Protokoll für Richtigkeit, Ordnung, Transport, Optimierung, Kanalunabhängigkeit, Ortsunabhängigkeit, Latenzminimierung und balancierte Lastverteilung - Das PROTOKOLL v2

Jaro Fietz & Noah Bergbauer

Contents

Abstract

This document defines version 2 of the Protokoll für Richtigkeit, Ordnung, Transport, Optimierung, Kanalunabhängigkeit, Ortsunabhängigkeit, Latenzminimierung und balancierte Lastverteilung, short the PROTOKOLL. The PROTOKOLL is a UDP-based protocol optimized for file synchronisation between peers. It is designed for a single-user multiple-clients scenario and interacts nicely with both short-range networks and long fat pipes. It features interruptible, resumable, secure, parallel synchronization of files of any size.

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. The 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.

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.

`inotify`

On Linux based systems inotify[1] provides a mechanism for monitoring file system events.

FUSE

Filesystem in Userspace is an interface to allow userspace programs to create custom file systems, handling all reads, writes and events themselves.

WebDAV

WebDAV is an extension to HTTP allowing file operations.

Samba

Samba is a network protocol for file transfers.

[1]http://man7.org/linux/man-pages/man7/inotify.7.html

NFS

Network File System specifies a protocol for communication in a distributed network storage system.

Design

This specification describes three main different aspects, before combining them into the final protocol. First, the frontend represents the virtual filesystem to the user. It takes the information from the blockdb to assemble files. Second, the blockdb is the internal representation of blocks of files. Third, the node network topology will be discussed.

Frontend

The frontend is used to generate the actual file tree and file content. It uses the information stored in the blockdb to assemble files. It is informed about changes in the blockdb, which it MUST propagate to the actual files. Additionally, it MUST detect changes of files, translating them into blockdb changes.

The actual file-backend of the frontend are deliberately unspecified to allow for use-case implementation specific optimizations. Examples for different file-backends are plain files with `inotify`, FUSE, a webdav / HTTP API or other protocols like Samba or NFS.

The reference implementation uses plain files with periodic polling.

BlockDB

The blockdb is the core of the protocol.

It stores files and the file tree as an efficient internal representation. This allows certain optimisations like only needing to store equal files once, reducing file transfer through usage of existing chunks and chunk level encryption.

The blockdb is an independent storage of raw blocks. It delivers information to the frontend, allowing it to render the file-tree. In general it's a modified version of a Merkle-Tree with use-case specific optimisations. The blockdb "backend", i.e. the physical storage is unspecified. For instance, blocks can be stored as plain file or in a database. The reference implementation uses an (ephemeral) in-memory database to simplify the implementation.

Invariants

The blockdb has several invariants. It is content-addressable: each block is identified by its blockid, which is the hash of the encrypted block. Thus blocks are copy on write (CoW). The hash algorithm is configurable, but not compatible. If it's changed, the entire blockdb must be converted accordingly on every node.

The suggested hash algorithm is Keccak-256. The length of blocks is variable, blocks can be arbitrarily large. By default each file and directory is a single block. Storing partial blocks allow resumption of incomplete or aborted transfers and SHOULD be supported by implementations to reduce network overhead.

The root of the blockdb is a single blockref without any hints (thus only containing a blockid and its key). The root MUST point to a directory block.

Blocks

Each block is identified by its blockid, which is the hash of the encrypted data. There are three different types of blocks. Directory blocks contain blockrefs to child directories or files and holds their metadata. Leaf blocks contain actual file contents, which can be any arbitrary payload data. File meta blocks represent a file consisting of multiple blocks. Before the different block types are discussed in detail, blockrefs are needed.

Blockrefs

Blockrefs are used to identify and reference blocks and contain all information needed to decrypt those blocks. It consists of the blockid of the referenced block, its decryption key and it MAY contain hints. Hints indicate that a block can be created from other blocks without needing to download the whole block if other blocks are present. Hints are a list of blockrefs, each with an offset and a length. If hints are provided, the referenced block can be created by concatenating the respective decrypted subranges of the referenced blocks each from their respective offset until their offset plus their respective length and then re-encrypting with the key of this block. This can be used to elide the transfer of blocks that can be combined from local data as well. This behaviour can be used when changes are detected within a file. Given a file is currently represented as a single block and that file is changed in the middle, the file can be split into three new leaf blocks referenced from a file meta block. The first leaf block hints at the beginning of the file's original block. The last part hints at the end of the file's original block. The middle part is a leaf block without any hints, because it is new, containing the new middle content.

It is possible for any block to be referenced at multiple places. This is useful if two files are equivalent (e.g. copied), because that file need only be saved once. The frontend implementation MAY provide a mechanism to deduplicate files.

If a hint is provided, it MUST be correct. If a hint is provided, it MAY be used. Hints SHOULD be implemented for leaf blocks to reduce the number of downloads of blocks and to allow for efficient partial file modifications. If hints are used to create a block, the correctness of the hints MUST be verified by checking the hash of the final encrypted block against its blockid.

Hints SHOULD be preserved as much as possible when modifying data. Note that a hint contain blockrefs which can in turn contain another hint.

While the above example demonstrates hints for leaf blocks, they MAY also be implemented for file meta blocks to support very large files (e.g. 2 PB), which changes very often with less overhead. They MAY also be supported for directory blocks to optimize small changes in a flat file tree (e.g. 1 million files in a single directory).

Directory Blocks

Directory blocks are a list of blockrefs to the directory's children. Each child MUST be annotated with its block type. Metadata MAY be provided along each blockref, like its name, date of creation (only available on file systems supporting it like NTFS, not e.g. on ext4), last modified, permissions, owner and group. Children can be other directory nodes, leaf nodes or file meta blocks.

Leaf Blocks

Leaf Blocks contain any arbitrary data. They can be used as whole files, or as file parts which are concatenated through file meta blocks.

File Meta Blocks

File meta blocks contain a list of blockrefs. The concatenation of the blocks referenced by blockrefs creates the final file.

Example Merkle Tree

In fig. 1 an example modified merkle tree can be seen. The root references `5891`, which is a directory node containing two blockrefs. One blockref points to the file `ff0b`, which has the name `bar` in that directory. The referenced directory itself contains a file named `qux`, which is the same as `bar` of the parent. Thus it's able to reference the same block. The file `baz` is a file meta block, stating that the file is comprised of the leaf nodes `110f` and `bd9d`. Additionally, `110f` has a hint, stating that it's equal to the subrange from byte 10 to 30 of `21bb`. Thus, if the block `21bb` already exists, the leaf block `110f` can be created without needing to download it.

Bootstrap

The initial state of the blockdb of every client in the very beginning, before any file is setup is the empty block. Everyone spontaneously starts out in the same state, where the blockid of the blockref of the root points to the empty block. Once an actual directory or file is added, the root blockref's blockid will be updated to point to that respective block.

Conflict Resolution

Within the blockdb there are no conflicts because it is CoW. The only conflict that can arise are root updates. If two clients try to update the root at the same time, a conflict resolution strategy MUST be applied.

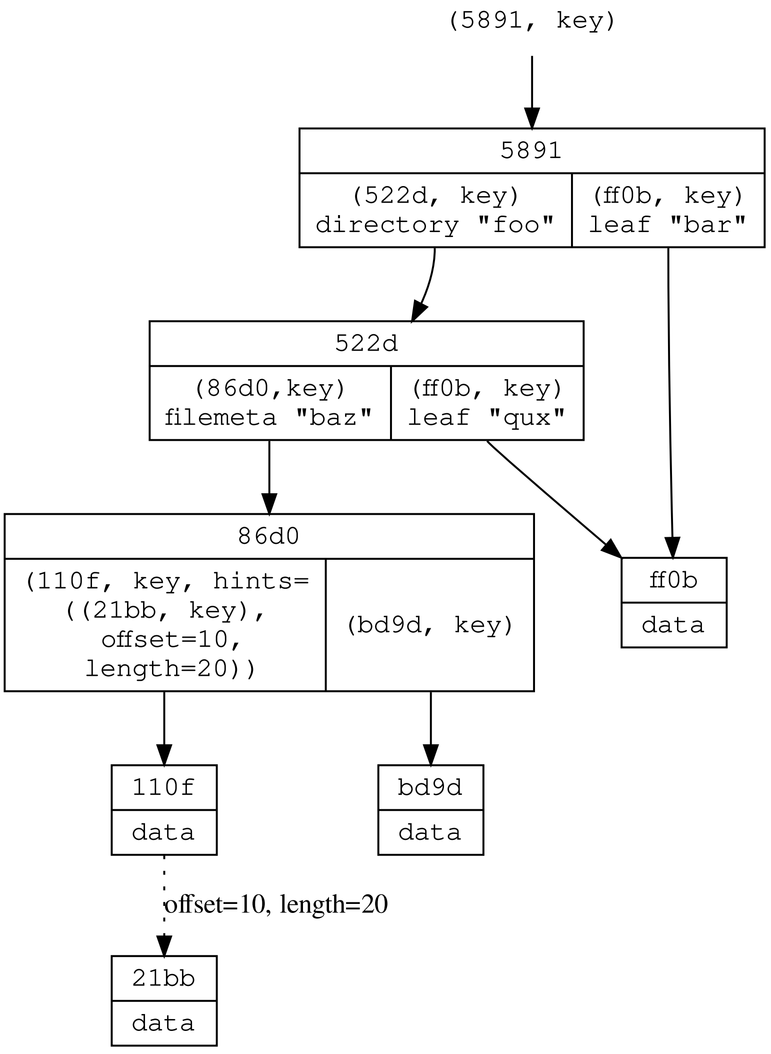


Figure 1: Example Merkle Tree

This specification does not enforce an explicit conflict resolution strategy, because it can be implementation dependent within every single client without breakage of other implementations.

A simple conflict resolution possibility would be to let the longest chain win, in case of a tie use the numerically lower hash of the root to break ties. `scsync`, striving to be a minimal reference implementation of this specification, uses a simple mutex on the server and is thus first come first serve. In case of a conflict, the rejected client MUST fetch the new state and apply its changes to the new root again, then attempt another root update.

This is essentially a classic 3-way-merge problem. As a simple solution we propose that the frontend agregates its changes as incrementally. Once the node attempts to perform this update it either succeeds or fails. If it fails, it will update its blockdb to the most recent state. The frontend MUST then re-apply the recorded changes: it can do an auto-merge on the directory level and compute a union of the new folder and the added elements. If there is a conflict on the file-level, both versions are linked into the directory tree (similar to how dropbox handles this case).

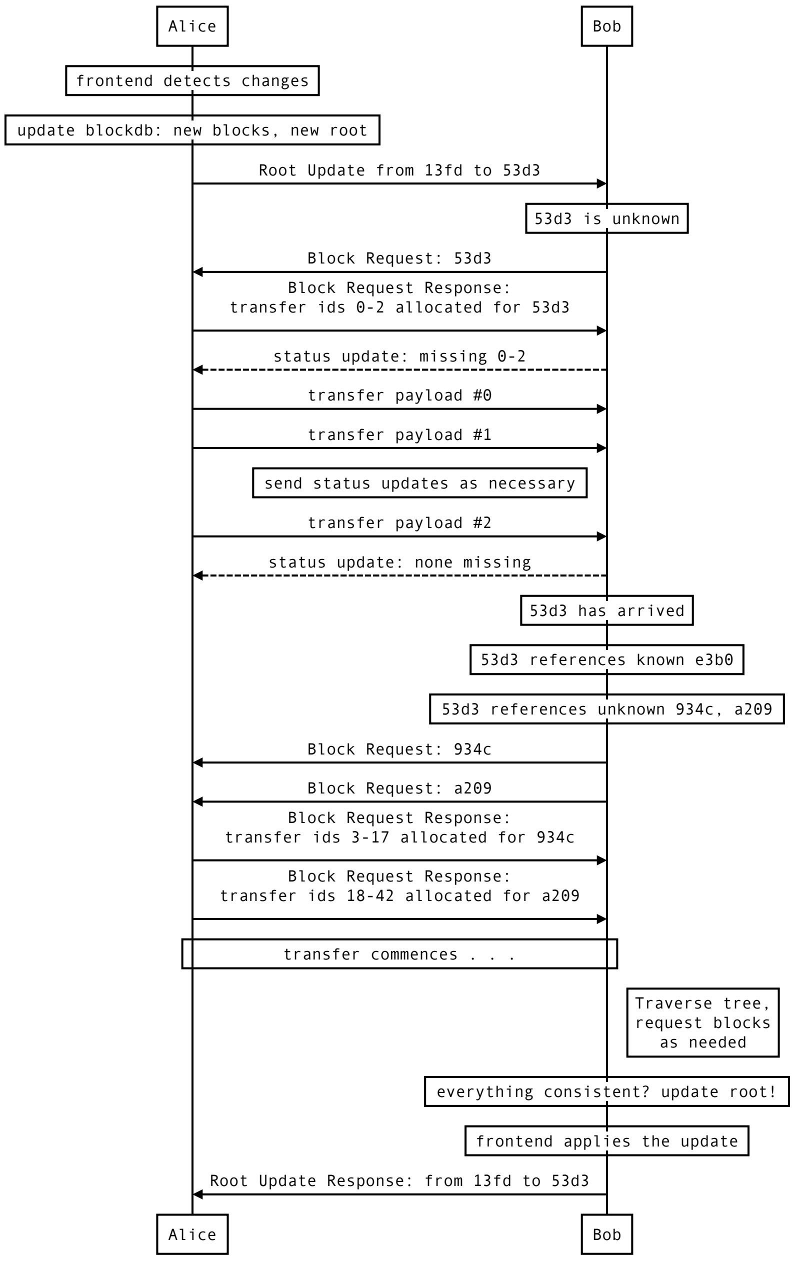
Protocol

Version 2 of The PROTOKOLL uses version 1 for block transfers. Additionally, it adds a control protocol on top.

General Design

In fig. 2 the basic protocol sequence is shown. Bob is chosen to be the server and Alice wants to synchronize changes. It is assumed that Alice and Bob have already established a connection. First, her frontend detects changes and updates the blockdb, creating a new root. A Root Update message is sent to Bob, which states that the root should be updated from `13fd` to `53d3`. Bob's root is at `13fd`, so the update is valid, but `53d3` is unknown. Thus he requests that block from Alice, who answers with a block request response allocating chunkids 0 through 2 to transfer that block. The transfer is initiated with Bob sending his first Status Update to Alice. After transferring that block, Bob requests two new missing blocks at the same time from Alice, who allocates different chunkids for both blocks. Those blocks can be transferred simultaneously, while Bob keeps traversing the new merkle tree, requesting all unknown blocks asynchronously. After having received all unknown blocks, Bob traverses the tree, verifying its integrity and consistency. Following, the root is updated, the frontend notified to update the files and a Root Update Response sent to all clients as push-notification.

While the server does provide push-notifications, it is not guaranteed that they won't be lost; they aren't acknowledged. It is expected by clients to poll regularly to be notified of updates.



Alice
Bob
frontend detects changes
update blockdb: new blocks, new root
Root Update from 13fd to 53d3
53d3 is unknown
Block Request: 53d3
Block Request Response: transfer ids 0-2 allocated for 53d3
status update: missing 0-2
transfer payload #0
transfer payload #1
send status updates as necessary
transfer payload #2
status update: none missing
53d3 has arrived
53d3 references known e3b0
53d3 references unknown 934c, a209
Block Request: 934c
Block Request: a209
Block Request Response: transfer ids 3-17 allocated for 934c
Block Request Response: transfer ids 18-42 allocated for a209
transfer commences . . .
Traverse tree, request blocks as needed
everything consistent? update root!
frontend applies the update
Root Update Response: from 13fd to 53d3
Alice
Bob

Figure 2: Basic Protocol Sequence

Control Protocol

The Control Protocol is used as meta-layer to handle updates of the blockdb. It is mostly stateless, holding only minimal connection information. It has small packets and sends minimal data, without any contents. The actual block transfer, and thus file transfer, is performed with the Transfer Protocol. The Control Protocol can be used at the same time the Transfer Protocol is synchronizing files. Thus, most of the bandwidth is assumed to be occupied by the Transfer Protocol. Control packets are assumed to occur rarely compared to transfer packets, and they are assumed not to be too relevant. As long as a transfer is in progress, losses in the control protocol aren't relevant, because the main purpose of the control protocol is to initiate further transfers. Due to these arguments, the Control Protocol can have a simple structure.

The Control Protocol is similar to a simple version of TCP. Packets are acknowledged by proper responses to the according package. If a control packet hasn't been acknowledged after $2 \cdot RTT$, it is assumed to be lost and the packet MUST be retransmitted.

The control protocol consist of five different packet types.

Root Update

The Root Update is used to indicate a root update from a given block to a new block. It consists of the blockid of the from-block and the blockref of the new block.

The root update is used for multiple purposes. Its main purpose is for a client to notify the server about an update in the file system. A root update is sent containing the from-blockid of the root to-be-updated and the to-blockref pointing to the new root. That message is automatically acknowledged by the next control message from the server, which is either a Block Request, if the merkle tree contains an unknown block, or a Root Update Response, indicating a successful (or unsuccessful, if a different client was faster) update.
Whenever a client wants to connect to the server, it MUST send a root update as first packet, where the from-blockid and the to-blockid are equal and point to the client blockdb's root. The server MUST respond with a root update response, where the from-blockid and the to-blockid are equal to its blockdb's root. Additionally, the client SHOULD update its state to match the server state if they are different. RTT information is only required to retransmit possibly lost control packets in a timely fashion. The client has RTT information from the 2-way-handshake. If the client needs to update its state, it'll respond with a Block Request, from which the Server is able to extract RTT information. The server only needs RTT information if it needs to request blocks and thus send control packet requests. This is only required if the client updates the root to a new state. In that case the server will get RTT information from a Block Request Response from the client after the server sent its Block Request.
As long as no RTT information is available, a sane default for retransmissions should be taken, like a few seconds.

The root update is also used as means to perform state polling. The 2-way-handshake is reperformed, initiated by the client, such that the client detects possible modifications. Clients should poll in timely fashions depending on the use-case.

Root Update Response

The Root Update Response is used by the server to inform clients about a changed root of the merkle tree. Just as the Root Update, the Root Update Response contains the from-blockid and to-blockref. This packet MAY be sent to every client whenever the root changes on the server, to inform clients about possible root changes. This is similar to a push notification, but it is not acknowledged. If a client receives the update, it MAY update its internal state instantaneously to the new root (by fetching unknown blocks). If that packet is lost, that's fine, because clients will fetch at a later point anyway. Thus, the push notification is optional.

Block Request

Block Requests are used by clients to request unknown blocks from the server, or by the server to request unknown blocks from clients. They contain the blockid of the requested block. It MUST be answered with a Block Request Response, which acknowledges the reception of the Block Request.

Block Request Response

Block Request Responses are used as responses to Block Requests. They contain the blockid of the requested block and the transfer protocol's allocated chunkids to transmit that block. It initiates the transfer protocol for that block. They are implicitly acknowledged by the first Status Update within the transfer protocol.

Transfer Protocol

The Transfer Protocol is a slight adaption of the PROTOKOLL version 1 file transfer protocol. First, the protocol is now bidirectional. One channel instance is created in each direction. Instead of a single transfer, there can be multiple, fixed-size, dynamically allocated transfers. Each channel has its own global incrementing ids. Version 1 was able to determine the size of a file transfer, which allowed the use of fixed-size chunkids. In version 2, the number of blocks and thus transfer chunks is unknown. Thus, instead of fixed-length chunkids, varints are used. Additionally, while PROTOKOLL v1 includes the file name / path in its initial packet, the name is already known from directory nodes within the merkle tree. Thus it is not part of the packets of the adapted PROTOKOLL v1 for v2.

After a client connecting to a server, one transfer channel is created in each direction, starting out idle with a random chunkid. Implementations SHOULD use a secure random to generate at least a 32-bit random start chunkid. Due to the random starting chunkid, the bitmap will usually start out with that number of zeroes. Thus another change is that the bitmap in v2 starts counting from zeroes instead of ones. When a Block Request is received, PROTOKOLL v1 is applied and the number of chunks required from the current chunkid is calculated. Based on the chunkid this number can be different for same-length payloads due to the varint encoding of the chunkid. The current chunkid is used as start and the current chunkid plus the number of required chunkids is used as end of the transfer for that blockid. That chunkid range is allocated for that block and a Block Request Response sent to the request origin with the start and end chunkids.

The transfer protocol can be performed independent from the control protocol. If the end of one allocation is reached, the transfer protocol can continue sending the next allocation if one exists without any interruptions. When no further allocation exists, and the whole transfer is acknowledged and the channel is idle again. Any status update causes the transfer protocol to jump back and un-idle.

Connection Management

Connection setup is established by the 2-way-handshake described in Root Update. The handshake only establishes a connection if the nodes determine that a block transfer is required.

The server assumes the connection is assumed to be dead if the client hasn't polled within 2.5 times its polling interval. The client assumes the connection to be dead if the server fails to respond to 3 root update pollings in a row. In that case, the connection is removed and a new connection needs to be established.

There is no graceful connection teardown. Peers close the connection by simply ceasing to communicate.

Protocol State Machine

In fig. 3 the statemachine flow chart of the protocol can be seen.

Encoding

The transfer protocol encoding is performed as described in PROTOKOLL v1, leaving out the filename. Control message data is encoded using Concise Binary Object Representation (CBOR) as described in RFC 7049.

The first octet of each message determines the message type.

Currently defined values:

- transfer payload: 0
- transfer status: 1
- root update: 2
- root update response: 3
- block request: 4
- block request response: 5

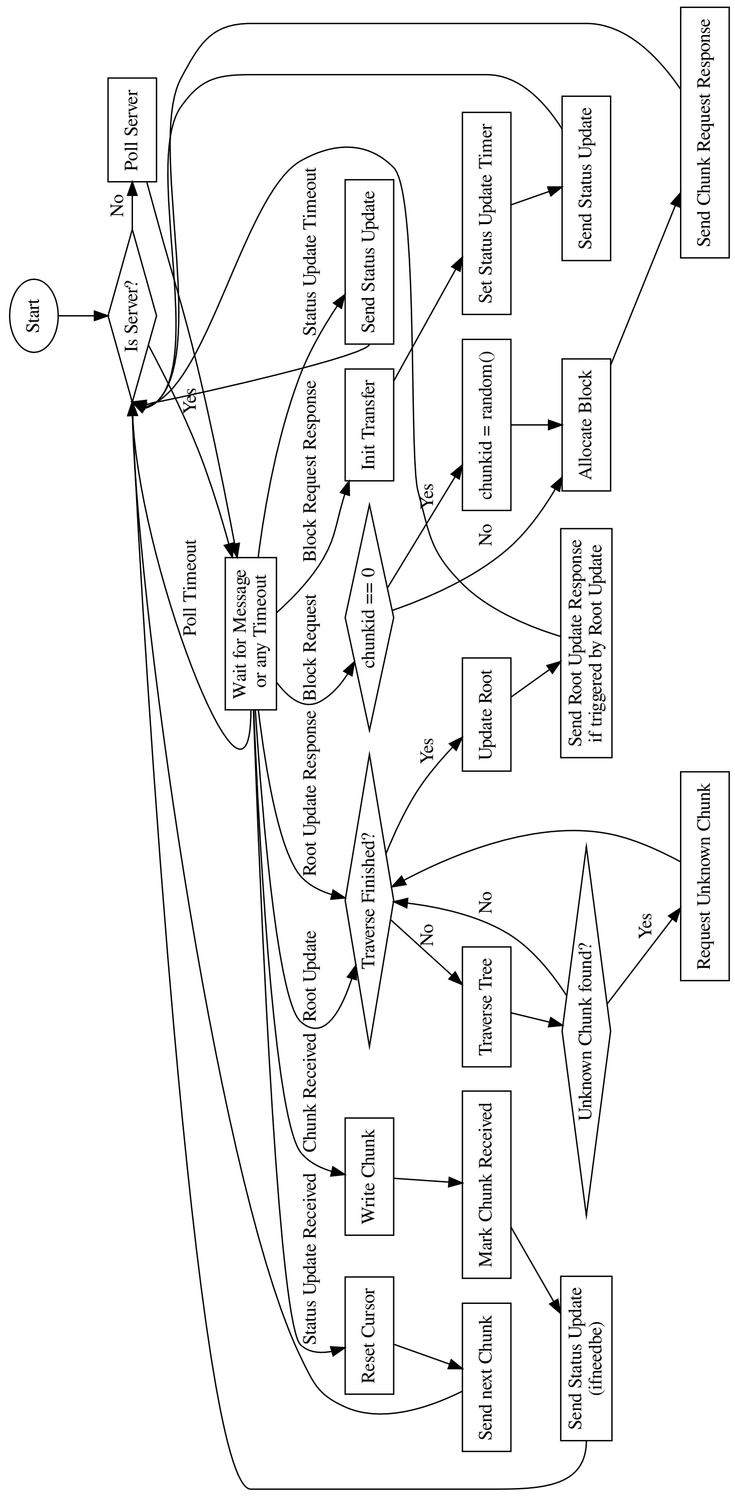


Figure 3: Protocol State Machine

All remaining values are reserved for future extensions.

The structure of the rest of the packet is determined by its type.

The protocol is mainly sent without encryption or signatures, except for the Root Update, which is AEAD'd as described in Cryptography.

Primitives

Blockref

```
blockid: [u8; N], // fixed length
key: [u8; K], // fixed length
hints: List<Blockref>, // list of blockrefs, optional
```

Block Type

The block type is an enum represented as one octet.

```
DIRECTORY = 1,
FILE_META = 2,
LEAF = 3,
```

All remaining values are reserved for future extensions.

Child

```
name: TextString, // name of file without path
metadata: Custom, // may be used in implementations, null otherwise
type: BlockType, // type of child
blockref: Blockref,
```

Root Update

```
nonce: [u8; L], // fixed length
{
  from_blockid: [u8; N], // fixed length
  to_blockref: Blockref,
},
```

Root Update Response

```
from_blockid: [u8; N], // fixed length
to_blockid: [u8; N], // fixed length
to_key: [u8; N], // fixed length
```

Block Request

```
blockid: [u8; N] // fixed length
```

Block Request Response

```
blockid: [u8; N], // fixed length
start_id: varint,
end_id: varint,
```

Leaf Block

```
data... // untagged byte-array
```

File Meta Block

```
parts: List<Blockref>, // list of blockrefs
```

Directory Blocks

```
children: List<Child>,
```

Cryptography

Security (confidentiality and authenticity) is handled by the blockdb. The protocol layer only needs to protect itself from denial of service, replay and spoofing attacks and thus contains barely any cryptography.

Encrypting the actual user data (instead of relying on transport encryption) enables several interesting use cases (protocol extensions - future work), most notably nodes that can store blocks (and provide them to other nodes) without having access to the contents. Concrete examples would be a cloud storage provider that utilizes this to provide end-to-end security for their users or multi-user file synchronization with directory-level security (the key in a blockref could be stored multiple times, each copy encrypted with the public key of one authorized user).

This (first) version assumes a single user with multiple fully trusted devices (one of which is the server). Consequently, a (symmetric) shared secret can be established, either by deriving one from a password (legacy) or ideally by generating random bytes on one device and securely transferring them to the others (e.g. by scanning a QR code).

This shared secret serves as the master key for both encryption and authentication.

BlockDB

The blockdb naturally ensures authenticity because it's content-addressable: blocks are identified by their hash sum from a cryptograhically strong hash function and thus unforgeable - because the hash function is secure, coming up with different data for a given hash sum is infeasible.

To support subslicing of blocks they are encrypted using a stream cipher (e.g. AES-CTR). Their most notable weakness is malleability (modifications to the ciphertext perfectly apply to the plaintext) which is completely countered by our authentication however.

Whenever a block is created, a random key is generated to encrypt it. Blockrefs contain both a blockid and the key so while traversing the directory tree you can always identify, authenticate and decrypt each block.

Hash as key

Implementations MAY offer a configuration option to use a hash of the block's plaintext as encryption key. This option MUST be disabled by default. Doing this offers deduplication through the blockdb (the same file stored twice is only stored and transferred once) but it opens up a cryptographic weakness as it allows eavesdroppers to efficiently confirm if a given file is stored in the shared folder. This is problematic in case file contents are reasonably guessable - for example storing a picture of your credit card PIN is fine whereas storing it in a text file can be trivially broken. Implementations SHOULD instead offer other mechanisms for deduplication.

Transmission Crypto

Because each block is encrypted independently, from a security point of view there is no reason to protect the encrypted block contents. Simply knowing a block id entitles you to its contents, you can just request it and there are no further permission checks. Even correctly guessing a block id is not a problem.

The control protocol manages the transfer of completely insensitive data and thus does not perform any cryptography for most of its tasks (just like TCP).

The only exception to this is the root update message which effectively updates the entire state and serves as the starting point for both the authentication chain as well as the key chain: it contains the root block reference with both a block id and the corresponding key and is thus used to recursively decrypt and traverse the entire filesystem tree. For this reason it is protected using an AEAD cipher (e.g. AES-GCM) with a random initialization vector that is included in the packet. The encryption key is the user's master key. This prevents eavesdroppers from learning the root block's key and it also allows each genuine node to verify the update's legitimacy. The nonce's uniqueness is critical for cryptographic security but this is fine for this use case as a single user can't reasonably produce huge numbers of update transactions.

Threat Model

The attacker is not the user, i.e. does not know the master key (shared secret).

We consider a passive attacker “Eve” that can eavesdrop on all network communications, an attacker “Frank” that can’t modify or even read our communication but inject/spoof arbitrary packets and an active attacker “Mallory” that can read/send/modify network traffic arbitrarily.

Confidentiality is ensured by a chain of encryption keys starting from the secure master key. Eve and Mallory can learn block ids as well as the lengths (arguably a non-negligible side channel) of blocks but never the actual contents. The entire block transfer protocol is deliberately constructed to never deal with sensitive data. The only sensitive message is the root update message which is however protected by an AEAD cipher.

While Mallory can disrupt the block transfer state and thus perform a denial of service attack she can trivially prevent any protocol from succeeding simply by withholding all traffic. The only improvement on this is that by altering the transfer payloads she can additionally cause both participants to waste resources on unsuccessful transfer attempts: block ids always cover the entirety of a block so if any data was changed the receiver can only notice this once the entire block has arrived and is forced to discard all of it (there’s no way to tell what part was modified). One notable special case of this is that Mallory can destroy a long-running (hours) block transfer with negligible effort (modifying a single packet is enough). Note however that this attack can’t be performed by Frank due to the randomization of block ids. The AEAD cipher prevents Mallory from modifying root update messages however and the merkle tree style structure below that precludes any other modifications.

The position of Frank is a classic candidate for both denial of service and distributed denial of service (amplification) attacks. As we discuss above, Frank can not disrupt block transfers because he can’t guess the chunk ids (with non-negligible probability). While Frank is fully capable of forging valid status message (which are completely unprotected) this does not pose a problem. He never able to produce a valid root update message since he doesn’t know the master key, forging a root update response requires the nonce from the request, block requests may allocate a transfer (assuming he somehow learns a valid block id) but the peers can just carry on and ignore this transfer and finally block request responses are ignored unless the peer has specifically requested this block and is still waiting for a response.

Amplification attacks are only possible against the server (clients drop packets from anyone but the server). However, connections are always initiated with a root update message. Even if Frank were to somehow learn an old valid root update message this only gets him a root update response stating that the root update was rejected (which is shorter than the request).

Classic resource exhaustion attacks attack a protocol’s handshake (“SYN flood”). The PROTOKOLL’s root update mechanism is effectively a stateless (even idempotent) handshake that only establishes a connection if the update is valid. While servers MAY remember clients regardless to push update notifications they MUST implement an upper limit on this memory. Both Mallory and Frank could cause the sender to allocate additional block transfers during an existing transfer. However the amount of resources consumed this way is limited to the number of valid blockids they know (a single transfer will never contain the same block twice).

The root update message is idempotent and thus secure to replay attacks.

Future Work

The encryption scheme is designed with extensibility in mind. While the version specified here only supports a single master key future versions could improve on this with elaborate systems like public key infrastructure and fine grained directory/file-level permissions.

The protocol is designed for peer to peer use cases. While this specification describes a server-client architecture it can be adapted to a peer to peer protocol with few changes.

While the current specification requires a peer to recursively request all blocks in the current filesystem state this is by no means an inevitability. A protocol extension could be designed that stores blocks on a network of nodes using some load balancing heuristic (e.g. the block hash). You could also have "thin clients" that fetch blocks as they are accessed, keeping only a local cache of the most commonly accessed blocks (or maybe not even that).

On "long fat pipe" networks the current specification performs badly when transferring very deep and sparse folder hierarchies as every level requires roundtrips to set up and then perform the new block transfer. The design attempts to amortize this by transferring other blocks during those roundtrips but an actual solution requires "block inlining", i.e. embedding very small blocks inside parent blocks to avoid the block transfer overhead and ensure that the transfer layer is never idle.

Since computation has become significantly faster than disk I/O in recent years modern filesystems commonly offer filesystem-level compression as a way to both improve throughput and save space. Real-world users report average compression ratios as high as 2x (as well as a proportional increase in throughput). Because the PROTOKOLL is essentially a distributed file system, block-level compression would be a useful feature - especially considering that this also improves transfer speeds.