Mumble protocol 1.2.X reference (WIP)

Stefan Hacker

October 15, 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 It reflects the state of the protocol implemented in the Mumble 1.2.2 client and might be outdated by the time you are reading this.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

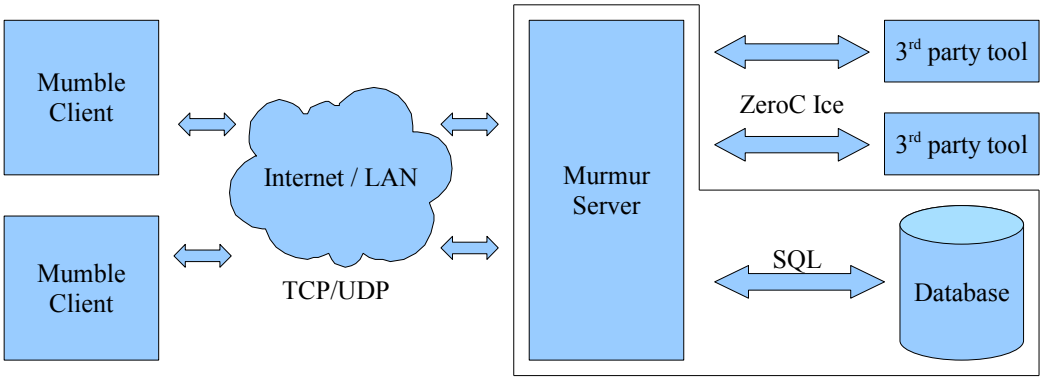

Figure 1: Mumble system overview

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.

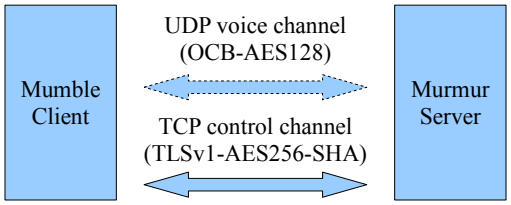

Figure 2: Mumble crypto types

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)

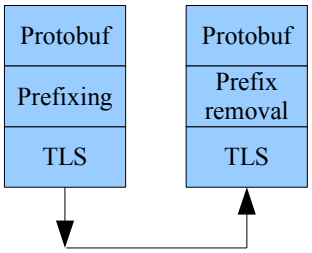

Figure 3: Mumble protocol stack

Mumble has a shallow and easy to understand stack. Basically it uses Googles 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.

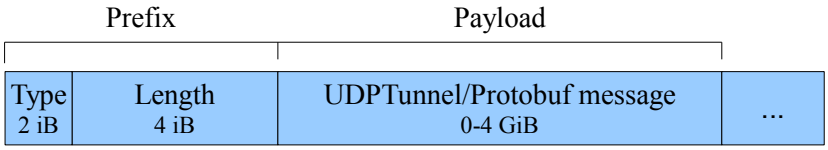

Figure 4: Mumble packet

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

The following section is going to describe the communication between the server and the client during connection establishing, note that the first part of this section only contains the procedures for the TCP connection. After this the client will be visible to the other clients on the server and able to send other types of messages.

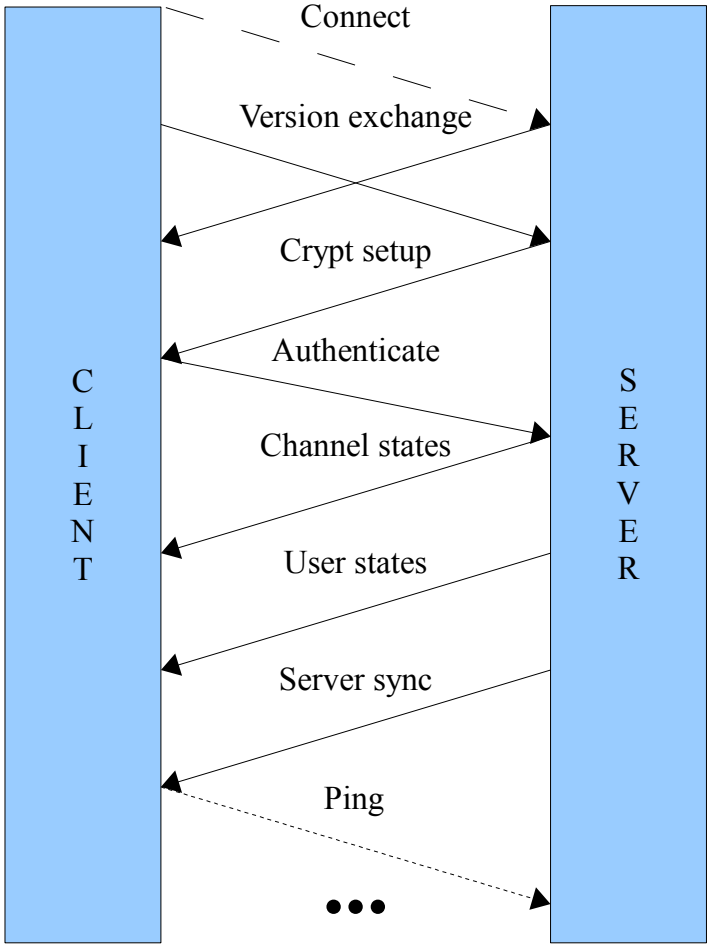

Figure 5: Mumble connection setup

4.1 Connect

As a basis for the synchronization procedure you first have to 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 your client provides a

strong certificate to the server. This however is not mandatory, you can connect to the server without providing a certificate, we do recommend to check the servers certificate though.

4.2 Version exchange

Once the TLS handshake is completed the server will send a Version packet to the client containing following information:

Version	
version	uint32
release	string
os	string
os_version	string

The client is supposed to send a Version packet with his information before any other messages. The version field of the packet contains the (major, minor, patch) tuple (e.g. 1.2.0) encoded like described in the following figure.

Major	Minor	Patch
2B	1B	1B

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

4.3 Crypt setup

Once the Version packets are exchanged the server will send a CryptSetup packet to the client. It contains the necessary cryptographic information to establish the OCB-AES128 encrypted UDP Voice channel. This will be described later in the section.

CryptSetup	
Key	ByteString
ServerNonce	ByteString
ClientNonce	ByteString

4.4 Authenticate

Before the client can be synchronized with the server state it has to authenticate itself to the server. This is done by sending an Authenticate packet.

Authenticate	
username	string
password	string
tokens	repeated string

The username and password are encoded as simple strings. Be aware that the server can impose restrictions on the username, also once the client registered a certificate with the server this field is only displayed in brackets behind the name the client posessed when he 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 called tokens contains a list of zero or more strings called tokens which are basically password which can give you access to a certain ACL group without actually being a registered member in them, again see the server documentation for more information.

4.5 Channel states

After the client is successfully authenticated the server starts synchronizing the state by transmitting a ChannelState message for every channel on this server. Note that these do not yet contain channel links. These are transmitted as updated directly after every channel has been transmitted. It contains the following information:

For more information pease refer to Mumble.proto in the appendix.

4.6 User states

When the channels are synchronized the server send a UserState message for every user connected to the client containing the following data:

For more information pease refer to Mumble.proto in the appendix.

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 TODO seconds it will disconnect the client.

5 Voice data

5.1 Enabling the UDP channel

By default the voice data is transmitted by tunneling the UDP packets through the TCP connection as it is not certain if the client can be reached with UDP. The tunneling packets are described below in section 5.3. The server enables the UDP connection when it receives the first UDP transmission (Or must it be UDP Ping?) from the client and continues it while it keeps receiving these packets. For this reason the client should start implementing the UDP voice data transmission by implementing an UDP ping to the server. The UDP ping packet is described below in figure 6.

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

Figure 6: UDP Ping packet

5.2 Data

The voice packets consist of header portion, followed by repeated data segments and an optional position part. The full packet structure is shown in figure 8. The protocol transfers 64 bit integers using variable length encoding. This encoding is specified in section **??**.

The first byte of the header contains the packet type and additional target specifier. The current types are listed in table 1 and the targets are 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

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` that informs the receiver whether

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 7: UDP Voice packet

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

there are more audio frames after this one. The last audio frame in the UDP packet has the terminator bit turned off. All the preceeding frames have the terminator bit on. 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 **??**)

Type	Length	UDP Packet
1B	3B	0-2048 KiB

Figure 8: 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 **??**.

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.

5.5 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 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
```

```
message        Version {
        optional uint32 version = 1;
        optional string release = 2;
        optional string os = 3;
        optional string os_version = 4;
}

message UDPTunnel {
        required bytes packet = 1;
}

message Authenticate {
        optional string username = 1;
        optional string password = 2;
        repeated string tokens = 3;
        repeated int32 celt_versions = 4;
}

message Ping {
        optional uint64 timestamp = 1;
        optional uint32 good = 2;
        optional uint32 late = 3;
        optional uint32 lost = 4;
        optional uint32 resync = 5;
        optional uint32 udp_packets = 6;
        optional uint32 tcp_packets = 7;
        optional float udp_ping_avg = 8;
        optional float udp_ping_var = 9;
        optional float tcp_ping_avg = 10;
        optional float tcp_ping_var = 11;
}

message Reject {
        enum RejectType {
                None = 0;
                WrongVersion = 1;
                InvalidUsername = 2;
                WrongUserPW = 3;
                WrongServerPW = 4;
                UsernameInUse = 5;
                ServerFull = 6;
                NoCertificate = 7;
        }
```

```
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
```

```
message UserState {
        optional uint32 session = 1;
        optional uint32 actor = 2;
        optional string name = 3;
        optional uint32 user_id = 4;
        optional uint32 channel_id = 5;
        optional bool mute = 6;
        optional bool deaf = 7;
        optional bool suppress = 8;
        optional bool self_mute = 9;
        optional bool self_deaf = 10;
        optional bytes texture = 11;
        optional bytes plugin_context = 12;
        optional string plugin_identity = 13;
        optional string comment = 14;
        optional string hash = 15;
        optional bytes comment_hash = 16;
        optional bytes texture_hash = 17;
        optional bool priority_speaker = 18;
        optional bool recording = 19;
}

message BanList {
        message BanEntry {
                required bytes address = 1;
                required uint32 mask = 2;
                optional string name = 3;
                optional string hash = 4;
                optional string reason = 5;
                optional string start = 6;
                optional uint32 duration = 7;
        }
        repeated BanEntry bans = 1;
        optional bool query = 2 [default = false];
}

message TextMessage {
        optional uint32 actor = 1;
        repeated uint32 session = 2;
        repeated uint32 channel_id = 3;
        repeated uint32 tree_id = 4;
        required string message = 5;
}
```

```
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];
169                 optional bool inherited = 3 [default = true];
170                 optional uint32 user_id = 4;
171                 optional string group = 5;
172                 optional uint32 grant = 6;
173                 optional uint32 deny = 7;
174         }
175         required uint32 channel_id = 1;
176         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;

        optional uint32 udp_packets = 6;
        optional uint32 tcp_packets = 7;
        optional float udp_ping_avg = 8;
        optional float udp_ping_var = 9;
        optional float tcp_ping_avg = 10;
        optional float tcp_ping_var = 11;

```

```
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 }
```