The Delegate Protocol

First Edition

Version: 1.0.0
Last Edited: 10/22/22

Preface

The Delegate Protocol has been again rewritten, due to reasons of losing the heavily modified documents that were new editions, as well as for reasons of making the protocol more concise. Many aspects of the protocol will remain the same, but differing design choices will be implemented in this new release.

The version should be defined by a major number (such as 1) with its minor number (such as 0), then the patch number, all delimited by periods. A differing major number will indicate that the protocol has changed such that cross-compatibility is impossible.

This protocol describes the communication method between the server and the client (and the libraries, etc); however, it may also describe ways in which the server should be designed, for maximum efficiency, albeit this will be lackluster in order to ensure that this document is not painfully long. Additionally, this document will only describe the text-oriented side of Delegate, where any voice/video features will be described in another document, along with other spin offs of Delegate, such as theoretical mail servers, captcha servers, file upload servers, etc.

Editing & License

The author can make whatever changes he wants to the protocol, as of now, since it is in a heavy development stage—requiring immediate changes and modifications. The creator may change the governing system of the changing of this protocol at any time.

This protocol may be modified and forked, but it cannot be called the Delegate Protocol. In addition to that, any edits or forks of the Delegate Protocol must state that they are a derivative work of the Delegate Protocol.

Introduction

The Delegate Protocol describes a systems of communication similar to that of Discord, Matrix, and IRC where the server is centralized—though with Delegate being fully open source, in contrast to Discord—such that there are private communications, as well as community-oriented communications, where IRC has channels and Discord has servers.

Delegate attempts to emulate services such as Discord in a subtle way, emulating many of its modern—relative to IRC—features in an open source manner; however, by no means do we wish to obtain even a diminutive fraction of their user base, as Discord serves the common person's needs quite well. Delegate is intended for those who—perhaps irrationally according to some—fear the consolidation of freedom by centralized and proprietary tech, along with the censorship that inevitably follows. Delegate also establishes a respect on privacy by default, not requiring phone numbers, but instead finding other ways to validate users, albeit with a more tedious nature whilst preserving privacy—arguably, a good tradeoff.

Delegate is meant to be self-hosted, within a small community of friends or like-minded individuals. Due to this fact, many server implementations (and, possibly, the protocol itself) will not be particularly scalable or efficient for many thousands of users, which is okay. However, due to how abstract and idiomatic the protocol is, this can be ameliorated by more efficient server implementations. A user-base we also wish to establish ourselves in is the developer support group community, where the verbose features of Delegate may prove useful for those creating communities surrounding development and whatnot.

Unfortunately, decentralized services that lack the infrastructure to provide adequate moderation are prone to developing a radicalized and/or illicit user base—where Delegate is not exempt from this ubiquitous outcome. Delegate will not, in the grand scheme of things, contribute to a large portion of this possible behavior, considering that it already is found abundantly on other services, such as IRC and Discord.

Delegate works off of a Client-Server model, where the clients may connect to the server, where the server will serve as a *delegate* in order to send any messages to other people who are connected to that server or other servers within the network. Though, modules will be available for Delegate implementations to have REST-apis for those who find them necessary. Delegate, though, is based off of JSON completely, therefore sharing similarities to that of an HTTP request.

Delegate is not a distributed system like Matrix. It is instead centralized to a specific network with servers managing the requests. It is such because some networks may wish to be segregated from others. It is also such because this level of centralization makes it easier for administrative tasks. However, if you do not like your Delegate network, you can always make your own as the protocol and its implementation are open source.

In spite of this, a Delegate client can have access to a directory containing a list of public servers, along with a protocol for automatically registering an account for those servers, given information stored on the client. In this way, Delegate servers can be more akin to IRC or game servers.

Delegate recognizes that there is a need for privacy on the internet, through the usage of Tor and other methods of obscuring one's identity. Tor and other mechanisms to secure one's anonymity are supported in the Delegate protocol itself, with provisions being made to allow those kinds of users, albeit with more captchas and systems of verifications. A Delegate server or network can be hosted off of a hidden service, as it shall be made to support doing so.

Delegate is a text-based protocol, specifically designed for communication via text; however, it has the capabilities to incorporate video, audio, and other external services. Delegate sets up the medium for which these services can be given to users, through an abstraction known as *streams*. Not only can audio and video be implemented through the usage of *streams*, but also games and alike, considering how abstract the concept is. The Delegate Protocol provides the abstract concepts that can form a cohesive community.

Networking

The Delegate Protocol uses WebSockets for communication with a Delegate server. The Delegate Protocol also encourages usage of WebSockets with TLS, which is known by WWS. A Delegate server will also expose certain HTTP(s) endpoints for auxiliary usage. These should be hosted off of specific ports, which are listed below.

- WWS (WebSockets with TLS): **9999**
- WS (WebSockets plaintext): **9998**
- HTTPS endpoint: **9997**
- HTTP endpoint: **9996**
    - HTTPS and HTTP endpoints use the HTTP(x) protocols. More on their usage shall be described in the *HTTP Endpoints* section of this protocol.

A standard certificate authority can be used; alternatively, networks may declare themselves certificate authorities, to prevent potential hijacking concerns. Of course, TLS is not required when using Tor—a protocol which has built-in encryption, not relying on certificate authorities to ensure that a secure connection is not being tampered with. Additionally, Delegate holds the philosophy that external parties shall not be entrusted with the task of end-to-end encryption: two or more direct parties shall implement that themselves, for the servers cannot be trusted to provide end-to-end encryption between users! More on that later, though.

Delegate will require two WebSockets connections to the endpoint: one for sending/receiving commands and response codes (in a more synchronous fashion), and another for receiving events solely (in a more asynchronous fashion). These connections both contribute to the maximum number of connections that a user account can have concurrently; they are not counted differently. However, an event connection cannot connect if there is not already an initial normal connection; it will fail with *E_USER_EVENT*.

Users

Users are essentially the fundamental building block of Delegate, where they represent a list of sockets that a person or a channel may message to. Unlike IRC, users must be registered using commands that will be described below. A client will connect to a Delegate server, and they will be unable to execute most other commands while being kept-alive, until they register and sign in.

Users may be deleted after they are registered, but it should be known that these users were deleted, and their usernames should be permanently banned from ever being recreated in order to prevent impersonations. The server should also delay the deletion process by a certain amount specified at their own volition—we recommend 1-week.

In contrast to IRC yet again, multiple connections may be held on a single user account, but the maximum should be defined as 3—perhaps more, if the server allows for it. It is to also be noted that a *connect* event should be broadcasted upon a connection to a user, which would alert others of a possible hacking attempt.

Users may have *settings* which they can use to store useful information about themselves, but also for the purposes of exposing information about the user to others. There is an entire section dedicated to the purposes and the abstractions that settings will provide.

The constraints for the usernames may vary from different server implementations and their unique settings, but in general it should be kept that: usernames must be a minimum of **4** characters and a maximum **32** characters in length; they must also adhere to the REGEX rule of **[a-zA-Z0-9 _.]**.

Users may message each other privately or partake in groups called channels, identical to that of a Discord server and similar to that of an IRC channel. This will generate an event called *message* which will be heavily described in the following sections, namely *Events*.

2FA is available through TOTP 2FA, but is by default disabled. A user wishing to enable 2FA on their account will use the *2fa* command, which will return a secret key and backup codes. After such a command is run, a 2FA code will be required in addition to a password on each login. 2FA can be disabled again by toggling the *&2fa* user setting.

Bot Accounts

Bot accounts are user accounts; user accounts may also be used for botting purposes. Unlike Discord, we do not care about whether a user account is being controlled by a bot, logging messages or acting as a chatbot—we only care about rate limiting any kind of malicious behavior. A user account may become a bot account when passing a flag to the *uregister* command, which will set the setting *bot* to true indefinitely on that user account.

Something which is **important** to note is that bot accounts will have their usernames prefixed with *bot_* after the account is registered as a bot. You do not need to provide the name prefixed upon registration, but all interactions with that bot account must include the prefixed *bot_*, namely when signing into the bot account. This is to make a clear separation between users and bots and to protect the username space from that of bots.

Bot accounts will have their desired permissions conveyed through the usage of the *perms* setting. Bots may not be messaged by other users, nor have private interactions with them, for bot accounts are expressly for channel purposes. Bots also have other stipulations placed upon them, especially with regards to differences in rate limiting compared to regular accounts.

Channels

As aforementioned, channels are congregations of users who wish to talk with each other in a contained space, similar to that of servers on the Discord platform. They bear permissions and roles, settings, and many commands useful for moderating and keeping peace. Users must register channels before they can join them, where they will receive the role *owner* upon doing so—and a channel may only have one owner!

Channels may be broken down into subunits called *subchannels*, as to organize the various topics that may arise in the server, as well as provide containment areas for certain users. These must be created by people who bear the permissions to do so. The main subchannel is called *main* and it is, in fact, the center of the entire server, as it will always exist and cannot be deleted.

The textual format for notating channels and their subchannels is ***#channel/subchannel***. Channel names may only be **4-64** characters long, where their subchannels have a limit of **2-192** characters in length; channels and subchannels have a REGEX rule of (which specifically allows for Base64 encoded text):

[a-zA-Z0-9+\/=-]

- This is important for allowing encrypted channels to exist—they may want to encrypt their subchannel names, so we should allow them to do so. This is allowing for Base64 encoded cipher text.

Channels do not require—in fact, do not allow—hashtags in their name, as opposed to IRC, but it is useful for notation.

Channels may also have settings. Additionally, the users within the channel also bear their own settings specific to that channel for moderation purposes. All of these will be described in their respective section of this document, *Settings*.

Channels may have their attributes queried by their queryables—settings that are relationally queryable for the purposes of finding channels with a specific criteria. This is in contrast to Discord where servers are typically not standardly findable, especially by not many metrics. Again, these will be further elaborated on in its respective section, *Queryables*.

Group Channels

There is a need for something called group channels, but we should not bloat our protocol by making them separate. Essentially, group conversations allow users to be within a group, but without additional subchannels or many settings; additionally, the client may want to make notification settings different from that of channels, which is where group conversations come in (e.g., notification of call or message).

In order for a channel to become a group conversation, you must use the *cregister* command with the *group* field set to true. This will automatically initialize the server with the immutable *$group* setting, which will make certain commands not work (e.g., that of subchannel commands).

The only subchannel that will exist within the group channel will be called *main*, as is the default subchannel created on every channel. You may not add or remove subchannels within the group channel, and the setting of certain channel parameters pertaining to subchannels will yield an error or do nothing.

Roles & Permissions

Permissions within a channel allow certain users to execute administrative tasks, but are not typically given manually unless for bots. Instead, roles will be used, where they are collections of permissions under a name, ranked by a hierarchy, so that they may not kick, ban, mute, or change the role of those higher than them, in a very similar fashion to Discord.

Channel permissions are what allow for users of a channel to have certain privileges. Roles within a channel are a collection of these permissions. There are default roles (e.g., administrator, moderator, etc), with custom ones which can be created by channel owners.

The roles in a channel are event-based: if a custom role has been renamed, the server will send an event to all users of the channel that it has been renamed, so the client can update its information. Role information is additionally sent when querying the users who are within a channel.

Unlike Discord, users may not be under multiple roles. Roles are only for establishing a permission hierarchy in Delegate. If one wishes to label a user as being a specific kind of member within the community, without any permissions or administrative functions associated with that, then the User-Channel setting of *labels* or *~labels* shall be used.

The list of channel permissions is as follows, with their string representations and descriptions:

I. **talk - Talk in a channel**
II. **read - Read messages within a channel**
III. **remove - Remove messages within a channel**
IV. **subchannel - Add or remove subchannels**
V. **metadata - Change channel/subchannel description or image or other metadata.**
VI. **set - Set channel/subchannel settings.**
VII. **kick - Kick users**
VIII. **ban - Ban users**
IX. **mute - Mute users.**
X. **role - Set the role of users**
XI. **invite - Invite other users**
XII. **password - Change the join password, if it exists.**
XIII. **order - Can order roles, setting the role hierarchy.**
XIV. **vote - Can vote. (Experimental; do not implement as of now)**
XV. **cast - Can cast a vote (start one). (Experimental; do not implement as of now)**
XVI. **summon - Summon a bot to the channel.**
XVII. **admin - Is equivalent to having every permission, besides that of channel deletion, which only the owner has.**

Subchannel Permissions

Subchannel permissions are permissions that are specific to a subchannel. A subchannel may define a role to have these specific permissions within its premises. Subchannel permissions are *in addition* to permissions that are globally available. For example, if a user already has the *talk* and *read* permissions, they will automatically have them in the subchannel; however, if one were to give them the *remove* permission within the subchannel, it would only be available within that subchannel, given that they do not have the *remove* permission on a channel-wide basis already.

The list of them is as follows, in a similar format to the channel-wide permissions displayed above:

I. **talk - talk in that subchannel**
II. **read - read in that subchannel**
III. **remove - remove a message in that subchannel**
IV. **set - set a setting in that subchannel.**
V. **vote - can vote in that subchannel. (Experimental; do not implement as of now)**
VI. **cast - can cast a vote in that subchannel. (Experimental; do not implement as of now)**
VII. **metadata - can change subchannel metadata attributes.**

The default role allotted to new users is “default” which include the following permissions in channel—and as a result, in the subchannels:

I. talk,
II. read

Owners of a channel will have the “owner” role, which will include all permissions described in this protocol above. Only channel owners will be allowed to delete the channel. This role of “owner” cannot be changed, unless the owner transfers ownership of the channel to another person.

If the owner of a channel is banned from the server, deletes their account, or leaves their channel, then the *heirs* channel setting will determine who becomes the next owner. If the setting *heirs* is blank or the server is unable to transfer ownership based off of it, then the server will fallback on the following options, in order of success:

I. It will get the users of the highest ranking role (see role order for reference) and give the one who received the role the earliest the role “owner”.
II. If this fails, for some reason, then the person who joined the channel the earliest—not considering their outstanding history throughout the channel, just the time of the join—shall receive the role of “owner”.

Unless otherwise specified through channel configuration, roles and default roles will have the same permissions within the specific subchannels. These may be changed by using the *role* command on a subchannel.

Auditing

Channel auditing is an important aspect in ensuring integrity among channels. Channel audits are *always* available to everyone (not merely owners or moderators) so that reasonable cause can always be verified. Channel audits record whenever a person of power within a channel moderates others. Channel audits may not be deleted or modified.

Channel audits will include the fields *by* (the person who did it) and *timestamp* (the time at which it was done, in the format of a UNIX timestamp, of course). Below are the actions which will be audited and their respective fields. The top portion will refer to the value which goes under the *audit* field. Audits will be returned by a response code (S_CHAN_AUDIT).

ban — Someone banned another.
{
who: who got banned,
message: the banning message,
duration (int): for how long (-1 forever)
}

unban — Someone unbanned another.
{
who: who got unbanned,
message: unbanning message
}

kick — Someone kicked another.
{
who: who was kicked,
message: the kicking message
}

mute — Someone muted another.
{
who: who was muted,
duration: for how long,
subchannel: which subchannel
}

role — Roles were changed.
{
prev: {*roles and their respective permissions*},
new: {*roles and their new respective permissions*}
}

order — Role order was changed.
{
prev: [*order of roles*],
new: [*new order of roles*]
}

remove — A message was removed.
{
uuid: what message,
who: whose message,
subchannel: in what subchannel
}

subchannel — Subchannel creation/deletion
{
subchannel: what subchannel,
delete: was it deleted or created?
}

set — A setting was changed.
{
setting: what setting,
prev: previous value,
new: new value,

subchannel: if present, it is a subchannel setting,
username: if present, it is a channel-user setting
}

Government

This section of the Delegate Protocol is overengineered, so it is subject to heavy modification or outright removal. Do not implement this section, for it may disappear.

Channel administration may be democratically held to a certain degree. There are certain configurations to this: one configuration may include the administrator or those with the permission *cast* to be able to cast a vote for a specific administrative action to be taken on the server, where those with the permission *vote* are able to vote.

Voting is limited to one vote per IP address, username, and/or unique identifiers. A fundamental limitation to the Matrix protocol is that IP verification is made exponentially more difficult due to the nature of the protocol. However, Delegate can prevent abuse by IP. This is also an advantage over Discord as well, since Discord does not incorporate voting into its server or protocol, thus one must utilize a bot to handle voting which cannot verify the integrity of the vote.

Commands

Commands are the method for which the client may interface with the server to tell it what it should do, within some constraint. As previously mentioned, commands are in the format of JSON and most of them may not be issued whilst not signed into a valid user account.

In the JSON object, the *command* field should represent the commands which will be listed below. Any commands which are available without sign in will be marked with an asterisk (but are not actually when sent).

An exception to the asterisk notation is abundantly clear: password-protected servers. When a server is password-protected, no commands other than *authenticate* will be available, instead erroring with *E_PASSWORD* until the correct password is passed into the *authenticate* command.

A sequence number (or string) is to also be associated with each command, as to be able to keep track of responses. An example JSON object which shall convey a command issuance:

```
{
        "command": a command name which will be described in this section,
        "seq": the sequence number (or string) of the command, for keeping track of things,
        … (more data which will depend on the command being issued)
}
```

All fields within the JSON object, unless explicitly stated, are of the string type, and if the fields are required and not optional, they will be marked with an asterisk. We hope you have fun reading these commands.

Server Commands

*quit — Close the server connection cleanly. There are no arguments needed or to be had.
*ping — Ping the server to test your connection or to avoid a timeout. No arguments needed.

- Returns:
- S_PING when ping was done successfully.

*get — Obtain server settings—rather, constants, since they are not modifiable by anyone other than the server administrator.

```
{
        *settings: [array of settings to get]
}
```

- Returns:
- S_SETTING when settings are obtained successfully, which also contains the values of the settings returned.

*getall — Get all server settings/constants

- Returns:
- S_SETTING which contains all available server settings/constants, in the format as described above for the *get* command.

*authenticate — If the server is password protected, enter a password to gain entry.
{
    *password: the password (what else?)
}

- Returns:
- S_PASSWORD when the server is password-protected and the password is correct.
- E_INVPASSWORD when the server password inputted above is incorrect.
- E_NONAPPLIC when this command is issued and the server isn't password-protected.

notifications — Get notice of previously unread messages or of certain events.
{
    *unread_only (bool): retrieve unread events only,
    *peek (bool): retrieve unreads but do not mark them as read if *true,*
    *page_len (int): how many entries to get per iteration,
    timestamp (int): get messages before this,
}

- Returns:
- S_NOTIFICATIONS upon success; no failure code makes sense.
- By issuing this command, the unread messages and the unread events will be marked as read, unless the *peek* argument is *true*.
-

reportmsg — Report a message for being against the terms of service of the server.
{
    *origin: *"channel"* or *"user"*,
    username: the username if origin "user",
    channel: the channel if origin "channel",
    *uuid: the UUID of the message
}

User Commands

*user — Sign in.
{
    *username: the username which you wish to sign in with,
    *password: the user's password,
    *event (bool): are you signing in as an event listener?,
    *stream (bool): are you signing in to listen to a stream?,
    2fa: the universal OTP 2fa code, if applicable
}

- Returns:
- S_USER_LOGIN when done successfully,
-
- E_USER_PASS on password authentication failure,
- E_USER_EVENT if signing in as an event listener but no corresponding regular connection,
- E_USER_NOENT if the user is not registered,
- E_USER_2FA if TOPT 2FA authentication failed,
- E_USER_MANY if too many users are connected.
    - The server shall dictate what is too many in this context.
- This command will send a *login* event to all people currently signed in with the username above, EXCEPT for the new event listener if this command is being used that way.

logout — Sign out of a user account.
{
	*all (bool): log out your current connection or all others on the account?
}

- This command will also work on an event listener connection.
- Returns:
- S_USER_LOGOUT when successful

*uregister — Register an account.
{
	*username: username to register,
	*password: password for the user account,
	*bot: true for bot account, false for user account.
}

- Note: the client should check whether the passwords match!
- Returns:
- S_USER_REG when successful,
-
- E_USER_EXISTS if user is already registered,
- E_USER_WEAK if password is weak,
    - We recommend a minimum password length of 16 characters. We additionally recommend that the password have an acceptable amount of entropy. The password should ideally be randomly generated from a password manager of the user's choice—NOT EVER by the server—so that even if databases are leaked, the password will be very difficult to crack.

    - To those who are making implementations of the Delegate Protocol: implement password security correctly! Use *argon2* or a similarly slow, expensive hash function to hash passwords; this will make it more expensive for people to crack the passwords. Consider adding a secret server-wide passphrase to all passwords when hashing them, as to make it more difficult to crack the hashes.
- E_USER_LONG if username is too long,
- E_USER_REGEX if username violates REGEX,
- E_USER_RESV if username previously existed
- E_USER_LIMITED if you are limited on making new accounts

2fa — Set up TOPT 2FA on the user account. Additionally, this will reset the stored secret, if one wishes to do that.

- Returns S_USER_2FA, which returns the secret required for 2FA code generation, as well as backup codes. Seek *Responses/User* for more information regarding this.
- The user setting *&2fa* will become true. To disable 2FA, the user setting must be set to false.
- If 2FA is already enabled, this command will reset the shared secret required for 2FA code generation—be really careful!

upasswd — Change or modify the password of your user account.

{

*prev: the previous password,

*new: the new password

}

- Note: the client should check if the new passwords are correct!
- Returns:
- E_USER_PASS if the previous password is incorrect.
- E_USER_WEAK if the password is less than *user_pass_len* (default: 8) characters.

udelete — Delete a user account

{

*password: the password for the user
}

- Note: the client should ask for confirmation!
- Returns:
- E_USER_PASS if password is incorrect

uexists — Does an account exist? Did it ever exist, and was it banned, or deleted?
{
*username: the username,
}

uget — Get settings from a user
{
*username: the user; if *null*, you're getting it from yourself,
*settings: [an array of settings to get]
}

uset — Set your user settings
{
*settings: {an object of settings to set}
}

- Returns:
- E_SET_TYPE if another object is set within a setting.

upriv — Set specified settings to visible or private
{
*settings: {an object where the settings correspond to booleans: true if private, false if visible}
}

- For example,
- *{setting1: false, setting2: true … } — setting1* is visible, whereas *setting2* is private.

- There are exceptions: certain users may be whitelisted based off of some user settings; see *Settings/User Settings*.
- Note: only user settings which have no prefix in front of them can be affected by this command and feature.
- *E_SET_NOENT* will be returned if one of the settings does not exist.
- *E_CMD_OBJECT* will be returned if each key in *settings* does not have a corresponding boolean value.
- *E_SET_PREFIX* will be returned if attempting to private a user setting which has a prefix in front of it.
- Setting a setting which is already private to private will do nothing; similarly, setting a setting which is already visible to visible will do nothing. No worries.

uprivwhitelist — Put certain users or groups of people on a whitelist for any given private setting
{
    *settings:
    {
        “setting1”: [user1, user2],
        “setting2”: null,
        “setting3”: []
        …
    }
}

- In *setting1*, the users *user1* and *user2* are spared from the normally private setting; any arbitrary number of users who exist in Delegate may be put in here for the whitelist.
    - If any user does not exist within these arrays, then *E_USER_NOENT* will be raised, detailing what the problem was.
- In *setting2*, the value of *null* indicates that only friends shall be able to see the normally private setting.
- In *setting3*, the value of an empty array *[]* indicates that there was a previous whitelist for *setting3*, but we now wish to remove it and make it private to everyone. If attempting to delete a field that does not already exist, *E_SET_WHITEDEL* will be returned as an error.
- If any setting provided does not exist, *E_SET_NOENT* shall be yielded.
- If any setting provided is not already privated, *E_SET_NOTPRIV* will be thrown.
- If a key within the *settings* object does not have a corresponding value that conforms to the valid ones shown above, *E_CMD_OBJECT* will be returned.

usend — Message another user
{

*username: whom you are going to message,
*type: *null* for chatting purposes; “bot” for bot commands (or a custom one),
*message: what are you sending them
*format: message format (or, abstractly, metadata), if applicable (*null* if not present).
}

- Returns:
- E_USER_NOENT if username is not found,
- E_UMSG_LONG if the message was too long,
- E_USER_BLOCKED if the user has you blocked
- E_FORMAT_LONG if the format was too long,

frequest — Send a friend request to a user.
{
*username: user whom you do,
message: the message, if applicable.
}

- The request must comply with the user’s privacy rules (see Settings/User Settings).
    - *E_USER_FRIEND* if the friend request is rejected because:
        - The issuer fails to share a channel with the user in question.
        - Friend requests are disabled by the user in question.
- You must not be blocked by the user.
    - *E_USER_BLOCKED* if so.
- You must not already be friends with the user.
    - *E_DONT_CARE* if so.

friend — Accept or deny a friend request.
{
*username: the request from who,
*accept (bool): if true, accept the request; if not, reject it,
*notify (bool): if true, alert the user about the acceptance or denial; if false, don’t
}

- Returns:
- E_FREQ_NOENT if the friend request was not found.

unfriend — Unfriend another user
{
	*username: who to unfriend?,
	message: if applicable, a final message why
}

- Returns:
- E_FRIEND_NOENT if the user is not within your friends list.

block — Block or unblock a user
{
	*username: who to block/unblock,
	*block (bool): *true* to block, *false* to unblock if already blocked.
}

usubscribe — Subscribe/unsubscribe to a user's special settings.
{
	*username: the username in question,
	*subscribe: true or false (false to unsubscribe).
}

- Note: this is useful if you are not friends with a user, but still want to be updated on their status.
- Returns:
- E_USER_NOENT if the user does not exist.

umsgslen — Reports information about the number of messages available for query
{

}

umsgquery — Query messages from a user interaction.
{
	*username: the user in question,
	*query: the query as described in *Queryables/Messages*,
	*page_len: length of a page,
	timestamp: get messages before this (keyset pagination)
}

- *page_len* should be between 1-100 messages or a range dictated by the server.
- If *timestamp* is not provided, it will get the latest at most *page_len* number of messages.

- Returns:
- E_USER_LENGTH if the page length is outside of the bounds mentioned above.
- E_USER_NOENT if the user does not exist
- E

ueventquery — Query events from a user interaction.
{
	*username: the user in question,
	*query: what is your query?
}

utyping — State to a user that you are about to send a message to them; you're typing
{
	*username: to what username?,
	*typing (bool): have you begun typing (*true*) or have you just stopped (*false*)?
}

Channel Commands

cregister — Register a channel.
{
	*channel: name of the channel,
	*group (bool): should it be registered as a group channel?
}

- Returns:
- S_CHAN_REG on success,
- E_CHAN_EXISTS if the channel is already registered
- E_CHAN_LENGTH if the channel name is too long or too short
- E_CHAN_REGEX if the channel name violates the REGEX set

cdelete — Delete a channel.
{
	*channel: name of the channel
}

- Only the channel owner may issue this command.
- Returns:
- S_CHAN_DELETE on success
- E_CHAN_PERM if not owner,
- E_CHAN_NOENT if the channel does not exist
- E_CHAN_NIN if you are not in the channel

subchannel — Create or destroy a subchannel.
{
	*channel: channel in question,
	*destroy (bool): if this is false, we are creating; if it is true, we are destroying,
	*subchannel: the subchannel to be created or destroyed
}

- Requires the *subchannel* permission.
- You cannot destroy /main—it is the main subchannel, thereby the main hub.
- Returns:
- S_SCHAN_REG on successful subchannel creation,
- S_SCHAN_DELETE on successful subchannel deletion,
- E_CHAN_NOENT if the channel does not exist,
- E_CHAN_NIN if you are not in the channel,
- E_CHAN_PERM if you lack the permission(s),

- E_CHAN_MAIN if you are attempting to delete /main
- E_SCHAN_EXISTS if that subchannel already exists
- E_CHAN_GROUP if operating on a group channel.

role — Create or destroy a role within a channel or subchannel.
{
    *channel: channel in question,
    *role: name of the role,
    *destroy (bool): destroy it?

    If destroy is false,
    subchannel: if provided, what subchannel?
    *permissions: [the permissions that this role has]
}

- You may not give the role permissions that you, the issuer, do not have yourself.
- You may not modify a role that is higher than your current role.
- You may not change the default roles.
- Creating a role that already exists will overwrite it.
- You must have the permission *role*.
- When creating a role in the subchannel, it must exist in the parent channel. You are not making a new role, but rather specifying the permissions given for that role within that subchannel.
- Returns:
- E_CHAN_INSUB if you give permissions to a role that you do not have,
- E_CHAN_ROLE if you are modifying a role higher than you,
- E_CHAN_PERM if you do not have the permission *role*.
- E_CHAN_NIN if you are not within the channel
- E_CHAN_ROLE if you are modifying a default role.
- E_CHAN_GROUP if operating on a group channel.

order — Change the hierarchy of roles within the channel.
{
    *channel: channel in question,
    *roles: [order of roles]
}

- Leftward roles are more powerful than rightward roles.

    - Roles cannot moderate people of roles to the left of them.
- You may not change the places of your role or the roles above you currently.
- You must include all existing roles.
- You must have the permission *order*.
- Returns:
- E_CHAN_INSUB if you are changing the order of your role or of roles higher than yours,
- E_CHAN_ORDER if all existing roles were not provided.
- E_CHAN_PERM if you lack the permission of *order*.
- E_CHAN_NIN if you are not within the channel,
- E_CHAN_GROUP if operating on a group channel.

csend — Send a message within a channel and subchannel.
{
    *channel: the channel you wish to send to,
    *subchannel: the subchannel,
    *message: the message,
    *type: *null* for chat messages; “bot” for bot commands, etc
}

- The *type* parameter can be arbitrarily set.
- You may not send too many messages within a fast period (regulated by server).
- The message must be of proper length.
- Returns:
- E_CHAN_NIN if you are not in the channel,
- E_CHAN_NOENT if it does not exist,
- E_SCHAN_NOENT if the subchannel does not exist,
- E_CMSG_LONG if the message is too long
- E_CMSG_REGEX if in violation of REGEX rules

join — Join a channel.
{
    *channel: name of channel,
    message: message upon joining if allowed and complying with server and channel message regulations.
    password: if enabled on the channel, a join password is required for entry.
}

- You may not join a channel twice.
- Returns:

- S_CHAN_JOIN if join is successful
- E_CHAN_NOENT if channel does not exist,
- E_CHAN_BANNED if you are banned from this channel,
- E_CHAN_PASS if the join password is enabled and is incorrect
- E_CMSG_LONG if the message is too long
- E_CMSG_REGEX if the message violates REGEX

kick — Kick somebody from the channel.
{
    *channel: name of the channel,
    *username: the name of that person,
    reason: the reason behind the kick
}

- You may not kick without the *kick* permission.
- You may not kick somebody with a higher role than you.
- You cannot kick yourself.
- Returns:
- E_CHAN_PERM if you lack permissions to do so,
- E_CHAN_INSUB if kicking someone higher than you,
- E_CHAN_NOENT if the channel does not exist,
- E_CHAN_NIN if you are not within the channel,
- E_USER_NOENT if the user is not within the channel,

mute — Mute/unmute someone from the channel.
{
    *channel: name of the channel,
    *username: the person to mute,
    *unmute: are you unmuting?

    If *unmute* is false:
    *duration: duration in minutes (-1 for forever),
    subchannel: if provided, mute them only in a specific subchannel,
    reason: the reasoning behind it
}

- Muting is not merely revoking the *talk* permission. This should be implemented separately.
- You may not unmute somebody muted by a higher role.
- You cannot mute yourself.
- Returns:
- E_CHAN_PERM if you lack the *mute* permission,
- E_CHAN_INSUB if muting someone higher than you or unmuting as described above.
- E_CHAN_NOENT if the channel does not exist,
- E_CHAN_NIN if you are not within the channel,
- E_USER_NOENT if the user does not exist

ban — Ban/unban somebody from the channel.
{
    *channel: the name of the channel,
    *username: whom to ban,
    *unban: are you unbanning?

    if *unban* false:
    *duration: time in seconds (0 for forever),
    *reason: the reason behind the ban (*null* if none)
}

- You must have the *ban* permission to ban.
- You cannot ban somebody of a higher role than you
- You may not unban somebody who was banned by a higher role than you.
- You cannot ban yourself.
- Returns:
- E_CHAN_PERM if you don't have permissions,
- E_CHAN_INSUB if you are kicking someone higher or unmuting as described above.
- E_CHAN_NOENT if the channel does not exist
- E_CHAN_NIN if you are not within the channel
- E_USER_NOENT if the user is not within the channel
- E_CHAN_GROUP if operating on a group channel.
- E_WTF if you try to ban yourself

For *ban* and *mute*, both the user account and the IP address are affected. This includes all IP addresses that are currently connected onto that user account when the administrative action is taken.

invite — Send invitation to a channel or group channel.
{
*channel: channel that you are sending an invite to,
*username: the user you are inviting,
message: invitation message.
}

- The issuer must have the *invite* permission in the channel.
- E_CHAN_GROUP if operating on a group channel.

summon — Summon a bot to the channel.
{
*channel: which channel,
*username: the username of the bot,
permissions: [permissions to manually give]
}

- The username in question must be corresponding to a bot account.
    - If not, E_NOT_BOT will be returned.
- The server will get the bot's setting *perms* to give by default if not manually given above.
- Only people with the *summon* permission may issue this command for a channel.
    - Lest, E_CHAN_PERM will be thrown as an error.
- You cannot summon a bot which would be granted more permissions than you have
    - Lest, E_CHAN_INSUB will be thrown as an error.
- E_CHAN_GROUP if operating on a group channel.
- If *permissions* is provided as an argument, if there is a permission passed which does not exist, then E_CHAN_NOPERM will be raised as a consequence.

gsummon — Summon a user to the group channel/conversation.
{
*channel: the group channel,
*username: the username in question,
*message: the message, if applicable (*null* if none)
}

- You must be friends with the username in question and of course, the username must exist.
    - E_USER_NOFRIEND is raised if you are not friends.
    - E_USER_NOENT if the user in question does not exist.
- A user may opt to have a confirmation of being summoned to a group channel.
- Most importantly, *channel* must be corresponding to that of a group channel
    - If not, then E_CHAN_GROUP will be thrown.

- E_CHAN_NOENT if the channel itself does not exist.

dup — Duplicate a subchannel
{
    *channel: the channel,
    *source: the subchannel to copy/duplicate,
    *destination: the name of the copy/duplicate
}

- E_CHAN_NIN if you are not within the channel.
- E_CHAN_PERM if you do not have the *subchannel* permission.
- E_SCHAN_NOENT if the *source* subchannel does not exist.
- E_SCHAN_EXISTS if the *destination* subchannel already exists.

transfer — Transfer ownership of a channel
{
    *channel: the channel,
    *username: the person to transfer it to (the person who shall get the role “owner”),
    *role: the role you want to become after losing ownership
}

- E_CHAN_NIN if the channel does not exist,
- E_CHAN_USER if the heir is not within the channel,
- E_CHAN_NROLE if the role does not exist,
- E_CHAN_NOWNER if you are not owner of the channel and cannot make this decision.

leave — Leave a channel.
{
    *channel: name of channel you wish to leave,
    *message: parting message, given it conforms like the join message (*null* if none)
}

- If an owner leaves a channel, the ownership will be transferred to the next heir, which is explained in channel settings.
- E_CHAN_NIN if you are not in the channel.
- E_CMSG_LONG if the message is too long

cget — Obtain channel/subchannel settings.
{
    *channel: name of channel,
    subchannel: if provided, these are subchannel settings,
    username: if provided, these are Channel-User settings we are obtaining,
    *settings: [array of settings to obtain]
}

- Note: subchannel and username are mutually exclusive. You cannot have both.
- Returns:
- E_CHAN_NIN if you are not within the channel,
- E_SCHAN_NOENT if the subchannel does not exist,
- E_CHAN_USER if the user is not within the channel.
- E_CMD_MUT if username and subchannel are both provided, due to mutual exclusivity in those two fields.

cset — Set channel/subchannel settings.
{
    *channel: name of channel,
    subchannel: if provided, these are subchannel settings,
    username: if provided, these are Channel-User settings we are setting,
    *settings: {object representing the settings to set}
}

- Requires the *set* permission.
- Note: subchannel and username are mutually exclusive. You cannot have both.
- Returns:
- E_CHAN_NIN if you are not within the channel,
- E_CHAN_PERM if you do not have the permissions required,
- E_CHAN_USER if the user does not exist,
- E_SCHAN_NOENT if the subchannel does not exist,
- E_CMD_MUT if username and subchannel are both provided.

cpriv — Set visibility of channel/subchannel settings.
{
    *channel: name of channel,
    subchannel: if provided, these are subchannel settings,
    username: if provided, these are Channel-User settings we are working with,
    *settings: {an object representing the visibility; refer to *upriv* for examples}
}

- Requires the *priv* permission.
- Note: subchannel and username are mutually exclusive. You cannot have both

ctags — Set the tags for the channel, for discovery.
{
    *channel: the channel in question,
    *tags: [*the tags of the channel*]
}

- Requires the *metadata* permission.
- You may not include the same tag twice.
    - E_CMD_DUP if this occurs.
- You may not have an empty list.
    - E_CMD_NULL is raised if this occurs.
- A list of tags may be obtained via a database registry provided by the *tagsregistry* server constant.

cquery — Query for channels by their queryables.
{
    *query: {object containing the queryables to search for}
}

- Note, the format for querying will be described in the respective section, *Queryables*.
- Compatible queryables will also be shown in that section.

cmsgquery — Query channel messages.
{
    *channel: the channel in question,
    *query: the query for the channel messages as described in *Queryables*
}

ceventquery — Query channel events
{
	*channel: the channel in question,
	*query: the query for the channel events
}

cauditquery — Query the audit log of a given channel.
{
	*channel: the channel in question,
	*query: the query for the channel audits
}

csubscribe — Subscribe/unsubscribe to *all* of the events of a channel.
{
	*channel: the channel in question,
	*yeah: *true* to subscribe; *false* to unsubscribe
}

- *All* special settings that could trigger an event within that channel will be sent to you.
    - This includes *all* channel special settings.
    - This includes *all* subchannel special settings.
    - This includes *all* user special settings, of those who are in a channel.
    - This includes *all* user-channel special settings.
    - This includes *all* user-subchannel special settings.
- This will not persist after the user has disconnected and has expired its state in the memory of the Delegate server implementation.

Events

Events are the method for which the Delegate protocol alerts all connected clients that something has occurred, be it a sign-in, a user or channel message, or a captcha. They were previously called “actions” but that is confusing and the word “events” makes it much clearer what the purpose of these are. There are events for multiple types of concepts within the protocol, such as user events, channel/subchannel events, and server events—we also make message events their own group since they are unique.

As specified in *Networking*, events are NOT sent through the connection that is for sending commands and for receiving the corresponding response codes. You must connect in such a way that your socket is only used for receiving events. The *user* command has an option for connecting as an event listener on a user account. In order to connect as an event listener, you must have a corresponding regular connection on that user account. If you do not, *E_USER_EVENT* will be yielded.

When events are sent out to an array of users (such as with a message being delivered to all those users and their variable amount of file descriptors within a channel), they should be sent out asynchronously, as order does not matter here.

All protocol-standard events will be defined in the field “event” with no preceding symbols or characters. Then, the bodies of those events will be further elaborated on in the sections that describe them. The Delegate protocol, like the Matrix protocol, allows the manifestation of custom events, but these must be prefixed with a @. Fields that are imperative will be marked with an asterisk (*), similar to commands.

Messages

message — A message was sent.
{
	*timestamp: the UNIX time at which the message was sent,
	*type: type of message (text message is *null*),
	*uuid: the UUID4 id generated,
	*origin: 0 for server, 1 for user, and 2 for channel
	*username: who sent it,
	*format: the format of the message
	*contents: the contents of the message,

	If type is 1

	channel: the channel of where the message came from,
	subchannel: and its subchannel
}

edit — A message was edited.
{
	*timestamp: when was the message edited,
	*uuid: which message,
	*type: 0 for user, 1 for channel,
	*format: the format,
	*contents: the new contents

	If type is 1

	channel: in what channel,
	subchannel: in what subchannel
}

delete — A message was deleted.
{
	*timestamp: when was the message edited,
	*uuid: which message,
	*type: 0 for user, 1 for channel,

	If type is 1

	channel: the channel,
	subchannel: the subchannel
}

For events *edit* and *delete*, the original messages shall be kept for logging purposes. The client should ideally update its displaying of messages to reflect the updates, but also allow viewing of the older messages which will be kept in the database. Of course, the client could simply choose not to care about the message events, which is perfectly acceptable—everyone should be wise about what they send, afterall!

typing — Somebody started or stopped typing.
{
	*type: 0 for user, 1 for channel,
	*username: who did,
	*is (bool): is typing or is not typing,

	If type channel

	channel: the channel,
	subchannel: and its subchannel
}

there — People are/are not currently watching.
{
	*type: 0 for user, 1 for channel,
	*usernames: [*people who are watching*],
	*are: are they watching (true/false),

	If are is true:
	*degree: 0 for glance, 1 for look, 2 for stare
}

read — People read the message.
{
	*type: 0 for user, 1 for channel,
	*usernames: [*people who read the message*],
	*degree: 0 for glance, 1 for look, 2 for stare
}

User

block — You have been blocked or unblocked by another user.
{
	*username: by whom,
	*block (bool): if true, you have been blocked; if false, you have been unblocked.
	message: the blocking or (weirdly, but allowed) unblocking message.

	If block is true
	duration (int): minutes
}

friend — Another user wishes to become friends with you.
{
	*username: who does?,
	message: their friend request message, if applicable
}

frequest — Something happened with the friend request you sent.
{
	*username: regarding who?,
	*accepted (bool): was it accepted or not?
}

- A user may opt to not show the fact that they denied your friend request!

login — Somebody logged into your user account.
logout — Another client disconnected from your user account.
upasswd — Your password has been changed and you will now be disconnected, but you should know something about the person who did it.

especial — Special settings were changed!
{
	*username: whose changed,
	*settings: {*setting1*: *value1*, *setting2*: *value2*, … }
}

Channel

join — Somebody joined the channel you're in.
{
	*username: who?,
	*message: their message, if it exists (*null* if none was provided)
}

subchannel — A subchannel has been deleted, created, or otherwise changed.
{
	*subchannel: which?,
	*status: "created", "deleted", or "changed"
}

leave — Somebody left the channel you're in.
{
	*username: who?,
	*message: if they provided one, their message (*null* if none was provided),
	*circumstance: "normal" (voluntarily left), "kicked", "banned"
}

banned — You were banned from the channel!
{
	*channel: in what channel,
	*username: by whom, if anonymous, then *null*,
	*duration (int): duration in minutes, if 0, then forever,
	reason: the reason,
	message: the ban message.

}

- Reasons are one-liners, whereas messages may be more in-depth; additionally, reasons are stored on the database.

muted — You were muted in a channel.
{
    *channel: what channel,
    *username: by whom, if anonymous, then *null*,
    *duration (int): duration in minutes, if 0, then forever,
    reason: the mute reason
}

kick — You have been kicked from a channel.
{
    *channel: what channel,
    *username: by whom, if anonymous, then *null*,
    reason: the kick reason
}

cmessage — A private message has been issued to you within the context of the channel.
{
    *channel: what channel,
    *subchannel: what subchannel,
    *username: by whom,
    *contents: the contents of the message
}

userrole — A user's role has changed!
{
    *channel: what channel,
    *username: whose,
    *prev: previous role,
    *new: new role.
}

role — A role's definition has changed!
{
    *channel: what channel,
    *prev: what role,

*permissions: *[permissions of the role]*,
*new: new name, if applicable.
}

- Note: if the role's new name is that of *null*, then it has been deleted.

cspecial — Special settings were changed!
{
*channel: whose changed,
*subchannel (bool): was it in a subchannel?
*settings (object): {*setting1*: *value1*, *setting2*: *value2*, … }
}

cdeleted — A channel was deleted!
{
*channel: what channel was deleted?
}

Subscriptions

To be added later

Responses

Responses are quite similar to events, but they seek to respond to commands and send that response only to the issuer of that command—or merely just to one single connection that is on a user, as opposed to events which send their information to all connected clients. Responses are used for success codes, error codes, and information returned from commands, treating them as if they were functions, and responses being the return value.

Responses will always come in the form of the following JSON body:

{
“code”: any of the codes described in this section,
“seq”: the sequence number of the response,
… (any other data, if any, which will be explained further below)
}

Success and error codes are numerical enumerations and have ranges for their respective context in which they describe. Success codes are positive integers, while error codes are negative ones. These responses may also have additional information attached to them which will be described. Responses also get a name, for references in this document, as well as assisting with creating the enumerations in a programming language library for Delegate. Note: as the protocol develops, it is okay if the enumerations break their definition of enumeration—because, guess what, negative numbers growing downwards aren’t exactly enumerations either!

Error codes are prefixed with E_; Success codes are prefixed with S_, followed by their respective section name—though, we may be brief sometimes if we can. They should be condensed and they should look as if they were constants and macros within the C programming language libraries—we in fact use *ENOENT* to describe something not existing! They must be brief, yet readable.

0 - Server

Successes*:*

0 - Server connection successful.	**S_OK**
1 - Ping successful.	**S_PING**
2 - Setting(s) successfully obtained	**S_SETTING**
{	
settings: { … }	
}	
3 - Authentication success. You are now allowed entry to the server.	**S_PASSWORD**
4 - You didn’t do anything wrong yet, but you need a password.	**S_PASSREQ**

5 - Notifications that you may have missed. **S_NOTIFICATIONS**

{

notifications: [

notification_object

]

}

- **If there are no notifications available, the *notifications* field will be that of *null*.**

A *notification_object* will be defined as follows:

{

"origin": "server" or "channel" or "user",

"type": "message" or "event",

the contents of such a message or event …

}

Errors:

-1 - Server is not open for usage. **E_CLOSED**

-2 - General exception occurred. **E_EXCEPTION**

{

exception: the exception that occurred (specific to language; is for debugging purposes),

message: the exception message

}

-3 - You are banned! **E_BANNED**

{

duration: how long in minutes, 0 if forever,

reason: what was your reason?

}

-4 - You have been kicked from the server. **E_KICKED**

{

reason: why

}

-5 - Something that wasn't JSON was sent to the server. **E_NOTJSON**

-6 - The command was too long. **E_LONG**

~~**-7 - Invalid or non-existent length header. (Raw TCP only)** **E_HEADER~~**

- **Deprecated.**

-8 - Invalid password. **E_INVPASSWORD**

-9 - Password required before issuance of any command. **E_PASSWORD**

-10 - Non applicable. **E_NONAPPLIC**

- **If a command is issued without the appropriate setting being turned on, it is issued uselessly. For example, if a user tries to input the server password, but the server is not password protected, we need something to scold them about.**

-11 - Parameter given was blank or null inappropriately. **E_NULL**

-12 - DIDN'T ASK; DON'T CARE **E_DONT_CARE**

- **This is given if a command issued by the user is just useless and not worth creating a unique error for. For example, if a user is already friends with a user, then they issue another friend request to that user, what do I say? I just don't care.**

-13 - Not implemented by the server implementation. **E_NOT_IMPLEMEN**

-14 - WHAT THE FUCK **E_WTF**

- **If an illogical, stupid thing is done by a user.**

100 - User

Successes:

100 - Successfully signed in. **S_USER_LOGIN**
101 - Successfully registered. **S_USER_REG**
102 - Log out. **S_USER_LOGOUT**

103 - User message info **S_USER_MSGINFO**
{
len (int): how many messages?
}

104 - 2FA enabled/changed **S_USER_2FA**
{
“secret”: ***the secret for TOTP 2FA in Base32 format,***
- ***Yes, you heard that right, Base32, not Base64.***

}

105 - User settings successfully obtained **S_USER_SET**
{
“username”: from who (*null* if from self),
“settings”: {
“setting1”: “value1”,
“setting2”: “value2”,
…
}
}

106 - User account successfully deleted. You will now be logged out. **S_USER_DELETE**

Errors:

-100 - User password incorrect. **E_USER_PASS**
-101 - Username exists; cannot be registered. **E_USER_EXISTS**
-102 - Weak password. **E_USER_WEAK**
-103 - Username length error. **E_USER_LENGTH**
-104 - Username violates REGEX. **E_USER_REGEX**
-105 - Username previously existed; it cannot be registered. **E_USER_RESV**
-106 - Username does not exist. **E_USER_NOENT**

-107 - User has you blocked. **E_USER_BLOCKED**
-108 - Maximum connections to the user reached. **E_USER_MANY**
-109 - You are prohibited from creating more user accounts. **E_USER_LIMITED**
-110 - Already signed in. Sign out to change accounts. **E_USER_IN**
-111 - Friend request not found. **E_FREQ_NOENT**
-112 - Not a bot account. **E_NOT_BOT**
-113 - Pertaining to this user account being that of a bot account. **E_BOT**
-114 - 2FA verification error or not provided. **E_USER_2FA**
-115 - Cannot become friends with this user. **E_USER_FRIEND**
-116 - Cannot send messages to this user. **E_USER_MESSAGE**
-117 - Friend does not exist. **E_FRIEND_NOENT**
-118 - Not subscribed to or already subscribed to. **E_USER_SUBSCRIB**
-119 - Cannot do that because not friends. **E_USER_NOFRIEND**
-120 - The event listener connection was not permitted. **E_USER_EVENT**

200 - Channel

Successes:
200 - Channel successfully joined. **S_CHAN_JOIN**
201 - Channel successfully registered. **S_CHAN_REG**
202 - Subchannel created. **S_SCHAN_REG**
203 - Subchannel destroyed. **S_SCHAN_DELETE**
204 - Channel successfully deleted. **S_CHAN_DELETE**

205 - Audit complete. **S_CHAN_AUDIT**
{
The audit body as described in Channel/Auditing
}

206 - List successful **S_CHAN_LIST**
{

}

Errors:
-200 - Channel does not exist. **E_CHAN_NOENT**
-201 - Channel already exists. **E_CHAN_EXISTS**
-202 - You are banned. **E_CHAN_BANNED**

-203 - You lack the permissions to do that. **E_CHAN_PERM**

{

permissions: [*the permissions you lack*]

}

~~**-204 - Weak master password.** **E_CHAN_MWEAK**~~

- **Deprecated**

-205 - Channel name length error. **E_CHAN_LENGTH**
-206 - Channel violates REGEX. **E_CHAN_REGEX**
-207 - Subchannel does not exist. **E_SCHAN_NOENT**
-208 - Subchannel already exists. **E_SCHAN_EXISTS**
-209 - Subchannel name length error. **E_SCHAN_LENGTH**
-210 - Subchannel name violates REGEX. **E_SCHAN_REGEX**
-211 - Trying to moderate a role higher than you. **E_CHAN_INSUB**
-212 - Channel is invite only. **E_CHAN_INVITE**
-213 - Channel requires a password—or you got it wrong. **E_CHAN_PASS**
-214 - Tor users aren't allowed! **E_CHAN_TOR**
-215 - Channel isn't allowing joins at the moment. **E_CHAN_LOCKED**
-216 - Weak join password. **E_CHAN_WEAK**
-217 - You are not in the channel. **E_CHAN_NIN**
-218 - You are in the channel, already. **E_CHAN_IN**
-219 - That password was incorrect. **E_CHAN_PASS**
-220 - Main cannot be deleted or changed like that. **E_CHAN_MAIN**
-221 - Order does not contain all fields. **E_CHAN_ORDER**
-222 - You may not modify those roles (i.e., default roles or higher roles). **E_CHAN_ROLE**
-223 - The user does not exist within the channel. **E_CHAN_USER**
-224 - An error relating to the specificity of group channels occurred. **E_CHAN_GROUP**
-225 - Role does not exist. **E_CHAN_NROLE**
-226 - You are sending messages too quickly for the subchannel/channel. **E_CHAN_FLOMSG**
-227 - You cannot do that: you are not the channel owner. **E_CHAN_NOWNER**
-228 - Permission does not exist? **E_CHAN_NOPERM**

300 - Commands

Errors
-300 - Invalid syntax; imperative arguments missing. **E_CMD_INVALID**
-301 - Command not found. **E_CMD_NOENT**

-302 - Invalid types passed. **E_CMD_TYPE**
-303 - Permission denied for that command (you must be server admin). **E_CMD_DENIED**
-304 - You are not signed in, so you cannot use that command. **E_CMD_USER**
-305 - Fields you provided are mutually exclusive. **E_CMD_MUT**
-306 - Duplicate fields. **E_CMD_DUP**
-307 - An empty value was provided. **E_CMD_NULL**
-308 - Object not used correctly; it goes too deep. **E_CMD_OBJECT**

400 - Settings

Successes
400 - Settings successfully obtained. **S_SET_GET**
{
settings: {*setting1*: *value1*, *setting2*: *value2*, … }
}

401 - Settings successfully set. **S_SET_SET**
{
settings: [*all settings that were successfully set*]
}

Errors
-400 - Some settings were private! **E_SET_PRIV**
{
settings: [*settings that were private*]
}

-401 - A scalar type is required. **E_SET_SCALAR**
-402 - An array type is required. **E_SET_ARRAY**
-403 - An object type is required. **E_SET_OBJECT**

For errors -401, -402, and -403, the response body will also include:
{
***settings*: [*where what type was required*]**
}

-404 - Immutable setting. **E_IMMUTABLE**

- **Note: it is evident which settings are immutable by default. This never changes, as it is foolish to allow the user to make settings immutable (why would they do that, to never**

change it again?). Therefore, we don't require a response detailing which settings were immutable.

-405 - Invalid type, for regulated settings. **E_SET_TYPE**

{

setting: name of the setting where the error occurred,
given: type given to the setting,
required: type that is required by the regulated setting

}

- **Setting types will be in Pythonic format and may be any of the following (in text):**
  - ***str***
  - ***list***
  - ***dict***
  - ***bool***
  - ***int***
  - ***float***
- **If the implementation of the Delegate protocol server does not want to support this, then *given* and *required* shall be *null* respectively.**

-406 - Mutually exclusive **E_SET_EXCLUSIVE**

{

settings*: [*which settings were mutually exclusive]

}

-407 - Not within enumeration. **E_SET_ENUM**

{

settings: {setting1: value1, setting2: value2, … }

}

-408 - Too long for the server to handle. **E_SET_LONG**

{

***settings*: [*the setting which were too long*]**

}

-409 - Not within range. **E_SET_RANGE**

{

setting: name where the range error occurred,
range: [*min*, *max*] (what the server defines to be acceptable range)

}

-410 - The data provided was wrong. **E_SET_WRONG**

- **This error is a general error. It may mean that an image URL passed into the *avatar* variable—for instance—does not exist or that a specific username entered into a setting does not exist.**

-411 - The setting provided does not exist; cannot operate on it. **E_SET_NOENT**

{

setting: ***the setting that does not exist***

}

-412 - The setting isn't private—cannot whitelist it. **E_SET_NOTPRIV**

{

setting: ***the setting that isn't private***

}

-413 - The setting has a prefix—cannot private it. **E_SET_PREFIX**

-414 - The setting isn't within the whitelist—cannot delete it. **E_SET_WHITEDEL**

{

setting: ***the setting that isn't within the whitelist for deletion***

}

500 - Queryables

Successes:

500 - Query successful **S_QUERY_OK**

{

results: [*the results of the things you have queried*]

}

Error:

-500 - Query field does not exist. **E_QUERY_NOENT**

{

fields: ***[which fields]***

}

-501 - Query field invalid operation **E_QUERY_MISUSE**
{
field: fieldname,
operation: operation string (*null* if str)
}

-502 - Array operation on non-array queryable type. **E_QUERY_ARRAY**
-503 - Non supported queryable type. **E_QUERY_TYPE**

600 - Messages

Successes:
600 - Messages retrieved from the database. **S_MESSAGES**
{
messages: [messages that of the format of message events described in Events/Messages]
}

Errors:
-600 - Message was too long for the server. **E_MSG_LONG**
-601 - Message was too long for the user. **E_UMSG_LONG**
-602 - Message was too long for the channel. **E_CMSG_LONG**
-603 - You are sending too many messages to the server. **E_MSG_RATE**
-604 - You are sending too many messages for the channel. **E_CMSG_RATE**
-605 - Null or blank message. **E_MSG_NULL**
-606 - Format too long. **E_FORMAT_LONG**

700 - Captchas

Successes
700 - Captcha displayed successfully. **S_CAP_OK**
{
type: "ascii" or "ansi", (or something else, god forbid they use ReCaptcha),
captcha: *ASCII/ANSI art containing the captcha image* or something else
}

701 - Captcha completed successfully. **S_CAP_DONE**

Errors

-700 - Captcha failed; please try again. **E_CAP_FAILED**
-701 - Do another one, you're still suspicious. **E_CAP_SUS**

800 - Flood Control

Errors:
-800 - You have generally done too many things too quickly. **E_FLOOD_GEN**
-801 - You have sent too many messages in a certain amount of time. **E_FLOOD_MSG**

900 - Streams

Successes:
900 - Stream started successfully.

Settings & Constants

We have mentioned settings many times in the previous sections in the document, and their purpose there is fairly well-documented, but officially, settings within the Delegate Protocol are variables (or constants, with server-wide settings) which can be applicable to users, channels, the users within a channel, and on a server-wide scope. They can assume different data types, such as int, string (by default), int enum (where different integer values apply to different settings), string enum similar to that of int enums but with string definitions, and booleans; they will also have varying dimensions, such as linear/scalars, arrays, and even objects (not really a separate dimension, but I do not want to be too verbose).

Settings are used for relaying information from and to the Delegate Server, as well as for other users or bots who may find them useful. For this reason, users can set unofficial settings that the protocol makes no establishment on by prefixing them with an **@**. Settings can, for instance, set the channel mode for any given channel, or it may do something unofficial such as facilitating the transferring of public keys between users. Settings marked with **@** are called custom settings, which are limited to 256 bytes in their total data size; custom settings may only have 32 entries in each respective category.

They will be saved in memory initially. However, setting updates will be placed upon a queue which will save them to the database after a certain amount of time. If a user or a channel are not initialized, the mere request for their information will warrant their settings to be placed into memory.

Settings have varying qualifiers as well, with some of them being immutable or private. Settings that are by default immutable are marked with **$**; settings that are by default private from the eyes of other users are marked with **&**; finally, settings marked with **!** are both immutable and private. Immutability cannot be toggled, but privacy can if the user or channel administrators wish so.

If a setting does not exist, then a *get* on them through whatever means shall return *null* in the response body for that specific setting.

Settings may be more intelligent than others, where it will be noted if they are. For instance, they may return an error if they are given a value that they do not like. Their errors and their format of conveying things will be explained also in *Responses/Settings*. Some settings may also regulate the values which can be supplied to them, such as settings which use enumerations only allowing the values of the enumerations to be used; other settings may return an error if there are qualities that are mutually exclusive. Types are generally regulated. These shall be called *regulated settings*.

Even more, the changing of settings may issue an action to users, depending if the protocol deems them to be important or not (e.g., profile image change, channel name change, anything that the clients will need to know, etc). These are called *special settings*. They will issue an action stating what settings were changed; however, the client will still have to retrieve them manually.

Settings are like what the file is to the UNIX system. They are a unified concept which allows the server to communicate to users and vice versa. Some may even expose features of a channel to users and take administrative action.

Settings should exist within memory on the server instance, for speed and easy access. It should then be placed on a queue so that it can be cached to the database so that it may be loaded later into memory either for the entire user or for accessing a user that is offline.

The default setting value will be situated next to the setting, denoted by ": *default"* if there is a default setting, besides settings immutable to the user/owner and set instead by the server. If there is no mention of a default value, then the server implementation should provide one of its own. Settings should **at least** be initialized.

Server Constants

Server settings may only be changed by the maintainers of the server itself, by whatever means it is implemented as (it could be as simple as changing the values in an associative array). Thus, they are not changeable from within the protocol itself, and they only act as constants for the client to know what they should expect, in a similar fashion to the kinds of information that the HTTP protocol may expose to the client, for instance. Server settings have no qualifiers because it is implied that they are always immutable and always public.

The implementation of the Delegate Server shall set default setting values.

name: the name of the server
description: the description of the server
image: the server image URL
banner: the server banner URL
version: version string of the server
admin: the username of the administrator of the server
password (bool): is a password required?
msglen (int): maximum message length
timeout (int): timeout in seconds.
username_len (int array): [*minimum username length, maximum username length*]
username_regex: username regex
password_len (int array): [*minimum password length, maximum password length]*
channel_len (int array): [*minimum channel name length, maximum channel name length*]
channel_regex: channel regex
subchannel_len (int): [*minimum subchannel name length, maximum subchannel name length*]
subchannel_regex: subchannel regex

safelinks (string array): domains and websites that the server approves of.
http_endpoint: where is the HTTP endpoint located? Provide a URL with a schema and everything.
default_avatars (string array): [*HTTP links to pictures of default avatars*]
default_channel_avatars (string array): [*HTTP links to pictures of default channel avatars*]

User Settings

In the case of user special settings, there is no way for the server to know who should get these events. We know that friends should be able to automatically subscribe to each others' events; however, we are unsure of who else, be it a recent conversation. For this reason, any clients who want to keep tabs on statuses that may change on a user account may use the *usubscribe* command.

Any special settings will be marked by having their names in italics.

***name*: the unrestricted, UTF-8 name of the user. : *default* null (if not further implemented)**

- **It is recommended that the server, by default, set it to the username.**
- **It should be between 1-24 characters long.**
- **It may not be nothing.**
    - **If there is no name, it should be set to *null*.**

***dnd* (bool): are they in *Do Not Disturb*? : *default* false**

- **The client shall silence notifications when it detects that you are in dnd-mode.**
- **The server will not send anything to your pager, if it is even configured.**

***status_text*: what is their status text? : *default* null**

- **It should be between 1-32 characters long.**
- **It may not be nothing; *null* if you want nothing.**

***description*: their description message/bio. : *default* null**

- **It should be between 1-360 characters long.**
- **You get the trend: can't be nothing.**

***avatar*: the URL of their profile picture. : *default* null (if random image not implemented)**

- **It is recommended that the server set it to a default image instead of *null***
    - **If it does, the kind of default image should be randomly chosen from the list of avatars from the server constant *default_avatars***
- **The client, if paranoid, should check if the link comes from *safelinks*.**
- **It may not exceed 256 characters.**

$creation (int): UNIX timestamp of their creation date, which should be an integer.

!channels (string array): [*channels joined*] : *default* []
!gchannels (string array): [*group channels joined*] : *default* []
!blocked (string array): [*those who are blocked*] : *default* []
!friends (string array): [*your friends*: : *default* []
!subscribedto (string array): [*who you are subscribed to*] : *default* []
!subscribedtome (string array): [*who are subscribed to me*] : *default* []

!privatedsettings (string array): [*settings which are privated*] : *default* []
!privatewhitelist (object) : *default* {}
{
"setting1": [*user1*, *user2*],
"setting2": *null*,
…
}

- **Even though *setting1* is private to most users, *user1* and *user2* are able to retrieve the contents of the setting.**
- **Even though *setting2* is private to most users, a value of *null* indicates that only friends can see it.**

$bot (bool): are they a bot? : *default* false

- **This is set by the server, obviously.**

perms (string array): if summoned to a channel as a bot, what permissions do I want? : *default* []

- **You cannot have more permissions than exist in the Delegate Protocol.**
  - **If a permission in here does not exist, *E_CHAN_NOPERM* will be returned as an error code.**
- **This has no effect on user accounts; it's a useless setting there.**
  - ***E_USER_NOTBOT* will be raised if changed on a user account.**

&2fa (bool): enable 2fa on your account? : *default* false

Modes (some are mutually exclusive):

&invisible (bool): do you want to be invisible from queries? : *default* true

&asocial (bool): no one can private message them. : *default* false

- ***E_USER_MESSAGE* if someone tries to private message them.**

&friends_only (bool): only friends can message them. : *default* false

- ***E_USER_MESSAGE* if a non-friend attempts to private message them.**

&lone (bool): nobody can become friends with them. : *default* false

- ***E_USER_FRIEND* will occur when someone attempts to friend them, if this is true**

&friendly (bool): if true, people outside of mutual channels can message them : *default* true

&skeptic (bool): only people in mutual channels can become friends. : *default* false

- **Note: *asocial* conflicts with *friends_only*; *lone* conflicts with *skeptic*.**
- **As implied above, conflicts will return an error.**
- ***E_USER_FRIEND* will occur when someone attempts to friend them, without being in a mutual channel.**
- **Returns:**
- ***E_SET_EXCLUSIVE* on conflict, detailing the two in conflict.**

***$status* (int enum): is the user currently online, away, or offline? : *default* 0 (online)**

- **online = 0,**
    - **Active on any device within *user_away_duration* (default: 5 minutes) minutes. A connection will automatically yield this value.**
- **away = 1,**
    - **No user activity for *user_away_duration* (default: *at least* 5 minutes) minutes.**
- **offline = 2**
    - **No device connections on that user account are known to the server/network. This includes both event and regular connections.**

Paging:

Paging is a feature that is on the server-backend which allows a user to receive push notifications if they are offline or away. The user can customize where such paging will be sent, as well as the amount of information contained in each pager request.

&pager: the HTTP(s) endpoint for message paging. : *default* null

- **If this is null, paging is disabled.**
- **May not exceed 256 characters.**
    - **This yields *E_SET_LONG***

&pager_level (int): the level of secrecy that the pager has: : *default* 0

- **0: Only notify that a message has been received, no author or contents.**
- **1: Only notify that a message has been received, including its author and origin.**
- **2: Send the entire message.**
- ***E_SET_ENUM* will be raised if *pager_level* is not within the range [0, 2].**

When a pager request is sent to the endpoint, it will send the following information, via JSON:

{

"timestamp": UNIX timestamp of message,

"channel": the channel it came from,

- *If it didn't come from a channel or pager_level is 0, this is null*

"subchannel": the subchannel it came from,

- *If it didn't come from a channel or pager_level is 0, this is null*

"username": the user it came from,

- *This is null if pager_level is 0*

"message": the message contents

- *This is null if pager_level is 0 or 1.*

}

A pager request may also notify when a person has connected with their account, akin to the *login* event that is generated.

Channel Settings

Channels may have settings and characteristics just like any other type of setting mentioned in this document. Only users who have the *set* permission within the channel may attempt to modify any of these settings. However, notable exceptions apply, such as when setting a setting which is regulated by the server, such as the *description* or *image* setting, which only requires the *metadata* permission.

The following settings expose valuable information about the channel and its users to clients, of which are managed solely by the server, hence their prefix of **$**:

$creation (int): the UNIX timestamp of when the channel was created.

$heirs (string array): the usernames of those who shall inherit the channel if the owner is to be banned or to delete their account.

$owner: the owner of the channel (subject to change with change of ownership)

- **The owner shall be set by the person who initially registers the channel.**
- **If the initial owner transfers ownership, this variable shall be changed to the new owner.**

$roles (object): the roles and their permissions. For example:

```
{
        "role1": [
                talk,
                read,
                admin
        ],

        "role2": [
                talk,
                read,
                delete,
                kick
```

],
…
}

$banned (object): {*those who are banned from the server, by username when* and *duration*} : *default* {}. For example:

{
“username1”: {
“duration”: in seconds as an int,
“reason”: the reason why as a string,
“when”: the UNIX timestamp of when as an int
},
…
}

$muted (object): {*those who are muted*} : *default* {}. For example, similar to above:

{
“username2”: {
“duration”: in seconds as an int,
“reason”: the reason why as a string,
“when”: the UNIX timestamp of when as an int
},
…
}

$userno (int): how many users are in the channel.
$userlist (string array): the list of users in the channel.
$userroles (object): an object correlating the users to their role. *{user: role, … }*

- **As a client, this may be more efficient than *$users*.**

$group (bool): should the channel act like a group conversation? : *default false*

- **The client should additionally make a distinction between groups and channels (i.e., groups should be a part of the friend’s list).**

$subchannels (string array): the subchannels within the channel.

- **If the channel is hidden from the user by their username or by their role, they will be unable to see it when they query for this setting.**

Special Settings:

name: the unrestricted name of the channel (only requires *metadata*). : *default* null

- **It should be 4-24 characters max, lest *E_CHAN_LENGTH* is raised.**
- **It should obey the REGEX rule of [a-zA-Z0-9], lest *E_CHAN_REGEX* is raised.**
- **Only requires *metadata***

description: the description of the channel (only requires *metadata*).

- **It should be 1-128 characters max, lest *E_CHAN_LENGTH* is raised.**
- **Only requires *metadata***

image: the image url which represents the channel (only requires *metadata*).

- **It must come from *safelinks*, if the client wishes to protect their users**
    - **Clients *MUST* be careful. They can also choose to use a proxy for the URL here.**
- **It must exist.**
- **Only requires *metadata***

categories (object): group and categorize subchannels. The format must be, for example: *default* {}

{
    "programming": ["python", "cpp"],
    "images": ["selfies", "memes"],
    "help": ["linux", "windows"]
}

- **If a subchannel listed in one of the arrays does not exist within the channel, *E_SCHAN_NOENT* is raised.**
- **If a subchannel is listed more than once, *E_SET_EXCLUSIVE* is raised as an error.**
- **If there are more than 16 categories, *E_SET_RANGE* is raised.**
- **Only requires *metadata***

default_role: what is the default role given to new members? : *default* "default"

- **If the role provided here does not exist within the channel, *E_CHAN_NROLE* is sent as an error.**

default_label: what is the default channel-given label given to new members? : *default* null

- **If *null*, then none are given.**
- **Only requires *metadata*.**

auto_roles (object): if chosen, automatically set users roles when they reach a certain level milestone. For example: : *default* {}

```
{
        3: "cool",
        10: "trusted",
        25: "OG"
}
```

auto_labels (object): if chosen, automatically add/append channel-set labels to a user when they reach a certain level milestone. Refer to the *auto_roles* example above for an idea on what this setting should contain : *default* {}

For an idea of what levels mean in terms of how many messages, this may be useful: https://www.desmos.com/calculator/71rozcxuab. Ultimately, it is less of a burden on the server to only apply autoroles and auto labels on each level upgrade, not on each message: this is why autoroles and auto labels are not based on the number of messages sent.

Some channel modes are as described:

invisible (bool): should the channel be invisible to queries? : *default* true
lockdown (bool): if true, nobody new can join. : *default* false
invite (bool): should the channel require invites? : *default* false
password (bool): should the channel require a password for join? : *default* false
tor (bool): should tor users be allowed? : *default* true
concealed (bool): should people who conceal their IP address be allowed? : *default* true

ban_by_ip (bool): should users be banned by their IP in addition to their username? : *default* false

captcha (bool): should a captcha be required to join? : *default* false
captcha_count (int): how many captchas? : *default* 2

- **Note: The server may regulate the amount. The protocol recommends 1-9.**
    - **If the number is outside of the limit, *E_SET_RANGE* is raised.**

tor_captcha_count (int): how many captchas under tor? : *default* 5

- **Note: this is also regulated. The protocol recommends 3-16**
    - **If the number is outside of the limit, *E_SET_RANGE* is raised.**
- **Note: the setting *tor* will override this if it is set to false.**

concealed_captcha_count (int): how many captchas whilst concealed? : *default 3*

- **Note: this is also regulated. The protocol recommends 3-9**
    - **Yet again, if the number is outside of the limit defined by the server, *E_SET_RANGE* is raised.**
- **Note: the setting *concealed* will override this if it is set to false.**

join_message: what should the channel tell newcomers? : *default* is determined by server

- **The length of this is regulated: we recommend 3-24 characters.**
    - ***E_SET_RANGE* if violated.**

leave_message: what should the channel tell those who leave? : *default* is determined by server

- **The length of this is also regulated: we recommend 3-24 characters.**
    - ***E_SET_RANGE* if violated.**

Subchannel Settings

$creation: the UNIX timestamp of when the subchannel was created.
$roles (object): an object corresponding roles to their permissions. For example:

```
{
        "role1": [
                "talk",
```

“read”,
“admin”
],

“role2”: [
“talk”,
“read”,
“delete”,
“kick”
]
}

Special Settings (triggers an event):

description: the description of the subchannel (only requires *metadata*)

- **The description should only be 1-32 characters long.**
  - ***E_SET_RANGE* if violated.**

image: the image of the subchannel (only requires *metadata*)

- **The URL must be in *safelinks*.**

private (bool): a private subchannel (only the users, as described below, may use it). : *default* false

Normal Settings:

allowed_roles (string array): [*roles that are allowed*] : *default* []

- **Roles that are placed inside of this array will be the only ones able to access the subchannel and use it. The subchannel will be inaccessible and invisible to those outside the roles listed in this setting.**
- **Roles at or above your own role cannot be made inaccessible to this channel, for that would violate the channel role hierarchy. *E_CHAN_INSUB* will be raised if this unfortunate event occurs.**
- ***E_CHAN_USER* is raised if roles are put in that do not exist within the channel.**

User-Channel Settings

These are settings that apply to users within a channel. Settings marked with ~ are immutable by the user but not by the server and/or channel administrators (those with the *set* permission). If a user without the *set* permission attempts to change a setting prefixed with ~ within a channel, then *E_CHAN_PERM* is

raised as an error. As a reminder, settings marked with $ are only set by the server to expose information in an abstracted manner.

$join (int): timestamp of when a user most recently joined

~labels (string array): the channel labels for a user. : *default* []

- **Each label must be a maximum of 16 characters.**
- **Only channel admins may set this. Users are not supposed to change these kinds of labels.**

nickname: what is their channel nickname : *default* null

- **The maximum length for nicknames should be 1-24 characters, if it exists.**
    - ***E_SET_LENGTH* if violated.**
- **If it is *null*, the user has no channel nickname.**

image: their profile picture for the channel. : *default* null

- **The URL must be in *safelinks*.**
- **If this is *null*, their user profile picture shall be displayed by the client.**

labels (string array): the labels for a user. : *default* []

- **Each label must be between 1-16 characters.**
    - ***E_SET_LENGTH* if this is violated.**
- **Use this for labeling users (e.g., this person uses Arch Linux, this person identifies a certain way, etc). Do not use roles.**

$sent (int): how many messages have they ever sent in the channel? : *default* 0
$level (int): how much have they talked in the channel? : *default* 0

- **Ultimately, server implementations may define their own way for considering the user level, but the protocol recommends the following mathematical function:**
    - $level(sent) = floor(log_{1.75}(sent + 1))$
        - **This means that each new level will require 75% more messages to be sent than for the previous level. The requirement can be tweaked by changing the base of the logarithm. The $+ 1$ is required to prevent the logarithm from being *undefined*.**

Queryables

It is useful to have characteristics of an entity (i.e., a channel or a user) be queryable for searching. Queryables are closely associated with the aforementioned concept of settings, but may be queried along a long list of entities. There are certain queryable fields with their own respective rules which abide by the protocol.

There is a unified format in which queries shall be made, almost becoming their own JSON SQL language. One must pass a specific format into any of the query commands. There are certain operators that may be used. An object must be passed that corresponds to the queryable field to the operator which makes a search query on that condition. We shall detail them below along with some examples:

- **(no operator): is equals to,**
    - **This can work on *any* type, as long as the inputted type matches the type that the queryable holds.**
- **"-": within range of two numbers (e.g., "5-10")**
    - **You may not use this on a non-int type.**
- **">": greater than a number (e.g., ">5")**
    - **You may not use this on a non-int type.**
- **"<": less than a number (e.g., "<5")**
    - **You may not use this on a non-int type.**
- **"{": contains a string (e.g., "{arch")**
    - **You may not use this on a non-string type.**
- ***["AND",, strings]*: contains these strings on an *AND* basis (e.g., "['linux', 'hacking', 'programming']"; useful for tags).**
    - **You may not use this on a non-array queryable type.**
    - **This must be passed in as an array type, not as a string.**
- ***["OR", strings]*: contains these strings on an *OR* basis**
    - **You may not use this on a non-array queryable type.**
    - **This must be passed in as an array type, not as a string.**

The total length of the queryable shall be capped by the server, as to prevent an extremely long and expensive query. If there are any logical loops or bottlenecks, the server can deny the request for the queryable.

Every single queryable queried will be queried on an AND-basis. Even the same queryable can be queried multiple times within a single queryable request, where it shall be queried on an *AND* basis among the other instances of the queryable.

An example:

```
{
    "name": "{linux",
    "users": ">10",
    "users": "<100",
    "admin": "Delegate",
    "tags": ["OR", 'linux', 'hacking', 'programming']
```

}

"Find channels that have 'linux' contained within their name somewhere ***AND*** *where the users are above 10* ***AND*** *where the users are below 100* ***AND*** *where the admin's name is* ***exactly*** *"Delegate"* ***AND*** *where the tags must contain either 'linux'* ***OR*** *'hacking'* ***OR*** *'programming'."*

The response will be that of S_QUERY_OK, detailing the entities with the specified criteria above. Find out more about the response in *Responses/Queryables*.

For more information about Queryables from a server-backend perspective, please view the auxiliary *Delegate Queryables* document that should be within close reach of this document. As said before, the Queryables interface has broadened its scope beyond the Delegate Protocol, being its own SQL transpiler of some sorts.

Tags

Tags are a useful way to group entities like users or channels. Even though matchmaking is not within the purview of the Delegate protocol, users could be tagged for specific purposes—it is on the user's volition to do so, although it should not be expected for clients to implement this. Tags are especially useful for querying for channels, though.

Tags must obey the REGEX rule of **[a-z0-9 _]** on the protocol standard along with a character limit of **16** characters. Tags shall preferably be of words describing the channel (or user), such as: [programming, linux, osdev].

Tags shall be queried on an **AND** basis, where channels queried must meet every tag provided. Tags may be added with the ***ctag*** command; user tags are not of concern unless the issue arises.

There is no database of valid tags, and the tag system is biased towards the English language. It is important that tags are kept concise to allow for the discovery of channels. The protocol only provides the medium for channel discovery; it is not within the purview of the protocol to establish known tags. It is, perhaps, that there should be a protocol for universal tags (e.g., *1 = programming, 2 = linux, …*).

Channels

The following fields may be queried (with their own stipulations onto what can be queried on them):

- users (int): how many users are in the channel,
- tags (string array): tags that define what the channel is about,
- name: what is the (nick)name of the channel,
- description: its description,
- owner: who is the owner,
- open (bool): are they open,
- invite (bool): are they invite only,
- tor (bool): do they allow tor users,
- limit (int): what is their limit on the number of users, if it exists?

Messages

Since Delegate assumes that clients will utilize some sort of encryption among themselves, not a lot of information on messages and user interactions can be queried. However, some metadata should be preserved, for user convenience reasons. The protocol will assume that there are some instances where encryption is not occurring, therefore it will provide message querying that would assume no encryption.

The following fields can be queried:

- from: the username it is from,
    - This is for channel messages only. Using this metric for user messages will result in an error.
- between (int array): [*this, that*]
    - Between the two UNIX timestamps *this* and *that*.
- before (int): before a UNIX timestamp
- after (int): after a UNIX timestamp
- type: what type?
    - Remember, default messages are *null* typed.
- subchannel: from what subchannel
    - Only if querying from a channel! Again, using this metric for user messages will result in a query error.

HTTP Endpoints

The sole reliance of our own protocol is quite inefficient for certain purposes, especially for requests that do not require a consistent connection or for the usage of webhooks. This section of the protocol will be dedicated to describing the various HTTP endpoints that can exist within the Delegate Protocol (and which should be implemented by implementations of the Delegate Protocol). To see what ports these HTTP endpoints should be hosted on, look in the *Networking* section of this protocol specification. Anything in bold is to be replaced by values at the discretion of whomever is using the endpoint (i.e., they are variables).

An HTTP endpoint may not be called by the same IP more than once within a 30-second period. If that happens, the request will be denied, yielding *403 Forbidden* as an HTTP response code. If any placeholder value has no corresponding existences within the server (i.e., a channel name does not exist on the server), *404 Not Found* shall be returned as an HTTP response code.

GET /info/ - Gets all information about the server. No arguments needed. The information that will be returned will be the same information as if you issued the *getall* command.

POST /webhooks/**channel_name/webhook_id** - Send a message to a channel's webhook **webhook_id** as specified by **channel_name**. The body POSTed to this URL must be akin to:

{
"message": message to send,
"type": the message type,
"format": the message formatting or metadata,
"avatar": URL of avatar to provide, (at the client's discretion to proxy),
"name": name of the thing being sent (how it shall be represented as)
}

(Experimental*) **POST** /query/users/ - Query all usernames on the server, without discrimination of one username. The POST body must contain the following:

{
"query": the query format as described in the Queryables section of this document.
}

(Experimental*) **GET** /query/users/**username/** - Query the username as specified by **username**. There are no arguments required. All queryables and visible user settings will be returned by this specific query.

(Experimental*) **POST** /query/channels/ - Query all channels on the server, not one specific channel. The POST body must contain the following:

{
“query”: the query format as described in the Queryables section of this document.
}

(Experimental*) **GET** /query/channels/**channel_name** - Query the channel as specified by **channel_name**. This will return values which would be returned by a queryable or by *cget*. This requires no arguments.

Streams

The Delegate Protocol does not make an official establishment on voice and video calling, among many other features, but it ought to provide the abstracted medium for doing so. The concept of *Streams* in Delegate can help to achieve this goal. A *Stream* acts a lot like a standard call on a platform like Discord or Skype, but has an abstracted interface. Theoretically, one can send anything through a stream—as long as it is valid text or JSON—but that can be used to implement a calling protocol among clients, along with other things such as games.

A *Stream* may have a type as indicated by a string. The two stream types are reserved for an implementation of voice and video calling to be detailed in another document, called “voice” and “video” respectively.

Among users and group channels, a particular user may issue a stream call, meaning that an event will be sent to all parties (either just the other user or all of the other members in the group channel), which signifies that there is a request to start a stream call. An invitation to a stream call will only be available to deny or accept for a certain fixed amount of time: we recommend 30 seconds, but that can be set by the server variable *stream_call_duration*. One may use the *gstart_stream* and *ustart_stream* Delegate commands to achieve this.

~~To handle such a call, one must accept or deny it. The event which signifies that one is receiving a call will have a *call token* attached to it. This call token is a Base64-encoded representation of the following JSON object + a hash:~~

~~*{*~~
~~*“origin”: “gchannel” or “user”,*~~
~~*“type”: “voice”, “video”, or something custom,*~~
~~*“invoker”: username of person who called,*~~
~~*“timestamp”: when was the call issued*~~

~~*if origin is “gchannel”:*~~
~~*“gchannel”: the name of the gchannel is came from*~~
~~*}*~~

~~Then, once this JSON object is deserialized to text in compressed form, a server secret key is directly appended to it, then it is thrown through a SHA512 hashing algorithm. Then the original JSON will be given a new field *hash* where the SHA512 digest—in string hexadecimal form—will be. Finally, this JSON will be deserialized into Base64. This is what a *gchannel* or *user* stream token will look like, and this is for the purposes of putting a time limit on the calls.~~

Flood Control & Captchas

With any online service, it is paramount to protect against users flooding the service with unwanted spam/fraudulent requests. There have been many innovations to combat the proliferation of spam and abuse of services, such as the captcha and the requirement of phone verification. However, methods such as phone verification can pose a threat to the privacy of users, for requiring a phone de-anonymizes the user and makes their identities virtually known to the service in question. With Delegate, we want a method of ensuring that bots are not being used for spam, but without the curtailment of privacy—a fundamental right amongst those who use the internet. In the following paragraphs, we shall provide recommendations that should prevent the abuse of any Delegate server; however, the exact numbers are of course at the discretion of the specific Delegate server in question, where they may set their own values as they see appropriate.

Flood Control

It is recommended that 3 (normal users), 6 (VPN/proxy users), or 12 (Tor users) captchas shall be required upon registering a user. 2 (normal users), 4 (VPN/proxy users), or 6 (Tor users) captchas shall be required upon the registration of a channel. If more than 45 (normal users), 19 (Tor users), 30 (VPN/proxy users) messages—in total, regardless if they are spread among multiple users or channels—are sent within a minute on a normal user (not bot) account, a captcha shall be required to progress or they shall have to wait 30 (normal users), 45 (VPN/proxy users), or 60 (Tor users) seconds before being able to send another message to any party.

An IP may only make 3 (normal users), 2 (VPN/proxy), or 1 (Tor users) user accounts per week; conversely, they are allowed to make 4 (normal users), 3 (VPN/proxy), or 2 (Tor users) bot accounts per week.

An HTTP endpoint, as previously stated in the *HTTP Endpoints* section of this protocol, is subject to regulations on the number of requests that can be made to it from an IP within any interval of time. The maximum amount of requests that can be made to any endpoint within any given minute is 9 (normal users), 6 (VPN/proxy users), or 3 (Tor users).

Captchas

In order to preserve compatibility across low-powered machines and slower internet connections, captchas shall be preferably made from ANSI/ASCII (ANSI preferred!) art which contain distorted lettering of different fonts. Within this obscuring, logical or mathematical tests may also be given, as to increase the effectiveness of these measures as proper captchas. Culture or language dependent captchas should be avoided at all costs. Captchas should be easy for humans, but hard for robots; additionally, due to the sheer number of captchas, this will slow down and incur charges for services that use human labor in order to solve captchas, mitigating a problem which is rarely a concern of most services which wish to prevent such spam. Commands for captchas are discussed in the *Commands/Server* section of this protocol, unsurprisingly.