Protokoll für Richtigkeit, Ordnung, Transport, Optimierung, Kanalunabhängigkeit, Ortsunabhängigkeit, Latenzminimierung und balancierte Lastverteilung - Das PROTOKOLL

Jaro Fietz & Noah Bergbauer

Contents

Abstract

This document defines the *Protokoll für Richtigkeit, Ordnung, Transport, Optimierung, Kanalunabhängigkeit, Ortsunabhängigkeit, Latenzminimierung und balancierte Lastverteilung*, short *Das PROTOKOLL*. Das PROTOKOLL is a UDP-based protocol optimized to upload files to a remote machine with support for multiple clients and uploads at the same time. It is designed to interact nicely with both short-range networks and long fat pipes. It features low data overhead, resumability if a connection is aborted and provides easy means for extension. It supports files with a size of up to 16 EiB.

Introduction

Most modern file upload servers and services use a HTTP-based protocol for uploading files. HTTP is based on TCP, which has its advantages like automatic congestion control and reordering, and thus less complexity. But these advantages are also the reason why TCP is not optimized for file upload. For example if a packet is lost, the application won't get any following packet until that packet is retransmitted and received. For file transfer this is undesired behaviour, because following data chunks do not depend on previous chunks and can be written to disk at the chunk's corresponding position. Das PROTOKOLL tries to eliminate these disadvantages of TCP by choosing UDP as underlying transport protocol.

Together with this specification comes a reference implementation of the proposed protocol called `csync`. Whenever implementation decisions are described, the decision of the reference implementation is discussed.

Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Terms and Definitions

User Datagram Protocol (UDP)

The User Datagram Protocol is an OSI layer 4 (transport layer) protocol specified by the IETF.

UDP Flow

The UDP flow for Das PROTOKOLL is defined for the server by the 2-tuple of the client's IP and port. For the client the UDP flow is defined as the 2-tuple of the server's IP and port.

Maximum Segment Size (MSS)

While MSS is a term defined by TCP, this specification uses MSS similarly for UDP. The maximum segment size as used in this specification defines the maximum number of data octets within a UDP packet. The MSS is calculated with $MTU - packet\_overhead$ where the $packet\_overhead$ for UDP is 40 bytes. Common values are usually 1460 for the internet and 65496 for localhost connections of linux systems.

Round-Trip Time (RTT)

The time between sending a packet and receiving an answer to that packet.

Inter-Packet Time (IPT)

The time between reception of two packets in the same UDP flow.

General Design

Das PROTOKOLL starts with the client sending a login packet to the server, authenticating itself and initiating the connection with the command it wants to use. The server responds with a packet that may contain additional information as specified by the command. Both client and server then start executing the protocol specified by the command.

Only one command can be performed per connection. Otherwise delayed packets from an old command could be misinterpreted by the new command within the same UDP flow. Each connection must use a different UDP flow, e.g. by using a different source port, or be delayed at least 10 seconds.

Integrity check of packets is not handled by this protocol. This specification requires the UDP checksum to be used to provide data integrity of packets. UDP Checksums are required in IPv6, but are optional in IPv4. Nevertheless, all modern systems provide and use UDP checksums by default for IPv4. Thus, this protocol should work on all modern systems. Otherwise care must be taken to ensure that UDP checksums are enabled both on the server and the client side.

Primitive Encoding

Primitives in the protocol are encoded in the following different ways.

Varint

Varint encoding is used to encode positive integers of unknown length. Varints are defined in the Google Protobuf Encoding of Base 128 Varints. Integers are divided into septepts and represented as little-endian. The most significant eighths bit is used to indicate if the following septet belongs to the same number or if the following septet is the last one in this integer, similar to UTF-8 encoding of characters.

Implementations MUST decode varints at least to a 64bit integer to enable files with a length of up to 16 EiB, and MAY support larger internal representations. `csync` uses 64 bit unsigned integers to represent varints internally.

Fixed-Length Integers

Fixed length integers have a length property associated with it. That length is not encoded, but required to properly encode integers as fixed-length. To encode an integer to fixed-length, it is represented in little-endian and padded to `length` number of bytes.

Length-Prefixed Data

To encode length-prefixed data the length of the data is prepended as varint, followed by the encoded data.

Runlength Encoding (RLE)

The runlength encoding is the number of consecutive same bits, starting with 1. That number is encoded as varint.
Usually in RLEs first the number is written followed by the number of times that number follows consecutively. In the runlength encoding used within this specification bits are counted instead of bytes or numbers, which makes prepending the bit redundant. Thus, counting starts with set bits (ones) and alternates between the two states of bits.

Login Packet

The connection starts with the client logging in to the server. The login packet consists of a length-prefixed client token, which SHOULD be generated randomly by a cryptographically secure pseudo-random generator when connecting to the server for the first time. After the first connection the same client token SHOULD be used to get access to previous uploaded files. Its length SHOULD be chosen to guarantee uniqueness among all client that connect to the server, like 16 bytes.

The server MUST ensure that the same client token has access only to the files that client token has uploaded except for a negligible probability. `csync` calculates the SHA256 sum of the client token and uses the resulting hex-encoded digest as folder name for that client.

The client token is followed by the encoded command, which defines further communication packets and the further protocol used within this connection.

Command

The command is encoded starting with the type discriminator of the command. The type discriminator is one byte long and used as tag to indicate the command. The following data is defined by the respective command.

Currently the only valid command is the *Upload Request*, indicated by the type-id `0`.

Upload Request

The upload request uses the type-id `0`. The type-id is followed by the length-prefixed path of the file. The path MUST be encoded as valid UTF-8. Foreslashes are interpreted as path separators. Foreslashes inside folder- and filenames MUST be escaped with a leading backslash. After that the length of the file is written as varint. The upload request initiates the upload sequence.

Upload Sequence

After receiving the upload request from the client, the server checks if that file has already been uploaded in the past. If there is a bitmap associated with the file, which is not complete, the upload is resumed. If the associated bitmap file is full, which can only happen if the connection is aborted during end of transmission, it is treated as normal upload resumption. The server will send the full bitmap as status update to the client, which will acknowledge it with the extension message `0` (FIN). If the file does not yet exist or there is no bitmap associated with the file the bitmap is reset to all zeroes and the file must be (re-)uploaded completely. The server creates the file and sets its length to the file length provided by the client. The server MUST answer with a status update as described in Status Update. The client then starts uploading Chunks.

Chunks

Chunk packets are used to transfer the actual data of the file to the server. Each chunk is indexed sequentially starting from zero. That index is written as fixed-length integer at the beginning of each chunk packet. The length of the id is calculated from the file-length with $\lceil \log_2(\textit{number of chunks} + 1)/8 \rceil$. The *number of chunks* is the number of chunks required to send the whole file to the server with the configured MSS.

The chunk-id is followed by the chunk data. Each chunk except for the last one MUST completely fill the UDP segment size. For example if the chunk-id has a length of one byte and the segment size is 1460, then each chunk's data (except the last one) must have a length of 1459 bytes.

When receiving a chunk the server has already received (it is `1` in the bitmap), the server MAY choose to discard the chunk instead of handling it. When handling a chunk the server MUST seek to the chunks position in the file: $pos = chunk\_size * chunkid$. It then writes the chunk to that position in the file. The chunk-id thus solves the problem of out-of-order packets.

Chunk ids larger than the number of chunks required to transmit all contents of the file are used for extension messages. The discriminator of extension messages is gotten by subtracting the number of chunks from the chunk-id.
Currently the only officially supported extension message has the discriminator `0` and is used by the client during End of Transmission handling.

Status Update

The server MUST hold a list of received chunk ids in some internal representation. This list MUST be persisted to allow resuming file upload if the connection is aborted. The internal and persisted representation SHOULD be a bitmap of received chunks.

The status update is a packet consisting of the run-length encoded bitmap of received chunks, truncated to the MSS. The server MUST send status updates periodically to the client. The interval between two status updates is defined by two different metrics, whichever occurs first. The metrics are based on the round-trip time and inter packet times. The RTT is gotten on the server side by measuring the time between sending the first status update (the second packet of the connection) and getting the first chunk from the client. The inter-packet time (IPT) is a moving average over chunk packets received from the client. Packets per second (PPS) are calculated with $pps = 1/ipt$.
The first metric defines a status interval, which is the number of chunk packets received from the client until the next status update is sent. That number of packets is defined by $num\_packets = \lfloor pps/\ln(pps + 1) \rfloor$.
If a burst-loss of packets occurs or the IPT increases, that metric may result in a very late or delayed status update. Thus, a second metric is added as timeout. The timeout is calculated by adding the RTT to the expected time it should take to receive *num_packets* packets from the client: $timeout = ipt * num\_packets + rtt$. If the first metric is not met within the timeout, a status update is sent regardless. Whenever a status update is sent by one of

those two metrics, both metrics are reset.
Additionally, whenever the number of zeroes in the bitmap is a power of two or zero, an additional status update is sent, not influencing the periodic status update metrics. This ensures that at the end of the connection, when the number of missing chunks is low, enough status updates are sent to client, such that the client has quicker feedback instead of waiting until the timeout triggers.

The client is free to maintain the bitmap in-memory in any way. Whenever a new status update is received by the client, the client MUST update its internal bitmap. Packets are assumed to be lost during transmission iff a chunk the client sent more than $1.5 * RTT$ ago is not marked as received in the server's bitmap sent to the client. The client SHOULD retransmit lost packets starting from the oldest lost one. This ensures that the amount of ones in the beginning of the bitmap is maximal and thus the RLE minimal. If no lost packet is left to retransmit, the client continues uploading the next chunk it has not yet sent to the server.

The bitmap is expected to have a large number of ones in the beginning, representing received chunks, and a large number of zeroes in the end, representing chunks not yet sent by the client. Between those two large groups may be an area of mixed zeroes and ones, which is the current "working area" of sent and lost chunks from the client.

The RLE compresses the ones in the beginning and zeroes in the end very well. Packet loss usually occurs in bursts. Thus, the working area is expected to be bursts of ones and zeroes, which will be encoded nicely as well. The worst case for RLE is a strictly alternating bitmap, which results in n bits being encoded into n bytes. Such an encoding, or any encoding with a lot of alternating bursts, may be larger than the MSS. This is handled by truncating the runlength encoded bitmap to the MSS. Such a truncation is expected to hold enough information about missing packets for the client until the next status update is sent and received. Even if it doesn't, the result will be gratuitous retransmissions, leading only to reduced performance and not influence the correctness of the protocol in any way.

A status update can be lost during transmission without many implications. The client will not know which packets have been lost and thus will continue sending not yet sent chunks. Thus, the only downside is that the RLE will become larger if packet loss occurs.

End of Transmission

The connection is aborted if there is no incoming or outgoing packet for more than 10 seconds on either side. If the connection is aborted, it needs to be reestablished starting with the login.

The connection is finished successfully if the client acknowledges the reception of a full bitmap from the server with a chunk packet with extension message `0`. After successful termination, both the client and the server SHOULD keep the connection open for another 10 seconds to catch any old packet that might still be in transit but have not yet reached their target.

Due to the increased number of status updates in the end of the connection, the server sends a status update as soon as it has received and written the last chunk. If that update is lost, the next one will be triggered by the timeout of periodic status updates. If all of those status updates are lost, the 10 second connection timeout will trigger and the connection will be aborted. On the next connection from the client the server will see a full bitmap and will directly send the FIN packet. The same occurs if that message from the client does not reach the server. As the server will continue sending status updates to the client until it has received the acknowledgement, the client SHOULD resend its FIN packet if it received another full status update during the 10 second shutdown period.

MSS

The maximum segment size is configurable, but must be fixed between the server and client over all connections if incomplete transmissions are going to be resumed. This is due to the chunk bitmap being based on a fixed chunk_length, which is dependent on the MSS. A different MSS would result in different chunk_length calculations resulting in an invalid bitmap.

The minimal MSS is 14 bytes.
The minimum chunk packet size is 9 bytes. The chunk size is 1 byte with a chunk-id of 8 bytes to support 16 EiB large files.
A status update must be able to encode at least the number of leading ones to indicate one missing chunk or the successful reception of all chunks. With a chunk-id of 8 bytes, the status update must be able to encode the number $2^{64} - 1$. The storage space for a varint integer in bytes is $\lceil \log_2(n)/7 \rceil$. Thus 10 bytes are required to encode $2^{64} - 1$.
The login packet consists of the client token and the command. Assuming a single client, the client token can be empty, resulting in a length-prefix of one byte and an empty client token. The command has a one-byte tag, a length-prefixed path and the file length as varint. Assuming a path length of only a single character, we end up with the total length of the login packet $1 + 0 + 1 + 1 + 1 + 10 = 14$ bytes.

If a smaller MSS is required, a non-standard implementation may split up the login packet into multiple smaller packets, leaving that implementation with a minimum MSS of 10 bytes.

Extensibility

This protocol is easily extensible. New commands can be introduced by defining a new type-id. Within the upload sequence new packet types can be added by means of extension messages.

Security

Security is out of scope of this protocol. Due to the configurability of the MSS for Das PROTOKOLL it can interoperate with data link layer and transport layer security protocols. For example Das PROTOKOLL can be used on top of IPSec as layer 3 protocol.

Error Handling

Currently error messages are not part of the protocol. If an error occurs on either side, the connection SHOULD be aborted. The side aborting the connection SHOULD drop every packet from the same UDP flow for the next 10 seconds to make the other side trigger the timeout.

Future Work

Error Handling

Implementing proper error handling into the protocol would enable the connection to terminate more gracefully on an error instead of dropping packets for 10 seconds in order to fire the timeout on the other side.

File Checksums

Another further extension is to exchange checksums of files the sever has that the client wants to upload in order to not upload existing unchanged files.

Proper RTT

Currently the RTT is only gotten in the beginning of the protocol during the handshake. The current RTT is not precise, because it includes the time on the client that it takes to read the first chunk of the file, which has a high overhead. A proper RTT implementation would ensure that no disk overhead or overhead of any sort is present when measuring the round-trip time. This could either be done by sending packets without IO overhead, or by including the time needed for an operation within the packet. For example in the current handshake the client could measure the time it takes to read the chunk and send that time together with the chunk to the server.
Furthermore the RTT should be measured periodically within the connection and a moving average be calculated.

End of Transmission Timeout

The end of transmission timeout is current set to the fixed value of 10 seconds. It would be better to base that value on the RTT both for a quicker timeout for fast connections, and for better handling of long pipes.
Together with this, a change to the handshake would be useful to not have to wait for 10 seconds to find out if a packet was lost, but instead react more dynamically to possible packet loss within the handshake.

MSS Probing

Another future extension would be to implement MSS probing and support differing MSSs between connections. MSS probing would allow automatic MSS discovery and move the burden of configuring the correct MSS away from the user to the program itself. It would also make use of the largest MSS possible, reducing overhead from the protocol.
Different MSSs could be supported by saving the MSS of a bitmap together with the bitmap. When loading the bitmap, the bitmap is adapted to the new MSS with old chunks which would be missing data with the new MSS being reset to zero.

Varint vs Fixed-Length for Chunk Ids

A very small microoptimization is to find out for a given file-size if varint encoding of the chunk ids is better than fixed-length encoding and use the better one dynamically. The advantage of varint encoding is that for the first few chunks the chunk id is smaller, while it is larger than fixed-length encoding for larger chunk ids. There is a range of values of number of chunks where varint encoding has less overhead than fixed-length encoding, but fixed-length encoding has less overhead for more values, which is why the current specification uses those.

Congestion Control

Adding congestion control would highly benefit the protocol and reduce packet loss in connections with other data flow. Currently the client just writes data as fast as possible without any rate control or limitation.

Out of Scope

There are a few edge cases which are out of scope of this protocol specification. They should be handled on the application layer and not on the protocol layer. These include, but are not limited to, a file changing while it is being uploaded, a file being partially uploaded before the connection aborts, after which the file changes, before the upload continues, and how to handle a client updating the same file twice at the same time.