Mumble Protocol

Release 1.2.5-alpha

May 10, 2014

Contents:

CHAPTER 1

Introduction

This document is meant to be a reference for the Mumble VoIP 1.2.X server-client communication protocol. It reflects the state of the protocol implemented in the Mumble 1.2.2 client and might be outdated by the time you are reading this. Be sure to check for newer revisions of this document on our website url{http://www.mumble.info}. At the moment this document is work in progress.

CHAPTER 2

Overview

Mumble is based on a standard server-client communication model. It utilizes two channels of communication, the first one is a TCP connection which is used to reliably transfer control data between the client and the server. The second one is a UDP connection which is used for unreliable, low latency transfer of voice data.

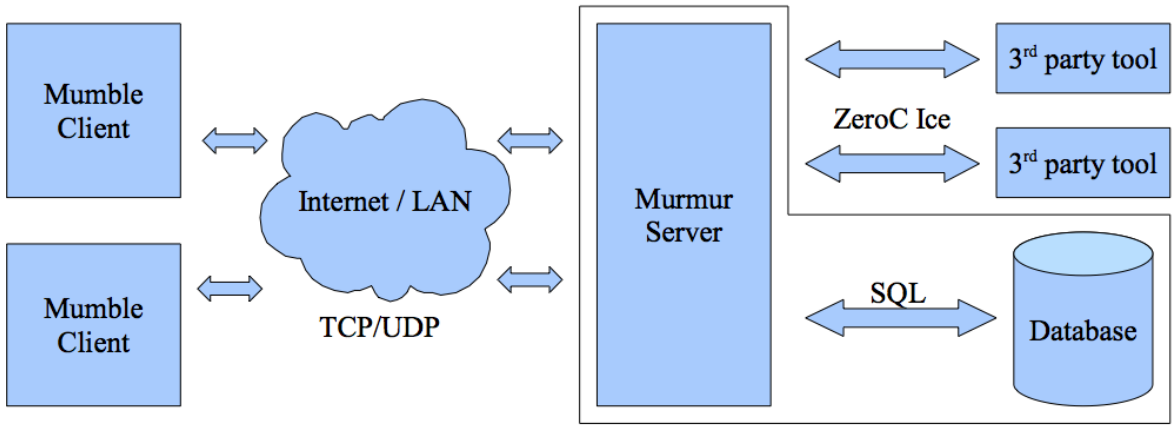

Figure 2.1: Mumble system overview

Both are protected by strong cryptography, this encryption is mandatory and cannot be disabled. The TCP control channel uses TLSv1 AES256-SHA [1] while the voice channel is encrypted with OCB-AES128 [2].

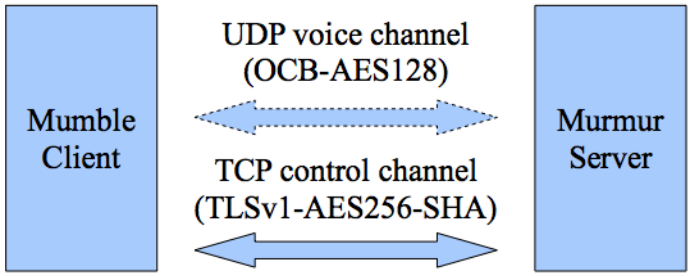

Figure 2.2: Mumble crypto types

[1] http://en.wikipedia.org/wiki/Transport_Layer_Security
[2] http://www.cs.ucdavis.edu/~rogaway/ocb/ocb-back.htm

While the TCP connection is mandatory the UDP connection can be compensated by tunnelling the UDP packets through the TCP connection as described in the protocol description later.

CHAPTER 3

Protocol stack (TCP)

Mumble has a shallow and easy to understand stack. Basically it uses Google's Protocol Buffers [1] with simple prefixing to distinguish the different kinds of packets sent through an TLSv1 encrypted connection. This makes the protocol very easily expandable.

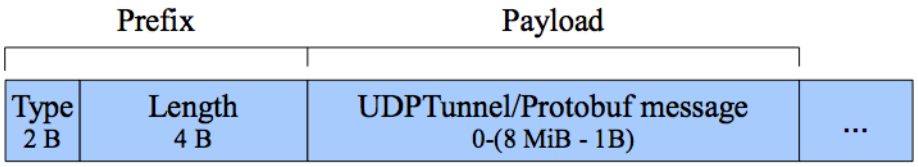

Figure 3.1: Mumble packet

The prefix consists out of the two bytes defining the type of the packet in the payload and 4 bytes stating the length of the payload in bytes followed by the payload itself. The following packet types are available in the current protocol and all but UDPTunnel are simple protobuf messages. If not mentioned otherwise all fields outside the protobuf encoding are big-endian.

[1] http://code.google.com/p/protobuf

Table 3.1: Packet types

Type	Payload
0	Version
1	UDPTunnel
2	Authenticate
3	Ping
4	Reject
5	ServerSync
6	ChannelRemove
7	ChannelState
8	UserRemove
9	UserState
10	BanList
11	TextMessage
12	PermissionDenied
13	ACL
14	QueryUsers
15	CryptSetup
16	ContextActionModify
17	ContextAction
18	UserList
19	VoiceTarget
20	PermissionQuery
21	CodecVersion
22	UserStats
23	RequestBlob
24	ServerConfig
25	SuggestConfig

For raw representation of each packet type see the attached Mumble.proto [2] file.

[2] https://raw.github.com/mumble-voip/mumble/master/src/Mumble.proto

CHAPTER 4

Establishing a connection

This section describes the communication between the server and the client during connection establishing, note that only the TCP connection needs to be established for the client to be connected. After this the client will be visible to the other clients on the server and able to send other types of messages.

4.1 Connect

As the basis for the synchronization procedure the client has to first establish the TCP connection to the server and do a common TLSv1 handshake. To be able to use the complete feature set of the Mumble protocol it is recommended that the client provides a strong certificate to the server. This however is not mandatory as you can connect to the server without providing a certificate. However the server must provide the client with its certificate and it is recommended that the client checks this.

4.2 Version exchange

Once the TLS handshake is completed both sides should transmit their version information using the Version message. The message structure is described below.

Table 4.1: Version message

Version	
version	uint32
release	string
os	string
os_version	string

The version field is a combination of major, minor and patch version numbers (e.g. 1.2.0) so that major number takes two bytes and minor and patch numbers take one byte each. The structure is shown in figure ref{fig:versionEncoding}. The release, os and os_version fields are common strings containing additional information.

Table 4.2: Version field encoding (uint32)

Major	Minor	Patch
2 bytes	1 byte	1 byte

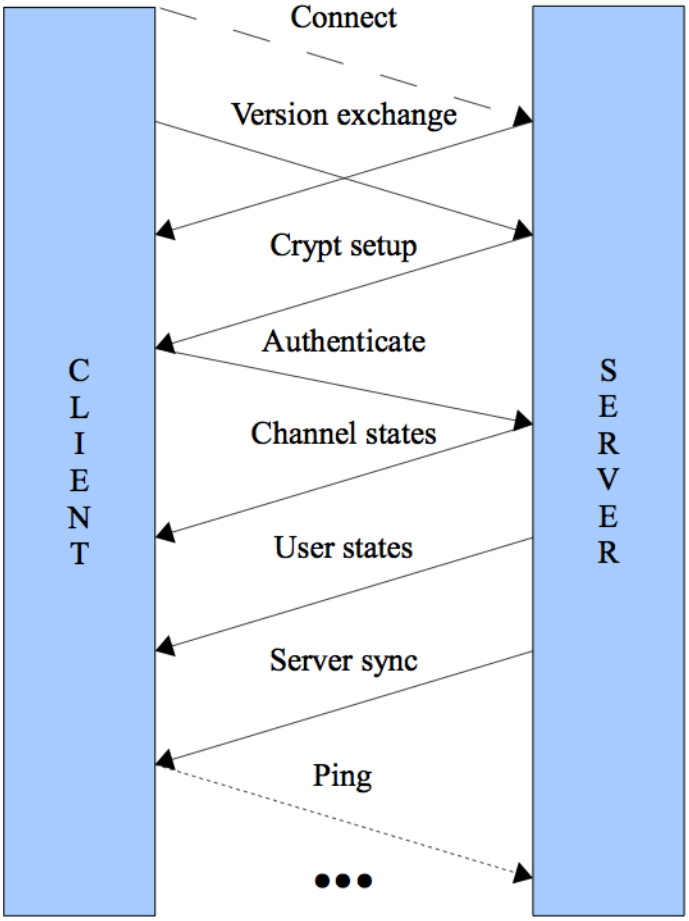

Figure 4.1: Mumble connection setup

The version information may be used as part of the *SuggestConfig* checks, which usually refer to the standard client versions. The major changes between these versions are listed in table below. The *release*, *os* and *os_version* information is not interpreted in any way at the moment.

Table 4.3: Mumble version differences

Version	Major changes
1.2.0	CELT 0.7.0 codec support
1.2.2	CELT 0.7.1 codec support
1.2.3	CELT 0.11.0 codec
1.2.4	Opus codec support, SuggestConfig message

4.3 Authenticate

Once the client has sent the version it should follow this with the Authenticate message. The message structure is described in the figure below. This message may be sent immediately after sending the version message. The client does not need to wait for the server version message.

Table 4.4: Authenticate message

Authenticate	
username	string
password	string
tokens	string

The username and password are UTF-8 encoded strings. While the client is free to accept any username from the user the server is allowed to impose further restrictions. Furthermore if the client certificate has been registered with the server the client is primarily known with the username they had when the certificate was registered. For more information see the server documentation.

The password must only be provided if the server is passworded, the client provided no certificate but wants to authenticate to an account which has a password set, or to access the SuperUser account.

The third field contains a list of zero or more token strings which act as passwords that may give the client access to certain ACL groups without actually being a registered member in them, again see the server documentation for more information.

4.4 Crypto setup

Once the Version packets are exchanged the server will send a CryptSetup packet to the client. It contains the necessary cryptographic information for the OCB-AES128 encryption used in the UDP Voice channel. The packet is described in figure below. The encryption itself is described in a later section.

Table 4.5: CryptSetup message

CryptSetup	
key	bytes
client_nonce	bytes
server_nonce	bytes

4.5 Channel states

After the client has successfully authenticated the server starts listing the channels by transmitting partial ChannelState message for every channel on this server. These messages lack the channel link information as the client does not yet have full picture of all the channels. Once the initial ChannelState has been transmitted for all channels the server updates the linked channels by sending new packets for these. The full structure of these ChanneLState messages is shown below:

Table 4.6: ChannelState message

ChannelState	
channel_id	uint32
parent	uint32
name	string
links	repeated uint32
description	string
links_add	repeated uint32
links_remove	repeated uint32
temporary	optional bool
position	optional int32

The server must send a ChannelState for the root channel identified with ID 0.

4.6 User states

After the channels have been synchronized the server continues by listing the connected users. This is done by sending a UserState message for each user currently on the server, including the user that is currently connecting.

Table 4.7: UserState message

UserState	
session	uint32
actor	uint32
name	string
user_id	uint32
channel_id	uint32
mute	bool
deaf	bool
suppress	bool
self_mute	bool
self_deaf	bool
texture	bytes
plugin_context	bytes
plugin_identity	string
comment	string
hash	string
comment_hash	bytes
texture_hash	bytes
priority_speaker	bool
recording	bool

4.7 Server sync

The client has now received a copy of the parts of the server state he needs to know about. To complete the synchronization the server transmits a ServerSync message containing the session id of the clients session, the maximum bandwidth allowed on this server, the servers welcome text as well as the permissions the client has in the channel he ended up.

For more information pease refer to the Mumble.proto file [1].

4.8 Ping

If the client wishes to maintain the connection to the server it is required to ping the server. If the server does not receive a ping for 30 seconds it will disconnect the client.

[1] https://raw.github.com/mumble-voip/mumble/master/src/Mumble.proto

CHAPTER 5

Voice data

5.1 Enabling the UDP channel

Before the UDP channel can reliably be used both sides should be certain that the connection works. Before the server may use the UDP connection to the client the client must first open a UDP socket and communicate its address to the server by sending a packet over UDP. Once the server has received an UDP transmission the server should start using the UDP channel for the voice packets. Respectively he client should not use the UDP channel for voice data until it is certain that the packets go through to the server.

In practice these requirements are filled with UDP ping. When the server receives a UDP ping packet from the client it echoes the packet back. When the client receives this packet it can ascertain that the UDP channel works for two-way communication.

Table 5.1: UDP ping packet

UDP ping packet	
byte	type/flags (0010 0000 for Ping)
varint	timestamp

If the client stops receiving replies to the UDP packets at some point or never receives the first one it should immediately start tunneling the voice communication through TCP as described in the *UDP tunnel* section. When the server receives a tunneled packet over the TCP connection it must also stop using the UDP for communication. The client may continue sending UDP ping packets over the UDP channel and the server must echo these if it receives them. If the client later receives these echoes it may switch back to the UDP channel for voice communication. When the server receives an UDP voice communication packet from the client it should stop tunneling the packets as well.

5.2 Data

The voice data is transmitted in variable length packets that consist of header portion, followed by repeated data segments and an optional position part. The full packet structure is shown in the figure below, and consists of three parts. The decrypted data should never be longer than 1020 bytes, this allows the use of 1024 byte UDP buffer even after the 4-byte encryption header is added to the packet during the encryption. The protocol transfers 64-bit integers using variable length encoding. This encoding is specified in the *varint* section.

A voice packet starts with a header:

Voice packet header		
Type	Field	Description
byte	Type/Flags	Bit-13: Type, Bit 4-8: Target
varint	Session	The session number of the source user (only from server)
varint	Sequence	

Followed by one or more audio data segments:

Voice packet audio data		
Type	Field	Description
byte	Header	Bit 1: Terminator, Bit 2-8: Data length
byte[]	Data	Encoded voice frames

Followed by an optional set of positional audio coordinates:

Voice packet positional audio data		
Type	Field	Description
float	Position 1	
float	Position 2	
float	Position 3	

The first byte of the header contains the packet type and additional target specifier. The format of this byte is described below. If the voice packet comes from the server, the type is followed by a *varint* encoded value that specifies the session this voice packet originated from – this information is added by the server and the client omits this field. The last segment in the header is a sequence number for the first audio frame of the packet. If there are for example two frames in the packet, the sequence field of the next packet should be incremented by two.

The type is stored in the first three bits and specifies the type and encoding of the packet. Current types are listed in *UDP Types* table. The remaining 5 bits specify additional packet-wide options. For voice packets the values specify the voice target as listed in the table below:

Table 5.2: UDP Types

Type	Description
0	CELT Alpha encoded voice data
1	Ping packet
2	Speex encoded voice data
3	CELT Beta encoded voice data
4-7	Unused

Table 5.3: UDP targets

Target	Description
0	Normal talking
1	Whisper to channel
2-30	Direct whisper (always 2 for incoming whisper)
31	Server loopback

The audio frames consist of one byte long header and up to 127 bytes long data portion. The first bit in the header is the *terminator bit* which informs the receiver whether there are more audio frames after this one. This bit is turned on (value *1*) for all but the last frame in the current UDP packet. Rest of the seven bits in the header specify the length of the data portion. The data portion is encoded using one of the supported codecs. The exact codec is specified in the type portion of the whole packet (See the UDP types table). *The data in each frame is encoded separately.*

5.3 Codecs

Mumble supports two distinct codecs; Low bit rate audio uses Speex and higher quality audio is encoded with CELT. Both of these codecs must be supported for full support of the Mumble protocol. Furthermore, as the CELT bitstream has not been frozen yet which places requirements for the exact CELT version: The clients must support CELT 0.7.1 bitstream. The protocol includes codec negotiation which allows clients to support other codec versions as well, in which case the server should attempt to negotiate a version that all clients support. The clients must respect the server resolution.

5.4 Whispering

Normal talking can be heard by the users of the current channel and all linked channels as long as the speaker has Talk permission on these channels. If the speaker wishes to broadcast the voice to specific users or channels, he may use whispering. This is achieved by registering a voice target using the VoiceTarget message and specifying the target ID as the target in the first byte of the UDP packet.

5.5 Varint and 64-bit integer encoding

The variable length integer encoding is used to encode long, 64-bit, integers so that short values do not need the full 8 bytes to be transferred. The basic idea behind the encoding is prefixing the value with a length prefix and then removing the leading zeroes from the value. The positive numbers are always right justified. That is to say that the least significant bit in the encoded presentation matches the least significant bit in the decoded presentation. The *varint prefixes* table contains the definitions of the different length prefixes. The encoded **x** bits are part of the decoded number while the _ signifies a unused bit. Encoding should be done by searching the first decoded description that fits the number that should be decoded, truncating it to the required bytes and combining it with the defined encoding prefix.

See the *quint64* shift operators in https://github.com/mumble-voip/mumble/blob/master/src/PacketDataStream.h for a reference implementation.

Table 5.4: Varint prefixes

Encoded	Decoded
0xxxxxxx	1 byte with $7 \cdot 8 + 1$ leading zeroes
10xxxxxx + 1 byte	2 bytes with $6 \cdot 8 + 2$ leading zeroes
110xxxxx + 2 bytes	3 bytes with $5 \cdot 8 + 3$ leading zeroes
1110xxxx + 3 bytes	4 bytes with $4 \cdot 8 + 4$ leading zeroes
111100__ + **int** (4 bytes)	32-bit positive number
111101__ + **long** (8 bytes)	64-bit number
111110__ + **varint**	Negative varint
111111xx	Byte-inverted negative two byte number (~xx)

The variable length integer encoding is used to encode long (64-bit) integers so that short values do not need the full 8 bytes to be transferred. The encoding function is given below. While it might seem complex it is worth noting that the $(a_v, a_p)(b_v, b_p)$ function equals appending the a_p bits long value a_v to a byte stream that already has the b_p bits long value b_v.

5.6 TCP tunnel

If the UDP channel isn't available the voice packets must be transmitted through the TCP socket. These messages use the normal TCP prefixing, as seen in shown in figure *Mumble packet*: 16-bit message type followed by 32-bit message length. However unlike other TCP messages, the UDP packets are not encoded as protocol buffer messages but instead the raw UDP packet described in section *Data* should be written to the TCP socket directly.

When the packets are received it is safe to parse the type and length fields normally. If the type matches that of the UDP tunnel the rest of the message should be processed as an UDP packet without attempting a protocol buffer decoding.

5.7 Encryption

All the packets are encrypted once during transfer. The actual encryption depends on the used transport layer. If the packets are tunneled through TCP they are encrypted using the TLS that encrypts the whole TCP connection and if they are sent directly using UDP they must be encrypted using the OCB-AES128 encryption.

5.8 Implementation notes

When implementing the protocol it is easier to ignore the UDP transfer layer at first and just tunnel the UDP data through the TCP tunnel. The TCP layer must be implemented for authentication in any case. Making sure that the voice transmission works before implementing the UDP protocol simplifies debugging greatly. The UDP protocol is a required part of the specification though.

CHAPTER 6

Indices and tables

- *genindex*
- *search*