Mumble protocol 1.2.X reference (WIP)

Stefan Hacker

January 24, 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 It reflects the state of the protocol implemented in the Mumble 1.2.2 client and might be outdated by the time you are reading this.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

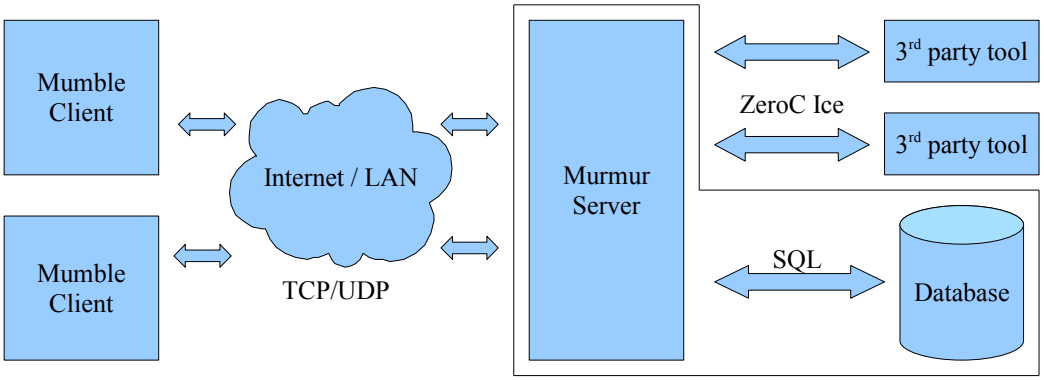

Figure 1: Mumble system overview

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.

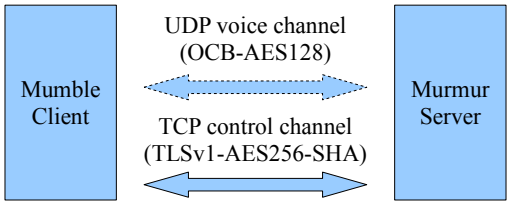

Figure 2: Mumble crypto types

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)

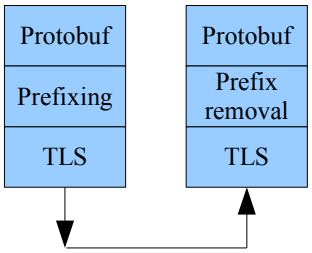

Figure 3: Mumble protocol stack

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.

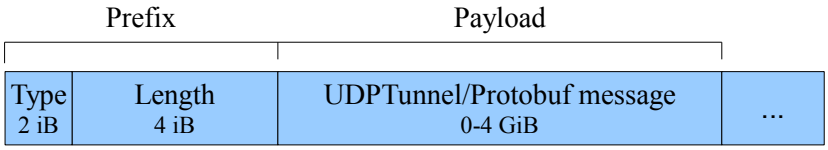

Figure 4: Mumble packet

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.

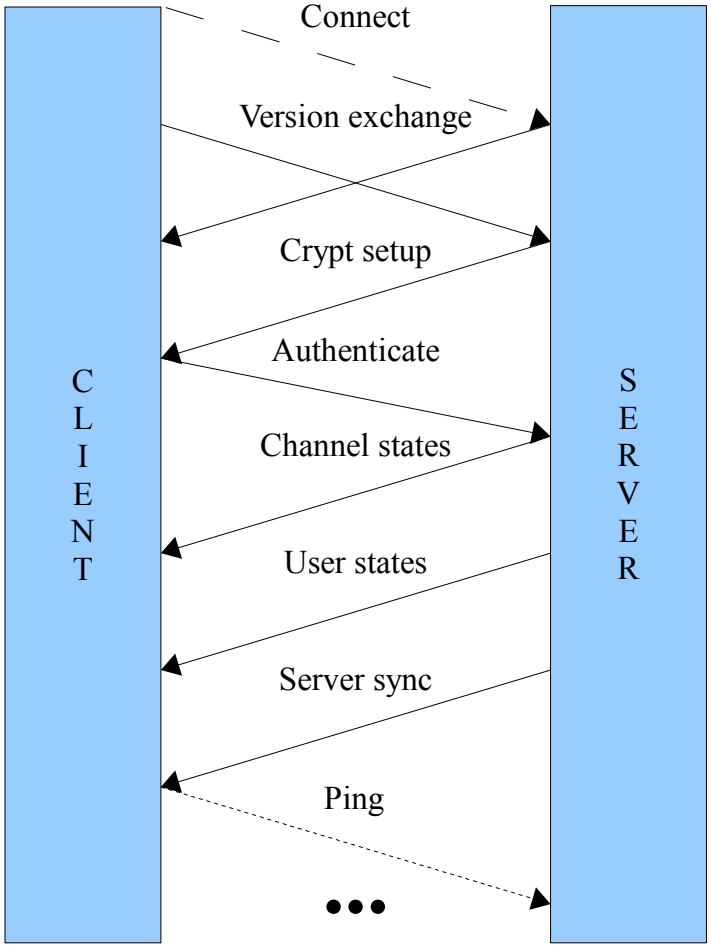

Figure 5: Mumble connection setup

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:

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.

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.

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.

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 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.

5 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;
}

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;
        }
        optional RejectType type = 1;
        optional string reason = 2;
}

message ServerSync {
        optional uint32 session = 1;
        optional uint32 max_bandwidth = 2;
        optional string welcome_text = 3;
        optional uint64 permissions = 4;
        optional bool allow_html = 5 [default = true];
}

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;
}

message UserRemove {
        required uint32 session = 1;
        optional uint32 actor = 2;
        optional string reason = 3;
        optional bool ban = 4;
}
```

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

```
message PermissionDenied {
        enum DenyType {
                Text = 0;
                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;
```

```
169         repeated ChanACL acls = 4;
170         optional bool query = 5 [default = false];
171 }
172
173 message QueryUsers {
174         repeated uint32 ids = 1;
175         repeated string names = 2;
176 }
177
178 message CryptSetup {
179         optional bytes key = 1;
180         optional bytes client_nonce = 2;
181         optional bytes server_nonce = 3;
182 }
183
184 message ContextActionAdd {
185         enum Context {
186                 Server = 0x01;
187                 Channel = 0x02;
188                 User = 0x04;
189         }
190         required string action = 1;
191         required string text = 2;
192         optional uint32 context = 3;
193 }
194
195 message ContextAction {
196         optional uint32 session = 1;
197         optional uint32 channel_id = 2;
198         required string action = 3;
199 }
200
201 message UserList {
202         message User {
203                 required uint32 user_id = 1;
204                 optional string name = 2;
205         }
206         repeated User users = 1;
207 }
208
209 message VoiceTarget {
210         message Target {
211                 repeated uint32 session = 1;
```

```
212                 optional uint32 channel_id = 2;
213                 optional string group = 3;
214                 optional bool links = 4 [default = false];
215                 optional bool children = 5 [default = false];
216         }
217         optional uint32 id = 1;
218         repeated Target targets = 2;
219 }
220
221 message PermissionQuery {
222         optional uint32 channel_id = 1;
223         optional uint32 permissions = 2;
224         optional bool flush = 3 [default = false];
225 }
226
227 message CodecVersion {
228         required int32 alpha = 1;
229         required int32 beta = 2;
230         required bool prefer_alpha = 3 [default = true];
231 }
232
233 message UserStats {
234         message Stats {
235                 optional uint32 good = 1;
236                 optional uint32 late = 2;
237                 optional uint32 lost = 3;
238                 optional uint32 resync = 4;
239         }
240
241         optional uint32 session = 1;
242         optional bool stats_only = 2 [default = false];
243         repeated bytes certificates = 3;
244         optional Stats from_client = 4;
245         optional Stats from_server = 5;
246
247         optional uint32 udp_packets = 6;
248         optional uint32 tcp_packets = 7;
249         optional float udp_ping_avg = 8;
250         optional float udp_ping_var = 9;
251         optional float tcp_ping_avg = 10;
252         optional float tcp_ping_var = 11;
253
254         optional Version version = 12;
```

```
255         repeated int32 celt_versions = 13;
256         optional bytes address = 14;
257         optional uint32 bandwidth = 15;
258         optional uint32 onlinesecs = 16;
259         optional uint32 idlesecs = 17;
260         optional bool strong_certificate = 18 [default = false];
261 }
262
263 message RequestBlob {
264         repeated uint32 session_texture = 1;
265         repeated uint32 session_comment = 2;
266         repeated uint32 channel_description = 3;
267 }
```