The Delegate Protocol

First Edition

E. Arner
arner@usa.com

Version: 1.0.11.22 Beta

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), with the date that it was modified, all delimited by periods. If the protocol is in a Beta release or Alpha release, then that should be declared by adding that as a suffix.

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

There shall be a committee that is appointed by the creator of this protocol where the creator is also involved in that committee. The committee should review concerns brought about by others and amongst themselves. They will have to vote on a supermajority to approve the change, where the creator of this protocol may have the final decision as to whether the changes should be implemented or not. The creator may make changes without committee approval—if a committee ever forms, anyway.

The creator may change the governing system of the changing of this 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. Currently, Delegate is mostly a project to show how such a service can be made.

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.

However, Delegate also holds a philosophy of contribution. Delegate should grow to reward users who share resources to a specific network. This can be in the form of a currency which allows special actions to take place on the platform; additionally, it may act as a payment processor in its own right, in which people mine currency by virtue of making contributions. This would be rather troublesome to efficiently and correctly implement, however.

Networking

The Delegate Protocol suggests that WebSockets over TLS be the default method of connecting to any of the server implementations, but alternatives should always be readily available for those who need them. Below are listed the connection methods in order of priority with their designated ports:

- **WWS (WebSockets with TLS): 9999**
- **WS (WebSockets plaintext): 9998**
- **TCP over TLS: 9997**
- **TCP plaintext: 9996**

WebSockets is the most compatible of these, which is why it is at the highest priority in the networking of the Delegate protocol. It can be used on the desktop, in the browser through JavaScript, and anywhere with TCP capabilities—just with an HTTP handshake before the connection is kept-alive and acts like a standard TCP connection. When this is combined with TLS, we obtain a very compatible and also very secure method of communicating with the Delegate server.

Server implementations should optionally also provide provisions for connecting via a modem and/or other services to make Delegate a portal to people everywhere, without all of the bloat. A compression algorithm that converts the string possibilities into integers in the various JSON blocks should be used, providing people with absolute minimalism if they desire so. Although, this should not be prioritized, as this would be a minute portion of the already minute user base of Delegate.

Any data sent to the main server is limited to a strict 16384 bytes to prevent vulnerabilities and denial-of-service attempts.

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 maximum **32** characters in length and must 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*.

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.

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.

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 will be registered with something called the master password, which is a password which only the owner should know. It will allow the owner, in a time where the channel has gone rogue or has been subject to a *coup d'état*, to regain control back over their channel with the usage of the password. These passwords are subject to regulations on their weakness. This is especially useful if the owner's account has been compromised.

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 32 characters long, where their subchannels only have a limit of **16** characters in length; channels and subchannels have a REGEX rule of **[a-z0-9 _]**. 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 Conversations

There is a need for something called group conversations, but we should not bloat our protocol by making them separate. Essentially, group conversations allow users to be within a group, but without 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).

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.

If a custom role has been deleted, then all users of that role will become members (i.e., role 3). 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.

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 the respective number and its description:

- **talk - 0 - Talk in a channel**
- **read - 1 - Read messages within a channel**
- **remove - 2 - Remove messages within a channel**
- **subchannel - 3 - Add or remove subchannels**

- **metadata - 4 - Change channel/subchannel description or image**
- **set - 5 - Set channel/subchannel settings.**
- **kick - 6 - Kick users**
- **ban - 7 - Ban users**
- **mute - 8 - Mute users.**
- **role - 9 - Set the role of users**
- **invite - 10 - Invite other users**
- **password - 11 - Change the join password**
- **order - 12 - Can order roles.**
- **vote - 13 - Can vote. (Experimental; do not implement)**
- **cast - 14 - Can cast a vote (start one). (Experimental; do not implement)**
- **summon - 15 - Summon a bot to the channel.**
- **admin - 16 - Is equivalent to an administrator.**

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.

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

- **talk - 0 - talk in that subchannel**
- **read - 1 - read in that subchannel**
- **remove - 2 - remove a message in that subchannel**
- **set - 3 - set a setting in that subchannel.**
- **vote - 4 - can vote in that subchannel. (Experimental; do not implement)**
- **cast - 5 - can cast a vote in that subchannel. (Experimental; do not implement)**

Channels will have the default roles (with their numbers and respective permissions);

- **owner - 0 - All permissions; this role can only be allotted to one person.**
- **admin - 1 - All permissions; cannot change password or delete channel.**
- **mod - 2 - *talk, read, remove, description, kick, ban*, *invite*,**
- **member - 3 - *talk, read***

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 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 UNIX). 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. There may be a more democratic implementation, where anyone may cast a vote. The channel may also implement simple majorities or supermajorities (i.e., a ⅔ vote).

A channel where everyone can vote and cast is a democracy; a channel where only a few (i.e., elected officials) can cast votes but everyone can vote on them would be a quasi-democratic system of governance; a channel where only elected officials can cast and vote would be a republican system of government for that channel. Channels can also be made into a federal republic because subchannels may have their own votes and elected officials. Most of this is, of course, for fun thinking: on a small scale, an authoritarian, yet just, form of channel control is more efficient. Channel governance is made more fair by mandatory auditing.

Voting is limited to one vote per IP address 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). 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.

Server

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

Returns:
S_PING when ping was done successfully.

***nop — Do nothing. Saves bandwidth, as opposed to a ping. No arguments needed.**

***get — Obtain server settings—rather, constants.**
{
***settings: [array of settings to get]**
}

- **Returns:**
- **S_SETTING when settings are obtained successfully**

User

***user — Sign in.**
{
***username: the username which you wish to sign in with,**
***password: the user's password**
}

- **Returns:**
- **S_USER_LOGIN when done successfully,**
-
- **E_USER_PASS on password authentication failure,**
- **E_USER_NOENT if the user is not registered,**
- **E_USER_MANY if too many users are connected**

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

- **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!**
- **A weak password should be defined as less than 8 characters.**
- **Returns:**
- **S_USER_REG when successful,**
-
- **E_USER_EXISTS if user is already registered,**
- **E_USER_WEAK if password is weak,**
- **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**

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 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.**

usend — Message another user

{

***username: whom you are going to message,**
***message: what are you sending them**
format: message format, if applicable.

}

- **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,**
- **E_FORMAT_BAD if the format is invalid**

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).**
- **You must not be blocked by the user.**
- **You must not already be friends with the user.**

friend — Accept or deny a friend request.

{

***username: the request from who,**
***deny: if true, deny the request.**

}

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

usubscribe — Subscribe/unsubscribe to a user's events.

{

***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.**

ulisten — Listen to user streams.
{
***username: the username to listen for, if they shall send your way**
***stream: which stream.**
}

- **Default streams: 0 for voice, 1 for video.**
- **Custom streams must be in string format. Integer streams are regulated.**

uemit — Emit information to a user stream.
{
***username: the user in question,**
***stream: to which stream,**
***payload: what to send (this shall be in text format).**
}

Channels

cregister — Register a channel.
{
***channel: name of the channel,**
***password: master password.**
}

- **Returns:**
- **S_CHAN_REG on success,**
- **E_CHAN_EXISTS if the channel is already registered**
- **E_CHAN_MWEAK if the master password is weak—less than 8 characters.**
- **E_CHAN_LONG if the channel name is too long**
- **E_CHAN_REGEX if the channel name violates the REGEX set**

cpasswd — Change the password of a channel.
{
***channel: name of the channel,**
***prev: the password of a channel,**
***new: the new password**
}

- **Only the channel owner may issue this command.**
- **Returns:**
- **E_CHAN_NOENT if the channel does not exist,**

- **E_CHAN_NIN if you aren't in the channel,**
- **E_CHAN_PASS if the *prev* password is incorrect,**
- **E_CHAN_MWEAK if the *new* password is weak (less than 8 characters).**

cdelete — Delete a channel.
{
***channel: name of the channel,**
password: if you lack the permissions, this password will do the trick.
}

- **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**
- **E_CHAN_PASS if the password is incorrect when supplied**

elevate — Obtain ownership of the channel, booting the other owner off of it.
{
***channel: name of the channel,**
***mpassword: master password for regaining control**
}

- **E_CHAN_NOENT if channel does not exist**
- **E_CHAN_PASS if password is incorrect**

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**

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.**

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

- **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,**

csend — Send a message within a channel and subchannel.
{
***channel: the channel you wish to send to,**
***subchannel: the subchannel,**
***message: the message.**
}

- **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 (-1 for forever),**
reason: the reason behind the ban

}

- **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**

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.**

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.**

gsummon — Summon a user to the group channel/conversation.
{
***channel: the group channel,**
***username: the username in question,**
message: the message, if applicable
}

- **You must be friends with the username in question.**
- **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.**

leave — Leave a channel.
{
***channel: name of channel you wish to leave,**
message: parting message, given it conforms like the join message
}

- **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.**

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.**
- **You may not have an empty list.**

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.**

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.

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,**
***uuid: the UUID4 id generated,**
***type: 0 for user, 1 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. *As of now,* no other fields!
logout — Another client disconnected from your user account. As of now, no other fields.
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
}

leave — Somebody left the channel you're in.
{
***username: who?,**
message: if they provided one, their message
}

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 3, 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*, … }**
}

Server

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.

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: { … }
}

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**

100 - User

Successes:

100 - Successfully signed in. **S_USER_LOGIN**

101 - Successfully registered. **S_USER_REG**

102 - Log out. **S_USER_LOGOUT**

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 too long. **E_USER_LONG**

-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**

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
}

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**
-205 - Channel name too long. **E_CHAN_LONG**
-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 too long. **E_SCHAN_LONG**
-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. **E_CHAN_IN**
-219 - That password (master or join) was incorrect. **E_CHAN_PASS**
-220 - Main cannot be deleted or changed like that. **E_CHAN_MAIN**
-221 - Listing successful **E_CHAN_LIST**
-222 - Order does not contain all fields. **E_CHAN_ORDER**
-223 - You may not modify those roles (i.e., default roles or higher roles). **E_CHAN_ROLE**
-224 - The user does not exist within the server. **E_CHAN_USER**
-225 - An error relating to the specificity of group channels occurred. **E_CHAN_GROUP**

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**

400 - Settings

Successes

400 - Settings successfully obtained. **S_SET_GET**

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

401 - Settings successfully set. **S_SET_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. **E_SET_TYPE**
-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. E_SET_LONG

{

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

}

-409 - Not within range. E_SET_RANGE

{

settings: {*setting1*: *[min, max]*, *setting2*: *[min, max]*, … }

}

-410 - The data provided was wrong. E_SET_WRONG

{

}

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

-411 - The protocol-compliant setting does not exist. E_SET_NOENT

{

settings: [*protocol-compliant settings that don't exist*]

}

- **Settings that are not marked with @ are protocol-compliant and the server instance will regulate what is put into them. If that setting does not exist, this error is thrown.**

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 misused.	**E_QUERY_MISUSE**

{
fields: {fieldname: value, … }
}

600 - Messages

Successes:
There are none? **It is a waste of resources to verify a message being sent, when it is sent back anyway; the client can verify on its own.**

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 - Message was not UTF-8.	**E_MSG_ENCODING**
-604 - You are sending too many messages to the server.	**E_MSG_RATE**
-605 - You are sending too many messages for the channel.	**E_CMSG_RATE**
-606 - Null or blank message.	**E_MSG_NULL**
-607 - Format too long.	**E_FORMAT_LONG**
-608 - Invalid format.	**E_FORMAT_BAD**

700 - Captchas

Successes

700 - Captcha displayed successfully. **S_CAP_OK**

{

captcha: ASCII art containing the captcha image.

}

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**

Settings

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
version: version string of the server
admin: the administrator of the server
msglen (int): maximum message length
timeout (int): timeout in seconds.
username_len (int): max username len
username_regex: username regex
channel_len (int): max channel name len
channel_regex: channel regex
subchannel_len (int): max subchannel name len
subchannel_regex: subchannel regex
safelinks (string array): domains and websites that the server approves of.

User Settings

Special Settings (these will generate events):

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 *csubscribe* command.

name: the unrestricted, UTF-8 name of the user.

- **It is recommended that the client, by default, set it to the username.**
- **It should be at maximum, 24 characters long.**
- **It may not be nothing.**

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

- **The client shall silence notifications when it detects that you are in dnd-mode.**

status_text: what is their status text?

- **It should be at maximum 32 characters long.**

description: their description message/bio.

- **It should be at maximum 360 characters long.**

avatar: the URL of their profile picture.

- **It is recommended that the server set it to a default image.**
- **It must come from *safelinks*.**
- **It must exist.**

$creation (int): UNIX timestamp of their creation date.
!channels (string array): [*channels joined*] : *default* []
!gchannels (string array): [*group channels joined*] : *default* []
!blocked (string array): [*those who are blocked*] : *default* []

$bot (bool): are they a bot? : *default* false
perms (int array): if invited to a channel as a bot, what permissions do I want? : *default* []

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
friends_only (bool): only friends can message them. : *default* false
lone (bool): nobody can become friends with them. : *default* false

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.**
- **Returns:**
- **E_SET_EXCLUSIVE on conflict, detailing the two in conflict.**

mobile (bool): are they on a mobile device? : *default* false
agent: what kind of client are they using (this string is defined by the client).

- **We recommend: [major number].[minor number].[client name]-[device]**
- **E.g., 1.3.Delegation-Android**

$method (int enum): how are they connected? : *default* 0

- **Websockets = 0,**
- **TCP = 1,**
- **Phone = 2**
- **Other = 3**

$speed (int enum): what is the quality of their network speed? : *default* 1

- **Fast = 0,**
- **Normal = 1,**
- **Slow = 2,**
- **Dialup (literally) = 3**
- **BBS (literally) = 4**

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

- **online = 0,**
- **away = 1,**
- **offline = 2**

For *mobile, agent, method, speed*, and *$status,* the most recently active connection—that is, they made an action, not merely just kept receiving events—will set these values, as multiple devices and methods of connection could be on the same user account!

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.
$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.**

$id: the UUID of the channel
$banned (string array): [*those who are banned from the server, by username*] : *default* []

$userlen (int): how many users are in the channel.
$users (string array): the list of users in the channel.
$roles (string array): the custom roles of 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?

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

Special Settings:

logging (bool): should the server log messages? If not, provide some alternatives of your own below. : *default* true

logging_servers (string array): [*the servers used for logging messages on the channel]. : *default* []

- **A channel may have at most 5 logging servers.**
- **If the array is null, nothing is being logged.**

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

- **It should be 24 characters max.**

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

- **It should be 128 characters max.**

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

- **It must come from *safelinks*.**
- **It must exist.**

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* true
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

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.**

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

- **Note: this is also regulated. The protocol recommends 3-16**
- **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**
- **Note: the setting *concealed* will override this if it is set to false.**

join_message: what should the channel tell newcomers?

leave_message: what should the channel tell those who leave?

Subchannel Settings

$creation: the UNIX timestamp of when the subchannel was created.

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

- **The description should only be 32 characters long.**

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

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).

$join (int): timestamp of when a user joined
$joins (int): how many times has this user joined
$kicks (int): how many times have they been kicked
$bans (int): how many times have they been banned
~labels (string array): the *channel* set labels for a user.

- **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 24 characters.**
- **If it is *null*, the user has no channel nickname.**

image: their profile picture for the channel.

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

labels (string array): the labels for a user.

- **Each label must be a maximum of 16 characters.**

- **Use this for labelling users (e.g., this person uses Arch Linux). Do not use roles.**

Formatting

Message formatting is an important aspect of communication on an instant messaging platform. Moreover, including various options for communication (i.e., allowing images, invite buttons, or other multi-media representations) is an extremely integral aspect of any of these services. Message formatting may be done by passing a format body (JSON) alongside a message sent. The Delegate server instance shall make minimal regulations or establishments on these; it is on the sole behalf of the clients to articulate and interpret these message formattings. This section will define a common protocol for doing such.

The format body shall be an object with the range of the formatters' influences being specified within the string, which will be delimited by a "-" (hyphen) (e.g., "0-9" refers to the 0th character of the message to the 9th character). Each range then shall be subject to an object of formatters which shall add format to the message, which may include colorization, font size, boldness, or of something else. A range which encompasses the entire text—or, rather, the lack thereof, if there is no underlying text message—shall be a mere *null*.

The protocol recommends that any server implementation limits message formatting to 256 bytes in size. Certain formatters may be catalogued in messaging databases so that one may search via a specific formatting type (e.g., if a message has an image).

This method of formatting text is extremely useful because it is extremely simple for clients to implement, as all that is needed is simple processing of JSON and understanding the ranges of text to modify. It is also respecting that certain clients may be minimal such that in-text formatting (e.g., that of Discord) may not be processed and therefore clutter messaging. For example: ```c hello```. Above all, it standardizes a common language for clients.

Message formatters will have properties on themselves (e.g., a font-size requires a size, a text-color requires a color, an image requires a URL, etc). A formatter shall resolve to its properties like so:

```
{
        "color": "red"
}
```

We shall define the various message formatters, their properties, and the acceptable values of those properties below:

color — Colorize text subjectively (the client shall decide what is, for instance, red).

- **0 - red**
- **1 - blue**
- **2 - green**

- **It is to be noted that colors should be limited and that the client is tasked with correlating the names of the colors with specific hex codes.**

italics — Make text italics.
bold — Make text bold.
strikethrough — Make text strikethrough.
underline — Underline the text.

super — Supertext. Provide what should be above.
sub — Subtext. Provide what should be below.

image — Tell the client to display an image. Provide a URL.

- **As previously stated, message formatting is not regulated by the server, so the client must receive safe URLs from *safelinks* or from its own table and obtain the image once it is decidedly safe. If the client refuses to implement these measures, an IP address could be leaked.**

video — Tell the client to display a video. Provide a URL.

- **Same note as above applies and hereafter all formattings including a GET request to a file.**

file — Represent a file to be downloaded. Provide a URL.

- **Again, same note to be heeded.**

channel — Represent a channel. Provide a channel name.
user — Represent a user. Provide a username.

link — A hyperlink. Provide the URL.

- **The client should receive the website metadata on its own discretion. Some clients may opt to use a proxy to retrieve the information in order to protect the user from their IP address from being leaked.**

latex — Represent a LaTeX equation. Provide the LaTeX equation.

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 in. 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,**
- **"-": 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.**
- ***[strings]*: contains these strings (e.g., "['linux', 'hacking', 'programming']"; useful for tags).**
    - **You may not use this on a non-array queryable type.**

An example:

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

The response will be that of S_QUERY_OK, detailing the entities with the specified criteria above.

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,
- concealed (bool): do they allow people who conceal their IP,

Streams

Streams are a method to add functionality to the Delegate Protocol without the Delegate Protocol having to change often to accommodate that—modularity and flexibility is a virtue of Delegate. Streams may be present on the user level or the subchannel level. They facilitate information about an external source in real time, without much regulation. They may, for instance, control voice and video features that correspond to an external server; it may also be used to, theoretically, synchronize a YouTube video between clients, if the client wishes.

Streams must be listened to with *clisten* or *ulisten* in order to receive events. A command *ucall* and *ccall* may inform the other user(s) that there is a stream that is session. The user(s) will accept this request by listening. A user may not send a stream of data with *uemit* or *cemit* without listening; a user who calls will automatically listen to the stream.

Emissions to a subchannel will only be sent to those who have listened. A subchannel may prevent unprivileged users from receiving emissions by withholding the *listen* permission for that subchannel.