Metarhia Protocol Documentation

Release

Metarhia

Jan 29, 2019

Contents:

Metarhia Protocol (`mhp`) is a protocol for RPC, event streams and two-way asynchronous data transfer that supports multiplexing and is capable of handling network errors gracefully.

Protocol Specification

1.1 Introduction

Metarhia Protocol is an RPC, session and binary data transfer protocol that provides two-way asynchronous data transfer, multiplexing applications, channels, event and binary streams over one socket, and graceful handling of short-time connection losses due to network errors with full and transparent session restoration. It also provides authentication mechanisms, offers data compression and supports multiple serialization formats with each of those being more appropriate or efficient for different kinds of data.

1.2 Terms

Important: 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.

Transport refers to a network protocol or another communication mechanism that provides full-duplex communication channel with ordered and reliable data flow, to which Metarhia Protocol delegates transmission of raw data streams.

Transport connection refers to an underlying transport socket, connection or other transport-specific communication channel, over which a Metarhia Protocol connection transmits data.

Metarhia Protocol connection (or just **connection** without additional adjectives) refers to an abstraction over an instance of transport connection managed by a Metarhia Protocol implementation, that hides implementation details of the transport and provides the functionality of Metarhia Protocol to user applications. Metarhia Protocol connections have one-to-one correspondence to their primary transport connections used for RPC, but may open additional ones internally for specific purposes.

Chunk refers to a data unit of Metarhia Protocol consisting of headers and optional payload. All data transmitted via Metarhia Protocol are split into chunks.

Channel refers to a set of chunks transmitted over the same connection, identified among other ones using a number that is unique throughout the currently active channels connection-wise. Channels provide multiplexing capabilities for connections.

Message refers to a channel that is characterized by a short lifetime and small number of chunks (only one chunk in most cases, with the maximal amount specified in section ???) that are buffered in memory until the message is received, and then processed as a single unit by the protocol.

Stream refers to a channel that is characterized by arbitrarily long lifetime (up to existing as long as the connection exists) and indefinite number of chunks, which may be processed by an application one by one immediately after they become available.

Session is a persistent association between applications on both sides of a connection. Sessions may be anonymous and authenticated. Channels are bound to corresponding sessions.

Tunnel refers to the set of sessions and channels that belong to a connection. On network failure, the tunnel can be restored transparently for an application.

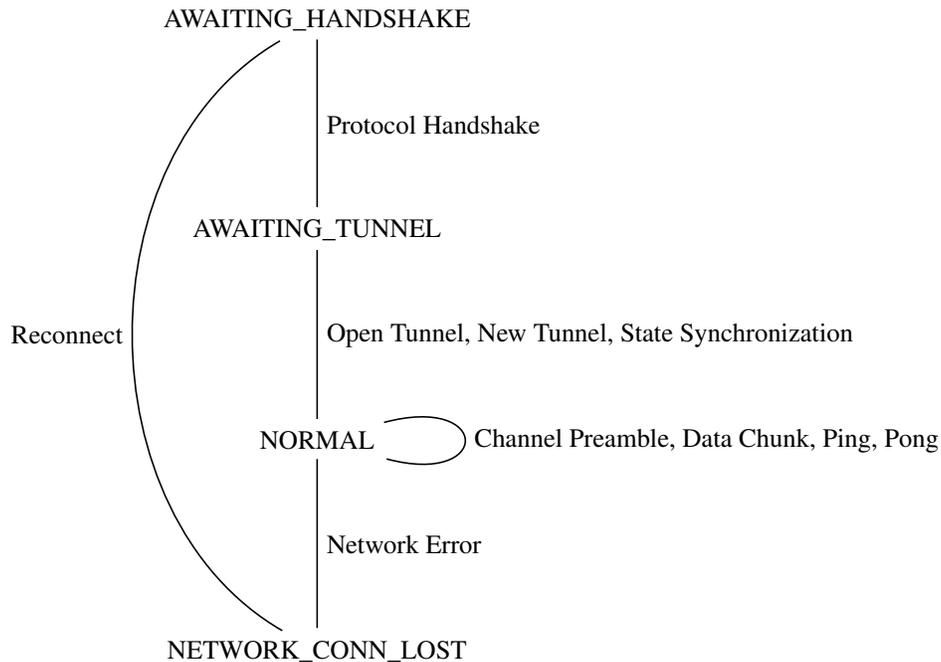**RPC** is an acronym for remote procedure calls.

AEAD is an acronym for Authenticated Encryption with Associated Data.

1.3 Transport Support

Metarhia Protocol implementations primarily targeted to be used in server-side environments MUST support TCP, TLS, WebSocket and WebSocket tunneled over TLS protocols as transports. Metarhia Protocol implementations primarily targeted to be used in server-side environments MAY support additional transports, for example, Unix domain sockets.

Metarhia Protocol implementations designed specifically to be used in client-side environments SHOULD support TCP and TLS transports. In those cases where it is not possible (for example, in implementations for Web browsers), such implementations MUST support WebSocket and WebSocket tunneled over TLS as transports, and MAY support them otherwise. In other words, at least one of "TCP and TLS" or "WebSocket and WebSocket over TLS" pairs of transports MUST be supported, with preference towards TCP and TLS. Client-side implementations MAY support other transports, if their implementors find it reasonable.

1.4 Connection States

```
              AWAITING_HANDSHAKE


                    │     Protocol Handshake
                    │

              AWAITING_TUNNEL

                    │
                    │     Open Tunnel, New Tunnel, State Synchronization
   Reconnect        │

                 NORMAL  ⟳  Channel Preamble, Data Chunk, Ping, Pong

                    │
                    │     Network Error
                    │

              NETWORK_CONN_LOST
```

1.4.1 `AWAITING_HANDSHAKE`

A transport connection has opened, but handshake hasn't been performed and there is no session established.

A client sends a *Protocol Handshake* chunk. When the handshake is performed successfully, the connection transitions into *AWAITING_TUNNEL* state.

A server waits a *Protocol Handshake* chunk from the client. When the handshake is performed successfully, the connection transitions into *AWAITING_TUNNEL* state.

1.4.2 `AWAITING_TUNNEL`

The client sends an *Open Tunnel* chunk. The server responds either with a *New Tunnel* chunk, or, in the case when an existing session is being restored, with a *State Synchronization* chunk, to which the client responds with a *State Synchronization* chunk too, and both sides re-send all the chunks they did not receive. After that, both connections transition into *NORMAL* state.

1.4.3 `NORMAL`

This is the main mode of operation. All the communication is performed using channels and ping/pong chunks. On network error, the connection transitions into *NETWORK_CONN_LOST* state.

1.4.4 `NETWORK_CONN_LOST`

The client buffers all outgoing chunks and tries to reconnect to the server. On success, the connection transitions into *AWAITING_HANDSHAKE* state.

The server buffers all outgoing chunks and awaits a new connection from the client.

1.5 Chunk Types

Each chunk transmitted in *NORMAL* connection state starts with a 1-octet field indicating the chunk type. This value MUST be equal to one of the following:

Name	Value
PING	0
PONG	1
MESSAGE_PREAMBLE	2
STREAM_PREAMBLE	3
DATA_CHUNK	4

1.6 Chunk Formats

Note: Metarhia Protocol uses little-endian byte order.

1.6.1 Protocol Handshake

Field	Bits
Version	16
Encryption	16
Payload	

Version field indicates the version of the protocol to use. This document describes Metarhia Protocol version 1.

Currently, the only possible value of Encryption is 0 and the payload is empty.

When new possible values of Encryption are added, they may require adding new handshake chunks to implement, e.g., key exchange. When Encryption is 0, no additional data is required for the protocol handshake, and Metarhia Protocol sessions may be opened or restored over the connection immediately.

1.6.2 Open Tunnel

Field	Bits
Token	256

Token is a 32-byte tunnel ID and tunnel secret key. 0 is a special value reserved to indicate that a new tunnel must be created, instead of restoring an existing one.

1.6.3 New Tunnel

Field	Bits
Token	256

`Token` is a 32-byte random string, obtained from a cryptographically secure source. It serves both as a tunnel ID and a tunnel secret key. `Token` must not be equal to `0`.

1.6.4 State Synchronization

Field	Bits
LastPingId	32
ChunksCount	32

`LastPingId` is an ID of the last ping chunk that a sending side has received, and `ChunksCount` is the number of chunks the side has received since then.

1.6.5 Ping

Field	Bits
ChunkType	8
PingId	32

`ChunkType` of Ping chunks is `PING` (see *Chunk Types*).

1.6.6 Pong

Field	Bits
ChunkType	8
PingId	32

`ChunkType` of Pong chunks is `PONG` (see *Chunk Types*).

1.6.7 Channel Preamble

This is an abstract channel preamble, that is, in practice, represented by *Message Preamble* and Stream Preamble. `Id` and `Compression` are generic channel preamble fields, pertaining to both of them. Stream Preamble doesn't have any additional fields, so this structure effectively describes it. *Message Preamble*, however, has additional fields that occupy the place of `MessagePreambleReserved` in the following table.

`ChunkType` of *Message Preamble* equals to `MESSAGE_PREAMBLE`, and `ChunkType` of Stream Preamble equals to `STREAM_PREAMBLE` (see *Chunk Types*).

Field	Bits
ChunkType	8
Id	32
Compression	8
MessagePreambleReserved	16
SessionId	64

`Id` field is an identifier of the channel in the connection. To avoid collisions because of unsynchronized channel counters on the sides of a connection, the most significant bit of the `Id` field is masked to be always equal to 0 for channels initiated by clients and 1 for channels initiated by servers by making the field an signed integer, two's complement. In other words, the valid values of the `Id` field of client-initiated channels are $[0, 2^{31} - 1]$ and the valid values of the `Id` field of server-initiated channels are $[-2^{31}, -1]$. The `Id` value MUST be unique throughout the currently active channels.

The `Compression` field indicates if the payload of subsequent *data chunks* in this channel is compressed. The field MUST be set to one of the following values:

Compression	
0	No compression
1	Gzip compression

`SessionId` is an identifier of the session to open a channel in. It is obtained during application handshake as a part of `HandshakeResponse`.

If `ChunkType` is `MESSAGE_PREAMBLE`, then the chunk is a *Message Preamble*.

1.6.8 Message Preamble

See *Channel Preamble*.

Field	Bits
ChunkType	8
Id	32
Compression	8
Encoding	8
MessageType	8
SessionId	64

This chunk type extends the generic *Channel Preamble*, adding two new fields instead of the `MessagePreambleReserved` field, namely, `Encoding` and `MessageType`.

The `Encoding` field specifies the format used to encode the payload fields of messages that require passing arbitrary data (e.g., arguments of RPC methods in `Call` messages). It MUST be set to one of the following values:

Encoding	
0	JSTP
1	JSON

The `MessageType` field MUST be set to one of the following values:

MessageType	
0	`HandshakeRequest`
1	`HandshakeResponse`
2	`Event`
3	`Call`
4	`Callback`
5	`Inspect`
6	`InspectCallback`

1.6.9 Data Chunk

Field	Bits
`ChunkType`	8
`ChannelId`	32
`Length`	16
`Flags`	8
`Payload`	

`ChunkType` of a data chunk is `DATA_CHUNK` (see *Chunk Types*).

The `ChannelId` field specifies a channel the chunk belongs to. The `Length` field contains the size of the payload in bytes. The `Flags` field has the following structure:

Flags	
Bits 7–1	Reserved
Bit 0	`More`

Flag `More` specifies if the channel has more chunks. Reserved flags MUST be set to `0`.

1.7 "Fast" UDP Events Encryption (ignore this for now)

Note: I drafted this section while I was in context so as to not forget about all I thought about this; the things this would be needed for aren't quite there yet ;)

1.7.1 AEAD Algorithm Requirements and Motivation

For chunks that use symmetric encryption (for example, "fast" UDP events), AEAD based on the ChaCha20 stream cipher and Poly1305 message authentication code algorithm with modifications from IETF (RFC 7539) MUST be used.

ChaCha20 and Poly1305 are modern, secure, high-speed algorithms developed by Daniel J. Berstein, that have undergone scrupulous analysis in multiple scientific papers and are under constantly growing adoption now. As some examples:

- Google has used their implementation of these algorithms for TLS traffic between Google Chrome on Android and Google's servers since 2014.

- TLS 1.3 draft has `TLS_CHACHA20_POLY1305_SHA256` cipher suite, and recommends implementing it.

IETF versions of ChaCha20, ChaCha20-Poly1305 and ChaCha20-Poly1305 AEAD specified in RFC 7539 modify Berstein's algorithm by changing 64-bit nonce to 96-bit nonce, so 64-bit block counter is reduced 32-bit block counter, effectively limiting the size of a message to 256 GB (instead of 2^{64} bytes).

Poly1305 is proved to be secure using the same key for at least 2^{64} messages, provided that nonces are never reused.

1.7.2 Symmetric Encryption Implementation

Upon creation of a `Tunnel` structure instance, the following fields relevant to the symmetric encryption facilities (with one of them not being limited to this scope only) are initialized:

- `secret` — a 32-byte unsigned integer value
- `nonce` — a 12-byte unsigned integer value

`nonce` value MUST be initialized with random data from a cryptographically secure source.

If the `Tunnel` structure is created on the side of a client, the least significant bit of `nonce` MUST be set to 0. If the `Tunnel` structure is created on the side of a server, the least significant bit of `nonce` MUST be set to 1.

If the `Tunnel` structure is created on the side of a server, `secret` value MUST be initialized with random data from a cryptographically secure source.

The server shares this value with the client during the handshake, as described in section ???. When the client receives this value, it MUST initialize the `secret` field of its `Tunnel` structure with the received value.

Danger: This procedure MAY be conducted over a connection that is not secured using TLS or other method of asymmetric encryption and server authentication in a local or trusted environment, or on a single machine during testing, but one SHOULD NOT do so over a publicly accessible network. Security may be compromised in such case. Only connections secured with TLS (or an alternative method) SHOULD be used with Metarhia Protocol in public networks.

When symmetric encryption of a chunk is requested, Metarhia Protocol implementations MUST follow the next algorithm:

1. **Let** *secret* := **Get** *secret* from *Tunnel*.
2. **Let** *nonce* := **Get** *nonce* from *Tunnel*.
3. **Let** *data* := **Input**.
4. **Let** *result* := AEAD_ChaCha20_Poly1305_IETF_Encrypt(*data*, *secret*, *nonce*).
5. **Set** *nonce* in *Tunnel* := *nonce* + 2.
6. **Output** := *result*.

When symmetric decryption of a chunk is requested, Metarhia Protocol implementations MUST follow the next algorithm:

1. **Let** *secret* := **Get** *secret* from *Tunnel*.
2. **Let** *data* := **Input**.
3. **Let** *result* := AEAD_ChaCha20_Poly1305_IETF_Decrypt(*data*, *secret*).
4. **Output** := *result*.