Workgroup:	Group I & Group VII
Internet-Draft:	draft-group18-rft-02
Published:	22 June 2022
Intended Status:	Informational
Expires:	24 December 2022
Authors:	A. Maslew	F. Schoenberger	J. Kusnierz	E. Berretta	A. Swierkowska
---	---	---	---	---	---

Robust File Transfer

Status of This Memo

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

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

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

This Internet-Draft will expire on 24 December 2022.

Copyright Notice

Table of Contents

1. Introduction

Robust File Transfer (RFT) is a file-transfer protocol on top of UDP [RFC0768]. This document defines version 0.2 of RFT. In spirit it is very similar to QUIC [RFC9000], albeit arguably a bit easier.

RFT is connection-oriented and stateful. A RFT *stream* is a unidirectional ordered sequences of byte between a *client* and a *server*. Streams support IP address migration, flow control and congestion control. The protocol guarantees in-order delivery for all messages belonging to a stream. There is no such guarantee for messages belonging to different streams.

RFT *messages* are either *chunk messages*, containing file data, or *control messages*. Control messages are used to request and signal state changes on either the server or the client. There's a 1:1 correspondence between a RFT message and a UDP datagram.

RFT employs flow and congestion control on a per-stream basis.

1.1. Terms and Definitions

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 BCP14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Commonly used terms in this document:

Client: An endpoint that participates in a stream and wants to receive a file.
Server: An endpoint that participates in a stream and wants to send a file.
Stream: An ordered unidirectional (Client to Server) sequence of bytes. It is guaranteed that each byte will arrive and that it will arrive in exactly the same order it was sent out.

1.2. Notation

We define `U8`, `U16`, `U32`, `U64` and `U128`, `U256` as unsigned 8-, 16-, 32-, 64-, 128-, or 256-bit integers. `I8`, `I16`, `I32`, `I64`, `I128`, and `I256` are signed 8-, 16-, 32-, 64-, 128-, or 256-bit integers. A `string` is a UTF-8 [RFC3629] encoded zero-terminated string. The syntax `DataType[]` defines a variable-length array of DataType; `DataType[N]` a fixed size array with `N` elements of type `DataType`.

Messages are represented in a C-style way (see Figure 1). They may be annotated by C-style comments. All members are laid out continuously on wire, any padding will be made explicit. If a field has a constant value we write it in the form of an assignment. We use the prefix `0x` to denote hexadecimal values.

```
ExampleMessage {
    U8 firstMember = 42,
    I8 secondMember = 0x01, // A hexadecimal value
    U16 padding0,
    string thirdMember,
    I32[16] array
}
```

Figure 1: Example Message

2. Streams

The protocol is based on streams. Streams are ordered sequences of bytes. RFT guarantees that all messages in a stream will arrive in-order (or will be retransmitted). There is not happened-before relation between different messages in different streams. A stream represents exactly one filepath on the server.

A stream can be established between a client and a server. A stream is identified by a **Stream ID**, a unique U16 value greater than 0. A stream ID is chosen by the server. A server **MAY** choose its stream IDs however it sees fit. Each stream contains the following state:

- **Encryption**. Right now, we don't support any form of encryption.
- **Compression**. Right now, we don't support any kind of compression.
- **Flow control**. We support flow control with a sliding window. See Section 2.6 for details.
- **Congestion control**. We support congestion control with an algorithm similar to TCP Reno. See Section 2.7 for details.
- **Connection migration**. We support migrating a stream. See Section 2.5.

2.1. Establishing a stream

A stream is established by a three way handshake:

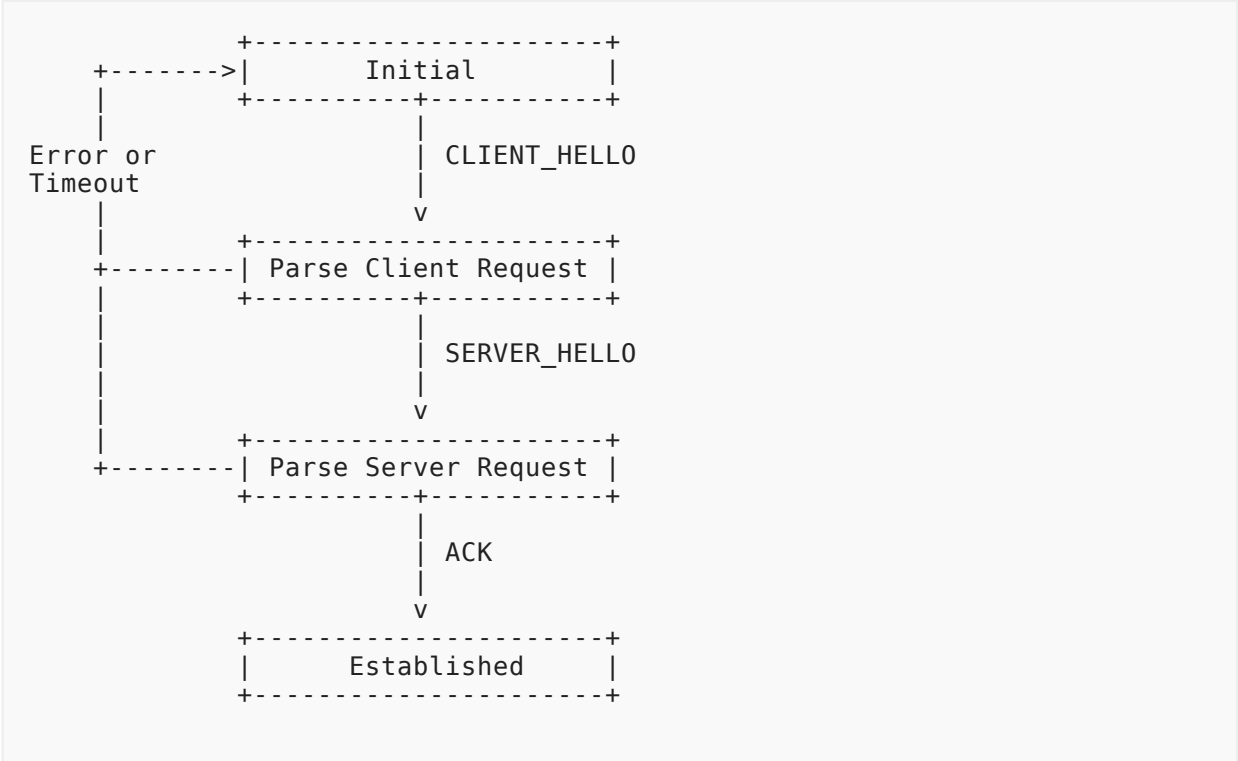

Figure 2: 3-way handshake of RFT

1. The client sends a CLIENT_HELLO (Section 3.1.1) message. The request **MUST** be filled with an appropriately sized `WindowInMessages`. To allow for the resumption of previous transmissions the client **MAY** choose an offset (in the chunk's payload size) of the file.

2. The server parses the CLIENT_HELLO message and tries to process the additional headers. Right now, none are defined, so they **MUST** be set to 0.

    1. In case of success, the server responds with a SERVER_HELLO (Section 3.1.2) message. It assigns a unique stream ID. The request **MUST** be filled with an appropriately sized `WindowInMessages` and the stream ID.
    2. In case of an error the server responds with an appropriate ERROR (Section 3.1.6) message. The server **MAY** pose arbitrary constraints. For example, it may not allow two streams to be created for the same file.

3. The client waits at most 5 seconds for the SERVER_HELLO message. If it was not received, it **MUST** retry to establish the stream (i.e., start at step 1). Any messages that are received later **SHALL** be discarded.

    The client parses the SERVER_HELLO message and tries to process the additional headers. Right now, none are defined, so they **MUST** be set to 0.

4. If the client accepts the headers it sends an ACK (Section 3.1.3) message to the server. The `SequenceNumber` **MUST** be set to 0. This concludes the stream establishing process.

2.2. Message sending

Both endpoints must validate all messages (e.g. by using the checksums). All valid messages in a stream except for ACKs themselves **MUST** be acknowledged using an ACK message. Each partner in a stream keeps track of the number of bytes the other side has sent. Both sides start out with a `SequenceNumber` of 0 after the initial handshake. Multiple messages **MAY** be acknowledged in one message (i.e., cumulative ACKs). ACKs **MUST** always contain sequence numbers that correspond to whole messages.

For each message, the `SequenceNumber` is increased by the size of the message in bytes (e.g., what you would get for the expression `sizeof(message)`). In other words: The sequence number of a message is the starting position (in bytes) of this particular message in the stream.

If a message arrives with a `SequenceNumber` larger than the one expected, the receiver **MUST** discard it and send an ACK with the last valid sequence number.

If the sender doesn't receive an ACK within 5 seconds of sending the message, the message is considered lost and **MUST** be retransmitted.

2.3. Sending data

After the handshake completes, the server **MUST** start sending the required file chunks (Section 3.2). A chunk of a file is 997 bytes. The server **MUST** respect the client's requested offset from the stream 3-way handshake. If less than 997 bytes of the file remain, the server **MUST** fill the rest of the message with 0x00. A client **MUST** ignore this padding.

2.3.1. File listing

A special case is the file with a path of "" (the empty string). In this case the server **MUST** produce a file listing, that prints the path to all available files separated by `\r\n`. Even though this file is purely virtual, the exact same mechanisms apply.

2.4. Closing a stream

A stream can be closed at any time by either the client or the server. This can be done by sending a FIN (Section 3.1.4) message. The other endpoint **MUST** ACK the message before resources can be freed. Upon reception all messages still in-flight **MAY** be discarded by both client and server. A client **MAY** choose to process outstanding CHUNK messages, though.

2.5. Stream Migration

Streams are identified by a stream ID. If the IP address of either the client or the server changes they **MUST** inform the other party with a CONNECTION_MOVED (Section 3.1.5) message. The receiver **MUST** use the information from the lower layers to change its state and send all further messages to the new address.

Any messages already in-flight **MAY** be processed by both the server and the client. If either opts not to receive them, normal error handling (Section 2.2) applies.

2.6. Flow control

Each ACK message (Section 3.1.3) contains `WindowInMessages`, the amount of messages a client may receive at any time. A server **MUST** take care to never exceed this limit. If the window remains zero for five consecutive messages the sender **MUST** assume the receiver has failed and terminate the stream.

2.7. Congestion control

Congestion control is done with an algorithm similar to TCP's congestion control [RFC5681].

The amount of messages in-flight is limited by a congestion window `cwnd`. The following rules **SHALL** apply to determine the size of the window.

Slow Start: `cwnd` is set to 1. For each received ACK or SERVER_HELLO message the window should be determined as `cwndNew = min(cwndOld * 2, receiverWindow)`. This phase stops when the slow start threshold is reached.

Congestion Avoidance: Once slow start has stopped, we reach this phase. Here we increase our `cwnd` by 1 if all messages in this window have been acknowledged. Our `cwnd` still may not grow larger than the advertised receiver window.

If we receive three duplicate ACKs we assume the message is lost and retransmit it immediately (fast retransmit). We then set our `cwnd` to half of its current size and continue with the congestion avoidance phase.

Once a timeout occurs `cwnd` is reset to 1 and the slow start threshold is set to half of the number of messages in-flight. We then go back to the slow start phase.

3. Message Formats

RFT knows two types of messages: `Control` and `Chunk` messages. Messages **MUST** have a little-endian format.

All RFT messages contain a stream ID as well as a message type. If there is no stream ID (e.g., because a stream is yet to be established) it **MUST** be set to `0`.

```
Message {
    U16 StreamId,
    U8 MessageType,
    U64 SequenceNumber,
    // More fields, depending on the message
}
```

Figure 3: All mandatory fields of any RFT message.

3.1. Control Messages

Control messages are typically very small and aren't secured by themselves. Instead, they rely on UDP's checksum for error correction. A message is a control message if its MessageType is greater than 0.

3.1.1. CLIENT_HELLO

A `CLIENT_HELLO` message is sent by a client to a server to establish a new stream. Since a stream is not established yet, the field **MUST** be set to 0 by the client. The message type **MUST** be set to 1.

The `Version` field carries information about the RFT version that the client can support. It **MUST** be set to the highest version a client can support. For RFT 0.2 the value **MUST** be 0x01.

The `NextHeaderType` and `NextHeaderOffset` fields aren't used in version 0.2 of RFT and **MUST** be set to 0 by the client. The server **MUST** reject all messages with different values. In the future they can be used for additional parameters like encryption and compression.

`WindowInMessages` tells the server the advertised receiver window.

`StartChunk` allows the client to specify an offset from which to start chunk transmission. The first chunk has an offset of `0`.

`Filename` contains the filename of the requested file. It **must** be absolute. It **MAY** be empty, in this case the server replies with a file listing.

```
ClientHello {
    U16 StreamId = 0x00,
    U8 MessageType = 0x01,
    U64 SequenceNumber = 0x00,
    U8 Version = 0x01,
    U8 NextHeaderType = 0x00,
    U8 NextHeaderOffset = 0x00,
    U16 WindowInMessages,
    U32 StartChunk,
    string Filename
}
```

Figure 4: A CLIENT_HELLO message.

3.1.2. SERVER_HELLO

The `SERVER_HELLO` message is sent as a response by the server.

- The `StreamId` field **MUST** contain the stream ID that the server has allocated for this stream.
- The `MessageType` **MUST** be set to 2.
- The `Version` field **MUST** contain `min(client version, maximum protocol version the server supports)`.
- The `NextHeaderType` and `NextHeaderOffset` fields aren't used in version 0.2 of RFT and **MUST** be set to 0 by the server. The client **MUST** reject all messages with different values.
- The `WindowInMessages` field tells the client the advertised receiver window.
- The `Checksum` field contains the SHA-3 hash [SHA3] of the entire file. [comment] # I think this one requires discussion: do we need to include the LastModified in the protocol?
- The `LastModified` field contains the number of seconds since 01. January 1970. It is in the server's timezone. Coordination about timezones between client and server is out of scope for this protocol.
- The `FileSizeBytes` contains the file's size.

```
ServerHello {
    U16 StreamId,
    U8 MessageType = 0x02,
    U64 SequenceNumber = 0x00,
    U8 Version,
    U8 NextHeaderType = 0x00,
    U8 NextHeaderOffset = 0x00,
    U16 WindowInMessages,
    U256 Checksum,
    I64 LastModified,
    U64 FileSizeBytes
}
```

Figure 5: A SERVER_HELLO message.

3.1.3. ACK

An `ACK` message is sent anytime someone wants to acknowledge something. [comment] # Do we need to send a WindowInMessages every single ACK? * `WindowInMessages` corresponds to the receive window of the sender. It's measured in messages. * `AckNumber` is the sequence number of the last message that should be acknowledged.

```
ACK {
    U16 StreamId,
    U8 MessageType = 0x03,
    U64 SequenceNumber,
    U16 WindowInMessages, //Number of chunk messages that can be held in
the client's buffer
    U64 AckNumber
}
```

Figure 6: An ACK message.

[comment] # Do we need FIN anyway? The transmission could end when sending the empty data packet, and this one creates a new condition branch to consider.

3.1.4. FIN

A `FIN` message can be sent to close a stream.

```
FIN {
    U16 StreamId,
    U8 MessageType = 0x04
    U64 SequenceNumber
}
```

Figure 7: A FIN message.

[comment] # According to discussion this thing could be taken care of by resending the last correct ACK from a new address instead - we know now where to start sending from again.

3.1.5. CONNECTION_MOVED

A `CONNECTION_MOVED` message can be sent by either the client or the server. It indicates that the sender's IP address has changed and any future communication should be sent to the new address.

```
ConnectionMoved {
    U16 StreamId,
    U8 MessageType = 0x07,
    U64 SequenceNumber = 0x00,
}
```

Figure 8: A CONNECTION_MOVED message.

3.1.6. ERROR

The `ERROR` message informs clients of an error. The tuple `(ErrorCategory, ErrorCode)` uniquely identifies each error. An optional `Message` field **MAY** give additional information, otherwise it **MUST** be empty ("").

```
Error {
    U16 StreamId,
    U8 MessageType = 0xFF,
    U64 SequenceNumber = 0x00,
    U8 ErrorCategory,
    U8 ErrorCode,
    string Message
}
```

Figure 9: A ERROR message.

ErrorCategory	ErrorValue	Meaning
0	According to the macros in glibc's error codes (see https://www.gnu.org/software/libc/manual/html_node/Error-Codes.html).	Same meaning as the glibc error codes.

Table 1: List of all valid error category/ error value combinations.

3.2. Chunk Messages

A `CHUNK` message contains the first 8 bytes of the SHA-3 ([SHA3]) checksum of the chunk. The rest are 1001 bytes (i.e. one *chunk* of the file). We chose 997 bytes so that the size of the entire UDP datagram is equal to 997 bytes.

```
ChunkMessage {
    U16 StreamId,
    U8 MessageType = 0x00,
    U64 SequenceNumber = 0x00,
    U8[8] Checksum
    U8[997] Payload
}
```

Figure 10: The structure of a chunk message

4. Future Work

4.1. Allowing a variable size chunk message

In the future it might prove advantageous to have chunk messages with variable sizes. We could support this by introducing a "length" field.

4.2. Hash algorithm negotiation

In the future there will be better hash algorithms than SHA-3. We could introduce some form of hash negotiation.

5. Normative References

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

[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/info/rfc3629>.

[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/info/rfc8174>.

6. Informative References

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

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[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/info/rfc9000>.

[SHA3] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <https://doi.org/10.6028/NIST.FIPS.202>.

Authors' Addresses

Alexander Maslew
Email: alexander.maslew@tum.de

Frederic Schoenberger
Email: frederic.schoenberger@tum.de

Jacek Kusnierz
Email: jacek.kusnierz@tum.de

Eugenio Berretta
Email: eugeniovinicioberretta@gmail.com

Aleksandra Swierkowska
Email: aleksandra.swierkowska@tum.de