Mumble Protocol

Release 1.2.5-alpha

Jun 20, 2017

Contents

1 Contents **1**

1.1 Introduction . . . 1
1.2 Overview . . . 1
1.3 Protocol stack (TCP) . . . 1
1.4 Establishing a connection . . . 3
1.4.1 Connect . . . 3
1.4.2 Version exchange . . . 5
1.4.3 Authenticate . . . 5
1.4.4 Crypto setup . . . 6
1.4.5 Channel states . . . 6
1.4.6 User states . . . 6
1.4.7 Server sync . . . 7
1.4.8 Ping . . . 7
1.5 Voice data . . . 7
1.5.1 Packet format . . . 8
1.5.2 Codecs . . . 10
1.5.3 Whispering . . . 11
1.5.4 UDP connectivity checks . . . 11
1.5.5 Tunneling audio over TCP . . . 11
1.5.6 Encryption . . . 11
1.5.7 Variable length integer encoding . . . 12

CHAPTER 1

Contents

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 It reflects the state of the protocol implemented in the Mumble 1.2.8 client and might be outdated by the time you are reading this. Be sure to check for newer revisions of this document at http://mumble-protocol.readthedocs.org/.

This document is a constant work in progress.

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.

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

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.

Protocol stack (TCP)

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

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

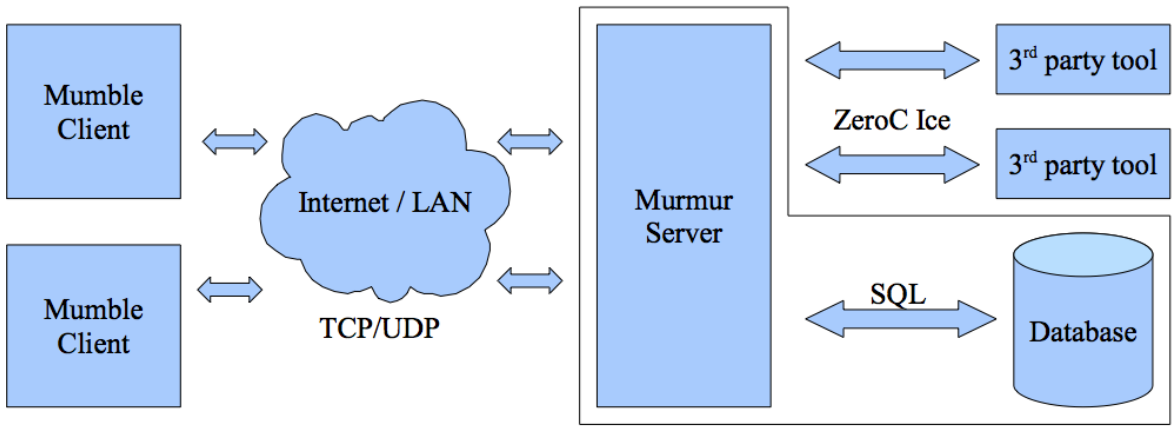

Fig. 1.1: Mumble system overview

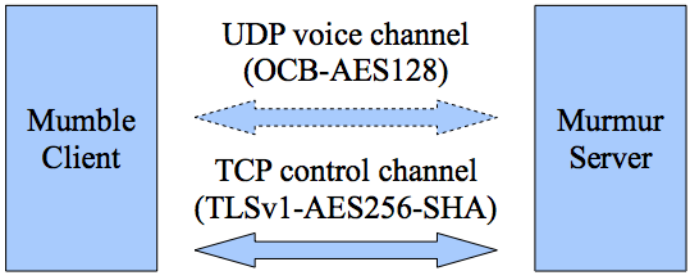

Fig. 1.2: Mumble crypto types

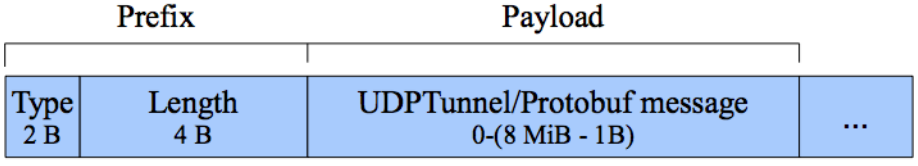

Fig. 1.3: Mumble packet

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

Table 1.1: Packet types

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

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

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.

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.

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

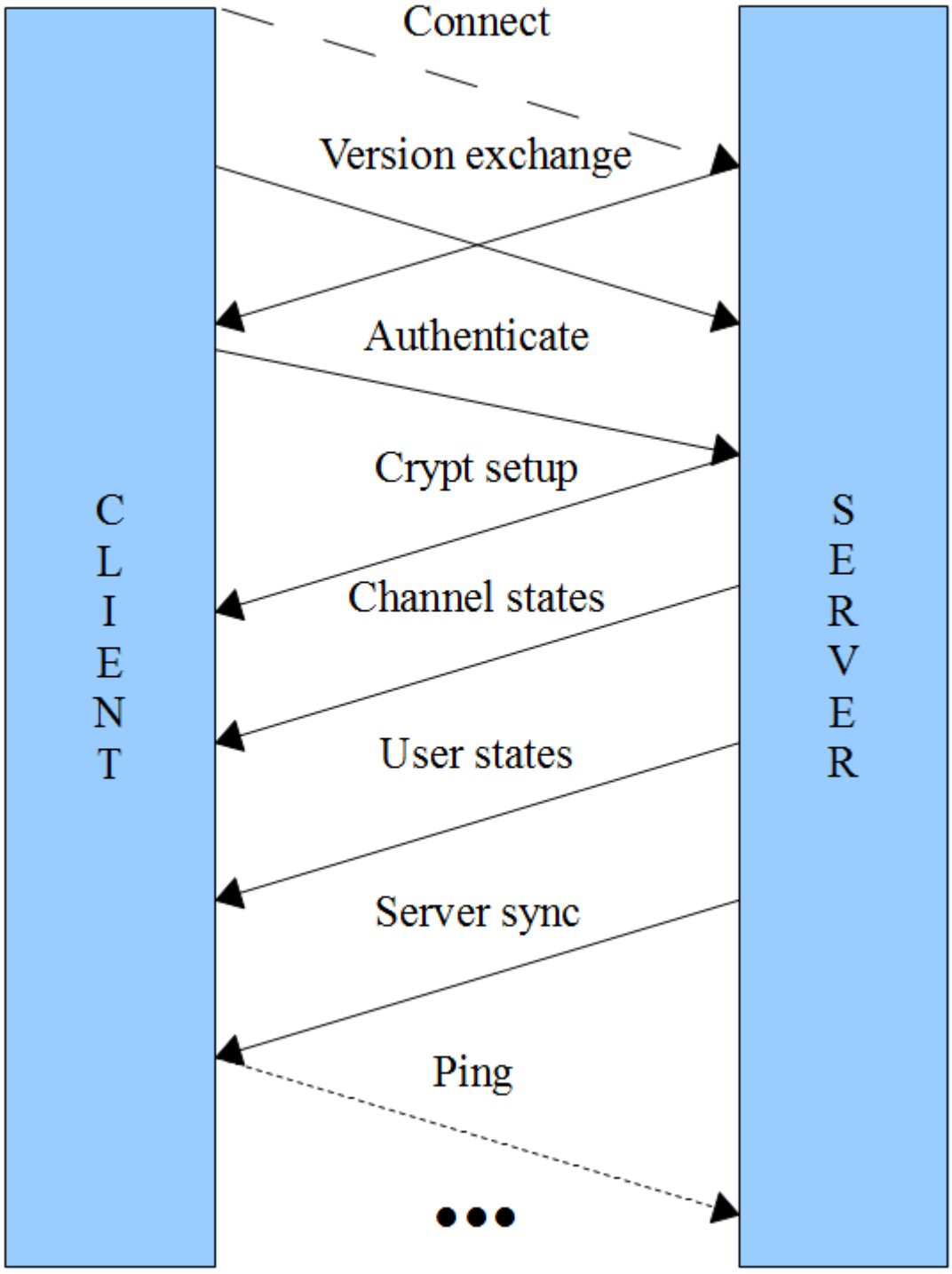

Fig. 1.4: Mumble connection setup

Version exchange

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

Table 1.2: Version message

Version	
version	uint32
release	string
os	string
os_version	string

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

Table 1.3: Version field encoding (uint32)

Major	Minor	Patch
2 bytes	1 byte	1 byte

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

Table 1.4: Mumble version differences

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

Authenticate

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

Table 1.5: Authenticate message

Authenticate	
username	string
password	string
tokens	string

The username and password are UTF-8 encoded strings. While the client is free to accept any username from the user the server is allowed to impose further restrictions. Furthermore if the client certificate has been registered with

the server the client is primarily known with the username they had when the certificate was registered. For more information see the server documentation.

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

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

Crypto setup

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

Table 1.6: CryptSetup message

CryptSetup	
key	bytes
client_nonce	bytes
server_nonce	bytes

Channel states

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

Table 1.7: ChannelState message

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

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

User states

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

Table 1.8: UserState message

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

Server sync

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

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

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.

Voice data

Mumble audio channel is used to transmit the actual audio packets over the network. Unlike the TCP control channel, the audio channel uses a custom encoding for the audio packets. The audio channel is transport independent and features such as encryption are implemented by the transport layer. Integers above 8-bits are encoded using the *Variable length integer encoding*.

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

Packet format

The mumble audio channel packets are variable length packets that begin with an 8-bit header field which describes the packet type and target. The most significant 3 bits define the packet type while the remaining 5 bits define the target. The header is followed by the packet payload. The maximum size for the whole audio data packet is 1020 bytes. This allows applications to use 1024 byte buffers for receiving UDP datagrams with the 4-byte encryption header overhead.

Table 1.9: Audio packet structure

Audio packet structure							
7	6	5	4	3	2	1	0
`type`			`target`				
Payload...							

type The audio packet type. The packets transmitted over the audio channel are either ping packets used to diagnose the transport layer connectivity or audio packets encoded with different codecs. Different types are listed in *Audio packet types* table.

Table 1.10: Audio packet types

Type	Bitfield	Description
`0`	`000xxxxx`	CELT Alpha encoded voice data
`1`	`001xxxxx`	Ping packet
`2`	`010xxxxx`	Speex encoded voice data
`3`	`011xxxxx`	CELT Beta encoded voice data
`4`	`100xxxxx`	OPUS encoded voice data
`5-7`		Unused

target The target portion defines the recipient for the audio data. The two constant targets are *Normal talking* (`0`) and *Server Loopback* (`31`). The range 1-30 is reserved for whisper targets. These targets are specified separately in the control channel using the `VoiceTarget` packets. The targets are listed in *Audio targets* table.

When a client registers a VoiceTarget on the server, it gives the target an ID. This voice target ID can be used as a target in the voice packets to send audio to specific users or channels. When receiving whisper-audio the server uses target 1 to specify the audio results from a whisper to a channel and target 2 to specify that the audio results from a direct whisper to the user.

Table 1.11: Audio targets

Target	Description
`0`	Normal talking
`1-30`	Whisper target • VoiceTarget ID when sending whisper from client. • 1 when receiving whisper to channel. • 2 when receiving direct whisper to user.
`31`	Server loopback

Ping packet

Audio channel ping packets are used as part of the connectivity checks on the audio transport layer. These packets contain only varint encoded timestamp as data. See *UDP connectivity checks* section below for the logic involved in the connectivity checks.

Table 1.12: Audio transport ping packet

Field	Type	Description
Header	`byte`	`00100000b` (`0x20`)
Data	`varint`	Timestamp

Header Common audio packet header. For ping packets this should have the value of 0x20.

Data Timestamp. The packet should be echoed back so the timestamp format can be decided by the original sender - the only limitation is that it must fit in a 64-bit integer for the varint encoding.

Encoded audio data packet

Encoded audio packets contain the actual user audio data for the voice communication. Incoming audio data packets contain the common header byte followed by varint encoded session ID of the source user and varint encoded sequence number of the packet. Outgoing audio data packets contain only the header byte and the sequence number of the packet. The server matches these to the correct session using the transport layer information.

The remainder of the packet is made up of multiple encoded audio segments and optional positional audio information. The audio segment format depends on the codec of the whole audio packets. The audio segments contain codec implementation specific information on where the audio segments end so the possible positional audio data can be read from the end.

Table 1.13: Incoming encoded audio packet

Field	Type	Description
Header	`byte`	Codec type/Audio target
Session ID	`varint`	Session ID of the source user.
Sequence Number	`varint`	Sequence number of the first audio data **segment**.
Payload	`byte[]`	Audio payload
Position Info	`float[3]`	Positional audio information

Table 1.14: Outgoing encoded audio packet

Field	Type	Description
Header	`byte`	Codec type/Audio target
Sequence Number	`varint`	Sequence number of the first audio data **segment**.
Payload	`byte[]`	Audio payload
Position Info	`float[3]`	Positional audio information

Header The common audio packet header

Session ID Session ID of the user to whom the audio packet belongs.

Sequence Number Audio data sequence number. The sequence number is used to maintain the packet order when the audio data is transported over unreliable transports such as UDP.

The sequence number might increase by more than one between subsequent audio packets in case the audio packets contain multiple audio segments. This allows the packet loss concealment algorithms to figure out how many audio frames were lost between two received packets.

Payload Audio payload. Format depends on the audio codec defined in the Header. The payload must be self-delimiting to determine whether the position info exists at the end of the packet.

Position Info The XYZ coordinates of the audio source. In addition to sending the position information, the user must be using a positional plugin defined in the `UserState` message. The plugins might define different contexts which prevent voice communication between users in other contexts.

Speex and CELT audio frames

Encoded Speex and CELT audio is transported as individual encoded frames. Each frame is prefixed with a single byte length and terminator header.

Table 1.15: CELT encoded audio data

Field	Type	Description
Header	`byte`	length/continuation header
Data	`byte[]`	Encoded voice frame

Header The length of the Data field. The most significant bit (`0x80`) acts as the continuation bit and is set for all but the last frame in the payload. The remaining 7 bits of the header contain the actual length of the Data frame.

Note the length may be zero, which is used to signal the end of a voice transmission. In this case the audio data is a single zero-byte which can be interpreted normally as length of 0 with no continuation bit set.

Data Single encoded audio frame. The encoding depends on the codec `type` header of the whole audio packet

Opus audio frames

Encoded Opus audio is transported as a single Opus audio frame. The frame is prefixed with a variable byte header.

Table 1.16: Opus encoded audio data

Field	Type	Description
Header	`varint`	length/terminator header
Data	`byte[]`	Encoded voice frame

Header The length of the Data field. 16-bit variable length integer encoded length and terminator bit value. The varint encoding is the same as with 64-bit values, but only 16-bit unencoded values are allowed.

The maximum voice frame size is 8191 (`0x1FFF`) bytes requiring the 13 least significant bits of the header. The 14th bit (mask: `0x2000`) is the terminator bit which signals whether the packet is the last one in the voice transmission.

Note: In CELT the "continuation bit" in the header defines whether there are more audio frames in the current packet. Opus always contains only one frame in the packet. In CELT the voice transmission end is signaled with a zero-byte CELT packet while in Opus we have a dedicated termination bit in the header.

Data The encoded Opus data.

Codecs

Mumble supports three distinct codecs; Older Mumble versions use Speex for low bitrate audio and CELT for higher quality audio while new Mumble versions prefer Opus for all audio. When multiple clients with different capabilities communicate together the server is responsible for resolving the codec to use. The clients should respect the server resolution if they are capable.

If the server resolves a codec a client doesn't support, that client is free to use any codec it prefers. Usually this means the client will not be able to decode incoming audio, but it can still send encoded audio out.

The CELT bitstream was never frozen which makes most CELT versions incompatible with each other. The two CELT bitstreams supported by Mumble are: CELT 0.7.0 (CELT Alpha) and CELT 0.11.0 (CELT Beta). While CELT 0.7.0 should technically be supported by most Mumble implementations, some servers might be configured to force Opus

codec for the users. Mumble has had Opus support since 1.2.4 (June 2013) so it should be safe to assume most clients in use support this now.

Whispering

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

UDP connectivity checks

Since UDP is a connectionless protocol, it is heavily affected by network topology such as NAT configuration. It should not be used for audio transmission before the connectivity has been determined.

The client starts the connectivity checks by sending a *Ping packet* to the server. When the server receives this packet it will respond by echoing it back to the address it received it from. Once the client receives the response from the server it can start using the UDP transport for audio data. When the server receives incoming audio data over the UDP transport it can switch the outgoing audio over to UDP transport as well.

If the client stops receiving replies to the UDP pings at some point, it should start tunneling the voice communication through the TCP tunnel as described in the *Tunneling audio over TCP* below. When the server receives a tunneled packet over the TCP connection it must also stop using the UDP for communication. The client should still continue sending audio ping packets over the UDP transport in case the UDP connection is restored and the communication can be switched back to it.

Tunneling audio over TCP

If the UDP channel isn't available the voice packets can be transmitted through the TCP transport used for the control channel. These messages use the normal TCP prefixing, as shown in figure *Mumble packet*: 16-bit message type followed by 32-bit message length. However unlike other TCP messages, the audio packets are not encoded as protocol buffer messages but instead the raw audio packet described in *Packet format* should be written to the TCP socket verbatim.

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

Implementation note

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.

Encryption

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

Variable length integer encoding

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

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

Table 1.17: Varint prefixes

Encoded	Decoded
`0xxxxxxx`	7-bit positive number
`10xxxxxx` + 1 byte	14-bit positive number
`110xxxxx` + 2 bytes	21-bit positive number
`1110xxxx` + 3 bytes	28-bit positive number
`111100__` + `int` (32-bit)	32-bit positive number
`111101__` + `long` (64-bit)	64-bit number
`111110__` + `varint`	Negative recursive varint
`111111xx`	Byte-inverted negative two bit number (`~xx`)