MFTP: the *Minimalist File Transfer Protocol*

Version 1.00 - 17/Jan/2012

George W. Taylor

https://www.TropicalCoder.com

Specification

1. Introduction

MFTP is a protocol for reliable data transport across the internet via the unreliable User Datagram Protocol (UDP). Ideal for embedded or IoT platforms with constrained resources where TCP's overhead is undesirable or impractical. Sends full 512 byte blocks of data with no overhead needed for ancillary information, enabled by acknowledgements with CRC32 of block received compared by the CLIENT with CRC32 of block sent.

While UDP has a checksum for basic data integrity checking, the 16-bit checksum has limited error-detecting capability. It can miss some types of errors, such as those that invert bits in a way that results in the same checksum (e.g., two-bit errors that cancel each other out).

For applications that require strong data integrity guarantees, additional measures must be taken at the application level to verify data integrity more robustly.

MFTP provides a stronger guarantee of data integrity by implementing additional CRC32 checks, which are used for the acknowledgments.

2. Data Types Involved

- `FILE_SIZE`: 32 bit unsigned value
- `CRC32`: 32 bit unsigned value
- `BLOCK_NUMBER`: 32 bit unsigned value
- `SHORT_MSG_SIZE`: 8 bytes
- `FILENAME`: variable size, null terminated, 1 to 256 bytes
- `HDR_MSG_SIZE`: variable size - 5 to 512 bytes zero padded such that `HDR_MSG_SIZE != SHORT_MSG_SIZE`
- `DUP_WARN`: 0xFFFFFFFF (optional - see below)
- `WAIT_TIME`: 2 seconds
- `MAX_WAIT_TIMES`: 10
- `MAX_RESENDS`: 10
- `SOCKET_TIMEOUT`: 20 seconds (adjust as required)

3. Messages

3.1 HDR_MSG

A `FILE_SIZE` followed by a `FILENAME` for a total of `HDR_MSG_SIZE` bytes.

(There is plenty of room here for other header information if desired.)

- Sent by the CLIENT who initiates the data transfer, the `HDR_MSG` must not be `SHORT_MSG_SIZE` in total so there is no chance of the SERVER mistaking it for a `SHORT_MSG`. This can be assured by simply appending a zero as necessary.
- The first packet received must be assumed to be the one and only `HDR_MSG`, except...
- If the SERVER is on a multi-homed platform - ie: if it has more than one IP address, since the first `HDR_MSG` sent is broadcast the SERVER may receive more than one copy. If this arrives immediately after the first, it will be swallowed by the redundancy checking.

However, it may arrive much later. Therefore the SERVER must save the `CRC32` from the `HDR_MSG` and compare it against all blocks received to see if it is a repeat of that first `HDR_MSG`, and if it is, it must be silently discarded. It is recommended that if a block arrives with that same `CRC32`, then its first 32 bit value be compared with the `FILESIZE` value to confirm that it is indeed a repeat before discarding it.

3.2 SHORT_MSG

A `CRC32` followed by a `BLOCK_NUMBER` for a total of `SHORT_MSG_SIZE` bytes. There are three types of `SHORT_MSG`:

- `ACK_MSG`: a `CRC32` followed by a `BLOCK_NUMBER`
- `REP_MSG`: a `CRC32` followed by a `BLOCK_NUMBER`
- `DUP_MSG`: a `CRC32` followed by a `DUP_WARN` (optional)

The type of message received can only be identified by the number of bytes received. Any message that is `SHORT_MSG` sized bytes must be one of these 3 types, but how do we know an `ACK_MSG` from a `REP_MSG` from a `DUP_MSG` when received?

The CLIENT only ever receives an `ACK_MSG`, so any `SHORT_MSG` sized message must be that. The SERVER can only receive either a `REP_MSG` or a `DUP_MSG`, and can distinguish between them by looking at the `BLOCK_NUMBER` to see if it is a `DUP_WARN`.

3.3 DATA_MSG

1 to 512 bytes, excluding `SHORT_MSG` sized bytes

- Full 512 byte blocks of file data are sent until the last block of the file, which may be anywhere from 1 to 512 bytes.
- Measures must be taken to ensure an 8 byte `DATA_MSG` cannot be mistaken for a `SHORT_MSG` by using an extra zero pad byte.

3.4 REP_MSG

A signal to the SERVER that the data in the next block is the same as the last, and it is to simply copy the previous buffer to file again. Then the CLIENT will expect an `ACK` with the same `CRC32` as before, but the block number incremented from there.

3.5 DUP_WARN_MSG

A signal to the SERVER that the next block has the same `CRC32` as the previous block but the data is different.

A `DUP_MSG` followed by a `DATA_MSG` is sent separately and sequentially, with no pause between them to wait for an ACK. The CLIENT will then expect an ACK for the `DATA_MSG`. If the ACK does not arrive, both the `DUP_MSG` and the `DATA_MSG` must be resent.

Optional - see section 6. "Duplicate Blocks of Data and CRC Collisions" below.

3.6 EOF_MSG

0 bytes

4. File Transfer Process

To begin, the SERVER is listening for UDP packets on a port known to the CLIENT. The CLIENT broadcasts a `HDR_MSG` every `WAIT_TIME` seconds to the SERVER at the expected port.

4.1 Receiving HDR_MSG

- The SERVER parses the `HDR_MSG` and prepares to create a file with the filename passed.
- It responds by sending an `ACK_MSG` to the CLIENT, with the `CRC32` calculated from the `HDR_MSG` received, and `BLOCK_NUMBER` field set to zero.

4.2 Sending Data

When the CLIENT receives the `ACK_MSG`, it verifies that it contains the `CRC32` of the header it sent and a `BLOCK_NUMBER` of 0. The CLIENT then proceeds to send a `DATA_MSG` with the first 512 byte block of file data. No header or ancillary data is sent with these blocks. The handshake is thus established. The CLIENT always waits for an `ACK_MSG` from the SERVER before sending each next message.

4.3 Block Numbering

- Blocks are numbered starting from 0. The CLIENT increments its internal current block number variable with each block sent.
- The SERVER increments its internal current block number variable with each block received.
- The SERVER writes each block to file in the sequence received. The SERVER always sends an `ACK_MSG` after each message received, consisting of the `CRC32` computed on that block plus the block number the SERVER assigned it.

4.4 Handling ACK_MSG

When the CLIENT receives an `ACK_MSG`, it checks the `CRC32` and block number in that message. If they are the same values it has recorded for the last block sent, it proceeds to send the next block. If these values correspond to the previous block sent, it assumes the latest block (or message) was not received, and resends it. If these values correspond to neither the last block nor the previous block, it displays an error and exits.

4.5 Resending ACK_MSG

If the SERVER does not hear from the CLIENT within a `WAIT_TIME` period, it resends its last `ACK_MSG`. The SERVER will continue resending the `ACK_MSG` until the CLIENT responds, or it has reached `MAX_RESENDS`.

4.6 Resending DATA_MSG

If the CLIENT does not receive an `ACK_MSG` within a `WAIT_TIME` period, it resends its last block or message. The CLIENT will continue resending until the SERVER responds, or it has reached `MAX_RESENDS`. If either of these reach `MAX_RESENDS` it will display an error and exit.

5. Resending Strategy

Since the CLIENT includes no headers or ancillary data along with the buffers it sends, processing the acknowledgement from the SERVER is key to ensure synchronization with the SERVER as sending of the file progresses. The SERVER only ever sends one kind of message, the `ACK`, which contains the `CRC32` of the last block it received and also the block number the SERVER assigned to that block.

Of course, the very first message received by the SERVER is not a block of file data, so 0 is returned in the `ACK` as the block number. The first block of file data will be assigned a block number of zero, and block numbers will increment from there.

Through analyses of the `ACK` the CLIENT learns how it is going with the SERVER. If the `CRC32` and block number agree with the CLIENT's record of the last block it sent, the CLIENT can rest assured that the SERVER has received that buffer and that all is well on that end. If the `ACK` instead contains the `CRC32` and block number for the previous block sent, the CLIENT knows that the SERVER never got the latest block it sent and that it should send it again.

If the `CRC32` or block number do not match up with either the values for the last block or the block sent before that, the CLIENT understands that the SERVER is hopelessly confused and terminates the session. There is nothing the CLIENT can do. Of course, a method could be developed to attempt a recovery, but in reality this never happens in bug-free builds. One could simply try sending the file again should the first attempt fail.

It is essential for the health of any connectionless network that unacknowledged packets be generated at exponentially decreasing rates. Depending on the application (e.g., in potentially high traffic networks), consider resending at exponentially decreasing rates, i.e., after 2 seconds, then again after 4 seconds, then again after 8 seconds, etc. The `SOCKET_TIMEOUT` value should be reconsidered in this case.

6. Duplicate Blocks of Data and CRC Collisions

If the CLIENT encounters a block of data in the file stream that computes to the same `CRC32` as the previous block, it compares the data in that block with the data in the previous block...

1. If the data is identical (expected), it does not send that data again, but rather, sends a `REP_MSG` to the SERVER with the `CRC32` of that block and the next block number in the sequence. (Example - sending a large executable file that may have two or more blocks of zeros in sequence.)
2. If the data is not identical (unexpected), it sends a `DUP_WARN_MSG`, consisting of the `CRC32` of that block and the next block number in the sequence in a `DUP_MSG`, then sending the block itself as a `DATA_MSG`. *This would be an extremely rare occurrence. There is a nearly infinitesimal probability that data would have two blocks with different data, but the same `CRC32` in sequence. However, handling for this is presented to complete the logic of the protocol.*

The SERVER employs alternating buffers to receive data such that the last block received is not overwritten by the next block or message to arrive.

When the SERVER receives a `REP_MSG` it writes the *previous* `DATA_MSG` received to file (again) and responds with an `ACK_MSG` as usual.

The SERVER employs redundancy checking on each `DATA_MSG` or `REP_MSG` that arrives to verify if it already has that block with that `CRC32`. If it is redundant, it silently discards the block - without reply.

When the SERVER receives a `DUP_MSG`, it disables its redundancy checking for the next block, awaits the next block and processes it as normal when it arrives. However, before processing it it checks the `CRC32` of that block against the last and the previous. If that `CRC32` is the same as the last and the one before the last, it is silently discarded as an inadvertent duplicate. Finally, it re-enables redundancy checking.

As the SERVER receives each `DATA_MSG` or `REP_MSG`, it verifies that redundancy checking has not been inadvertently left disabled by the expectation set up by a `DUP_MSG` that was not followed by a `DATA_MSG`.

7. End of Transfer

When the CLIENT completes sending all full 512 byte blocks of file data, the last remaining data may be any number of bytes between 1 and 512. If by coincidence the number of bytes happens to be `SHORT_MSG_SIZE` (8 bytes), it records the `CRC32` for that block before zero padding to `SHORT_MSG_SIZE` + 1 (9 bytes).

If the SERVER receives `SHORT_MSG_SIZE` + 1 bytes, it knows this must be the last block in the file, and that it might include a 0 pad byte. It compares what the final file size will be with the last byte against the file size it received in the beginning to know whether to drop the last byte. If it drops the last byte, it must recalculate the CRC for the `ACK` excluding that last byte.

When the CLIENT completes sending all file data, it may display a success message to the user and sends a sequence of (`MAX_RESENDS` + 1) EOF messages with a half-second pause between each and terminates, without looking for a response from the SERVER.

When the SERVER receives an EOF, it may display a success message to the user, closes the file, and terminates, sending no acknowledgment back to the CLIENT.

8. Security Considerations

MFTP does not address security considerations

9. Performance Enhancement

While the MFTP’s throughput may initially appear limited compared to conventional reliable data protocols, leveraging multi-path transmission across multiple ports significantly enhances its bandwidth capabilities. This approach requires a straightforward protocol for port negotiation, ensuring efficient data distribution.

In practice, the protocol would allocate file segments to ports sequentially, with the lowest port number handling the initial segment and the highest port number managing the final segment.

Once the port negotiation is complete, the MFTP can operate concurrently on each port without modification, thus providing a reliable and scalable solution for data transfer over UDP.

10. Relevant RFCs for MFTP

To understand the relevant RFCs for a protocol like MFTP that aims to provide reliable data transport over UDP, we can look into a few key documents that outline best practices and methodologies for such implementations:

RFC 8085 - UDP Usage Guidelines

This document provides comprehensive guidelines for using UDP, emphasizing the necessity of implementing mechanisms to handle varying path conditions, such as transmission delays and congestion levels. It also covers congestion control and other essential features for reliable data transport over UDP. Read more.

RFC 8304 - Transport Features of UDP and UDP-Lite

This RFC details the features and primitives of UDP and UDP-Lite, including how these protocols provide datagram services. It includes information on the APIs for sending and receiving datagrams, as well as the importance of checksum protection for ensuring data integrity. Read more.

Draft RFC on Reliable UDP (RUDP)

Although this is an expired draft, it discusses a reliable UDP protocol that builds on the principles of RFC 1151 and RFC 908. It outlines mechanisms for reliable, in-order delivery of packets and flexible design parameters suitable for various applications, including telecommunications signaling. Read more.