Mumble protocol 1.2.X reference (WIP)

Stefan Hacker, Mikko Rantanen

October 18, 2010

DISCLAIMER

Contents

1 Introduction 4

2 Overview 4

3 Protocol stack (TCP) 5

4 Establishing a connection 6
- 4.1 Connect . . . 6
- 4.2 Version exchange . . . 7
- 4.3 Authenticate . . . 7
- 4.4 Crypt setup . . . 8
- 4.5 Channel states . . . 8
- 4.6 User states . . . 8
- 4.7 Server sync . . . 9
- 4.8 Ping . . . 9

5 Voice data 9
- 5.1 Enabling the UDP channel . . . 9
- 5.2 Data . . . 11
  - 5.2.1 Whispering . . . 12
- 5.3 64-bit integer encoding . . . 12
- 5.4 TCP tunnel . . . 14
- 5.5 Encryption . . . 14
- 5.6 Implementation notes . . . 14

6 Messages 15

6.1 ACL . . . 15
6.1.1 ACL_ChanACL . . . 15
6.1.2 ACL_ChanGroup . . . 16
6.2 Authenticate . . . 16
6.3 BanList . . . 16
6.3.1 BanList_BanEntry . . . 17
6.4 ChannelRemove . . . 17
6.5 ChannelState . . . 17
6.6 CodecVersion . . . 18
6.7 ContextAction . . . 18
6.8 ContextActionAdd . . . 19
6.8.1 Enumeration: ContextActionAdd_Context . . . 19
6.9 CryptSetup . . . 19
6.10 PermissionDenied . . . 20
6.10.1 Enumeration: PermissionDenied_DenyType . . . 20
6.11 PermissionQuery . . . 20
6.12 Ping . . . 21
6.13 QueryUsers . . . 21
6.14 Reject . . . 22
6.14.1 Enumeration: Reject_RejectType . . . 22
6.15 RequestBlob . . . 22
6.16 ServerConfig . . . 23
6.17 ServerSync . . . 23
6.18 TextMessage . . . 23
6.19 UDPTunnel . . . 24
6.20 UserList . . . 24
6.20.1 UserList_User . . . 24
6.21 UserRemove . . . 24
6.22 UserState . . . 25
6.23 UserStats . . . 26
6.23.1 UserStats_Stats . . . 27
6.24 Version . . . 28
6.25 VoiceTarget . . . 28
6.25.1 VoiceTarget_Target . . . 28

7 This document is WIP 28

A Appendix 29
A.1 Mumble.proto . . . 29

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 It reflects the state of the protocol implemented in the Mumble 1.2.2 client and might be outdated by the time you are reading this.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 `http://www.mumble.info` . At the moment this document is work in progress.

2 Overview

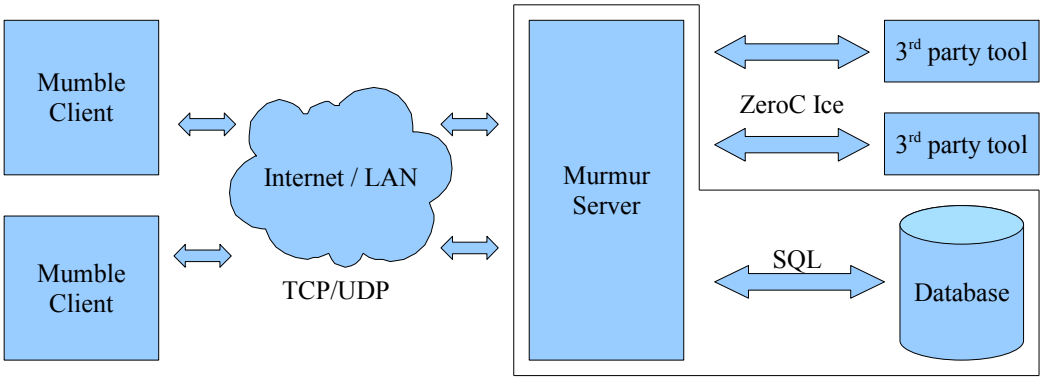

Figure 1: Mumble system overview

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.

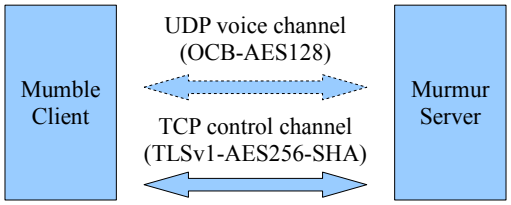

Figure 2: Mumble crypto types

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

[1] `http://en.wikipedia.org/wiki/Transport_Layer_Security`

is encrypted with OCB-AES128[2].

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.

3 Protocol stack (TCP)

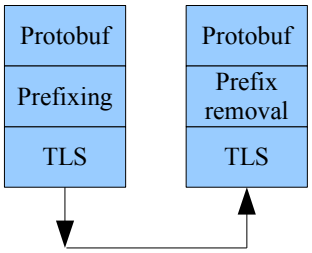

Figure 3: Mumble protocol stack

Mumble has a shallow and easy to understand stack. Basically it uses Google's Protocol Buffers[3] with simple prefixing to distinguish the different kinds of packets sent through an TLSv1 encrypted connection. This makes the protocol very easily expandable.

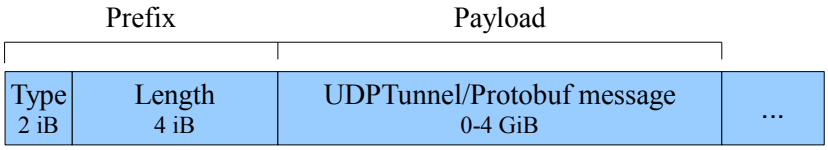

Figure 4: Mumble packet

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 are little-endian encoded.

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

[2] `http://www.cs.ucdavis.edu/~rogaway/ocb/ocb-back.htm`
[3] `http://code.google.com/p/protobuf/`

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.

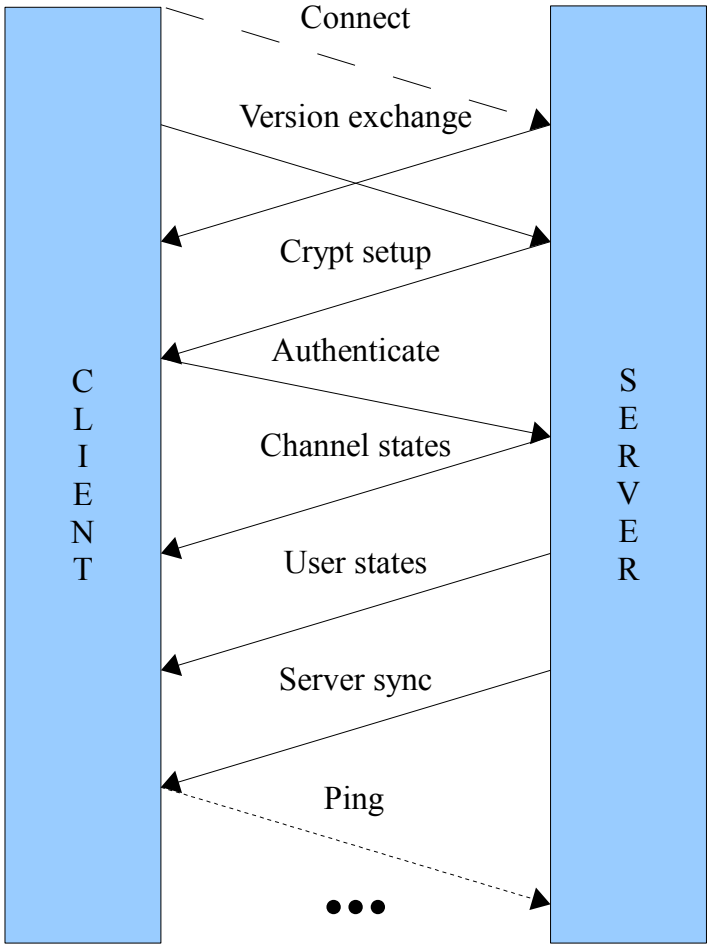

Figure 5: Mumble connection setup

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.

Version	
version	uint32
release	string
os	string
os_version	string

Figure 6: Version message

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 7. The release, os and os_version fields are common strings containing additional information. This information is not interpreted in any way at the moment.

Major	Minor	Patch
2B	1B	1B

Figure 7: `version` field structure

4.3 Authenticate

Once the client has sent the version it should follow this with the Authenticate message. The message structure is described below in figure 8. This message may be sent immediately after sending the version message. The client does not need to wait for the server version message.

Authenticate	
username	string
password	string
tokens	repeated string

Figure 8: Authenticate message

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 Crypt 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 . The encryption itself is described later in section **??**.

CryptSetup	
key	bytes
client_nonce	bytes
server_nonce	bytes

Figure 9: CryptSetup message

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 in 10.

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,

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

Figure 10: ChannelState message

including the user that is currently connecting. The message structure is shown in figure 11.

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 Mumble.proto in the appendix.

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.

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 the client should not

ChannelState	
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

Figure 11: UserState message

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 (See figure 12) 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.

byte	:	type/flags	`0010 0000` for Ping
varint	:	timestamp	Timestamp for the client.

Figure 12: UDP Ping packet

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 section 5.4. 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 a 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 figure 13. 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 section 5.3.

Header	byte	:	type/target	Bit 1-3: Type, Bit 4-8: Target
	varint	:	session	The session number of the source user
	varint	:	sequence	
Audio	byte	:	header	Bit 1: Terminator, Bit 2-8: Data length
Repeated	byte[]	:	data	Encoded voice frames
Position	float	:	Pos 1	Positional audio positions
Optional	float	:	Pos 2	Uses PacketDataStream encoding
	float	:	Pos 3	

Figure 13: UDP Voice packet

The first byte of the header contains the packet type and additional target specifier. The type is stored in the first three bits and specifies the type and encoding of the packet. Current types are listed in table 1. The remaining 5 bits specify additional packet-wide options. For voice packets the values specify the voice target as listed in table 2.

Table 1: UDP Types

Type	Description
0	CELT Alpha encoded voice data
1	Ping packet (See section 5.1)
2	Speex encoded voice data
3	CELT Beta encoded voice data
4-7	Unused

Table 2: 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 table 1). *The data in each frame is encoded separately.*

5.2.1 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 (See 6.25) and specifying the target ID as the target in the first byte of the UDP packet.

5.3 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 encoding function is

given below. While it might seem complex it is worth noting that the $(a_v, a_p) \triangleright (b_v, b_p)$ 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.

$$
\begin{aligned}
(a_v, a_p) \triangleright (b_v, b_p) &= (2^{b_p} a_v + b_v, a_p + b_p) \\
e : \mathbb{N} &\to \mathbb{N}^2_{\geq 0} \\
e(x) &= \begin{cases}
e_+(x, 1) & \text{when } 0 \leq x < 2^{28} \\
\left((2^8 - 2^4) \cdot 2^{8^4} + x, 2^{40}\right) & \text{when } 2^{28} \leq x < 2^{32} \\
\left((2^8 - 2^4 + 2^2) \cdot 2^{8^8} + x, 2^{72}\right) & \text{when } 2^{32} \leq x \\
(2^8 - 2^2 - x, 8) & \text{when } -4 < x < 0 \\
(2^8 - 2^3, 8) \triangleright e(-x) & \text{when } x \leq -4
\end{cases} \\
e_+(x, b) &= \begin{cases}
(p(b) + x, 8) & \text{when } r < 2^(8 - b) \\
e_+\left(\left\lfloor \frac{x}{2^8} \right\rfloor, b + 1\right) \triangleright (x \bmod 2^8, 8) & \text{when } r \geq 2^(8 - b)
\end{cases} \\
p(b) &= 2^8 - 2^{9-b}
\end{aligned}
$$

Essentially the first byte contains the number range information. For positive numbers this tells how many bytes the number takes. For numbers in the range $[0, 2^{28})$ the header bits keep growing steadily, starting from 1 and growing by one each time the number that should be encoded does not fit in the free space, as shown by $e_+(x, b)$. Numbers in the range $[2^{28}, 2^{32})$ have header byte `1111 0000` followed by the full 4 bytes long binary presentation of the number. Numbers greater than 2^{32} have header byte `1111 0100` followed by the full 8 bytes long binary presentation of the number. Negative numbers in the range $(-4, 0)$ are encoded by performing a bitwise or operation between `1111 1100` and the positive presentation of the number. Negative numbers less or equal to -4 have a header byte `1111 1000` followed by the varint encoding of their opposite number.

Decoding is performed by analyzing the first byte after which the rest of the number can be read from the byte stream.

$$
\begin{aligned}
s_0(x) &= 8 - \lfloor log_2(2^8 - 1 - x) \rfloor \\
f_x &: \mathbb{N}_{\geq} 0 \to [0, 2^8) \\
d &: f \to \mathbb{N}, f = \{f_1, f_2, f_3, ...\} \\
d(f) &= \begin{cases}
d_+\Big(f, s_0\big(f(0)\big)\Big) & \text{when } f(0) \leq 2^8 - 2^4 \\
\sum_{i=0}^{4} 2^{32-8i} f(i) & \text{when } f(0) = 2^8 - 2^4 \\
\sum_{i=0}^{8} 2^{64-8i} f(i) & \text{when } f(0) = 2^8 - 2^4 + 2^2 \\
-d(g : g(n) = f(n+1)) & \text{when } f(0) = 2^8 - 2^3 \\
(2^8 - 2^2) - f(0) & \text{when } f(0) \geq 2^8 - 2^2
\end{cases} \\
d_+(f, z) &= -2^{8z-7z} + \sum_{i=1}^{z} 2^{8z-8i} f(i-1)
\end{aligned}
$$

5.4 TCP tunnel

When the UDP packets are tunneled through the TCP tunnel they are prefixed with the TCP protocol header that contains the packet type and length and sent through the connection. (Figure 14)

Type	Length	UDP Packet
1B	3B	0-1020B

Figure 14: UDP Voice packet

5.5 Encryption

All the voice 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. The OCB-AES128 encryption is described in section **??**.

5.6 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 authenti-

cation 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.

6 Messages

6.1 ACL

Field	Type	Rule	Description
`channel_id`	`uint32`	Req.	Channel ID of the channel this message affects
`inherit_acls`	`bool`	Opt.	True if the channel inherits its parent's ACLs, **Default: true**
`groups`	`ChanGroup`	Rep.	User group specifications
`acls`	`ChanACL`	Rep.	ACL specifications
`query`	`bool`	Opt.	True if the message is a query for ACLs instead of setting them, **Default: false**

6.1.1 ACL_ChanACL

Field	Type	Rule	Description
`apply_here`	`bool`	Opt.	True if this ACL applies to the current channel, **Default: true**
`apply_subs`	`bool`	Opt.	True if this ACL applies to the sub channels, **Default: true**
`inherited`	`bool`	Opt.	True if the ACL has been inherited from the parent, **Default: true**
`user_id`	`uint32`	Opt.	ID of the user that is affected by this ACL
`group`	`string`	Opt.	ID of the group that is affected by this group
`grant`	`uint32`	Opt.	Bit flag field of the permissions granted by this ACL
`deny`	`uint32`	Opt.	Bit flag field of the permissions denied by this ACL

6.1.2 ACL_ChanGroup

Field	Type	Rule	Description
`name`	`string`	Req.	Name of the channel group, UTF-8 encoded
`inherited`	`bool`	Opt.	True if the group has been inherited from the parent. **Read only, Default: true**
`inherit`	`bool`	Opt.	True if the group members are inherited, **Default: true**
`inheritable`	`bool`	Opt.	True if the group can be inherited by sub channels, **Default: true**
`add`	`uint32`	Rep.	Users explicitly included in this group, identified by `user_id`
`remove`	`uint32`	Rep.	Users explicitly removed from this group in this channel if the group has been inherited, identified by `user_id`
`inherited_members`	`uint32`	Rep.	Users inherited, identified by `user_id`

6.2 Authenticate

Used by the client to send the authentication credentials to the server.

Field	Type	Rule	Description
`username`	`string`	Opt.	UTF-8 encoded username
`password`	`string`	Opt.	Server or user password
`tokens`	`string`	Rep.	Additional access tokens for server ACL groups
`celt_versions`	`int32`	Rep.	A list of CELT bitstream version constants supported by the client.

6.3 BanList

Relays information on the bans. The client may send the BanList message to either modify the list of bans or query them from the server. The server sends this list only after a client queries for it.

Field	Type	Rule	Description
`bans`	`BanEntry`	Rep.	List of ban entries currently in place
`query`	`bool`	Opt.	True if the server should return the list, False if it should replace old ban list with this one, **Default: false**

6.3.1 BanList_BanEntry

Field	Type	Rule	Description
`address`	`bytes`	Req.	Banned IP address
`mask`	`uint32`	Req.	The length of the subnet mask for the ban
`name`	`string`	Opt.	User name for identification purposes, does not affect the ban
`hash`	`string`	Opt.	TODO ??
`reason`	`string`	Opt.	Reason for the ban, does not affect the ban
`start`	`string`	Opt.	Ban start time
`duration`	`uint32`	Opt.	Ban duration in seconds

6.4 ChannelRemove

Sent by the client when it wants a channel removed. Sent by the server when a channel has been removed and clients should be notified.

Field	Type	Rule	Description
`channel_id`	`uint32`	Req.	The `channel_id` of the channel to be removed

6.5 ChannelState

Used to communicate channel properties between the client and the server. Sent by the server during the login process (See 4.5) or when channel properties are updated. Client may use this message to update said channel properties.

Field	Type	Rule	Description
`channel_id`	`uint32`	Opt.	Unique ID for the channel within the server.
`parent`	`uint32`	Opt.	`channel_id` of the parent channel.
`name`	`string`	Opt.	Channel name, UTF-8 encoded.
`links`	`uint32`	Rep.	A collection of `channel_id` values of the linked channels. Absent during the first channel listing (See 4.5).
`description`	`string`	Opt.	Channel description, UTF-8 encoded. Only if the description is less than 128 bytes
`links_add`	`uint32`	Rep.	A collection of `channel_id` values that should be added to `links`.
`links_remove`	`uint32`	Rep.	A collection of `channel_id` values that should be removed from `links`.
`temporary`	`bool`	Opt.	True if the channel is temporary. **Default: false**
`position`	`uint32`	Opt.	Position weight to tweak the channel position in the channel list. **Default: 0**
`description_hash`	`bytes`	Opt.	SHA1 hash of the description if the description is 128 bytes or more. See 6.15

6.6 CodecVersion

Sent by the server to notify the users of the version of the CELT codec they should use. This may change during the connection when new users join.

Field	Type	Rule	Description
`alpha`	`int32`	Req.	The version of the CELT Alpha codec
`beta`	`int32`	Req.	The version of the CELT Beta codec
`prefer_alpha`	`bool`	Req.	True if the user should prefer Alpha over Beta, **Default: true**

6.7 ContextAction

Sent by the client when it wants to initiate a Context action. Refer to Mumble documentation (TODO Context action source) for more information.

Field	Type	Rule	Description
`session`	`uint32`	Opt.	The target User for the action, identified by `session`
`channel_id`	`uint32`	Opt.	The target Channel for the action, identified by `channel_id`
`action`	`string`	Req.	The action that should be executed

6.8 ContextActionAdd

Sent by the server to inform the client of available context actions.

Field	Type	Rule	Description
`action`	`string`	Req.	The action name
`text`	`string`	Req.	The display name of the action
`context`	`uint32`	Opt.	Context bit flags defining where the action should be displayed, see 6.8.1

6.8.1 Enumeration: ContextActionAdd_Context

Name	Value	Description
`Server`	`0x01`	Action is applicable to the server
`Channel`	`0x02`	Action can target a Channel
`User`	`0x04`	Action can target a User

6.9 CryptSetup

Used to initialize and resync the UDP encryption. See section **??** for more information. Either side may request a resync by sending the message without any values filled. The resync is performed by sending the message with only the client or server nonce filled.

Field	Type	Rule	Description
`key`	`bytes`	Opt.	Encryption key
`client_nonce`	`bytes`	Opt.	Client nonce
`server_nonce`	`bytes`	Opt.	Server nonce

6.10 PermissionDenied

Field	Type	Rule	Description
`permission`	`uint32`	Opt.	The denied permission when type is `Permission`
`channel_id`	`uint32`	Opt.	`channel_id` for the channel where the permission was denied when type is `Permission`
`session`	`uint32`	Opt.	The user who was denied permissions, identified by `session`
`reason`	`string`	Opt.	Textual reason for the denial
`type`	`DenyType`	Opt.	Type of the denial
`name`	`string`	Opt.	The name that is invalid when type is `UserName`

6.10.1 Enumeration: PermissionDenied_DenyType

Name	Value	Description
`Text`	`0`	Operation denied for other reason, see reason field
`Permission`	`1`	Permissions were denied
`SuperUser`	`2`	Cannot modify SuperUser
`ChannelName`	`3`	Invalid channel name
`TextTooLong`	`4`	Text message too long
`H9K`	`5`	The flux capacitor was spelled wrong.
`TemporaryChannel`	`6`	Operation not permitted in temporary channel
`MissingCertificate`	`7`	Operation requires certificate
`UserName`	`8`	Invalid username
`ChannelFull`	`9`	Channel is full

6.11 PermissionQuery

Sent by the client when it wants permissions for a certain channel. Sent by the server when it replies to the query or wants the user to resync all channel permissions.

Field	Type	Rule	Description
`channel_id`	`uint32`	Opt.	`channel_id` of the channel for which the permissions are queried
`permissions`	`uint32`	Opt.	Channel permissions. TODO: Encoded how?
`flush`	`bool`	Opt.	True if the client should drop its current permission information for **all** channels, **Default: false**

6.12 Ping

Sent by the client to notify the server that the client is still alive. Server must reply to the packet with the same timestamp and its own good/late/lost/resync numbers. None of the fields is strictly required.

Field	Type	Rule	Description
`timestamp`	`uint64`	Opt.	Client timestamp. Server should not attempt to decode.
`good`	`uint32`	Opt.	The amount of good packets received
`late`	`uint32`	Opt.	The amount of late packets received
`lost`	`uint32`	Opt.	The amount of packets never received
`resync`	`uint32`	Opt.	The amount of nonce resyncs
`udp_packets`	`uint32`	Opt.	The total amount of UDP packets received
`tcp_packets`	`uint32`	Opt.	The total amount of TCP packets received
`udp_ping_avg`	`float`	Opt.	UDP ping average
`udp_ping_var`	`float`	Opt.	UDP ping variance
`tcp_ping_avg`	`float`	Opt.	TCP ping average
`tcp_ping_var`	`float`	Opt.	TCP ping variance

6.13 QueryUsers

Client may use this message to refresh its registered user information. The client should fill the IDs or Names of the users it wants to refresh. The server fills the missing parts and sends the message back.

Field	Type	Rule	Description
`ids`	`uint32`	Rep.	User IDs
`names`	`string`	Rep.	User names in the same order as `ids`

6.14 Reject

Sent by the server when it rejects the user connection.

Field	Type	Rule	Description
`type`	`RejectType`	Opt.	Rejection type
`reason`	`string`	Opt.	Human readable rejection reason

6.14.1 Enumeration: Reject_RejectType

Name	Value	Description
`None`	`0`	TODO ??
`WrongVersion`	`1`	The client attempted to connect with an incompatible version
`InvalidUsername`	`2`	The user name supplied by the client was invalid
`WrongUserPW`	`3`	The client attempted to authenticate as a user with a password but it was wrong
`WrongServerPW`	`4`	The client attempted to connect to a passworded server but the password was wrong
`UsernameInUse`	`5`	Supplied username is already in use
`ServerFull`	`6`	Server is currently full and cannot accept more users
`NoCertificate`	`7`	The user did not provide a certificate but one is required

6.15 RequestBlob

Used by the client to request binary data from the server. By default large comments or textures are not sent within standard messages but instead the hash is. If the client does not recognize the hash it may request the resource when it needs it. The client does so by sending a RequestBlob message with the correct fields filled with the hashes it wants to receive. The server replies to this by sending a new UserState/ChannelState message with the resources filled even if they would normally be transmitted as hashes.

Field	Type	Rule	Description
`session_texture`	`uint32`	Rep.	Hashes of the requested UserState textures
`session_comment`	`uint32`	Rep.	Hashes of the requested UserState comments
`channel_description`	`uint32`	Rep.	Hashes of the requested ChannelState descriptions

6.16 ServerConfig

Sent by the server when it informs the clients on server configuration details.

Field	Type	Rule	Description
`max_bandwidth`	`uint32`	Opt.	The maximum bandwith the clients should use
`welcome_text`	`string`	Opt.	Server welcome text
`allow_html`	`bool`	Opt.	True if the server allows HTML
`message_length`	`uint32`	Opt.	Maximum text message length
`image_message_length`	`uint32`	Opt.	Maximum image message length

6.17 ServerSync

ServerSync message is sent by the server when it has authenticated the user and finished synchronizing the server state. See section 4.7 for more information on the initial connection exchange.

Field	Type	Rule	Description
`session`	`uint32`	Opt.	The `session` of the current user
`max_bandwidth`	`uint32`	Opt.	Maximum bandwith that the user should use
`welcome_text`	`string`	Opt.	Server welcome text
`permissions`	`uint64`	Opt.	Current user permissions TODO: Confirm??

6.18 TextMessage

Used to send and broadcast text messages.

Field	Type	Rule	Description
`actor`	`uint32`	Opt.	The message sender, identified by its `session`
`session`	`uint32`	Rep.	Target users for the message, identified by their `session`
`channel_id`	`uint32`	Rep.	The channels to which the message is sent, identified by their `channel_id`s
`tree_id`	`uint32`	Rep.	The root channels when sending message recursively to several channels, identified by their `channel_id`s
`message`	`string`	Req.	The UTF-8 encoded message. May be HTML if the server allows.

6.19 UDPTunnel

Used to tunnel the UDP packets through the TCP channel. See section 5.4 for more information.

Field	Type	Rule	Description
`packet`	`bytes`	Req.	The data from the UDP packet

6.20 UserList

Lists the registered users

Field	Type	Rule	Description
`users`	`User`	Rep.	A list of registered users

6.20.1 UserList_User

Field	Type	Rule	Description
`user_id`	`uint32`	Req.	Registered user ID
`name`	`string`	Opt.	Registered user name

6.21 UserRemove

Used to communicate user leaving or being kicked. May be sent by the client when it attempts to kick a user. Sent by the server when it informs the clients that a user is not present anymore.

Field	Type	Rule	Description
`session`	`uint32`	Req.	The user who is being kicked, identified by their `session`, not present when no one is being kicked
`actor`	`uint32`	Opt.	The user who initiated the removal. Either the user who performs the kick or the user who is currently leaving
`reason`	`string`	Opt.	Reason for the kick, stored as the ban reason if the user is banned
`ban`	`bool`	Opt.	True if the kick should result in a ban

6.22 UserState

Sent by the server when it communicates new and changed users to client. First seen during login procedure (See 4.6). May be sent by the client when it wishes to alter its state.

Field	Type	Rule	Description
session	uint32	Opt.	Unique user session ID of the user whose state this is, may change on reconnect
actor	uint32	Opt.	The session of the user who is updating this user
name	string	Opt.	User name, UTF-8 encoded
user_id	uint32	Opt.	Registered user ID if the user is registered
channel_id	uint32	Opt.	Channel on which the user is
mute	bool	Opt.	True if the user is muted by admin
deaf	bool	Opt.	True if the user is deafened by admin
suppress	bool	Opt.	True if the user has been suppressed from talking by a reason other than being muted
self_mute	bool	Opt.	True if the user has muted self
self_deaf	bool	Opt.	True if the user has deafened self
texture	bytes	Opt.	User image if it is less than 128 bytes
plugin_context	bytes	Opt.	TODO ??
plugin_identity	string	Opt.	TODO ??
comment	string	Opt.	User comment if it is less than 128 bytes
hash	string	Opt.	The hash of the user certificate
comment_hash	bytes	Opt.	SHA1 hash of the user comment if it 128 bytes or more. See 6.15
texture_hash	bytes	Opt.	SHA1 hash of the user picture if it 128 bytes or more. See 6.15
priority_speaker	bool	Opt.	True if the user is a priority speaker
recording	bool	Opt.	True if the user is currently recording

6.23 UserStats

Used to communicate user stats between the server and clients.

Field	Type	Rule	Description
session	uint32	Opt.	User whose stats these are
stats_only	bool	Opt.	True if the message contains only mutable stats (packets, ping), **Default: false**
certificates	bytes	Rep.	Full user certificate chain of the user certificate in DER format
from_client	Stats	Opt.	Packet statistics for packets received from the client
from_server	Stats	Opt.	Packet statistics for packets sent by the server
udp_packets	uint32	Opt.	Amount of UDP packets sent
tcp_packets	uint32	Opt.	Amount of TCP packets sent
udp_ping_avg	float	Opt.	UDP ping average
udp_ping_var	float	Opt.	UDP ping variance
tcp_ping_avg	float	Opt.	TCP ping average
tcp_ping_var	float	Opt.	TCP ping variance
version	Version	Opt.	Client version, see 6.24
celt_versions	int32	Rep.	A list of CELT bitstream version constants supported by the client of this user.
address	bytes	Opt.	Client IP address
bandwidth	uint32	Opt.	Bandwith used by this client
onlinesecs	uint32	Opt.	Connection duration
idlesecs	uint32	Opt.	Duration since last activity
strong_certificate	bool	Opt.	True if the user has a strong certificate, **Default: false**

6.23.1 UserStats_Stats

Field	Type	Rule	Description
good	uint32	Opt.	The amount of good packets received
late	uint32	Opt.	The amount of late packets received
lost	uint32	Opt.	The amount of packets never received
resync	uint32	Opt.	The amount of nonce resyncs

6.24 Version

Field	Type	Rule	Description
`version`	`uint32`	Opt.	2-byte Major, 1-byte Minor and 1-byte Patch version number
`release`	`string`	Opt.	Client release name
`os`	`string`	Opt.	Client OS name
`os_version`	`string`	Opt.	Client OS version

6.25 VoiceTarget

Sent by the client when it wants to register or clear whisper targets. See 5.2.1 for more information. **Note: The first available target ID is 1 as 0 is reserved for normal talking. Maximum target ID is 30**

Field	Type	Rule	Description
`id`	`uint32`	Opt.	Voice target ID
`targets`	`Target`	Rep.	The receivers that this voice target includes

6.25.1 VoiceTarget_Target

Field	Type	Rule	Description
`session`	`uint32`	Rep.	Users that are included as targets
`channel_id`	`uint32`	Opt.	Channels that are included as targets
`group`	`string`	Opt.	TODO ??
`links`	`bool`	Opt.	True if the voice should follow links from the specified channel, **Default: false**
`children`	`bool`	Opt.	True if the voice should also be sent to children of the specifiec channel, **Default: false**

7 This document is WIP

SORRY BUT THIS DOCUMENT IS WORK IN PROGRESS. AT THE MOMENT IT LACKS A LOT OF IMPORTANT INFORMATION BUT WE HOPE TO BE ABLE TO FINISH THIS DOCUMENT SOMEDAY :-)

A Appendix

A.1 Mumble.proto

```
1  package MumbleProto;
2
3  option optimize_for = SPEED;
4
5  message        Version {
6          optional uint32 version = 1;
7          optional string release = 2;
8          optional string os = 3;
9          optional string os_version = 4;
10 }
11
12 message UDPTunnel {
13         required bytes packet = 1;
14 }
15
16 message Authenticate {
17         optional string username = 1;
18         optional string password = 2;
19         repeated string tokens = 3;
20         repeated int32 celt_versions = 4;
21 }
22
23 message Ping {
24         optional uint64 timestamp = 1;
25         optional uint32 good = 2;
26         optional uint32 late = 3;
27         optional uint32 lost = 4;
28         optional uint32 resync = 5;
29         optional uint32 udp_packets = 6;
30         optional uint32 tcp_packets = 7;
31         optional float udp_ping_avg = 8;
32         optional float udp_ping_var = 9;
33         optional float tcp_ping_avg = 10;
34         optional float tcp_ping_var = 11;
35 }
36
37 message Reject {
38         enum RejectType {
39                 None = 0;
```

```
                WrongVersion = 1;
                InvalidUsername = 2;
                WrongUserPW = 3;
                WrongServerPW = 4;
                UsernameInUse = 5;
                ServerFull = 6;
                NoCertificate = 7;
        }
        optional RejectType type = 1;
        optional string reason = 2;
}

message ServerConfig {
        optional uint32 max_bandwidth = 1;
        optional string welcome_text = 2;
        optional bool allow_html = 3;
        optional uint32 message_length = 4;
        optional uint32 image_message_length = 5;
}

message ServerSync {
        optional uint32 session = 1;
        optional uint32 max_bandwidth = 2;
        optional string welcome_text = 3;
        optional uint64 permissions = 4;
}

message ChannelRemove {
        required uint32 channel_id = 1;
}

message ChannelState {
        optional uint32 channel_id = 1;
        optional uint32 parent = 2;
        optional string name = 3;
        repeated uint32 links = 4;
        optional string description = 5;
        repeated uint32 links_add = 6;
        repeated uint32 links_remove = 7;
        optional bool temporary = 8 [default = false];
        optional int32 position = 9 [default = 0];
        optional bytes description_hash = 10;
}
```

```
83
84  message UserRemove {
85          required uint32 session = 1;
86          optional uint32 actor = 2;
87          optional string reason = 3;
88          optional bool ban = 4;
89  }
90
91  message UserState {
92          optional uint32 session = 1;
93          optional uint32 actor = 2;
94          optional string name = 3;
95          optional uint32 user_id = 4;
96          optional uint32 channel_id = 5;
97          optional bool mute = 6;
98          optional bool deaf = 7;
99          optional bool suppress = 8;
100         optional bool self_mute = 9;
101         optional bool self_deaf = 10;
102         optional bytes texture = 11;
103         optional bytes plugin_context = 12;
104         optional string plugin_identity = 13;
105         optional string comment = 14;
106         optional string hash = 15;
107         optional bytes comment_hash = 16;
108         optional bytes texture_hash = 17;
109         optional bool priority_speaker = 18;
110         optional bool recording = 19;
111 }
112
113 message BanList {
114         message BanEntry {
115                 required bytes address = 1;
116                 required uint32 mask = 2;
117                 optional string name = 3;
118                 optional string hash = 4;
119                 optional string reason = 5;
120                 optional string start = 6;
121                 optional uint32 duration = 7;
122         }
123         repeated BanEntry bans = 1;
124         optional bool query = 2 [default = false];
125 }
```

```
126
127 message TextMessage {
128         optional uint32 actor = 1;
129         repeated uint32 session = 2;
130         repeated uint32 channel_id = 3;
131         repeated uint32 tree_id = 4;
132         required string message = 5;
133 }
134
135 message PermissionDenied {
136         enum DenyType {
137                 Text = 0;
138                 Permission = 1;
139                 SuperUser = 2;
140                 ChannelName = 3;
141                 TextTooLong = 4;
142                 H9K = 5;
143                 TemporaryChannel = 6;
144                 MissingCertificate = 7;
145                 UserName = 8;
146                 ChannelFull = 9;
147         }
148         optional uint32 permission = 1;
149         optional uint32 channel_id = 2;
150         optional uint32 session = 3;
151         optional string reason = 4;
152         optional DenyType type = 5;
153         optional string name = 6;
154 }
155
156 message ACL {
157         message ChanGroup {
158                 required string name = 1;
159                 optional bool inherited = 2 [default = true];
160                 optional bool inherit = 3 [default = true];
161                 optional bool inheritable = 4 [default = true];
162                 repeated uint32 add = 5;
163                 repeated uint32 remove = 6;
164                 repeated uint32 inherited_members = 7;
165         }
166         message ChanACL {
167                 optional bool apply_here = 1 [default = true];
168                 optional bool apply_subs = 2 [default = true];
```

```
                optional bool inherited = 3 [default = true];
                optional uint32 user_id = 4;
                optional string group = 5;
                optional uint32 grant = 6;
                optional uint32 deny = 7;
        }
        required uint32 channel_id = 1;
        optional bool inherit_acls = 2 [default = true];
        repeated ChanGroup groups = 3;
        repeated ChanACL acls = 4;
        optional bool query = 5 [default = false];
}

message QueryUsers {
        repeated uint32 ids = 1;
        repeated string names = 2;
}

message CryptSetup {
        optional bytes key = 1;
        optional bytes client_nonce = 2;
        optional bytes server_nonce = 3;
}

message ContextActionAdd {
        enum Context {
                Server = 0x01;
                Channel = 0x02;
                User = 0x04;
        }
        required string action = 1;
        required string text = 2;
        optional uint32 context = 3;
}

message ContextAction {
        optional uint32 session = 1;
        optional uint32 channel_id = 2;
        required string action = 3;
}

message UserList {
        message User {
```

```
                required uint32 user_id = 1;
                optional string name = 2;
        }
        repeated User users = 1;
}

message VoiceTarget {
        message Target {
                repeated uint32 session = 1;
                optional uint32 channel_id = 2;
                optional string group = 3;
                optional bool links = 4 [default = false];
                optional bool children = 5 [default = false];
        }
        optional uint32 id = 1;
        repeated Target targets = 2;
}

message PermissionQuery {
        optional uint32 channel_id = 1;
        optional uint32 permissions = 2;
        optional bool flush = 3 [default = false];
}

message CodecVersion {
        required int32 alpha = 1;
        required int32 beta = 2;
        required bool prefer_alpha = 3 [default = true];
}

message UserStats {
        message Stats {
                optional uint32 good = 1;
                optional uint32 late = 2;
                optional uint32 lost = 3;
                optional uint32 resync = 4;
        }

        optional uint32 session = 1;
        optional bool stats_only = 2 [default = false];
        repeated bytes certificates = 3;
        optional Stats from_client = 4;
        optional Stats from_server = 5;
```

```
255
256             optional uint32 udp_packets = 6;
257             optional uint32 tcp_packets = 7;
258             optional float udp_ping_avg = 8;
259             optional float udp_ping_var = 9;
260             optional float tcp_ping_avg = 10;
261             optional float tcp_ping_var = 11;
262
263             optional Version version = 12;
264             repeated int32 celt_versions = 13;
265             optional bytes address = 14;
266             optional uint32 bandwidth = 15;
267             optional uint32 onlinesecs = 16;
268             optional uint32 idlesecs = 17;
269             optional bool strong_certificate = 18 [default = false];
270     }
271
272     message RequestBlob {
273             repeated uint32 session_texture = 1;
274             repeated uint32 session_comment = 2;
275             repeated uint32 channel_description = 3;
276     }
```