Vilundo Protocol Specification

Table of contents

Preface

Several chatting protocols have been introduced over the years, some of which are elegant, while others involve unnecessary verbosity. Vilundo is an attempt to introduce a simple protocol that allows primitive chat operations such as chatting in channels, in privates, kicking, banning, and access lists, while at the same time removing verbosity by limiting the data exchanged by all parties via various methods, including caching and packing. At the same time, it is designed to be reliable by making sure all messages exchanged are delivered to their recipients.
On the other hand, it limits complexity by not including abilities such as multiple synchronized servers on the same network, file sharing, or other unnecessary features.

Definitions

Indication of Requirement Levels

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Servers

Vilundo, aiming at being a simple protocol, only allows one server per network. That single server is denoted as "*server*" in this specification.

Clients

All programs connecting to a server are denoted as "*clients*" in this specification. All clients connect directly to the server for communication, and no direct peer-to-peer communication is allowed between the clients, for the sake of preserving user privacy by not exchanging user IPs.

Connection termination

A connection can either be *hard*- or *soft*-terminated by either the server or the client. A hard-termination consists of a TCP/IP connection termination. On the other hand, a soft-termination consists of ignoring all data initiating from the other party, while waiting for the other party to acknowledge soft-termination (it will always be indicated by a special packet sent by the party soft-terminating the connection). After either party receives any packet indicating a soft-termination, it should hard-terminate the connection. A hard-termination SHOULD occur from the party initiating the soft-termination after a timeout of 60 seconds, if the other party does not close the TCP/IP connection.

Byte order

The byte order used in this specification is Big-endian. As such, it requires the major bytes to be sent before the minor ones. Hence, if we want to exchange, for example, the 4-bytes-long integer 2781693, we shall send ASCII 0 followed by ASCII 42, followed by ASCII 113, followed by ASCII 253.

Alphanumerosity

An alphanumeric string sequence in this specification shall indicate any string containing the latin characters a – z, A – Z, the digits 0 – 9 the symbols ".", ",", "!", "@", "#", "$", "%", "^", "&", "*", "`", "~", "_", "-", "+", "=", "/", "?", """, "'", "[", "]", "(", ")", "<", ">", ":", and ";", the space symbol and the character ASCII 13 indicating a line return. This limits these strings so that they cannot contain, for instance, the tab character, or ASCII 10.

Users

A *user* is the person using a client, and, within this specification, we refer to them, allowing the server to use their preferred method of authentication and registration. This protocol does not specify a way for registering and authenticating users itself, but this SHOULD be handled by the server. Therefore, user, depending on your server infrastructure, might refer to a user record in a database table of users.
For conciseness, users are referred between the server and a client by using a unique four-bytes identifier, denoted as "*userid*" throughout this specification. The userid is always greater than zero. Therefore, if, for instance, the server wants to disclose to a client the fact that a different user has joined a room, it will only disclose the userid of the joining user. Additional information about the user can be disclosed to the client, if required, with ways as such specified later in this document.
A userid MUST be unique for each user, even between different connections. Hence, if a specific user logs in twice on two different days, they MUST have the same userid on both cases.

User Privileges

Users in Vilundo have certain server-wide privileges. There are no room-specific privileges. There are four different levels of privileges that a user can have. A banned user is disallowed to connect to the server and should be handled the same way a soft-banned IP address is (see below). A normal user has no special privileges. A moderator has certain increased privileges. An administrator has the highest privileges possible, that should be able to allow them to handle all matters occuring on the server. A fifth special type of privilege is the developer privilege that can be temporarily obtained to handle special matters, but should not be used in a regular basis. It is up to the server to determine a method for assigning, modifying, and saving these privileges for users.

Banning

Access to Vilundo services provided by a specific server can be revoked by banning a given IP address or user. There are two types of bans, *hard*- and *soft*-bans. Hard banning consists of directly refusing the TCP/IP connection initiating from a certain IP address. Soft banning consists of disconnecting the client after a user-friendly message has been delivered, before any communication can actually take place. Soft-bans are applicable on either users or IP addresses, while hard-bans can only apply on IP addresses and hostnames. Soft-bans are RECOMMENDED, because a hard-ban does not clarify that a certain user has been banned. It is up to the server to establish a method for banning users, and this should be unrelated to the Vilundo protocol.

Authtokens

After a user logs in, using a server-specified method unrelated to this chat protocol, they should acquire a so-called 16-bytes authtoken that will be able to identify them, given along with a userid, for the current session. The authtoken MUST be greater than zero (i.e. some bytes can be zero, but not all 16 bytes at the same time can).

Rooms

Rooms are virtual places where users can join together to chat as groups at the same time. Each room has a unique name, and a unique identifier, denoted as “roomid” throughout this specification. A room can exist even when no users are in a room. Several users can join one room at the same time. For compacity, rooms are referred between the server and the client by using their two-bytes-long roomid. Roomids are always greater than zero. A user MUST be in at least one room at a given moment. The server can require certain user privileges to allow a user to join a room, if desired. The server might also want to disallow certain users from joining specific channels, no matter their privileges, if desired. It is up to the server to determine a method for banning users from rooms, and it should be unrelated to the Vilundo protocol (for example, a configuration file, or a web interface could be used).

Packets

Each block of bytes denoting a specific event, action, or data transaction is referred-to in this specification as a packet. All packets are sent between the server and the client sequentially. There is no separation characeter sequence for distinguishing packets. Rather, the end of each packet is either packet-type-specific or the packet has prespecified length. There are several different packet types; they can be either server-to-client or client-to-server. Most

packet types are identified by a two-bytes integer identifier, which is sent at the beginning of each packet, called the *packetid*.
At the beginning of the transmission of each packet, the two bytes of the packetid are sent. This is the *header* of the packet, and no further references will be made in further sections. Additional packet-specific data is then sent. Hence, if later in this specification we state that a packet is an *empty* packet, we essentially mean that it only contains a header; while if we state that a packet contains only a userid, we essentially mean that it contains the packet header followed by the userid. Throughout this specification, packets are referred-to by the packetid enclosed between hashes using four digits, the first two of which refer to the major byte in hexadecimal format, while the latter two refer to the minor byte (for example #0001#). Some packets, referred-to as *special* throughout this specification have no packetid. They can be identified because they are sent on special occasions, such as, for instance, immediately after another packet, or at the beginning of a connection.

Checksums

When referring to a *checksum* of a string message of any length in this specification, it shall mean a four-byte integer number, the 32-bit CRC checksum of the string.

Connecting

Accepting

The Vilundo server listens for connections on one predefined TCP port. To connect, a client initiates a connection from a user that has already logged in and obtained an authtoken. The server can either refuse the connection if it is initiating from a hard-banned address instantly, without sending back a message to the client, or accept it and handle soft-bans later.

Initiating

As soon as a connection is established, the client MUST send a client-to-server initiative packet to the server. This is a special packet. The initative packet consists of only two bytes that are always the same and constant, to make sure both parts are using the Vilundo protocol; these two bytes are "V" and "L" respectively. Upon recieving a client-to-server initiative packet, the server MUST reply with a server-to-client that is essentially the same, again as a Vilundo protocol identifier. Right after that, the server MUST start a version handshake.

Version handshaking

Version handshaking is an essential step before any Vilundo packets can be exchanged, that determines the version of the protocol used. It is a series of version proposals between the server and the client. A version proposal packet is again a special type of packet and requires no packetid. The server starts by proposing a version of the protocol that SHOULD be the most recent version of the protocol supported by the server. The client then either accepts the proposal, or counter-proposes a version. As counter-proposals occur, they can either be accepted, or counter-counter-proposed until an agreement occurs. All counter-proposals MUST be of older versions of the protocol. Following this scheme, if all other versions are incompatible, it will yeild to falling back to version 1.0 of the protocol. A proposal packet consists of two bytes, the first of which indicates the major version of the protocol, while the second indicates the minor version of the protocol (following our byte order convention). For

instance, for version 1.0, the bytes for the proposal will be ASCII 1 followed by ASCII 0. An acceptance occurs if the other party replies with a proposal packet of the same version. Take for instance the following scheme, where the server starts by proposing the use of version 3.0 of the protocol, then the client proposes to fallback to version 2.5, then the server proposes to fallback to version 2.0, followed by the client accepting:

1. SERVER: 0300
2. CLIENT: 0205
3. SERVER: 0200
4. CLIENT: 0200

As soon as a version handshaking is complete, the two parties can start exchanging normal packets.

Client identification

After a version handshaking is complete, the client MUST send a client identification packet. This is again a special packet. It consists of 2 to 255 bytes that contain an alphanumeric string sequence that contains the client software name and version. Following that, an ASCII 0 character MUST be send.

Server identification

After a client identification is received by the server, it MUST send a server identification packet. Exactly as the client identification packet, this is a special packet. It consists of 2 to 255 bytes that contain an alphanumeric string sequence that contains the server software name and version. Following that, an ASCII 0 character MUST be send.

Authenticating

Client authentication

The client authentication is a special kind of packet. It consists of a userid at the beginning, followed by an authtoken of 16 bytes. The server MUST either reply with an authentication failure packet, or with a #0002#. Each client should authenticate right after it has received a valid server identification packet.

Authentication failure

An authentication failure packet is a special packet sent from the server to the client after a client authentication packet if the authentication fails. A so-called soft-ban might be applicable at this point. It starts with one byte, ASCII 255 and is followed by another byte indicating the reason of failure. It can be one of the following:

- ASCII 0: Authentication details are incorrect. The authtoken and userid combination is invalid.
- ASCII 1: Your IP, userid, or both have been soft-banned from the server.
- ASCII 2: The server is unable to currently handle more connections, as it has reached the connections limit.
- ASCII 3: Another error has occurred that does not allow you to connect at the moment.

After that, the server SHOULD soft-terminate the connection.

MOTD

#0001# Request MOTD

The client can re-request a #0002# at any time by sending a #0001#. This packet is empty (i.e. It contains just the two bytes describing the packetid).

#0002# MOTD

After the client has authenticated successfully, the server MUST reply with a #0002# MOTD (message of the day) packet. Following the two packetid bytes (these two bytes will not be specifically indicated in the following non-special packets), an alphanumeric message from 0 to 1024 bytes long, that can contain server rules, and other important notices MUST be send. An ASCII 0 byte follows, indicating the end of the MOTD packet. After the client receives the first #0002# it MUST send a #0003# packet, as a user can only be connected to the server if and only if they have joined a room. The server SHOULD NOT accept any packets from the client if they have not joined a room, apart from #0003# (for instance, private messaging SHOULD NOT be possible). An 0-length MOTD indicates that no MOTD is specified, or that it is currently unavailable.

Joining and Leaving

#0003# Request Join

The client can request to join a room by sending a #0003# at any moment. This packet contains two bytes, the roomid of the target room.

#0004# Joined

When the current user joins a room, or when a user joins a room the current user is in, the server MUST send a #0004# packet. This packet consists of 6 bytes, the first 4 being the userid of the joining user, while the latter 2 being the target roomid. If the packet contains your own userid, it should, of course, be handled specially by the client.

#0005# Join Failure

This packet is sent by the server to the client to indicate inability for the user to join a given room, after the user has attempted to join it. It is followed by two bytes indicating the roomid and another byte indicating the reason, which can be one of the following:

- ASCII 0: The room doesn't exist.
- ASCII 1: You do not have enough privileges to join the given room.
- ASCII 2: You have been banned from joining the given room.
- ASCII 3: The room is unable to handle any more users, as it has reached the users limit.
- ASCII 4: Another error has occurred that does not allow you to join at the moment.
- ASCII 5: The user is already in the given room.

#0006# Request Leave

The client can request to leave a room by sending a #0006# at any moment. This packet contains two bytes, the roomid of the target room.

#0007# Left

When the current user leaves a room, or when a user leaves a room the current user is in, the server MUST send a #0007# packet. This packet consists of 6 bytes, the first 4 being the userid of the leaving user, while the latter 2 being the target roomid. If the packet contains your own userid, it should, of course, be handled specially by the client.

#0008# Leave Failure

This packet is sent by the server to the client to indicate inability for the user to leave a given room, after the user has attempted to leave it. It is followed by two bytes indicating the roomid and another byte indicating the reason, which can be one of the following:

- ASCII 0: The room doesn't exist.
- ASCII 1: You have been disallowed to leave the given room.
- ASCII 2: Another error has occurred that does not allow you to leave at the moment.
- ASCII 3: You are not in the given room.
- ASCII 4: This is the only room you are in. You cannot leave this room.

Disconnecting and Checking Connection

#0009# Request Disconnect

The client can request to be disconnected by sending a #0009# at any moment and soft-closing the connection. This packet contains one reason byte, which can be one of the following:

- ASCII 0: The user decided to quit.
- ASCII 1: A fatal client error has occurred, requiring the client to disconnect.

After receiving this packet, the server should hard-disconnect and send a #0007# packet to all the users in the rooms the disconnecting user was on.
The server can also request the client to disconnect by sending a #0009# and soft-closing the connection at any moment. This packet contains one reason byte, which can be one of the following:

- ASCII 128: The connection was killed by a moderator.
- ASCII 129: The user was banned.
- ASCII 130: The server is experiencing an overload, and therefore is terminating connections.
- ASCII 131: The server is being upgraded. Please reconnect in a few minutes.
- ASCII 132: A fatal server error has occurred, requiring the server to disconnect.

#000a# Request Ack

This packet can be sent from either the server to the client or from the client to the server to verify that the other party is still connected. It consists of two bytes, a choice of the sender, that are used to identify the packet. If no identification is needed, they can be two ASCII 0 bytes. Upon receiving such a packet, the other party SHOULD reply with a #000b#. It would be wise, when developing both a server and a client, to set an interval of inactivity after

which an Ack Request must be made. If so, this interval should be adjusted by a small amount of randomness to avoid sending two Ack Requests if both the intervals of the server and the client are set to the same value. If no reply is received for a given #000a#, either peer can hard-close the connection after a predetermined timeout.

#000b# Ack

An #000b# packet MUST only be sent as a reply to a #000a#. It MUST consist of two bytes, the same bytes as received by #000a#. If a #000B# packet is received without a former #000a# having been sent, the party receiving the packet SHOULD ignore the incoming packet.

Requesting Information

#000c# Request Userinfo

Since users are referred-to by their userids, it is often necessary for the client to obtain more information about a specific user. This information SHOULD be cached by the client for the current connection, and SHOULD NOT be requested twice for the same user. To request information about one or more users, the client should send a #000c#. It contains four bytes for each user for which information needs to be obtained (their userids), followed by four ASCII 0 bytes. The number of userids grouped in the same packet MUST NOT exceed 16 (a total of 70 bytes maximum, including the header and sentinel).

#000d# Userinfo

After receiving a #000c#, the server should reply with a number of #000d# packets, one for each user requested. The packet starts with four bytes indicating the userid in question. The next byte is ASCII 255 if the user does not exist, or if requesting userinfo about the particular user has been disallowed and the packet ends with that. If not, it indicates the moderation permissions of the user, and can be one of the following:

- ASCII 0: The user is banned.
- ASCII 10: No special information exists about this user.
- ASCII 30: The user is a moderator.
- ASCII 50: The user is an administrator.
- ASCII 60: The user has temporary developer access.

If the user was found, the alphanumeric username of the user followed by ASCII 0 follows. Normal users should only be able to obtain userinfo about a given online user, while moderators and above should be able to obtain information about any user, even offline. The server might want to decide to conceal the privileges of a user for additional security.

#000e# Request Roominfo

Since rooms are, just like users, referred-to by their roomids, it is often necessary for the client to obtain more information about a specific room. This information SHOULD be cached by the client for the current connection, and SHOULD NOT be requested twice for the same room. To request information about one or more rooms, the client should send a #000e#. This contains two bytes for each room for which information needs to be obtained (their roomids), followed by two ASCII 0 bytes. The number of roomids grouped in the same packet MUST NOT exceed 8 (a total of 20 bytes maximum, including the header and sentinel).

#000f# Roominfo

After receiving a #000e#, the server should reply with a number of #000f# packets, one for each room requested. The packet starts with two bytes indicating the roomid in question. The next byte is ASCII 255 if the room does not exist, or if requesting roominfo about the particular room has been disallowed. If not, it indicates the permissions required to join the room and can be one of the following:

- ASCII 0: The permission level required cannot be revealed.
- ASCII 10: No special permissions are required to join this room.
- ASCII 30: Only moderators can join.
- ASCII 50: Only administrators can join.

The alphanumeric room name followed by ASCII 0 follows. If the room name cannot be revealed or the room does not exist, the room name is the empty string. It is to be determined by the server security settings whether room names and existence for rooms requiring higher privileges should be revealed to non-privileged users.

#1000# Request user list

This packet can be used by the client to request a listing of users who are in a specific room. This packet can be used to request listing users for more than one rooms, if desired. This information SHOULD be cached by the client while the user is in a room, and SHOULD be updated accordingly with joins and parts. Generally, requesting user listings can be done directly after receiving a #0004# packet on the client-side. To request a user list for one or more rooms, the client should send a #1000#. This contains two bytes for each room for which a listing needs to be obtained (their roomids), followed by two ASCII 0 bytes. The number of roomids grouped in the same packet MUST NOT exceed 8 (a total of 20 bytes maximum, including the header and the sentinel).

#1001# User list

After receiving a #1000#, the server should reply with a number of #1001# packets, one for each room requested. The packet starts with two bytes indicating the roomid in question. The next byte can take one of the following values:

- ASCII 0: The full list of users for this room is available and will follow.
- ASCII 1: You do not have permission to list the users of this room.

In case of ASCII 1, this should indicate the end of this packet. In case of ASCII 0, a list of users will follow. This contains four bytes for each user in the room (their userids), followed by four ASCII 0 bytes. Notice that this packet may take a few more milliseconds to deliver than others, as it might grow larger than usual packets if the number of users in the room is remarkably large.

Overdosing

Overdosing in Vilundo refers to informing other parties that the current user is typing a message, either in a private window, or in a room. A user is considered to start typing when they press a key while the room or private window (or any other type of GUI) is focused, but they haven't already started typing. The user is considered to stop typing when 5 seconds have passed since the last key was pressed.

#0010# Request Overdose

When a user starts or stops typing inside a private or room window, the client SHOULD send a #0010#. The first byte of this packet MUST be either of the following:

- ASCII 0: The user starts typing inside a room.
- ASCII 1: The user stops typing inside a room.
- ASCII 8: The user starts typing within a private.
- ASCII 9: The user stops typing within a private.

If that byte is 0 or 1, the roomid follows. If that byte is 8 or 9, the target userid follows. If a user is continuously typing for more than 30 seconds, an overdose packet with the start flag set SHOULD be resent, indicating that the user is still typing.
The server SHOULD then distribute the overdose to either the users present in a room, or the target private user.

#0011# Overdose

The server SHOULD share overdoses with remote parties of privates as well as with users inside a room. The server can decide not to share an overdose if it takes place in a very popular channel, to limit server load. If so, it should refrain from disclosing both the start-typing and the stop-typing packet, not just the start or the stop packet.
An overdose packet starts with one byte indicating the type of overdose:

- ASCII 0: A user starts typing inside a room.
- ASCII 1: A user stops typing inside a room.
- ASCII 8: A user starts typing within a private.
- ASCII 9: A user stops typing within a private.

Four bytes indicating the user typing follow. After that, if the first byte was 0 or 1, the roomid follows.
When a client receives an overdose packet, it should notify the user with an appropriate GUI. If no overdose packet with the start bit set during the next 40 seconds, or if an overdose packet with the stop bit set is received, the indication should disappear.

Messaging

Messaging is the core part of the Vilundo protocol. Vilundo makes it possible to deliver messages safely and compactly.

p2p through Server

Private conversations cannot be performed directly with a TCP/IP connection between clients; they have to go through the server for security, since that will avoid requiring to disclose the IP address of either party to the other. It is also so that no client can exploit another client in case they have some security leak, by sending invalid data. In addition, it solves all problems clients might have being behind routers, by not requiring direct connections apart from only one. It can also be used for server-side logging.

Verifiability

Vilundo defines a way to confirm that a message has been delivered successfully. This is done by confirming that messages have been delivered at every node. Messages are delivered from

peer to peer only and verified only on each bone. Assume, for instance, that a client sends a message to the server, which then delivers the message to another client. There are two messages to be sent, one from the client to the server, and one from the server to the client. The verification will occur right after each of those messages is delivered. Hence, if the last message fails, the initial client will not be informed. This is done to avoid getting trapped into infinite loops of corrupted messages. However, the message is verified on each node, hence making sure it goes through the predetermined path. That way, if a message is lost during one transmission, the parties will have the ability to resume and resend the message, providing they cache accordingly.

Message Ids

Each message exchanged between a specific client and the server is identified by a two-bytes-long messageid. Messages can either be client-to-server or server-to-client; this is the *type* of a message. Messageids assigned to messages of different type are distinct. Hence, if two messages of different types are assigned the same messageid, they should be distinguished as separate messages.
The first message of each type SHOULD have messageid 1. Each next message SHOULD have messageid the previous messageid of the same type plus one. When any messageid reaches 65535, the next messageid of the same type SHOULD be, again 1. Messageid 0 MUST NOT occur. Keeping that in mind, it is possible for two different messages of the same type to have the same messageid. If each peer follows the recommendations in this specification, however, they will not occur near each other, avoiding conflicts.
Upon receiving a message from any client, the server MUST deliver the message to either the other peer of a private conversation, or to all users that have joined a channel.
Let us illustrate all this with an example, in an attempt to make it clearer.
Assume two clients are connected, and their two users, namely A and B, are attempting to have a private conversation. Assume that both have already sent and received some messages, and therefore their last messageids are:

- A's last sent messageid is 39.
- A's last received messageid is 7819.
- B's last sent messageid is 571.
- B's last received messageid 1780.

Now assume that B wants to send a private message to A. B creates a message with the next available id, 572, and sends it to the server. The server then delivers the message to user A, assigning it the next available id, 7820.

#0012# Send Private Message

A #0012# is sent by a user who wants to send a private message. The packet starts with the target userid followed by the messageid and, after that, the alphanumeric message, followed by an ASCII 0.

#0013# Private Message Sent

As soon as a server receives a #0012# it MUST send a #0013# and deliver the message to all recipients through #0015#. The packet contains only the messageid of the sent message.

#0015# Receive Private Message

The #0015# packet is deployed to the other peer of a private conversation by the server, right

after a #0013# is sent back to the initial sender. A #0015# starts with the userid of the sender followed by the messageid, and, after that, the alphanumeric message. Following is an ASCII 0 byte that indicates the end of the alphanumeric message. Notice that the messageid used at this point might not be the same as the one used when sending the message.

#0016# Private Message Received

Upon receiving a #0015#, the client MUST send a #0016# back to the server. The client can then display the message to the user. The packet contains only the messageid of the received message.

#0018# Send Room Message

A #0018# is sent by a user who wants to send a room message. The packet starts with the target roomid followed by the messageid, and, after that, the alphanumeric message. Following is an ASCII 0 byte that indicates the end of the alphanumeric message.

#0019# Room Message Sent

As soon as a server receives a #0018# it MUST send a #0019# and deliver the message to all recipients through #001b#. The packet contains only the messageid of the sent message.

#001b# Receive Room Message

The #001b# packet is deployed to the other peers of a room (i.e. the users currently in a room) by the server, right after a #0019# is sent back to the initial sender. A #001b# starts with the userid of the sender followed by the roomid followed by the messageid, and, after that, the alphanumeric message. Following is an ASCII 0 byte that indicates the end of the alphanumeric message. Following that, a checksum of the message indicates the end of the packet. Notice that the messageid will be different for each recipient and also different from the initial messageid used to send the message.

#001c# Room Message Received

Upon receiving a #001b#, the client MUST send a #001c# back to the server. The client can then display the message to the user. The packet contains only the messageid of the received message.

TODO

- Idle time of a user?
- Ability for a user to have an "away" status?
- Ability for a user to set a personal message?
- Encryption of authtokens and messages?
- Exchange of key through Diffie-Hellman?
- Verification of server identity?
- Ability for the server to inform the client of roominfo or userinfo changes.
- Unicode support.
- Server messages (ads etc.).
- Be able to determine the gender of a user as part of userinfo.

- Optimize initial handshake procedure.

License