Mumble protocol 1.2.X reference (WIP)

Stefan Hacker, Mikko Rantanen

October 16, 2010

DISCLAIMER

Contents

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 It reflects the state of the protocol implemented in the Mumble 1.2.2 client and might be outdated by the time you are reading this.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
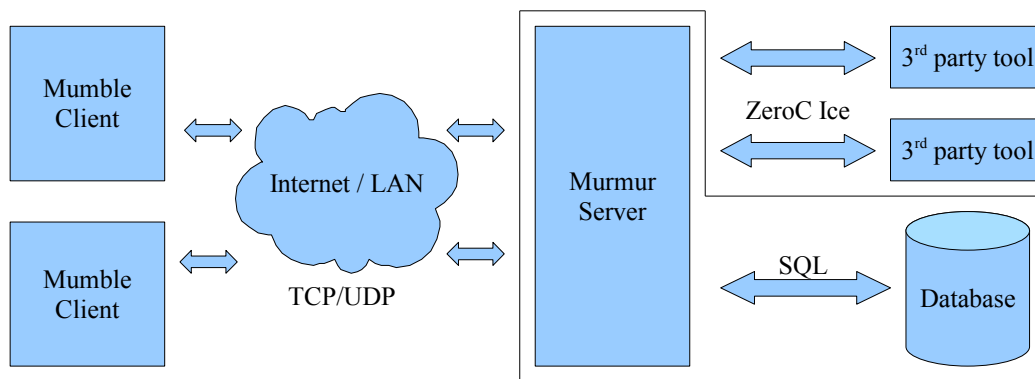
Figure 1: Mumble system overview

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.
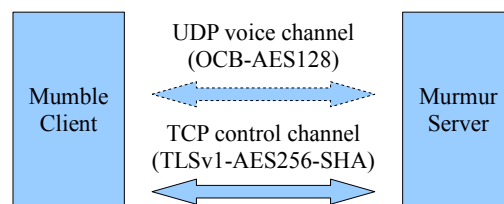
Figure 2: Mumble crypto types

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)
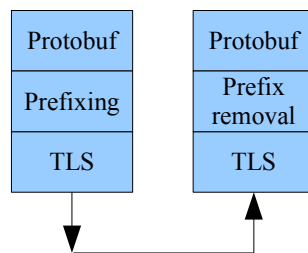
Figure 3: Mumble protocol stack

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.
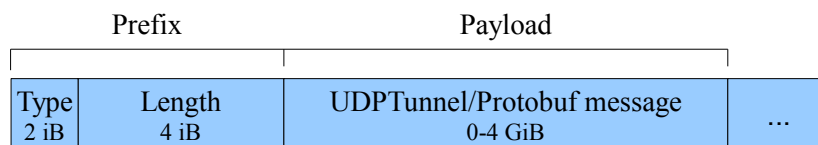
Figure 4: Mumble packet

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.
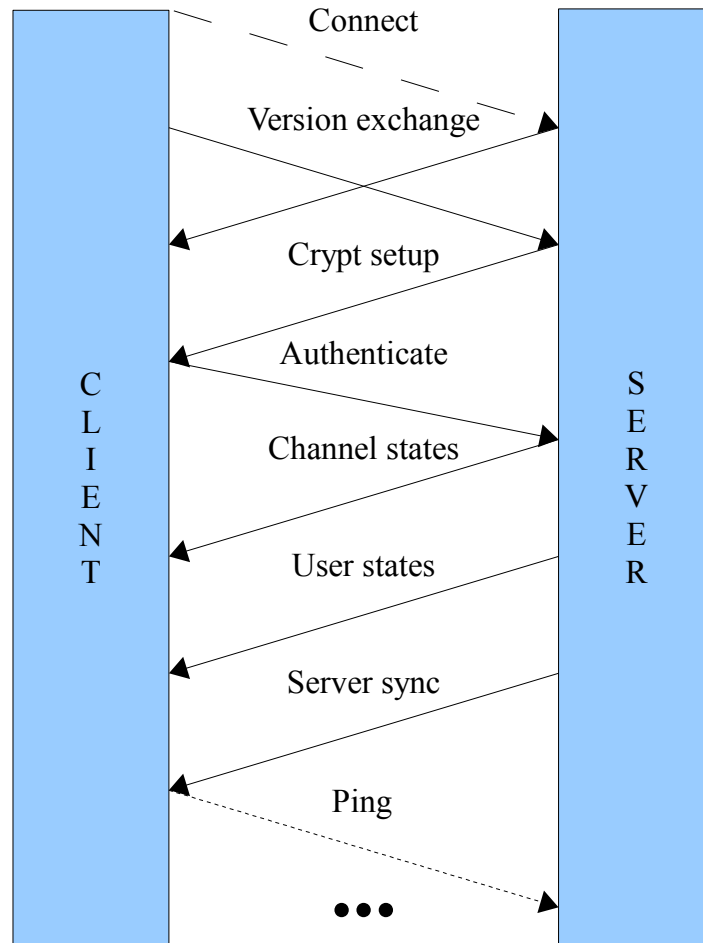
Figure 5: Mumble connection setup

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

9

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 5.3. 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 protocol transfers 64-bit integers using variable length encoding. This encoding is specified in section **??**.

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.

11

Current types are listed in table **??**. 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 (Refer to VoiceTarget, **??**). 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.*

5.3 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	0-2048 KiB

Figure 14: UDP Voice packet

5.4 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 **??**.

5.5 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.

5.6 PacketDataStream

The PacketDataStream class is used to serialize/deserialize the data packets received on the UDP connection or via the TCP-Tunneling. As the name implies it provides a stream based access to the data it contains. To pull data from it the user has to know what is located on the current position in the stream (e.g. a uint32, utf8 string and so on), the class itself is not aware of it's contents.

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
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

6.1.1 ChanACL

Field	Type	Rule	Description
apply_here	bool	Opt.	True if this ACL applies to the current channel
apply_subs	bool	Opt.	True if this ACL applies to the sub channels
inherited	bool	Opt.	True if the ACL has been inherited from the parent
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 ChanGroup

Field	Type	Rule	Description
name	string	Req.	Name of the channel group, UTF-8 enccoded
inherited	bool	Opt.	True if the group has been inherited from the parent. **Read only**
inherit	bool	Opt.	True if the group members are inherited
inheritable	bool	Opt.	True if the group can be inherited by sub channels
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.	TODO ??: Currently unused (mumble

Messages.cpp does not write)

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

6.3.1 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 states are updated. TODO: Sent by users if they modify channels?

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 **??**).
description	string	Opt.	Channel description, UTF-8 encoded.
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.	TODO ??. **Default: 0**
description_hash	bytes	Opt.	TODO ??

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

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: 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.	TODO
channel_id	uint32	Opt.	TODO
session	uint32	Opt.	TODO
reason	string	Opt.	TODO
type	DenyType	Opt.	TODO
name	string	Opt.	TODO

6.10.1 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

Field	Type	Rule	Description
channel_id	uint32	Opt.	TODO
permissions	uint32	Opt.	TODO
flush	bool	Opt.	TODO

6.12 Ping

Field	Type	Rule	Description
timestamp	uint64	Opt.	TODO
good	uint32	Opt.	TODO
late	uint32	Opt.	TODO
lost	uint32	Opt.	TODO
resync	uint32	Opt.	TODO
udp_packets	uint32	Opt.	TODO
tcp_packets	uint32	Opt.	TODO
udp_ping_avg	float	Opt.	TODO
udp_ping_var	float	Opt.	TODO
tcp_ping_avg	float	Opt.	TODO
tcp_ping_var	float	Opt.	TODO

6.13 QueryUsers

Field	Type	Rule	Description
ids	uint32	Rep.	TODO
names	string	Rep.	TODO

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 RejectType

6.15 RequestBlob

Field	Type	Rule	Description
session_texture	uint32	Rep.	TODO
session_comment	uint32	Rep.	TODO
channel_description	uint32	Rep.	TODO

6.16 ServerConfig

Field	Type	Rule	Description
max_bandwidth	uint32	Opt.	TODO
welcome_text	string	Opt.	TODO
allow_html	bool	Opt.	TODO
message_length	uint32	Opt.	TODO
image_message_length	uint32	Opt.	TODO

6.17 ServerSync

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

Field	Type	Rule	Description
session	uint32	Opt.	TODO
max_bandwidth	uint32	Opt.	TODO
welcome_text	string	Opt.	TODO
permissions	uint64	Opt.	TODO

6.18   TextMessage

Field	Type	Rule	Description
actor	uint32	Opt.	TODO
session	uint32	Rep.	TODO
channel_id	uint32	Rep.	TODO
tree_id	uint32	Rep.	TODO
message	string	Req.	TODO

6.19   UDPTunnel

Field	Type	Rule	Description
packet	bytes	Req.	TODO

6.20   UserList

Field	Type	Rule	Description
users	User	Rep.	TODO

6.20.1   User

Field	Type	Rule	Description
user_id	uint32	Req.	TODO
name	string	Opt.	TODO

6.21   UserRemove

Field	Type	Rule	Description
session	uint32	Req.	TODO
actor	uint32	Opt.	TODO
reason	string	Opt.	TODO
ban	bool	Opt.	TODO

6.22 UserState

Field	Type	Rule	Description
session	uint32	Opt.	Unique user session ID
actor	uint32	Opt.	TODO
name	string	Opt.	User name, UTF-8 encoded
user_id	uint32	Opt.	TODO
channel_id	uint32	Opt.	TODO
mute	bool	Opt.	TODO
deaf	bool	Opt.	TODO
suppress	bool	Opt.	TODO
self_mute	bool	Opt.	TODO
self_deaf	bool	Opt.	TODO
texture	bytes	Opt.	TODO
plugin_context	bytes	Opt.	TODO
plugin_identity	string	Opt.	TODO
comment	string	Opt.	TODO
hash	string	Opt.	TODO
comment_hash	bytes	Opt.	TODO
texture_hash	bytes	Opt.	TODO
priority_speaker	bool	Opt.	Is the user a priority speaker?
recording	bool	Opt.	Is the user currently recording?

6.23 UserStats

Field	Type	Rule	Description
session	uint32	Opt.	TODO
stats_only	bool	Opt.	TODO
certificates	bytes	Rep.	TODO
from_client	Stats	Opt.	TODO
from_server	Stats	Opt.	TODO
udp_packets	uint32	Opt.	TODO
tcp_packets	uint32	Opt.	TODO
udp_ping_avg	float	Opt.	TODO
udp_ping_var	float	Opt.	TODO
tcp_ping_avg	float	Opt.	TODO
tcp_ping_var	float	Opt.	TODO
version	Version	Opt.	TODO
celt_versions	int32	Rep.	TODO
address	bytes	Opt.	TODO
bandwidth	uint32	Opt.	TODO
onlinesecs	uint32	Opt.	TODO
idlesecs	uint32	Opt.	TODO
strong_certificate	bool	Opt.	TODO

6.23.1 Stats

Field	Type	Rule	Description
good	uint32	Opt.	TODO
late	uint32	Opt.	TODO
lost	uint32	Opt.	TODO
resync	uint32	Opt.	TODO

6.24 Version

Field	Type	Rule	Description
version	uint32	Opt.	TODO
release	string	Opt.	TODO
os	string	Opt.	TODO
os_version	string	Opt.	TODO

6.25 VoiceTarget

Field	Type	Rule	Description
id	uint32	Opt.	TODO
targets	Target	Rep.	TODO

6.25.1 Target

Field	Type	Rule	Description

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

```
package MumbleProto;

option optimize_for = SPEED;

message        Version {
        optional uint32 version = 1;
        optional string release = 2;
        optional string os = 3;
        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;
40                      WrongVersion = 1;
41                      InvalidUsername = 2;
42                      WrongUserPW = 3;
43                      WrongServerPW = 4;
44                      UsernameInUse = 5;
45                      ServerFull = 6;
46                      NoCertificate = 7;
47              }
48              optional RejectType type = 1;
49              optional string reason = 2;
50      }

51

52      message ServerConfig {
```

```
53          optional uint32 max_bandwidth = 1;
54          optional string welcome_text = 2;
55          optional bool allow_html = 3;
56          optional uint32 message_length = 4;
57          optional uint32 image_message_length = 5;
58      }
59
60      message ServerSync {
61          optional uint32 session = 1;
62          optional uint32 max_bandwidth = 2;
63          optional string welcome_text = 3;
64          optional uint64 permissions = 4;
65      }
66
67      message ChannelRemove {
68          required uint32 channel_id = 1;
69      }
70
71      message ChannelState {
72          optional uint32 channel_id = 1;
73          optional uint32 parent = 2;
74          optional string name = 3;
75          repeated uint32 links = 4;
76          optional string description = 5;
77          repeated uint32 links_add = 6;
78          repeated uint32 links_remove = 7;
79          optional bool temporary = 8 [default = false];
80          optional int32 position = 9 [default = 0];
81          optional bytes description_hash = 10;
82      }
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;
```

```
              SuperUser = 2;
              ChannelName = 3;
              TextTooLong = 4;
              H9K = 5;
              TemporaryChannel = 6;
              MissingCertificate = 7;
              UserName = 8;
              ChannelFull = 9;
        }
        optional uint32 permission = 1;
        optional uint32 channel_id = 2;
        optional uint32 session = 3;
        optional string reason = 4;
        optional DenyType type = 5;
        optional string name = 6;
}

message ACL {
        message ChanGroup {
                required string name = 1;
                optional bool inherited = 2 [default = true];
                optional bool inherit = 3 [default = true];
                optional bool inheritable = 4 [default = true];
                repeated uint32 add = 5;
                repeated uint32 remove = 6;
                repeated uint32 inherited_members = 7;
        }
        message ChanACL {
                optional bool apply_here = 1 [default = true];
                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];
}
```

```
182   message QueryUsers {
183           repeated uint32 ids = 1;
184           repeated string names = 2;
185   }
186
187   message CryptSetup {
188           optional bytes key = 1;
189           optional bytes client_nonce = 2;
190           optional bytes server_nonce = 3;
191   }
192
193   message ContextActionAdd {
194           enum Context {
195                   Server = 0x01;
196                   Channel = 0x02;
197                   User = 0x04;
198           }
199           required string action = 1;
200           required string text = 2;
201           optional uint32 context = 3;
202   }
203
204   message ContextAction {
205           optional uint32 session = 1;
206           optional uint32 channel_id = 2;
207           required string action = 3;
208   }
209
210   message UserList {
211           message User {
212                   required uint32 user_id = 1;
213                   optional string name = 2;
214           }
215           repeated User users = 1;
216   }
217
218   message VoiceTarget {
219           message Target {
220                   repeated uint32 session = 1;
221                   optional uint32 channel_id = 2;
222                   optional string group = 3;
223                   optional bool links = 4 [default = false];
224                   optional bool children = 5 [default = false];
```

```
225            }
226            optional uint32 id = 1;
227            repeated Target targets = 2;
228    }
229
230    message PermissionQuery {
231            optional uint32 channel_id = 1;
232            optional uint32 permissions = 2;
233            optional bool flush = 3 [default = false];
234    }
235
236    message CodecVersion {
237            required int32 alpha = 1;
238            required int32 beta = 2;
239            required bool prefer_alpha = 3 [default = true];
240    }
241
242    message UserStats {
243            message Stats {
244                    optional uint32 good = 1;
245                    optional uint32 late = 2;
246                    optional uint32 lost = 3;
247                    optional uint32 resync = 4;
248            }
249
250            optional uint32 session = 1;
251            optional bool stats_only = 2 [default = false];
252            repeated bytes certificates = 3;
253            optional Stats from_client = 4;
254            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    }
```