a command frame to the server. The stream ID is not negotiated but chosen by the client and send to the server in the command frame, for example in the ReadFrame:

```
ReadFrame (160 + len(Path)) {
  U8      TypeId = 7,
  U16     StreamId,
  U7      Reserved = 0,
  Bool    ValidateChecksum,
  U48     Offset,
  U48     Length,
  U32     Checksum,
  String  Path,
}
```

Figure 9: Read frame wire format

The client is therefore solely responsible for ensuring the time-local uniqueness of the stream ID. The server **MUST** answer commands with the same stream ID and in the context of the same stream.

Streams are closed when either the requested operation is completed, via an AnswerFrame or an empty DataFrame which signals EOF for transfers, or with an ErrorFrame in conflict cases.

Frames for connection global control traffic like AckFrames or FlowFrames do not carry a stream ID and are implicitly associated to the virtual stream 0. This stream ID **MUST NOT** be used for any other purpose.

The following sequence diagram shows the parallel execution of a read and stat command using two different streams (SID 1 and 2) on the same connection, ignoring acknowledgements:

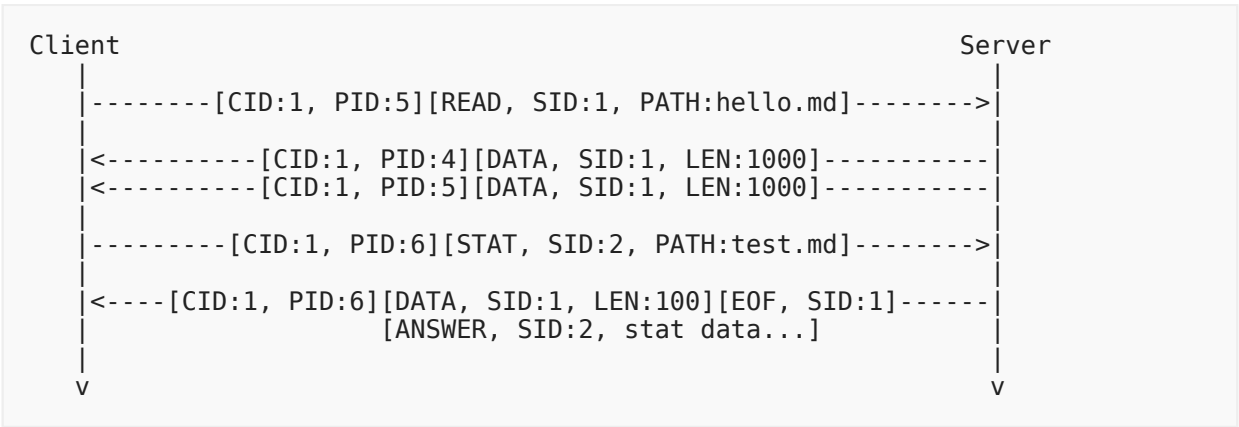


Figure 10: Simplified sequence diagram of read and stat commands executed in parallel, ignoring acknowledgements. Note the different stream IDs.

Both streams are terminated in the same packet, one with the EOF (empty DataFrame) and the other with an AnswerFrame.

In contrast the following example shows how the server returns an error in case of a duplicate stream ID:

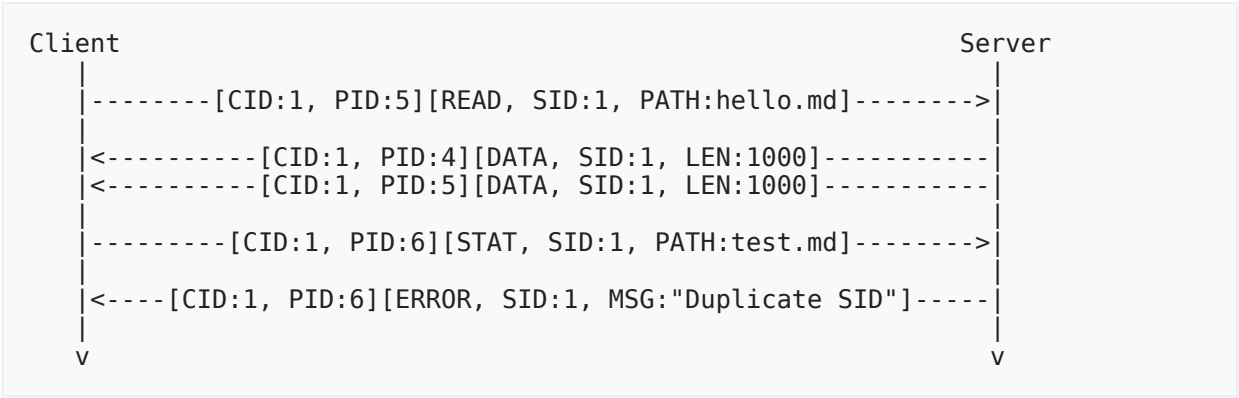


Figure 11: Simplified sequence diagram of an error due to duplicate stream ID

5. Connection

The protocol is connection-based. Connections are identified a singular connection ID (CID) unique on both sides.

5.1. Establishment

The connection establishment is and via a two-way handshake and is initiated by the client by sending a packet with connection ID 0. The server responds with the UDP packet having reversed IP addresses and ports, containing an RFT packet with the connection ID chosen by the server. The server knows all IDs of established connections and must make the new one is unique.

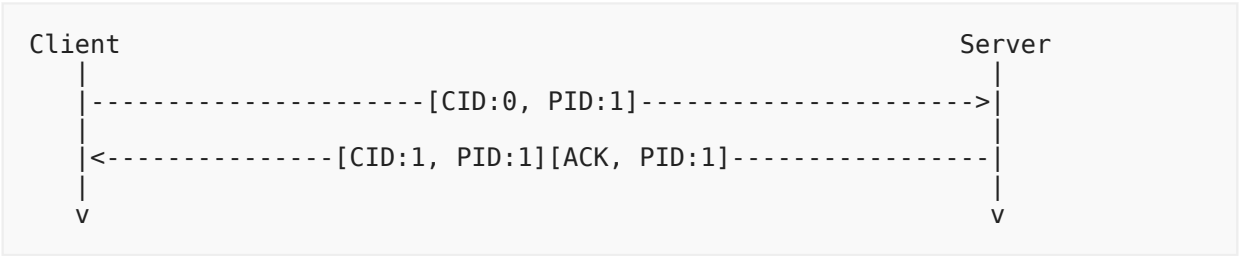


Figure 12: Sequence diagram of simple connection establishment

5.1.1. Connection ID Negotiation

This simple connection establishment is limited to a single handshake at a time per UDP source port. If the client wishes to establish multiple over a single port it can attach a ConnectionIdChangeFrame with a proposed connection ID for the new one (NEW) and 0 for the old one (OLD). The server acknowledges this and sends back the handshake response to that connection ID:

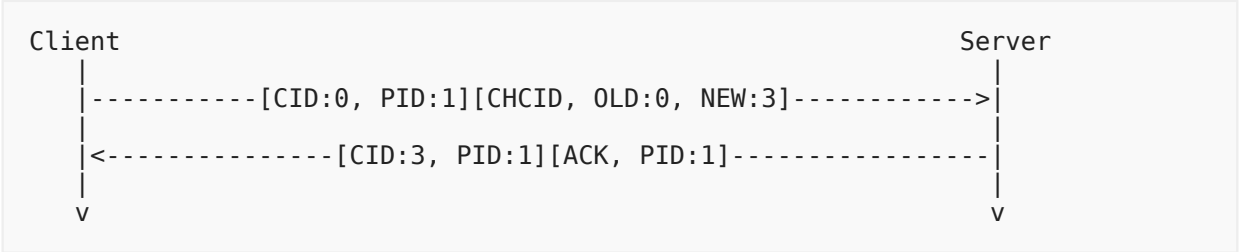


Figure 13: Sequence diagram of successful connection ID proposal

In case the proposal is already used for another connection attaches another ConnectionIdChangeFrame (CHCID) with the new unique connection ID chosen by the server.

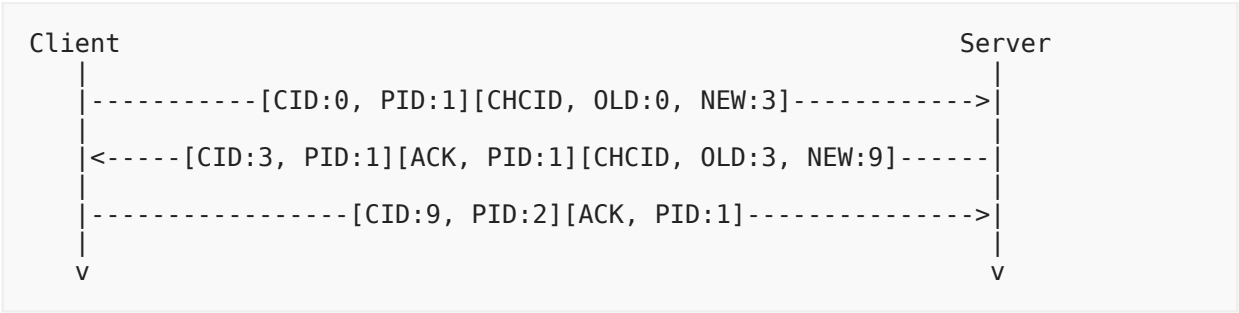


Figure 14: Sequence diagram of unsuccessful connection ID proposal

5.1.2. Unknown Connection ID

When a peer receives a packet for an unknown connection ID it **SHOULD** simply ignore it.

5.2. Termination

A connection can either be intentionally closed or timeout.

5.2.1. Tear Down

If a peer wishes to close the connection it simply sends a Exit frame.

Figure 15: Graceful connection tear-down

5.2.2. Timeout

If no packets were received for 5 minutes the connection is considered dead and the server **SHOULD** close it. Peers **MAY** send empty packets (i.e. packets without frames) to keep the connection alive beyond timeouts.

5.3. Migration

A connection is uniquely identified on both ends by the connection ID. As soon as a peer receives a packet for this connection ID from a different IP-address port pair, it must change its internal mapping and send all subsequent packets to the new address. Any packets lost in the meantime are subjects to retransmission. If a peer has nothing to send, but wishes to explicitly inform the other end of a migration, the peer **MAY** simply send an empty packet (thus a packet without frames).

5.4. Resumption

RFT does not explicitly support connection recovery, but allows for resuming file transfers by the means of partial reads and writes via the corresponding offset and length fields in the Read- and WriteFrames.

6. Robustness

The protocol has multiple mechanisms to ensure transmissions are complete, in-order and integrity is maintained, while not overwhelming the receiver or the network. It takes inspiration from both QUIC [RFC9000] and TCP [RFC0768].

6.1. In-Order Delivery

The Packet ID (Section 3.3) is a monotonically increasing counter for the packets send by a peer. It thus allows the receiving side to determine the order in which packets were sent out. Streams do not have a separate ordering mechanism, so implementations are **REQUIRED** to process frames only after all previous packets and thus frames have been completely handled. An implementation **SHOULD** be able to buffer packets for a short time within limits of a timeout to counteract reordering that might have occurred in the network before requesting retransmissions.

6.2. Acknowledgements

To ensure completeness the receiver acknowledges packets via AckFrames:

```
AckFrame (40) {
  U8   TypeId = 0,
  U32  PacketId,
}
```

Figure 16: Acknowledgement frame wire format

It contains the last consecutively received packet ID and thus acknowledges all packets up to that point cumulatively:

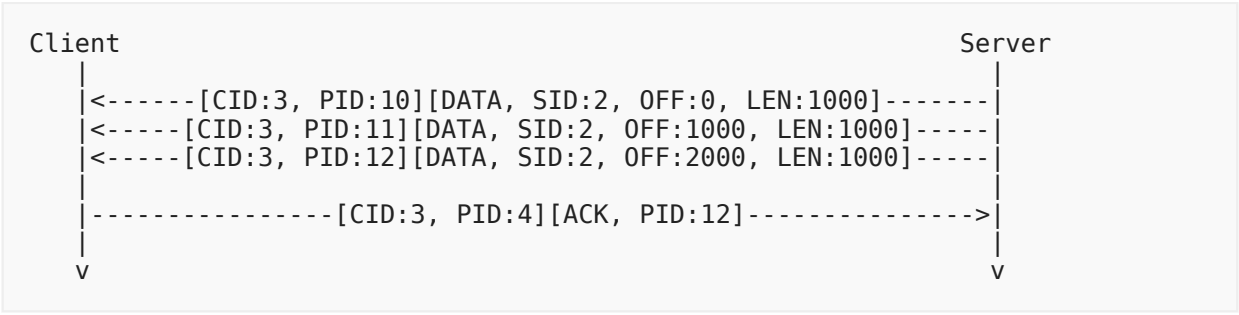


Figure 17: Example sequence diagram of cumulative acknowledgement

Empty packets or those with only an AckFrame do NOT NEED to be acknowledged to prevent an acknowledgment loop.

6.3. Retransmission

There are two ways retransmissions are triggered.

6.3.1. Retransmission Timeout

If sender does not receive an AckFrame for a packet or a later one within the retransmission timeout (RTO) of 1 second a normal retransmission (RT) is triggered:

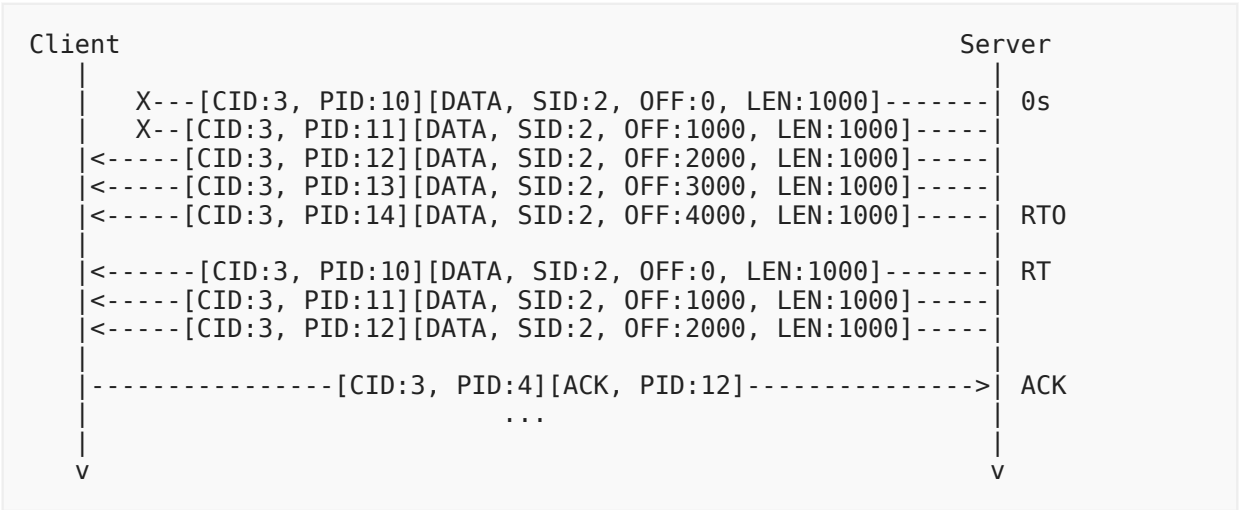


Figure 18: Example sequence diagram of retransmission (RT)

6.3.2. Fast Retransmission

The receiver can also request a fast retransmission (FRT) by sending a duplicate AckFrame (DACK) for the last packet it received:

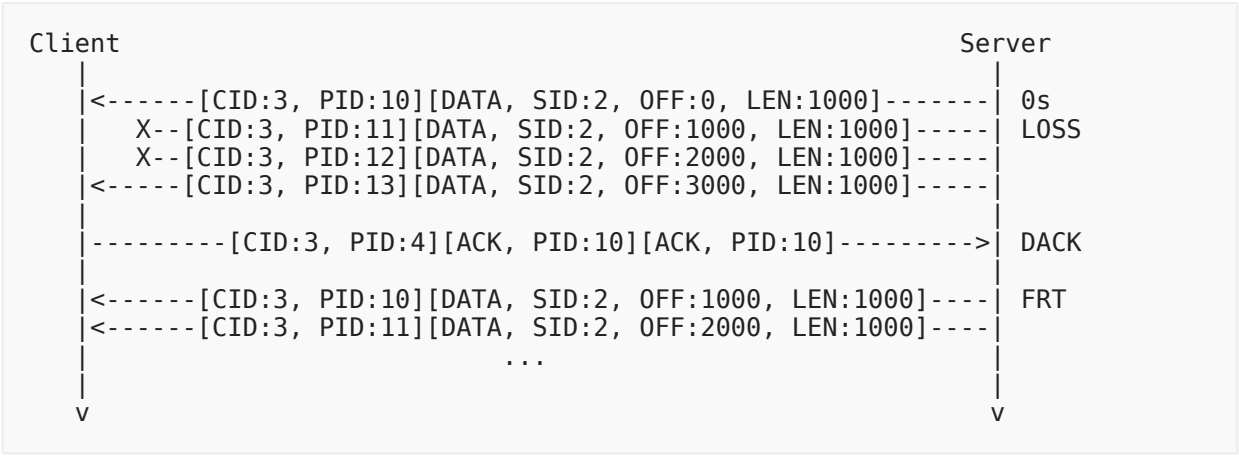


Figure 19: Example sequence diagram of fast retransmission (FRT)

6.4. Congestion Control

Congestion control is inspired by TCP's AIMD [RFC0768]. A congestion window (CWND) limits the amount of bytes in-flight. The window scaling goes through two phases:

6.4.1. Slow Start

The congestion window starts at 1 and is updated for each packet containing an AckFrame as "CWND_NEW = min(2 * CWND, FWND)" with FWND being the window indicated by flow control, and ends once the slow start threshold is reached.

6.4.2. Congestion Avoidance

After the slow start the AIMD (additive increase, multiplicative decrease) algorithm is used. The congestion window is increased by one for each acknowledged packet. In case a retransmission is necessary congestion is assumed and the congestion window is halved and avoidance continues from there. A timeout causes a reset of the congestion window to one and continues with a slow start where the threshold set to half the number of packets in-flight.

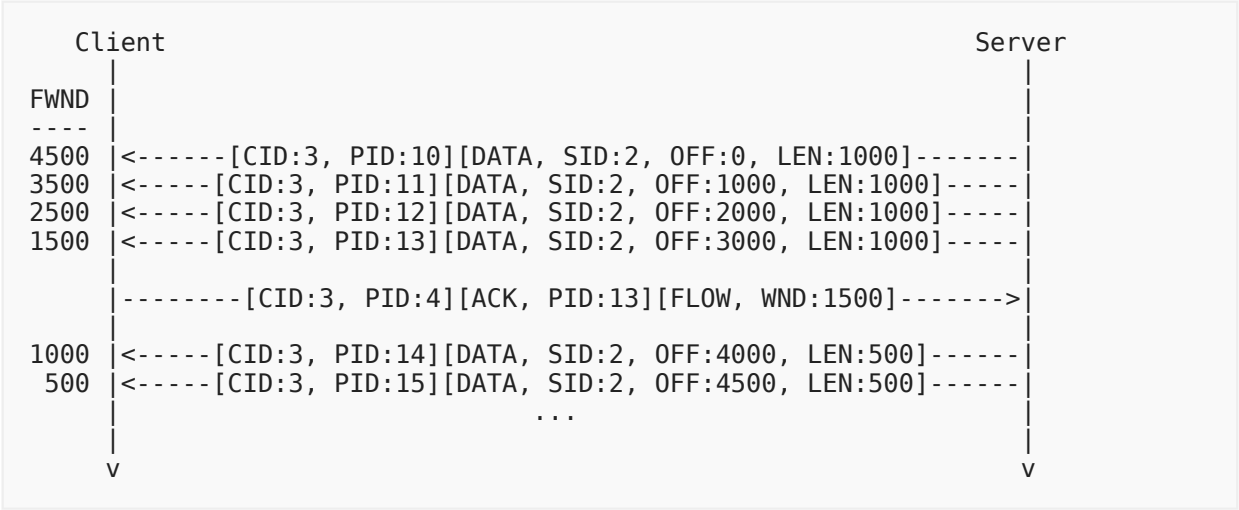
6.5. Flow Control

To avoid overwhelming the receiver it indicates its available receive buffer size for flow window (FWND) via FlowControl frames:

```
FlowControl (40) {
  U8   TypeId = 3,
  U32  WindowSize,
}
```

Figure 20: Flow control frame wire format

The window size is the number of bytes left in the receive buffer. When a sender receives this frame it **MUST NOT** exceed the indicated limit. The following example shows how the peer **SHOULD** react:



6.6. Checksumming

For the integrity validation of the transmission the partial CRC32 checksum in the packet header is used. Before processing a packet any further the receiver **MUST** verify that the checksum calculated over the remaining packet matches the one in the header. If it does not the packet **MUST** be discarded as not even the connection or packet IDs that are required to issue a fast retransmission can be trusted. The receiver has to wait for a timeout to trigger retransmission on the sender side.

7. File Transfer

File transfers in either direction are initiated by the client via the respective command (read or write) frames send on a new stream. Following that data and acknowledgement frames are exchanged until the transfer is complete, indicated by an empty end-of-file (EOF) data frame.

Both read and write frames indicate the path of the file together with the offset and length to be transferred. An offset of 0 indicates starting at the beginning of the file, a length of 0 indicates transferring everything from the offset up to the end of the file.

7.1. Read

To read a file from the server the client sends a ReadFrame:

```
ReadFrame (160 + len(Path)) {
  U8      TypeId = 7,
  U16     StreamId,
  U7      Reserved = 0,
  Bool    ValidateChecksum,
  U48     Offset,
  U48     Length,
  U32     Checksum,
  String  Path,
}
```

Figure 21: Read frame wire format

In case of reading an entire file the frame could look like this:

```
ReadFrame {
  TypeId            = 7,
  StreamId          = 1,
  Reserved          = 0,
  ValidateChecksum  = false,
  Offset            = 0,
  Length            = 0,
  Checksum          = 0,
  Path              = "./example/README.md",
}
```

Figure 22: Example read frame

The following sequence diagram puts such frame into context (Note that we omit fields that are not relevant for the example):

```
Client                                                    Server
|                                                           |
|----[CID:3, PID:4][READ, SID:5, OFF:0, LEN:0, PATH:readme.md]--->|
|                                                           |
|<---[CID:3, PID:10][ACK, PID:4][DATA, SID:5, OFF:0, LEN:1000]----|
|<--------[CID:3, PID:11][DATA, SID:5, OFF:1000, LEN:1000]--------|
|<--------[CID:3, PID:12][DATA, SID:5, OFF:2000, LEN:1000]--------|
|                                                           |
|-------------------[CID:3, PID:5][ACK, PID:12]------------------>|
|                                                           |
|<--------[CID:3, PID:13][DATA, SID:5, OFF:3000, LEN:1000]--------|
|<--------[CID:3, PID:14][DATA, SID:5, OFF:4000, LEN:177]---------|
|                         [DATA, SID:5, OFF:4178, LEN:0]          |
|                                                           |
|------------------[CID:3, PID:6][ACK, PID:14]------------------>|
|                                                           |
v                                                           v
```

Figure 23: Sequence diagram for an example file read

Aside from the already discussed fields the ReadFrame also contains a boolean ValidateChecksum flag together with the optional Checksum field. These allow the client to ensure that the already read portion is still the same before continuing to read it. When the client sets the ValidateChecksum flag to true, it **MUST** provide the CRC32 checksum of the already read portion and set the offset to the byte after that part. The server **MUST** then validate that the checksum matches for the file on its end and continue reading if it does. If the checksum does not match the server **MUST** back an ErrorFrame with a message "Checksum mismatch".

7.2. Write

To write a file on the server the client sends a WriteFrame:

```
WriteFrame (120 + len(Path)) {
  U8      TypeId = 8,
  U16     StreamId,
  U48     Offset,
  U48     Length,
  String  Path,
}
```

Figure 24: Example write frame

In case of writing a file without specifying its size ahead of time the frame could look like this:

```
WriteFrame {
  TypeId    = 8,
  StreamId  = 1,
  Offset    = 0,
  Length    = 0,
  Path      = "./example/README.md",
}
```

Figure 25: Example write frame

The following sequence diagram puts such frame into context (Note that we omit fields that are not relevant for the example):

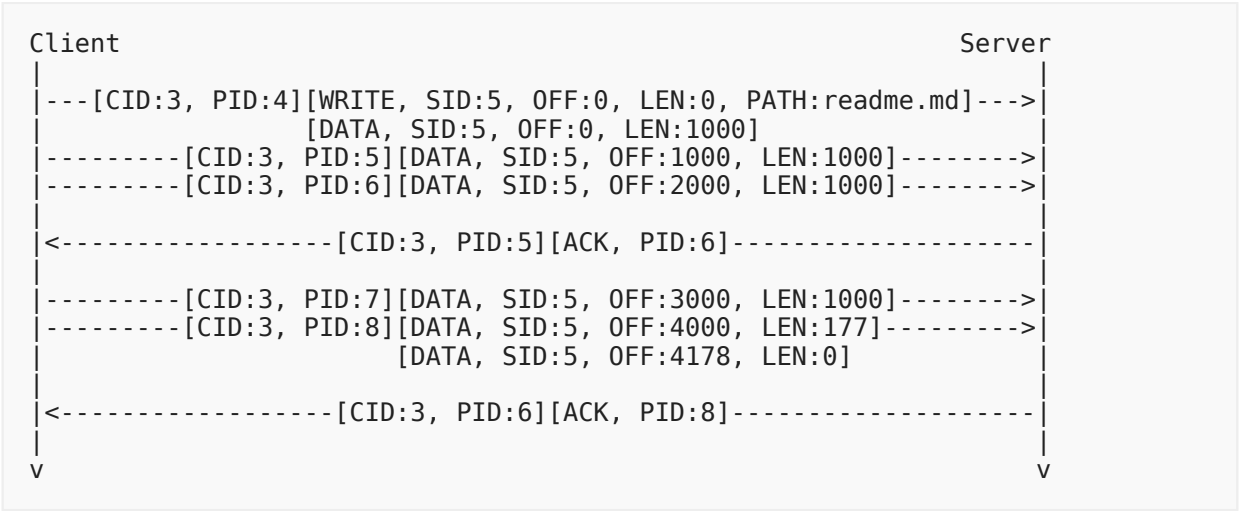


Figure 26: Sequence diagram for an example file write

WriteFrames do not contain a checksum field that requires any special handling.

7.3. Multiple Transfers

Multiple transfers can be executed in parallel by using different streams. The following sequence diagram shows how a client and server exchange two files:

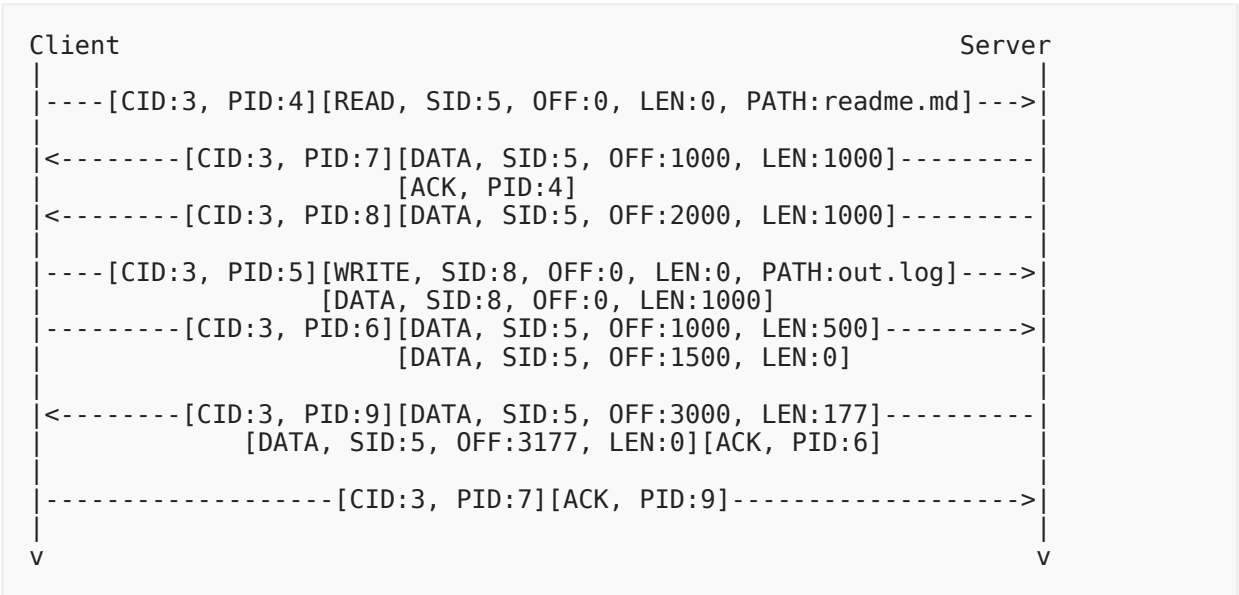


Figure 27: Sequence diagram for a parallel file read and write

Initially, the server has a file called "readme.md" and the client has a file called "out.log". The client reads the file from the server and concurrently also writes the other file to the server, so that in the end both systems have both files.

7.4. Recovery

As mentioned before, RFT does not have any explicit connection recovery mechanism. Offset and length fields on Read- and WriteFrames however allow the client to resume partially completed transfers in a new connection.

The following examples show how such a resumption of both a read and write could be performed. In both cases we assume that the file transferred is called "example.txt" and 4000 bytes long.

7.4.1. Read Recovery

In this case the client has already read the first 2000 bytes of the file before the connection is lost. For the second read command the client makes use of the offset and length fields to resume the transfer, and the optional checksum field to ensure the already read portion is still the same:

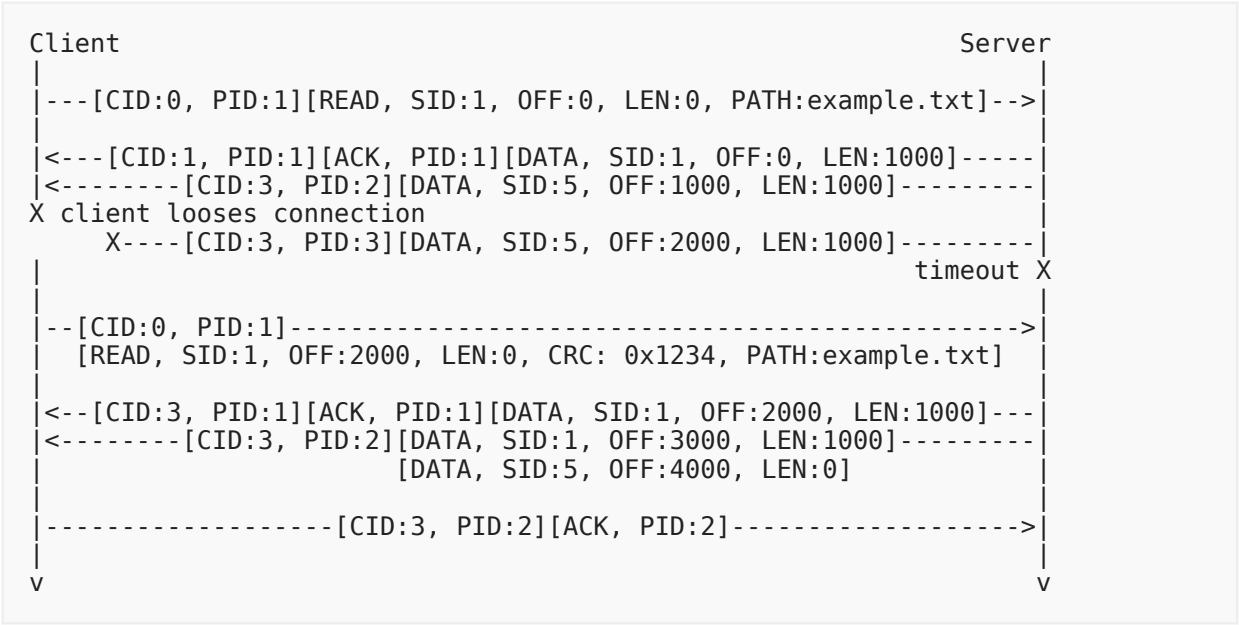


Figure 28: Sequence diagram for an example file read resumption after a client-side connection failure

The server will validate the checksum upon receiving the ReadFrame and continue reading the file from the offset provided if it matches. Otherwise it will send an ErrorFrame with a message "Checksum mismatch".

7.4.2. Write Recovery

In this case the server is assumed to have already received and acknowledged the first 3000 bytes of the file before the connection was lost. The client then simply uses the offset field to resume the transfer from there:

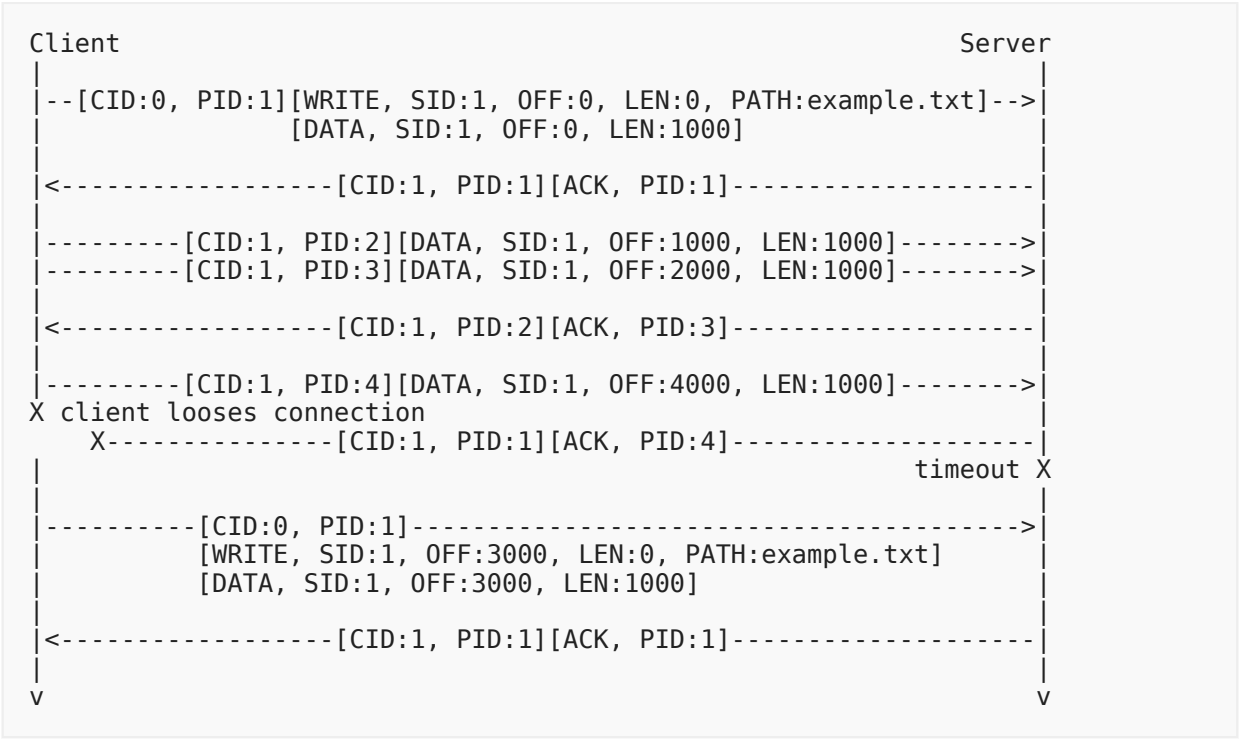


Figure 29: Sequence diagram for an example file write resumption after a client-side connection failure

Note that in contrast to the read case the protocol currently offers no mechanism on the WriteFrame to ensure the file has not changed on the server side nor would a ChecksumFrame completely eliminate this possibility.

8. Further Commands

While RFT is primarily designed for file transfers, many use cases require additional operations, therefore RFT has three additional commands: Checksum, Stat, and List, all discussed in the following.

8.1. Checksum

The ChecksumFrame initiates the computation of the SHA-256 [RFC6234] checksum of a file on the server side:

```
ChecksumFrame (24 + len(Path)) {
  U8      TypeId = 9,
  U16     StreamId,
  String  Path,
}
```

Figure 30: Checksum frame wire format

The server responds with an AnswerFrame containing the SHA-256 checksum of the entire file:

```
AnswerFrame (24 + len(Payload)) {
  U8     TypeId = 4,
  U16    StreamId,
  Bytes  Payload = {
    U8[32] Checksum,
  }
}
```

Figure 31: Answer frame for checksum command wire format

The following sequence diagram shows the process of a checksum computation:

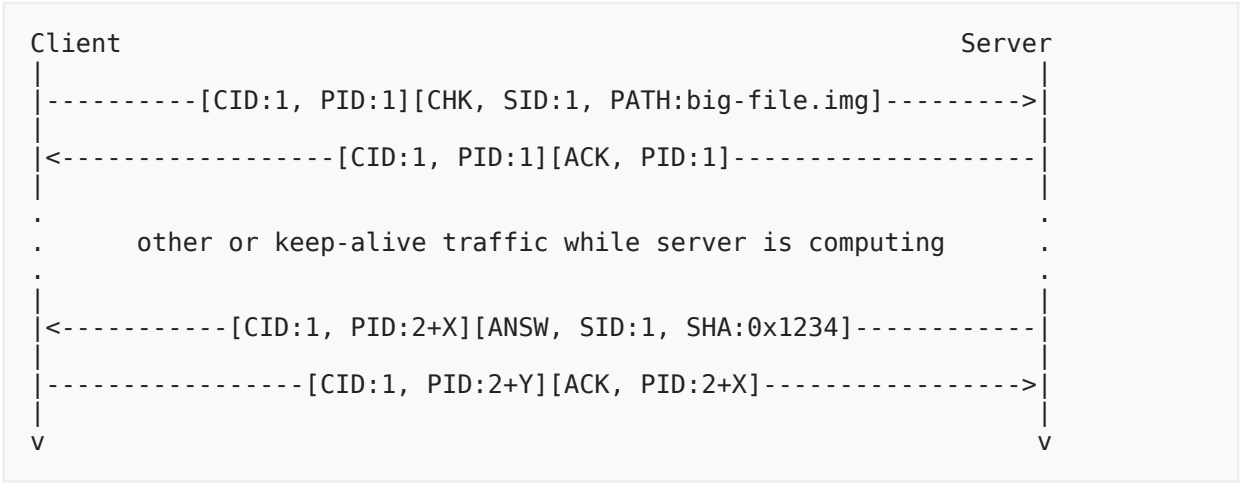


Figure 32: Sequence diagram for an example checksum computation

Note that in case of a larger file the server **MAY** send empty packets to keep the connection alive in absence of other traffic while the checksum is computed.

8.2. Stat

The StatFrame issues a request for file metadata:

```
StatFrame (24 + len(Path)) {
  U8      TypeId = 10,
  U16     StreamId,
  String  Path,
}
```

Figure 33: Stat frame wire format

The server responds with an AnswerFrame holding said metadata:

```
AnswerFrame (24 + len(Payload)) {
  U8     TypeId = 4,
  U16    StreamId,
  Bytes  Payload = {
    U4     FileType,
    U12    Permissions,
    U64    FileSize,
    U64    CreatedAt,
    U64    ModifiedAt,
    U64    AccessedAt,
  }
}
```

Figure 34: Answer frame for checksum command wire format

The file type is encoded in the first byte as follows:

File Type Value	File Type
0	- reserved -
1	Regular file
2	Directory
3	Symbolic link
4	Block device
5	Character device
6	FIFO
7	Socket
8 to 256	- reserved -

Table 2: File type definitions.

The file permissions follow Linux conventions:

Permission Bit	Permission
1	Set User ID
2	Set Group ID
3	Sticky Bit
4	Owner Read
5	Owner Write
6	Owner Execute
7	Group Read
8	Group Write
9	Group Execute
10	Other Read
11	Other Write
12	Other Execute

Table 3: Permission bit definitions.

The file size is the number of bytes in the file, while the timestamps are 64-bit UNIX timestamps.

The following sequence diagram shows the process of a stat operation:

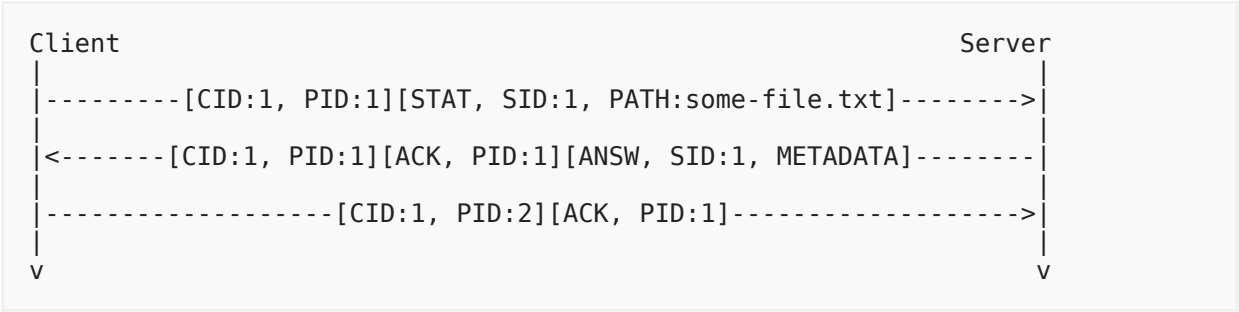


Figure 35: Sequence diagram for an example checksum computation

8.3. List

The ListFrame requests the list of entries in a directory:

```
ListFrame (24 + len(Path)) {
  U8      TypeId = 11,
  U16     StreamId,
  String  Path,
}
```

Figure 36: List frame wire format

Similar to the ReadFrame the server responds with sequence of DataFrames each containing DirectoryEntries:

```
DataFrame (72 + len(Payload)) {
  U8     TypeId = 6,
  U16    StreamId,
  U48    Offset,
  Bytes  Payload = {
    DirectoryEntry[n] Entries,
  }
}
```

Figure 37: Wire format of data frame for list command response

Each line of the payload contains the file type in the first byte, followed by the file name up to the line terminator:

```
DirectoryEntry (72 + len(Payload)) {
  U8      FileType,
  Char[m] Name,
  Char    Separator = '\n',
}
```

Figure 38: Directory entry wire format

The following sequence diagram shows the process of a list operation:

```
Client                                                    Server
|                                                           |
|---------[CID:1, PID:5][LIST, SID:1, PATH:/home/admin/]--------->|
|                                                           |
|<---[CID:3, PID:10][ACK, PID:4][DATA, SID:5, OFF:0, LEN:1000]----|
|<--------[CID:3, PID:11][DATA, SID:5, OFF:1000, LEN:1000]--------|
|<--------[CID:3, PID:12][DATA, SID:5, OFF:2000, LEN:1000]--------|
|                                                           |
|-------------------[CID:3, PID:6][ACK, PID:12]------------------>|
|                                                           |
v                                                           v
```

Figure 39: Sequence diagram for an example checksum computation

9. Wire Format

This section summarizes the wire format of the protocol.

9.1. Numbers

RFT encodes numbers in little endian format to make implementation easier on most platforms.

9.2. Arrays

Array types have a variable size and are prefixed with a 2-byte length field, sufficient for all practical purposes of RFT which is designed to avoid IP fragmentation.

9.2.1. Bytes

The Bytes type is used for arbitrary binary data like file chunks:

```
Bytes (2 + Length) {
  U16         Length,
  U8[Length]  Buffer,
}
```

Figure 40: Bytes wire format

9.2.2. String

The String type is a sequence of UTF-8 encoded characters, used for message fields:

```
String (2 + Length) {
  U16           Length,
  Char[Length]  Buffer,
}
```

Figure 41: String wire format

9.2.3. Path

The Path type is technically equivalent to the String type, but is specifically intended for file paths:

```
Path (2 + Length) {
  U16           Length,
  Char[Length]  Buffer,
}
```

Figure 42: Path wire format

9.3. Packet Format {#packet format}

The packet is the top-level structure of the protocol and consists of a header and an array of frames:

```
Packet (len(Header) + len(Frames)) {
  PacketHeader  Header,
  Array[Frame]  Frames,
}
```

Figure 43: Packet wire format

The header contains the version, connection ID, packet ID, and a partial CRC32 checksum:

```
PacketHeader (96) {
  U8   Version = 1,
  U32  ConnectionId,
  U32  PacketId,
  U24  PacketChecksum,
}
```

Figure 44: Packet header wire format

9.4. Frame Format

Frames come in different types identified by a type ID:

Frame Type Value	Frame Type
0	Acknowledgement Frame
1	Exit Frame
2	Connection ID Change Frame
3	Flow Control Frame
4	Answer Frame
5	Error Frame
6	Data Frame
7	Read Frame
8	Write Frame

Frame Type Value	Frame Type
9	Checksum Frame
10	Stat Frame
11	List Frame

Table 4: Frame type definitions.

9.4.1. Ack Frame

The AckFrame contains the packet ID to be acknowledged cumulatively:

```
AckFrame (40) {
  U8   TypeId = 0,
  U32  PacketId,
}
```

Figure 45: Acknowledgement frame wire format

9.4.2. Exit Frame

The ExitFrame terminates the connection and has no further fields:

```
ExitFrame (8) {
  U8  TypeId = 1,
}
```

Figure 46: Exit frame wire format

9.4.3. Connection ID Change Frame

The ConnectionIdChangeFrame contains the old and new connection ID for negotiation:

```
ConnIdChange (72) {
  U8   TypeId = 2,
  U32  OldConnId,
  U32  NewConnId,
}
```

Figure 47: Connection ID Change frame wire format

9.4.4. Flow Control Frame

The FlowControl frame contains the new flow window size:

```
FlowControl (40) {
  U8   TypeId = 3,
  U32  WindowSize,
}
```

Figure 48: Flow control frame wire format

9.4.5. Answer Frame

The AnswerFrame responds to the command on the same stream with Bytes payload of command-specific structure, which is described in more detail above (Section 8).

```
AnswerFrame (24 + len(Payload)) {
  U8     TypeId = 4,
  U16    StreamId,
  Bytes  Payload,
}
```

Figure 49: Answer frame wire format

9.4.6. Error Frame

The ErrorFrame returns an error message on the same stream:

```
ErrorFrame (24 + len(Message)) {
  U8      TypeId = 5,
  U16     StreamId,
  String  Message,
}
```

Figure 50: Error frame wire format

9.4.7. Data Frame

The DataFrame carries a chunk of the transferred file at the given offset.

```
DataFrame (72 + len(Payload)) {
  U8     TypeId = 6,
  U16    StreamId,
  U48    Offset,
  Bytes  Payload,
}
```

Figure 51: Data frame wire format

Note that they are also used to transmit the response to a list command, see above (Section 8.3).

9.4.8. Read Frame

The ReadFrame initiates a file read operation and is described in detail above (Section 7.1).

```
ReadFrame (160 + len(Path)) {
  U8      TypeId = 7,
  U16     StreamId,
  U7      Reserved = 0,
  Bool    ValidateChecksum,
  U48     Offset,
  U48     Length,
  U32     Checksum,
  String  Path,
}
```

Figure 52: Read frame wire format

9.4.9. Write Frame

The WriteFrame initiates a file write operation and is described in detail above (Section 7.2).

```
WriteFrame (120 + len(Path)) {
  U8      TypeId = 8,
  U16     StreamId,
  U48     Offset,
  U48     Length,
  String  Path,
}
```

Figure 53: Write frame wire format

9.4.10. Checksum Frame

The ChecksumFrame initiates a checksum computation for the given file, and is described in detail above (Section 8).

```
ChecksumFrame (24 + len(Path)) {
  U8      TypeId = 9,
  U16     StreamId,
  String  Path,
}
```

Figure 54: Checksum frame wire format

9.4.11. Stat Frame

The StatFrame initiates a stat operation for the given file, and is described in detail above (Section 8).

```
StatFrame (24 + len(Path)) {
  U8      TypeId = 10,
  U16     StreamId,
  String  Path,
}
```

Figure 55: Stat frame wire format

9.4.12. List Frame

The ListFrame initiates a list operation for the given directory, and is described in detail above (Section 8).

```
ListFrame (24 + len(Path)) {
  U8      TypeId = 11,
  U16     StreamId,
  String  Path,
}
```

Figure 56: List frame wire format

10. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/rfc/rfc768>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/rfc/rfc3629>.

[RFC3385] Sheinwald, D., Satran, J., Thaler, P., and V. Cavanna, "Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations", RFC 3385, DOI 10.17487/RFC3385, September 2002, <https://www.rfc-editor.org/rfc/rfc3385>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/rfc/rfc6234>.

[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/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

Authors' Addresses

Sandro-Alessio Gierens (EDITOR)
Technical University of Munich
Boltzmannstraße 3
85748 Garching
Germany
Email: sandro.gierens@tum.de

Niklas Stangl (EDITOR)
Technical University of Munich
Boltzmannstraße 3
85748 Garching
Germany
Email: niklas.stangl@tum.de

Johannes Pfannschmidt (EDITOR)
Technical University of Munich
Boltzmannstraße 3
85748 Garching
Germany
Email: johannes.pfannschmidt@cs.tum.edu

Désirée Rentz (EDITOR)
Technical University of Munich
Boltzmannstraße 3
85748 Garching
Germany
Email: desiree.rentz@tum.de

Yusuf Erdem Nacar (EDITOR)
Technical University of Munich
Boltzmannstraße 3
85748 Garching
Germany
Email: yusuferdem.nacar@tum.de

Isak Gustafsson (EDITOR)
Technical University of Munich
Boltzmannstraße 3
85748 Garching
Germany
Email: go68wuy@mytum.de