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

Jaro Fietz & Noah Bergbauer

Contents

TODO: examples of encoding

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. TODO: Minimum MTU

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 a 3-way-handshake. In the handshake the client starts with a login packet, authenticating itself and initiating the connection with the command it wants to use. The server responds with a packet that is either empty or contains additional information required for the command specified by the client. The client then starts executing the protocol specified by the sent command by sending the first packet.

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 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 we count bits and not bytes or numbers, which makes prepending the bit redundant. Thus, we start counting set bits (ones) and alternate 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. 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 MUST answer with a status update as described in Status Update. It can then create the file and SHOULD set its length to the file length provided by the client. 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 $\lceil \log_2(number\ of\ chunks)/8 \rceil$. The *number of chunks* is the number of chunks required to send the whole file to the server with the configured MTU.

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 (except the last one) must have a length of 1459 bytes.

When receiving 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.

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 marked as 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.

- Client has a "cursor" pointing into the chunk bitmap
  - start at the beginning, keep going forward as we send out chunks
  - obviously, only send chunks that are marked as "not received"
  - whenever client receivs a status update indicating lost packets, we reset the cursor to the beginning
  - we assume packet loss iff a chunk we marked as sent _more than 3*RTT ago_ is not marked as received in a status report from 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.

- 3-way-handshake to get RTT information
- Only one command per session
  - Otw an old chunk could reach server after new upload is initiated
  - New connection must be from different port
- End of transmission is indicated by largest chunk-id → no explicit end-message from client required
  - protocol is terminated by a status report from the server indicating that every chunk was received (bitmask of ones), which will **always** fit into a reasonably sized MTU due to RLE
- Checksums / data integrity handled by UDP
- Fixed, configurable MTU
- Extensible by introducing new commands

3-Way-Handshake

- Client collects all information beforehand
  - Filename, Size of File
  - Reduces delay during Handshake
  - → More precise RTT information
- Client → Server: Login Packet
  - Server must answer before doing anything else to not influence RTT
- Server → Client: Empty Packet
  - Calc RTT on Client
- Client → Server: Command Packet
  - Calc RTT on Server

Upload Request

- Path / Filename
  - Rust: Server must remove leading `/` before using `PathBuf::push`
  - Foreslashes inside filenames must be escaped to differentiate them from folders
- If the file is already on the server:
  - If it hasn't been uploaded completely, resume upload by sending bitmap to client
  - If it has been fully uploaded in the past, delete file and upload again

Chunk

- Chunk ID
  - Number of Chunk starting at 0
  - Encoded as little endian in n bytes
    - n = ceil(log2(number of chunks needed) / 8)
- If chunk has been received by the Server in the past (i.e. is `1` in the bitmap), it's discarded

Congestion Control

- ?? Initially Client sends burst of packets for `RTT / m` milliseconds ??
- probably: slow start like tcp (but mb a bit faster?)

Connection End

- All chunks have been received
- 10s without a packet from client

Problems:

- small MTU size such that runlength number is larger than MTU:
  - storage space for a varint integer in bytes `space(x) = ceil(log2(x) / 7)`
  - assuming the server status report contains no headers or anything else protocol is bounded by: `space(x) < MTU`
  - thus: `ceil(log2(x) / 7) < MTU` ; `x < 2 ^ (MTU * 7)`
  - where `x` is the RLE integer that counts the number of existing chunks at the start of the file
  - assuming a worst case chunksize of 1 byte, this means that we need an MTU of at least 10 bytes (plus overhead from headers etc) to support all 64-bit file sizes (16 EiB).
- MTU Probing
- MTU must have a size of least `max(10, maxlength of file path + length of chunk-index-field)`

Extensibility

Open Questions / TODO

- Error messages to Client
  - Handle `unwrap`s
- Which files does the server have?
- How to handle a client updating the same file twice at the same time?
- How to handle successful connection FIN?

  - What if last status update of server gets lost, so the client does not know that it's over?
- Slow Start
  - How often should a status update be sent in the beginning?
- Check usage of usize vs u64 everywhere to make sure we support u64 large files
- RTT: moving average
- Improvement: Send time needed on server / client for calculations between having received the chunk and sending the next chunk for more precise RTT information.

Out of Scope (for now)

- File changes during upload
- File partially uploaded, connection aborts, file changes, new connection, file upload continues
- Congestion Control