NAME

gma-mapper-protocol – GMA client/server protocol specification

EXPERIMENTAL

This document includes experimental changes to the protocol which have not yet been officially released.

DESCRIPTION

The `mapper`(6) program displays a battle grid map for the players to see their tactical positions with respect to their opponents and features of their environment (e.g., dungeon rooms). To take advantage of its full multi-user capabilities, multiple `mapper` clients may be connected to the GMA `server`(6). The following document is the definitive specification describing the protocol used to communicate all messages between these programs (as well as others which interact with the GMA server, such as `gma-console`(6)).

When a client is connected to a GMA server, they exchange commands to indicate changes to the map display or other game state information. To help keep communications simple and to help clients recover from malformed or missing data, each command is a newline-terminated line of text as described in detail below. If the mapper sends one of these commands to the service, it is indicating that the local user made those changes and requests that other connected map clients update themselves accordingly. If the mapper receives the command from the service, it should comply to make the corresponding change to itself.

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

The software implementing this protocol MUST allow Unicode text encoded as UTF-8 everywhere (of which 7-bit ASCII is a subset).

Each command (sent or received) consists of a command word followed by a JSON object holding the parameters appropriate to that command. (See the description of the `//` command for an exception to this rule.)

The JSON object MAY be omitted (particularly if the command in question has no parameters). Any expected parameter fields which do not appear in the JSON object are automatically assumed to have an appropriate “zero” value (i.e., empty string, zero numeric value, empty list, boolean false, etc.). Any fields sent which were not expected by the receiver are silently ignored.

JSON field names are matched case-sensitively. A client MAY match names regardless of case but MUST emit them as documented here.

Field values are strings unless otherwise noted.

Unless otherwise noted, there is no response expected from these commands. Even where a response is expected, it will be asynchronously sent. In any case the client SHOULD NOT wait for a server response before considering its request to be completed. As a special case, a client MAY be implemented to wait synchronously for a server response during the authentication negotiation stage.

This describes protocol version 415. These notes are intended to be detailed enough to implement a client from and should be considered the definitive standard reference to the protocol.

In the protocol command descriptions, `BOLD` text indicates literal text to be sent as-is, *Italics* indicate a parameter whose value should appear in place of the name shown, and [square brackets] surround optional items which may be omitted.

The normal course of actions at the start of the conversation is for the server to immediately send an initial greeting beginning with a `PROTOCOL` command followed by comments, `AC`, `DSM`, `UPDATES`, and/or `WORLD` commands, then send an `OK` command to indicate to the client its protocol version and to issue an authentication challenge (if configured to authenticate). It is RECOMMENDED that the initial greeting be limited only to comments, waiting until after authentication to send anything more specific about your game.

After the `OK` command, the client responds with an `AUTH` command if it is required to authenticate. The

server will respond with `GRANTED` or `DENIED` and then MAY issue more of the initial greeting commands listed above.

Finally, the server will issue a `READY` command to the client and begin normal interaction with the client.

The server MUST NOT send the `AC`, `GRANTED`, `OK`, `READY`, `UPDATES`, or `WORLD` commands after the initial `READY` is sent. Clients MUST NOT send the `AUTH` command after receiving the server's `READY` signal. (Note that prior to protocol 414, servers were not allowed to send `DENIED` to clients after the `READY` message. Now they can send a `DENIED` message at any time to indicate that they wish to terminate the client's session immediately.)

Command Summary

The messages sent by clients and servers is summarized in the following table and then explained in detail in the following paragraphs. In the table, Message is the standard message name recommended for identifier names in source code, Command is the string actually sent over the network, C↔S indicates whether the message is only sent from client to server, only from server to client, or if it could be sent in either direction. Priv indicates if the command may only be sent by a client authenticated as the GM. Note that Reply indicates the command which SHOULD (eventually) arrive in response, but no guarantee is made as to if or when that happens since all communications are asynchronous.

Message	Command	C↔S	Reply	Priv	Payload	Description
Accept	`ACCEPT`	C→S	—	no	JSON	Subscribe to a set of messages
AddCharacter	`AC`	C←S	—	N/A	JSON	Add primary PC to party
AddDicePresets	`DD+`	C→S	`DD=`	no	JSON	Add presets to user's set
AddImage	`AI`	C↔S	—	no	JSON	Add image to client's known set
AddObjAttributes	`OA+`	C↔S	—	no	JSON	Add values to an attribute
AdjustView	`AV`	C↔S	—	no	JSON	Scroll clients' views
Allow	`ALLOW`	C→S	—	no	JSON	Indicate supported features
Auth	`AUTH`	C→S	`GRANTED`	no	JSON	Authenticate to server
Challenge	`OK`	C←S	`AUTH`	N/A	JSON	Ask client to authenticate
ChatMessage	`TO`	C↔S	—	no	JSON	Send chat message
Clear	`CLR`	C↔S	—	no	JSON	Remove objects from client
ClearChat	`CC`	C↔S	—	no	JSON	Clear the chat history
ClearFrom	`CLR@`	C↔S	—	no	JSON	Remove elements from a map file
CombatMode	`CO`	C↔S	—	yes	JSON	Turn on/off combat mode
Comment	`//`	C↔S	—	no	any	Human-readable comment
DefineDicePresets	`DD`	C→S	`DD=`	maybe	JSON	Store new presets for user
DefineDiceDelegates	`DDD`	C→S	`DD=`	maybe	JSON	Assign die-roll delegates
Denied	`DENIED`	C←S	—	N/A	JSON	Server denies access
Echo	`ECHO`	C↔S	`ECHO`	no	JSON	Send data for server to echo back to client
FilterCoreData	`CORE/`	C→S	—	yes	JSON	Filter availability of the game database
FilterDicePresets	`DD/`	C→S	`DD=`	maybe	JSON	Remove some presets for user
FilterImages	`AI/`	C→S	—	yes	JSON	Filter saved image definitions
Granted	`GRANTED`	C←S	—	N/A	JSON	Server grants access
LoadArcObject	`LS-ARC`	C↔S	—	no	JSON	Add an arc to the map
LoadCircleObject	`LS-CIRC`	C↔S	—	no	JSON	Add an ellipse to the map
LoadFrom	`L`	C↔S	—	no	JSON	Load elements from file
LoadLineObject	`LS-LINE`	C↔S	—	no	JSON	Add a line to the map
LoadPolygonObject	`LS-POLY`	C↔S	—	no	JSON	Add a polygon to the map
LoadRectangleObject	`LS-RECT`	C↔S	—	no	JSON	Add a rectangle to the map
LoadSpellArea-OfEffectObject	`LS-SAOE`	C↔S	—	no	JSON	Add an area of effect to the map
LoadTextObject	`LS-TEXT`	C↔S	—	no	JSON	Add some text to the map
LoadTileObject	`LS-TILE`	C↔S	—	no	JSON	Add a graphic tile to the map
Marco	`MARCO`	C←S	`POLO`	N/A	none	Check if client is alive
Mark	`MARK`	C↔S	—	no	JSON	Visually mark a spot

PlaceSomeone	PS	C↔S	—	no	JSON	Place creature token
Polo	POLO	C→S	—	no	none	Acknowledge server's ping
Priv	PRIV	C←S	—	N/A	JSON	Priveleged command denied
Protocol	PROTOCOL	C←S	—	N/A	int	Signal protocol version in use
QueryCoreData	CORE	C→S	CORE=	no	JSON	Query the core game database
QueryCoreIndex	COREIDX	C→S	COREIDX=	no	JSON	Query the core database index
QueryDicePresets	DR	C→S	DD=	maybe	JSON	Request user's presets
QueryImage	AI?	C↔S	AI	no	JSON	Ask for definition of image
QueryPeers	/CONN	C→S	CONN	no	none	Request list of peer clients
Ready	READY	C←S	—	N/A	none	Server sign-on complete
Redirect	REDIRECT	C←S	—	N/A	JSON	Instruct client to use a different server
RemoveObjAttributes	OA-	C↔S	—	no	JSON	Remove values from an attribute
RollDice	D	C→S	ROLL	no	JSON	Initiate a die roll
RollResult	ROLL	C←S	—	N/A	JSON	Result of a die roll
Sync	SYNC	C→S	any CC	no	none	Request replay of commands
SyncChat	SYNC-CHAT	C→S	ROLL TO	no	JSON	Request replay of messages
Toolbar	TB	C↔S	—	yes	JSON	Turn on/off client toolbar
UpdateClock	CS	C↔S	—	yes	JSON	Change the current time of day
UpdateCoreData	CORE=	C←S	—	N/A	JSON	Update a core data item
UpdateCoreIndex	COREIDX=	C←S	—	N/A	JSON	Update core data index
UpdateDicePresets	DD=	C←S	—	N/A	JSON	Receive user's presets
UpdateInitiative	IL	C↔S	—	yes	JSON	Update the initiative list
UpdateObjAttributes	OA	C↔S	—	no	JSON	Update object attributes
UpdatePeerList	CONN	C←S	—	N/A	JSON	Notify of all connected peers
UpdateProgress	PROGRESS	C↔S	—	no	JSON	Indicate progress on task
UpdateStatusMarker	DSM	C↔S	—	yes	JSON	Define creature status marker
UpdateTurn	I	C↔S	—	yes	JSON	Indicate turn in combat
UpdateVersions	UPDATES	C←S	—	N/A	JSON	Advertise software updates
World	WORLD	C←S	—	N/A	JSON	Campaign world info

Command Details

The client/server messages and their parameters are detailed below, ordered by the server command string sent or received.

`//` This is a server comment. The entire command SHOULD be ignored by all clients. This is used to inject informative context and messages into the client/server conversation which may be of interest for debugging or interactive use.

The mapper client will interpret a server comment beginning with the text "`notice:`" which is received during the initial connection negotiation (i.e., before the `READY` message is given by the server) as a notice that the end user should see. The text following the word "`notice:`" is posted in an alert box. Clients MUST recognize the string in all lower-case if they choose to implement this feature; they MAY also recognize it regardless of case.

This is one of two exceptions to the rule that a command contains a JSON parameter object. In this case, the entire *line should be ignored by the client and not interpreted further.* This allows, for example, comments to be sent at the start of the server's signon message to provide a human-readable declaration as to the allowed usage of the server.

`PROTOCOL` *v*

This command (which—if used at all—MUST be the first command sent by the server to the client) gives an up-front indication to the client as to what protocol version is expected by the server. Older versions of the server waited until the `OK` command to notify the client, but as of protocol version 400 this is now too late for the client to correctly process the commands that may be sent before the `OK`. This, along with the `//` server comment command,

are the only two commands which do *not* have a JSON data payload attached to them. The protocol version number is simply sent as an ASCII string of digits separated from the `PROTOCOL` command word by a space. The reason for this is that this data format is valid in all protocol versions, including those before version 400, so it can be used to unambiguously identify the server's protocol to all possible clients.

Servers which implement protocol versions 400 and later MUST send this message at the start of their conversations. Servers which implement older protocol versions SHOULD send it.

`AC` Add a character to the pop-up menu for map clients. This makes it easy to place important character tokens on the map and ensures that such tokens are given consistent *id* values rather than generating random ones. Clients MUST NOT send this command to each other; it is intended for the server to send to the clients. The payload includes the same fields as the `PS` command.

`ACCEPT` A client sends this message to the server to indicate that from this point forward (until another `ACCEPT` command), only the commands listed should be sent to it by the server. The payload is a JSON object with this parameter:

`Messages` *(list)*
: A list of command names the client wishes to receive. This is a list of command names as they appear in this specification (e.g., `["AI","CO","TO"]`). If it contains the special value "`*`" or is empty, this means to accept all messages. Note that the server may still decide to send messages such as comments or the `MARCO` command (or others) at its discretion despite this request from the client.

`AI` Add an image to the map. (Supersedes the function of the `AI`, `AI:`, `AI.`, and `AI@` commands as of protocol version 400.) The payload is a JSON object with the following fields:

`Animation` *(object)*
: If this value is given and non-null, this image is to be animated from individual frames. If the client is capable of animating multi-frame files, the base filename can be used directly. Servers offering this file SHOULD provide the multi-frame animated image file in a variety of formats, and SHOULD provide single-frame files in all formats in which they provide static images.

  Individual frame files are named by prefixing the base filename with "`:`*n*`:`" where *n* is the frame number, starting with 0. The `Animation` object has the following fields:

  `Frames` *(int)*
  : The number of frames in the animation.

  `FrameSpeed` *(int)*
  : The number of milliseconds to delay between frames.

  `Loops` *(int)*
  : If present and non-zero, the animation will play this many times before stopping. Otherwise, the animation plays forever.

`Name` The name of the image as known within the mapper.

`Sizes` *(list of objects)*
: A list of sizes available for this object. This will be *merged* with any previously defined sizes for the same image name. Each element of this list is an object with the following fields:

  `File` The filename or server ID by which the image can be retrieved.

  `ImageData` *(base64-encoded bytes)*
  : If non-empty or not `null`, this provides the raw image data. This is DEPRECATED but still supported. Instead, images SHOULD be loaded to the

game server and their ServerID used by clients.

`IsLocalFile` *(bool)*
: If `true`, then `File` gives a local file pathname for the image; otherwise it gives the server-specific ID which the client will use to retrieve the file from the server.

`Zoom` *(float)*
: The magnification level this bitmap represents for the given image. Typically images are provided for zoom levels of 0.25, 0.5, 1, 2, and 4.

This does not *draw* the image on the map; it merely defines it so the client knows what to draw when an image of the given `Name` at the specified `Zoom` factor is called for.

Note on clients retrieving images by server ID when `IsLocalFile` is `false` and `File` contains a server-side ID: as currently implemented, the `mapper` client checks to see if it has a cached version of the file. If so, and that file is newer than 2 days, it is used without further checks. If the cached file is older, the server is queried to see if it has a newer version of the file; if so, that is retrieved and cached; otherwise, the existing cache is updated to note that it was the known latest version as of that moment. If no usable cache file is available, the file with the given id is obtained from the server. For example, if the image id were "`abcdefg`", the URL of the file to be retrieved would be *base*`/a/ab/abcdefg.gif` or *base*`/a/ab/abcdefg.png` where *base* is the base URL as specified to the `--curl-url-base=`*base* option (or `curl-url-base=`*base* configuration file line).

Servers SHOULD be set up to provide alternative file formats so that, for example, with ID `abcdefg` files such as `a/ab/abcdefg.jpg` (JPEG format) or `a/ab/abcdefg.png` (PNG format) may be retrieved for clients which use those other formats. Currently, `mapper.tcl` uses GIF or PNG format files.

`AI?` Request for the named image. Clients may send this if they need an image of the given *name* and *zoom* (as described above for the `AI` command) but no such image is defined yet in the client. This queries the server or connected peers to see if any of them know of the needed image. The client should continue to process tasks without waiting for a response (which is never guaranteed). If another peer knows of the requested image, it should respond with an "`AI`" command. The payload is a JSON object with the following fields:

`Name` The image name as known to the mapper.

`Sizes` *(list of objects)*
: The list of sizes for which the image data are requested. Each element of the list is an object with the following field:

`Zoom` *(float)*
: The requested magnification factor for the image.

`AI/` Filters (removes) all the saved image definitions held in the server's database whose names match a regular expression. The payload is a JSON object with the following fields:

`Filter`
: The regular expression used to select which image names will be removed.

`KeepMatching` *(bool)*
: If true, only those images whose names match the `Filter` expression will be kept; the others will be removed. Otherwise, the opposite is done: images whose names match `Filter` will be removed.

`ALLOW` The client MAY send this command to the server after logging in. This tells the server that the client supports one or more optional features. The JSON payload has the following field:

`Features` *(list of strings)*
: This is a list of feature names which the client wishes to enable. Every time an `ALLOW` command is given, it resets the entire set of features, rather than adding to

them. Currently, the following one features are recognized:

`DICE-COLOR-BOXES`
: The client supports formatting controls in the die-roll title string sent to the server in the `D` command and received from the server via the `ROLL` command. Specifically, the title may consist of multiple sub-titles separated by U+2016 characters (‖). Each of these may also have a suffix consisting of the U+2261 (≡) character followed by a color name or #*rrggbb* color code, indicating the foreground color for that part of the title. A second such suffix may be added to indicate the background color. If this feature is not enabled, these formatting codes are stripped out by the server before sending die roll results to a client which isn't prepared to interpret them.

`DICE-COLOR-LABELS`
: The client supports formatting controls in die-roll label fields in the roll details in a format as described above for sub-titles. This allows custom colors to be used for special die-roll modifiers that need to stand out visually from all the rest.

`AUTH` If the server included a challenge string (see the `OK` command below), then it requires a password before the client is allowed to join the server. This is intended to prevent accidental connections by clients to the wrong servers, and to filter out nuisance connections from spammers or other random connections, so it is a fairly simple authentication mechanism using a password shared by all clients in a play group. The payload is a JSON object with the following parameters:

`Client`
: This describes the client program. By convention, this value SHOULD include the version number (e.g., "mapper 4.0.1").

`Response` *(base64-encoded bytes)*
: The client's response to the server's challenge.

`User` If provided, the `User` field gives the local username of the player who is running the client. This name will be displayed for purposes such as chat-window conversations and dice rolling. If not provided, "anonymous" will be used, unless the authentication is successful for the GM role, in which case "GM" will be used regardless of any value sent by the client for the `User` field.

Calculation of the `Response` field is as follows.

Given:

C is the server's challenge as a binary string of bytes (after decoding from the base-64 string actually sent by the server),

i is the number of rounds of hash function to apply,

$\mathrm{H}(x)$ is the binary SHA-256 hash digest of x,

P is the password needed to connect to the server, and the notation

$x\|y$ means to concatenate strings x and y,

To calculate the `Response` string, the client must do the following:

(1) If no value for i was provided by the server, obtain i by extracting the first two bytes from C as an unsigned, big-endian, 16-bit integer value.

(2) Calculate $D=\mathrm{H}(C\|P)$.

(3) Repeat i times: Calculate $D'=\mathrm{H}(P\|D)$; then let $D=D'$

The binary value of D is then encoded using base-64 and sent as the `Response` value.

Starting with protocol version 332, the server is guaranteed to respond to the `AUTH` command with either a `DENIED` or `GRANTED` message (q.v.). The client MAY wait for this response before proceeding with its other operations, so that it knows for sure how the authentication went.

`AV` Adjust the map's viewport by making the grid square identified by the `Grid` parameter visible at the top-left of the display. If this field is not present or is empty, then clients MAY fall back to the older behavior of adjusting the view by setting the horizontal scrollbar to `XView` as a fraction of its full distance, where 0.0 is all the way to the left and 1.0 is all the way to the right; the vertical scrollbar is similarly set based on `YView` where 0.0 is all the way to the top and 1.0 is all the way to the bottom. The payload is a JSON object with the following fields:

`Grid` The name of the grid square which should be in the upper-left of the display. This uses the same naming convention as the grid is labelled on the display (e.g., the far upper-left grid square of the entire map is `A0`).

`XView` *(float)*
The fraction of full distance the *x*-axis scrollbar should be moved to.

`YView` *(float)*
As `XView`, but for the *y* axis.

`CC` Clear the chat history. The payload is a JSON object with the following fields:

`RequestedBy`
The name of the user who initiated this operation, if known. Clients SHOULD NOT set this field; it is set by the server when sending this message to clients.

`DoSilently` *(bool)*
If true, the client MAY clear its chat history without notifying the user; otherwise it SHOULD indicate the action to the user.

`Target` *(int)*
If 0 or missing, clear all messages in the history. Otherwise, if positive, clear all messages with `MessageID` less than the `target` value. If negative, clear all but the most recent −*target* messages (e.g., a *target* of −50 clears all but the most recent 50 messages).

`MessageID` *(int)*
This message counts the same as a chat message and is included in the history itself. Thus, the server sets this field to a unique identifier for this message. Clients SHOULD NOT set this field.

`CLR` Remove object with a given internal ID from the map. The payload is a JSON object with the following field:

`ObjID` This may be the object ID for the specific object to remove, or "`*`" to mean all objects should be removed; "`E*`" to mean all map elements should be removed; "`M*`" to mean all monster tokens should be removed; "`P*`" to mean all player tokens should be removed; or the form "[*imagename*=]*name*") may be used, which removes a creature token whose display name matches the `name` value.

`CLR@` This command tells the client to "unload" the contents of a map file. In other words, all the elements listed in that file are removed from the map canvas rather than being added to it. The payload is a JSON object with the following fields:

`File` The pathname or server ID which specifies the map file in question.

`IsLocalFile` *(bool)*
If true, `File` refers to a local file the client can read directly. Otherwise it is a server ID which can be used to fetch the file from its cache or from the server.

`CO` Sets the combat mode state in the client. The payload is a JSON object with the following attribute:

`Enabled` *(bool)*
: If true, the client should be in combat (initiative) mode, otherwise it should not.

`CONN` Update the list of connected peer clients to those described in the JSON payload, which contains the following field. Clients MUST NOT send this command. This supersedes the `CONN`, `CONN:`, and `CONN.` commands from protocol versions prior to 400.

`PeerList` *(list of objects)*
: A list of objects, each describing a single connected client, with the following fields:

  `Addr` The IP address and port the peer connected from.

  `User` The username as provided during authentication.

  `Client`
  : The name of the client program in use.

  `LastPolo` *(float)*
  : The number of seconds since the server last heard a `POLO` command from the peer.

  `IsAuthenticated` *(bool)*
  : If true, the client successfully negotiated the server's authentication process.

  `IsMe` *(bool)*
  : If true, this object describes the client which received this command.

`CORE` Query the server's core SRD database. The server SHOULD respond with a `CORE=` message containing the requested item or indicating that no such item was found. The payload is a JSON object with the following attributes:

`Type` The type of object being requested. Currently the supported types include `bestiary`, `class`, `feat`, `language`, `skill`, `spell`, and `weapon`.

`Code` The GMA unique code that identifies the entry. May be blank since only one of `Code` and `Name` is necessary.

`Name` The common name of the entry. May be blank since only one of `Code` and `Name` is necessary.

`RequestID`
: If you place an arbitrary string value in this field, it will be sent back to you in the corresponding `CORE=` response message.

`CORE=` This message from the server provides core SRD data for an item that the client might want to know about. Typically, this is in response to a `CORE` message from the client requesting this information. Clients MUST gracefully deal with the receipt of a `CORE=` message they didn't need or are no longer interested in, as well as the case where they requested core data but received no corresponding response. The payload is a JSON object containing the requested data, with the following attributes:

`NoSuchEntry` *(bool)*
: If true, the requested entry was not found in the database. Otherwise (including the case where this attribute is missing) the remaining attributes contain the requested information.

`IsHidden` *(bool)*
: If true, the requested entry is hidden from player view by the GM. No `Data` field will be given.

`IsLocal` *(bool)*
: If present and true, this entry was a local addition to the database. Otherwise it was imported from publicly-available SRD data for the game system.

`Code` The GMA code for the requested item.

`Name` The GMA name for the requested item.

`Type` The type of core data item being provided, as per the `Type` field of the `CORE` message (q.v.).

`RequestID`
: If present and non-empty, this is the `RequestID` value provided by the client when making the request that resulted in this reply.

`Data` This value is a JSON-encoded object containing the specific data relating to the item in question. Its format depends on the item `Type`.

The `Data` payload for each type is described in the following sections. **N.B. The descriptions that follow of the various Data objects are speculative at this point and entirely subject to change.**

`Data` fields for `bestiary` type items:

`ImageTag`
: If the GM maintains a library of creature token images for their local bestiary, and one is defined for this creature, then this field will contain the image tag name for this creature's token.

`Species`
: The name of the species of the creature.

`CR` The challenge rating of the creature. Fractions may appear as, e.g., `1/2`.

`XP` *(int)*
: XP reward for defeating the creature.

`Class` The class and levels if applicable.

`Alignment` *(object)*
: The alignment for the creature, as an object with the following fields:

  `Alignments` *(list)*
  : A list of alignment codes (`CG`, `LN`, etc.) which this creature may be.

  `Special`
  : Any special comment about alignment.

`Source`
: The reference to the source this item originally came from.

`Size` *(object)*
: The creature's size, as an object with the following fields:

  `Code` The size code as documented in the GMA encounter and familiar data.

  `SpaceText`
  : Any comment about the creature's threatened space.

  `ReachText`
  : Any comment about the creature's reach distance.

`Type` Creature type.

`Subtypes` *(list)*
: List of applicable subtypes.

`Initiative` *(object)*
: The creature's initiative modifier, as an object with the following fields:

  `Mod` *(int)*
  : The total initiative modifier value.

  `Special`
  : Any special information related to initiative.

`Senses`
: A description of the creature's senses.

`Aura` A description of the creature's aura effect.

`HP` *(object)*
: The creature's hit points as an object with the following fields:

  `Typical` *(int)*
  : Typical hit point total.

  `Current` *(int)*
  : The creature's current hit point total.

  `Special`
  : Any special comments about the creature's health.

  `HitDice`
  : The hit dice die-roll specification.

`Save` *(object)*
: The creature's saving throws, as an object with these fields:

  `Fort` *(object)*
  : The Fortitude saving throw, as an object with these fields:

    `Mod` *(int)*
    : The total saving throw modifier value.

    `Special`
    : Any special commentary about this saving throw.

    `NoSavingThrow` *(bool)*
    : If true, the creature has no such saving throw at all and the other fields should be ignored.

  `Refl` *(object)*
  : The creature's Reflex saving throw, in the same format as the Fortitude saving throw.

  `Will` *(object)*
  : The creature's Will saving throw, in the same format as the Fortitude saving throw.

  `Special`
  : Any special comments about the creature's saving throws.

`DefensiveAbilities`
: A description of the creature's defensive abilities.

`DR` *(object)*
: The creature's damage reduction, as an object with the following fields:

`DR` *(int)*
: The number of hit points resisted.

`Bypass`
: Description of what will bypass the DR.

`Immunities`
: A description of the things the creature is immune from.

`Resists`
: A description of the resistances the creature has.

`Speed` *(object)*
: The creature's movement speed, as an object with the following fields:

`Code` The encoded list of speeds.

`Special`
: Additional commentary about movement speed.

`SpecialAttacks`
: A description of the various special attacks the creature may make.

`Resists`
: A description of the resistances the creature has.

`Speed` *(object)*
: The creature's movement speed, as an object with the following fields:

`Code` The encoded list of speeds.

`Special`
: Additional commentary about movement speed.

`SpecialAttacks`
: A description of the various special attacks the creature may make.

`Abilities` *(object)*
: The creature's ability scores, as an object with the following fields:

`Str` *(object)*
: The Strength score, as an object with the following fields:

`Base` *(int)*
: The raw ability score value.

`Special`
: Any special notes about this ability score.

`NullScore` *(bool)*
: If true, the creature does not have this ability score at all, and the other fields should be ignored.

`Dex` *(object)*
: The Dexterity score, as an object in the same format as Strength.

`Con` *(object)*
: The Constitution score, as an object in the same format as Strength.

`Int` *(object)*
: The Intelligence score, as an object in the same format as Strength.

`Wis` *(object)*
: The Wisdom score, as an object in the same format as Strength.

`Cha` *(object)*
: The Charisma score, as an object in the same format as Strength.

`Combat` *(object)*
: The combat statistics for the creature, as an object with the following fields:

  `BAB` *(int)*
  : The base attack bonus.

  `CMB` *(int)*
  : The combat maneuver bonus.

  `CMD` *(int)*
  : The combat maneuver defense value.

  `CMBSpecial`
  : Any special notes on the CMB.

  `CMDSpecial`
  : Any special notes on the CMD.

`SQ`
: A description of the creature's special qualities.

`Environment`
: A description of the environment where this creature commonly appears.

`Organization`
: A description of the groups in which this creature is usually found.

`Treasure`
: The treasure type and amount usually found with this creature.

`Appearance`
: A short description of the creature's physical appearance.

`IsTemplate` *(bool)*
: If true, this is a template from which other creatures can be built.

`Strategy` *(object)*
: The creature's strategy in an encounter, as an object with these fields:

  `BeforeCombat`
  : How the creature will prepare for the encounter.

  `DuringCombat`
  : How the creature will manage its encounter with the players.

  `Morale`
  : How resolute the creature is in the face of deadly combat.

`IsCharacter` *(bool)*
: If true, this is a creature that may be a character.

`IsCompanion` *(bool)*
: If true, this creature may be a companion.

`IsUnique` *(bool)*
: If true, there is only one of this creature in existence.

`AgeCagetory`
: The age category for the creature.

`Gender`
: The creature's gender

`Bloodline`
: The creature's bloodline.

`Patron`
: The creature's patron deity.

`AlternateNameForm`

`DontUseRacialHD` *(bool)*
: If true, don't use racial hit dice when creating a character with levels.

`VariantParent`

`Mythic` *(object)*

- `IsMythic` *(bool)*
- `MR` *(int)*
- `MT` *(int)*

`OffenseNote`

`StatisticsNote`

`Gear` *(object)*

- `Combat`
- `Other`

`Schools` *(object)*

- `Focused`
- `Prohibited`
- `Opposition`

`ClassArchetypes`

`BaseStatistics`

`RacialMods`

`Mystery`

`Notes`

`Domains` *(list)*

`AC` *(object)*

- `Components` *(object)*
- `Adjustments` *(object)*

`AttackModes` *(list of objects)*

- `Tier` *(int)*
- `BaseWeaponID`
- `Multiple` *(int)*
- `Name`
- `Attack`
- `Damage`

```
            Critical (object)
                  CantCritical (bool)
                  Threat (int)
                  Multiplier (int)
            Ranged (object)
                  IsRanged (bool)
                  Increment (int)
                  MaxIncrements (int)
            IsReach (bool)
            Special
            Mode
      Languages (list of objects)
            Name
            IsMute (bool)
            Special
      Feats (list of objects)
            Code
            Parameters
            IsBonus (bool)
      Skills (list of objects)
            Code
            Modifier (int)
            Notes
      Spells (list of objects)
            ClassName
            CL (int)
            Concentration (int)
            NoConcentration (bool)
            PlusDomain (int)
            Description
            Special
            Spells (list of objects)
                  Name
                  AlternateName
                  Frequency
                  Special
                  Slots (list of objects)
                        IsCast (bool)
```

`IsDomain` *(bool)*

`MetaMagic` *(list)*

`Data` fields for `class` type items:

`Spells` *(object)*
Describes the spell-casting capability of the class, with the following fields:

`Type` The magic type (arcane, etc.)

`Ability`
The name of the ability score relevant to spells.

`HasBonusSpells` *(bool)*
True if the class allows bonus spells (e.g., clerics' domain spells)

`IsSpontaneousCaster` *(bool)*
True if the class casts spells without needing to prepare them in advance.

`CastPerDay` *(list)*
A list of the number of spells which may be cast per day. Each element is an object which has the following values:

`ClassLevel` *(int)*
The class level for which this applies.

`SpellLevel` *(int)*
The level of spell for which this applies.

`IsProhibited` *(bool)*
If true, this level of spell is not possible at this class level.

`IsUnlimitedUse` *(bool)*
If true, this level of spell can be used an unlimited number of times per day at this class level.

`Number` *(int)*
The number of spells granted at this spell and class level.

`PreparedPerDay` *(object)*
The number of spells which may be prepared per day by class level and spell level. Each element has the same values as for the `CastPerDay` attribute above.

`SpellsKnown` *(object)*
The number of spells which may be known by class level and spell level. Each element has the same values as for the `CastPerDay` attribute above.

`Data` fields for `feat` type items:

`Parameters`
Describes the parameters allowed for the feat, if any.

`Description`
The text description of the feat.

`Prerequisites`
The text description of the prerequisites for the feat.

`Benefit`
: Description of the benefit granted by taking this feat.

`Normal`
: Description of the situation if the feat is not taken.

`Special`
: Any special notes or circumstances.

`Source`
: The reference to the source book or other place where this feat origially came from.

`Race`

`Note`

`Goal`

`CompletionBenefit`

`SuggestedTraits`

`Types` *(list)*
: A list of feat types to which this belongs.

`FlagNames` *(list)*
: A list of flags which apply to this feat.

`MetaMagic` *(object)*
: If this is a metamagic feat, the following fields will appear in the `MetaMagic` attribute:

  `IsMetaMagicFeat` *(bool)*
  : If true, this is a metamagic feat and the other fields here will apply.

  `Adjective`

  `LevelCost` *(int)*
  : The number of levels this feat costs (i.e., the number of spell slots higher than normal this spell will use).

  `IsLevelCostVariable` *(bool)*
  : If true, the spell level cost is not fixed.

  `Symbol`
  : The symbol to use on GMA character record sheets and encounter run sheets for spells which were prepared with this feat.

`Data` fields for `language` type items:
: There is no additional data. The language name appears in the `Name` field.

`Data` fields for `skill` type items:

`ClassSkillFor` *(list)*
: A list of the class codes for which clases this skill is a class skill.

`Ability`
: Relevant ability score name.

`HasArmorPenalty` *(bool)*
: If true, this skill is subject to an armor check penalty.

`TrainingRequired` *(bool)*
: If true, the character must be trained to use this skill.

`Source`
: The reference to the original source for this skill.

`Description`
: The abbreviated text description of the skill.

`FullText`
: The full text description of the skill.

`ParentSkill`
: The name of the parent skill if applicable.

`IsVirtual` *(bool)*
: If true, this skill cannot be taken; one of its children must be taken specifically.

`IsBackground` *(bool)*
: If true, this is a background skill.

`Data` fields for `spell` type items:

`School`

`Descriptors` *(list)*

`Components` *(object)*

- `Components` *(list)*
- `Material`
- `Focus`
- `HasCostlyComponents` *(bool)*
- `MaterialCosts` *(int)*

`Casting` *(object)*

- `Time`
- `Special`

`Range` *(object)*

- `Range`
- `Distance` *(int)*
- `DistancePerLevel` *(int)*
- `DistanceSpecial`

`Effect` *(object)*

- `Area`
- `Effect`
- `Targets`

`Duration` *(object)*

- `Duration`
- `Special`
- `Concentration` *(bool)*
- `PerLevel` *(bool)*

`SR` *(object)*

SR

Special

Object *(bool)*

Harmless *(bool)*

Save *(object)*

SavingThrow

Effect

Special

Object *(bool)*

Harmless *(bool)*

IsDismissible *(bool)*

IsDischarge *(bool)*

IsShapeable *(bool)*

ClassLevels *(list of objects)*

Class

Level *(int)*

Deity

Domain

Description

Source

Bloodline

Patron

`Data` fields for `weapon` type items:

`Cost` *(int)*
: The cost in copper pieces for a standard weapon of this type.

`Damage` *(object)*
: The damage done by this weapon. The value is an object where the keys are size category codes and the values are the damage done by a weapon of that size category.

`Critical` *(object)*
: The critical hit stats for the weapon. This is an object with the following fields.

`CantCritical` *(bool)*
: If true, this weapon cannot score a critical hit at all. The other fields should be ignored.

`Multiplier` *(int)*
: The number of extra dice to add when a critical hit is scored.

`Threat` *(int)*
: The minimum die value to threaten a critical hit (e.g., 19 means to threaten on a natural 19–20).

`Ranged` *(object)*
: For ranged weapons, the fields in this attribute describe

those parameters.

`Increment` *(int)*
: The range increment for the weapon.

`MaxIncrements` *(int)*
: The maximum number of increments before the target is out of range.

`IsRanged` *(bool)*
: If true, this is a ranged weapon and the other fields are meaningful.

`Weight` *(int)*
: The weight in grams of the weapon.

`DamageType` *(list)*
: The type of damage done. This is a list of strings, with each indicating a damage type such as `B` for bludgeoning, `P` for piercing, and `S` for slashing.

`Qualities` *(list)*
: A list of weapon qualities. The codes in the list may include: `1` (one-handed), `2` (two-handed), `b` (brace), `D` (disarm), `d` (double), `E` (Elven), `f` (fragile), `G` (Gnome), `g` (grapple), `H` (Halfling), `L` (light), `M` (monk), `m` (martial), `O` (Orc), `R` (ranged), `r` (reach), `S` (non-lethal), `s` (simple), `t` (trip), `U` (unarmed), `W` (Dwarven), `X` (exotic), `x` (deadly), and `Z` (masterwork).

`CORE/` Filter the core database items which are visible to the players. The payload is a JSON object with the following fields:

`Filter`
: This is a regular expression matching the `Code` value of all the entries to be affected by this command.

`Type` The type of core data entry to target.

`IsHidden` *(bool)*
: If true, set the entries matching the `Filter` expression so they are hidden from the players. Otherwise, make them visible.

`InvertSelection` *(bool)*
: If true, all entries which do *not* match `Filter` are affected.

`COREIDX` Query the core database to retrieve a list of entries. This gives the information by which the actual data behind these entries may be retrieved using the `CORE` message. The payload includes the following fields:

`Type` The type of entry to index, as per the `CORE` message.

`CodeRegex`
: If present and non-empty, limit the responses to those whose `Code` field matches this regular expression.

`NameRegex`
: If present and non-empty, limit the responses to those whose `Name` field matches this regular expression.

`Since` *(RFC 3339-formatted datetime string)*
: If present and non-empty, limit the responses to those which were modified since the given date and time.

`RequestID`
: If a string value is provided here, it will be repeated in the server's response message.

`COREIDX=` The server response to a `COREIDX` message. Since the number of entries in the database may be quite large, the server will send a number of these reply messages, one per item from the database. It includes the following fields:

`RequestID`
: The `RequestID` sent to the server in the `COREIDX` request that initiated this update, if any.

`Type` The database item type

`IsDone` *(bool)*
: If true, this is the final response in this set.

`N` *(int)* The number of response this is, starting with 1.

`Of` *(int)*
: The total number of responses that will be given.

`Name` The item's name.

`Code` The item's code.

`CS` Update the client's clock. The JSON payload includes the following fields:

`Absolute` *(int64)*
: The absolute time on the GMA world clock (in tenths of seconds since the epoch).

`Relative` *(int64)*
: The elapsed time on the GMA world clock (in tenths of seconds since the GM set a reference point, e.g., the start of combat).

`Running` *(bool)*
: If true, the local client should continue advancing the clock in real-time locally after updating the time to the new absolute time. Otherwise, cancel real-time updates if they were running, leaving the clock at the given absolute time. The local real-time updating events should only be performed when not in combat mode.

`D` Roll dice using the server's built-in die rolling facility. The JSON payload contains the following fields:

`Recipients` *(list of strings)*
: The names of the people who should receive the results of the die roll. For global or GM results, this should be `null`.

`RequestID`
: If you put a string value in this field, it will be sent back in any `ROLL` response messages the server sends to you in response to this request. This allows a client to associate responses to the requests that generated them. You may leave this blank or omit it if you don't care to match requests with responses.

`ToAll` *(bool)*
: If true, this is a global die-roll, and the result should be sent to all clients.

`ToGM` *(bool)*
: If true, this die-roll will be sent "blind" to the GM only. Not even the requester will see the result of the roll, only the GM will.

`RollSpec`
: The die-roll specification, like "d20+12" or "6d6 fire". If this field is empty, the previous die-roll from this client is repeated, or "1d20" if there was no previous one.

`DD` Define (store) Die-Roll Presets. Clients send this command to the server to request it to store a set of die-roll presets for it to retrieve for use in subsequent sessions. The JSON payload is identical to the one sent by the server in the `DD=` command, except that it does not contain `DelegateFor` or `Delegates` fields.

The server SHOULD respond by issuing a `DD=` command to all peer clients logged in with the same username as the affected user as well as any authorized delegates.

`DD+` Add to Die-Roll Presets. Clients send this command to the server to add an additional set of die-roll presets to the collection it was already holding for that user. The JSON payload is identical to the one sent for the `DD` command. The server SHOULD respond by issuing a `DD=` command to all peer clients logged in with the same username as the requesting client and all authorized delegates.

`DD/` Filter die-roll presets. This removes a set of die-roll presets from the server's storage for the user. The server SHOULD respond by issuing a `DD=` command to all peer clients logged in with the same username as the affected user and all authorized delegates. The JSON payload includes the following fields:

`Filter`
: The value is a regular expression. The server will delete all die-roll presets whose `Name` attribute matches this regular expression.

`For` The name of the user whose presets are being filtered. This is optional; if omitted, the presets of the user making this request are affected. Only the GM and authorized delegates may filter other users' presets.

`DD=` This command defines the list of die-roll presets the client should know. This replaces any previous set of presets the client was using. Clients MUST NOT send this command. This supersedes the commands `DD=`, `DD:`, and `DD.` in protocol versions prior to 400. The JSON payload includes the following fields:

`For` The username of the player for whom these presets were stored. The default, if no name is given or if it is empty, is the currently logged-in user.

`DelegateFor` *(list of strings)*
: The list of all the other users for whom the target user is a delegate.

`Delegates` *(list of strings)*
: The list of delegates who also have visibility and control for the owner's presets. If empty or missing, there are no delegates.

`Presets` *(list of objects)*
: Each element in this list describes a single preset. It has the following fields:

`Name` The name by which this preset is identified to the user. This MUST be unique for that user. The client SHOULD sort the preset list by the `Name` field before displaying. If a vertical bar ("`|`") appears in the `Name`, all text up to and including the bar SHOULD be suppressed from display.
The formatting of the `Name` field is not mandated by this specification. However, the following standard interpretation for data to the left of the vertical bar (if any) is RECOMMENDED for clients using this feature:

1. Regular die-roll presets with a preferred display ordering SHOULD have a three-digit decimal zero-padded sequence number to the left of the bar, e.g., "`000|attack`", "`001|damage`", "`002|fort save`", etc. The numbers need not be strictly consecutive. Clients SHOULD use dictionary sorting for preset items so that the numeric sort-order sequence value is correctly arranged regardless of number of characters; however the recommendation here is also to pad them with zeroes to three digits for the benefit of clients which cannot or will not do so.

2. Stored variables MAY be stored as a die-roll preset with a `Name` field in the format

   "`§`*sequence*`;`*var*`;`*flags*`[;`*client*`]|`*displayname*".

   where `§` is Unicode character U+00A7 (section symbol), *sequence* is a three-digit decimal zero-padded display sequence number as described above, *var* is the name of the variable by which the contents of this preset may be inserted into another die-roll specification, *flags* are a set of zero or more single-letter flags affecting how the client should use this value, and *client* is an optional field with client-specific data not otherwise defined here. If the *client* field is not given, the semicolon immediately before it may be omitted as well. For example:

   "`§042;CL;e|caster level`"

   The *flags* are up to the client to interpret, but we RECOMMEND the flag "`e`" to mean the value should be enabled (or "on") by default and "`g`" to mean the die-roll expression to the left of it should be grouped together in parentheses.

   If the *var* value is empty (e.g., "`§000;;e|inspiration`"), clients SHOULD interpret this to mean that the preset's value should not be assigned to a variable name, but should be appended to any die-roll specification made while it is enabled.

3. It is RECOMMENDED that clients allow for grouping of related presets and/or modifiers by appending one or more group names to the end of the numeric sort order digits. Each such name is separated from the digits and each other with a right-pointing triangle Unicode U+25B6 (represented here as the ">" character). For example, `012|attack` is an ungrouped attack roll, `013>monster|attack` is an attack roll belonging to the "monster" group, `030>monster>save|Fortitde` is a Fortitude saving throw belonging to the "save" subgroup of the "monster" group, and `§042>monster;;|mod` is a modifier in the "monster" group. Clients SHOULD allow users to collapse all of the die rolls with the same group name(s) appearing consecutively in the die-roll preset list.

4. Clients MUST gracefully handle the existence of data formats not listed above without losing or corrupting any presets or metadata.

   Notably, the `gma`(6) program itself automates the creation and destruction of ephemeral presets for combatants on an encounter-by-encounter basis. The names for these presets include an additional component, *area*, which identifies the dungeon location or encounter identifier. This means the GMA preset names have these forms:

   `$[`*area*`]`*sequence*`|`*area*
   `$[`*area*`]`*sequence*`>`*monster*`|`*area*
   `$[`*area*`]`*sequence*`>`*monster*`>saves|`*area*
   `$[`*area*`]`*sequence*`>`*monster*`>skills|`*area*
   `$[`*area*`]`*sequence*`>`*monster*`>other|`*area*
   `$[`*area*`]`*sequence*`>`*monster*`>special|`*area*
   `§`*sequence*`;`*var*`;`*flags*`;$[`*area*`]|`*area*
   `§`*sequence*`>`*monster*`;`*var*`;`*flags*`;$[`*area*`]|`*area*
   (Again, the `>` here represents the Unicode codepoint U+25B6.)

`Description`
: A text description of the purpose of the preset.

`DieRollSpec`
: The die-roll specification to send to the server when rolling this preset.

`DDD`
: Define Die-roll Delegate. Clients send this command to the server to indicate that other user(s) are allowed to retrieve and store their die-roll presets. The server SHOULD immediately send `DD=` commands to the requesting user and all delegates. From this point on, any updates to the requesting user's presets will result in `DD=` commands sent to all delegates as well. The JSON payload includes:

  `For`
  : The user for whom these delegates are being defined. If empty or missing, the default is to define them for the requesting user. Only the GM may set delegates for someone else.

  `Delegates` *(list of strings)*
  : The list of usernames who are allowed to access the requesting user's die-roll presets. This replaces the previous set of delegates. If the list is empty (or the `Delegates` field missing), then all delegates are removed.

`DENIED`
: Access to the server is denied. The server will send this to a client; clients MAY NOT send it. When a client receives this message, it MUST not continue to communicate with the server. In fact, the server SHOULD terminate the network connection after sending the `DENIED` message. The JSON payload sent with this command includes the field:

  `Reason`
  : Text description of why the access was refused.

`DR`
: Retrieve the authenticated user's die roll presets. This SHOULD result in the server sending the `DD=` command back to the requesting client. The JSON payload may contain the following field:

  `For`
  : The name of the user whose presets are being requested. Only the GM or an authorized delegate may request other users' presets. If this field is empty or missing, the requesting user's presets are requested.

`DSM`
: Defines (or re-defines) a condition status marker which the client SHOULD use to mark creature tokens when the creature has the associated condition in effect. The JSON payload includes the following fields:

  `Condition`
  : The name of the condition being described. If there was already a marker defined for that condition, it is replaced with the new definition. If either `Shape` or `Color` is the empty string or `null`, then the condition is effectively removed from the list of conditions known to the mapper. Creature conditions are in effect if their name appears in the list value for that creature's `StatusList` attribute. The mapper comes with the following conditions pre-defined: `ability damage`, `ability drained`, `bleed`, `blinded`, `confused`, `cowering`, `dazed`, `dazzled`, `dead`, `deafened`, `disabled`, `dying`, `energy drained`, `entangled`, `exhausted`, `fascinated`, `fatigued`, `flat-footed`, `frightened`, `grappled`, `helpless`, `incorporeal`, `invisible`, `nauseated`, `panicked`, `paralyzed`, `petrified`, `pinned`, `poisoned`, `prone`, `shaken`, `sickened`, `stable`, `staggered`, `stunned`, and `unconscious`.

  `Shape` The shape of the marker to be placed if the creature has this condition. The mapper will attempt to arrange multiple markers with the same shape such that they are all visible at the same time. This value may be one of the following:

    `|v` A small downward-pointing triangle against the middle of the left edge of the token. (This is a lower-case "v".)

`v|` A small downward-pointing triangle against the middle of the right edge of the token. (This is a lower-case "v".)

`|o` A small circle against the middle of the left edge of the token. (This is a lower-case "o".)

`o|` A small circle against the middle of the right edge of the token. (This is a lower-case "o".)

`|<>` A small diamond against the middle of the left edge of the token.

`<>|` A small diamond against the middle of the right edge of the token.

`/` A slash (upper right to lower left) through the entire token.

`\` A back-slash (upper left to lower right) through the entire token.

`//` A double slash (upper right to lower left) through the entire token.

`\\` A double back-slash (upper left to lower right) through the entire token.

`-` A single horizontal line drawn through the center of the entire token.

`=` A double horizontal line drawn through the center of the entire token.

`|` A single vertical line drawn through the center of the entire token.

`||` A double vertical line drawn through the center of the entire token.

`+` A cross drawn through the entire token.

`#` A hash-mark drawn through the entire token.

`V` A large downward triangle drawn around the entire token. (This is an upper-case letter "V".)

`^` A large upward triangle drawn around the entire token.

`<>` A large diamond drawn around the entire token.

`O` A large circle drawn around the entire token. (This is an upper-case letter "O".)

`Color` The color to draw the marker in any of the forms documented above, or the special value "`*`", which means to draw the marker in the same color as the creature's threatened area.

If *color* begins with "`--`" (e.g., "`--red`"), then the marker is drawn with dashed lines instead of solid ones. If it begins with "`..`" (e.g., "`..blue`"), then the effect is the same, but the dashes are shorter.

`Transparent` *(bool)*
If true, the creature token should be displayed in a semi-transparent form whenever it has this condition (added in protocol 404).

`Description`
A description of the effects on the character of having that condition applied. This is intended to be shown to players (for example, if they hover their mouse over an affected creature token).

`ECHO` Ask that the server send this message back to you. This may be used for synchronization since it indicates to the client when the server got to this request amongst the others it received. The following fields may be populated:

`s` An arbitrary string value.

`i` *(int)* An arbitrary integer value.

`b` *(bool)* An arbitrary boolean value.

o *(object)* An arbitrary set of key/value pairs, with each value having any type.

The server will also populate the following additional fields in its reply, which a client can use to calculate the round-trip time for the request and identify how much of that was internal to the server and how much was network latency.

`ReceivedTime` *(RFC 3339-formatted datetime string)*
The time when the server received the packet from the client.

`SentTime` *(RFC 3339-formatted datetime string)*
The time when the server sent the response packet to the client.

`GRANTED` Access to the server is granted. The server will send this to a client; no client should send it. The JSON payload contains the field:

`User` The user name for this client. This is the name provided by the client, "`anonymous`" if the client didn't give one, or "`GM`" if the client authenticated as the GM.

`I` Update the time clock for initiative-based actions. If the mapper is in combat mode, the clock display is updated accordingly. The JSON object payload includes the following fields:

`ActorID`
The object identifier of the creature whose turn it is. This may be the unique object ID code, the creature name as documented in the `AC` command, the special string "`*Monsters*`" which indicates that all creatures with monster-type tokens have initiative, or may be of the form "*/regex*", which matches all creatures whose names match the regular expression *regex*.

`Hours` *(int)*
The number of hours elapsed since the start of combat.

`Minutes` *(int)*
The number of minutes elapsed since the start of this hour of combat.

`Seconds` *(int)*
The number of seconds elapsed since the start of this minute of combat.

`Rounds` *(int)*
The number of rounds elapsed since the start of combat.

`Count` *(int)*
The number of initiative slots elapsed since the start of the round.

`IL` Update the initiative list. The JSON payload includes the following field:

`InitiativeList` *(list of objects)*
The current initiative list is given as a list of initiative slots, each of which is an object with the following fields:

`Slot` *(int)*
The slot number. As currently implemented this is a number in the range [0,59] which gives the "count" (1/10th second) in the round where this creature may act.

`CurrentHP` *(int)*
The creature's current hit point total.

`Name` The creature's name as displayed on the map.

`IsHolding` *(bool)*
If true, the creature is holding their action.

`HasReadiedAction` *(bool)*
: If true, the creature is holding a readied action.

`IsFlatFooted` *(bool)*
: If true, the creature is flat-footed.

`L`
: Loads map elements into the map client, either replacing or adding to the current contents of the map. Supersedes the function of commands `L`, `M`, `M?`, and `M@` in protocol versions prior to 400. The payload is a JSON object with the following fields:

  `File` The local pathname or server ID identifying the map file to be loaded.

  `IsLocalFile` *(bool)*
  : If true, `File` refers to a local filename; otherwise it is a server ID.

  `CacheOnly` *(bool)*
  : If true, the server is only advising the client that it would be good to have a cached copy of the file on hand for later. The client MUST NOT actually load the file's contents to the displayed map at this time.

  `Merge` *(bool)*
  : If true, the map elements in `File` are merged with the map's current contents. Otherwise the map's current elements are replaced by the new ones in `File`.

`LS-`*type*
: This set of commands load a single map element into the map. As opposed tothe `L` command which directs the client to read a file to get one or more objects, this just sends an object directly to it. This may be used, for example, when one client draws an element interactively and wants the other clients to display it as well.

  These supersede the function of the commands `LS`, `LS:`, and `LS.` in protocol versions prior to 400.

  All of the following `LS-`*type* commands include as many of the following parameters as are applicable to them, in addition to type-specific parameters:

  `ID` The unique object identifier. This is a string containing upper- or lower-case letters, digits, underscores and octothorpes. By convention, we create these as hexadecimal UUID values, but they may be any arbitrary string, including human-readable IDs such as "PC1", etc.

  `X` *(float)*
  : The *x* coordinate of the "reference point" of the element, in standard map pixel units.

  `Y` *(float)*
  : The *y* coordinate of the "reference point" of the element, in standard map pixel units.

  `Points` *(list of coordinate objects)*
  : If an object needs more than a single coordinate pair to specify their location, (e.g., the diagonally opposite corner of a rectangle) the subsequent points are listed in this parameter. Each element in the list is an object with the fields:

    `X` *(float)*
    : The *x* coordinate in standard map pixel units.

    `Y` *(float)*
    : The *y* coordinate in standard map pixel units.

  `Z` *(int)* The *z* "coordinate" of the element is its vertical stacking order on the displayed map canvas. Higher numbers are drawn after lower numbers. If two objects have the same *z* value and physically overlap, the result is not defined.

  `Line` The color used to draw the shape's outline, as a standard color name or RGB string such as "#336699".

`Fill` As with the `Line` field, specifies a color to fill the interior of the element. If omitted or empty, the object will not be filled. Note that line objects are *filled* with the `Fill` color. They don't have an outline so don't use the `Line` value.

`Stipple`
If nonempty, the value is the name of a bitmap image which is assumed to be known to the client. The shape will be filled in with the `Fill` color using this bitmap to form a stipple pattern in that color. If the `Stipple` bitmap is empty or unknown to the client, a solid color fill is used instead. If the `Fill` field is empty, then `Stipple` is ignored. Clients SHOULD by default understand stipple bitmap names "`gray12`", "`gray25`", "`gray50`", and "`gray75`".

`Width` *(int)*
The width in pixel units of the element's outline.

`Layer` The map layer this element belongs to. Currently not implemented.

`Level` The dungeon level where this element appears. Currently not implemented.

`Group` The object group to which this element belongs. Currently not implemented.

`Dash` *(int)*
The outline of the element is to be drawn with the specified dash pattern:

0 Solid (the default)

1 Long dashes

2 Medium dashes

3 Short dashes

4 Long-Short pattern

5 Long-Long-Short pattern

`Hidden` *(bool)*
If true, this element MUST NOT be displayed on-screen. Currently not implemented by the mapper client.

`Locked` *(bool)*
If true, this element MUST NOT be edited further by clients.

Each of the following commands which begin with `LS-` defines a different kind of map element.

`LS-ARC`
Draws an arc on the canvas. The arc is drawn around the circumference of the ellipse inscribed in the rectangle defined by the reference point and the first point in the `Points` attribute (as opposite corners of a rectangle). The payload is a JSON object with these parameters in addition to those common to all map elements:

`ArcMode` *(int)*
Specifies how to draw the arc on-screen.

0 Pie slice (default); connects the arc endpoints to the center of the circle.

1 Arc; does not connect the endpoints to anything else.

2 Chord; connects the endpoints to each other via a straight line segment.

`Start` *(float)*
The number of degrees around the circle to begin drawing the arc.

`Extent` *(float)*
The number of degrees around the circle to end drawing the arc.

`LS-CIRC`
Draws an ellipse on the map canvas. The payload is a JSON object as described for all map elements above. The ellipse is defined as described for the `LS-ARC` command, but the entire circumference is drawn.

`LS-LINE`
Draws a straight line segment from the reference point to the each point in the `Points` attribute (if more than one point is in `Points` the result will be multiple connected line segments). The JSON payload has the following attribute in addition to the common ones described above:

`Arrow` *(int)*
The style of arrows to draw on the ends of the line:

0 No arrows

1 Arrow on the first point (the reference)

2 Arrow on the last point

3 Arrows on both first and last points

`LS-POLY`
Draws a polygon on the map canvas. The vertices of the polygon start at the reference point and continue through every point in the `Points` attribute, and finally connecting back to the reference point again. The JSON payload contains the following fields in addition to the common ones described above:

`Spline` *(int)*
The factor to use when smoothing the sides of the polygon between its points. 0 means not to smooth at all, resulting in a shape with straight edges between the vertices. Otherwise, larger values provide more smoothing.

`Join` *(int)*
The join style for the edges of the polygon:

0 Beveled corners

1 Mitered corners

2 Rounded corners

`LS-RECT`
Draws a rectangle defined by the reference point and the first point in the *points* attribute (as opposite corners of the rectangle).

`LS-SAOE`
Draws the zone of a spell's area of effect on the canvas. The JSON payload contains the following field in addition to those described above:

`AoEShape` *(int)*
The shape of the area of effect:

0 Cone: a 90° pieslice defined as described for a pieslice arc element.

1 Radius: an ellipse defined as described for a circle element.

2 Ray: a rectangle defined as described for rectangle elements.

`LS-TEXT`
Places some text on the canvas. The JSON payload includes the following fields in addition to those common to all elements:

`Text` The text to be displayed.

`Font` *(object)*
: The typeface to use for the text. This is an object with the following fields:

  `Family`
  : The name of the font family as recognized by Tk.

  `Size` *(float)*
  : The font size as recognized by Tk. The client MAY truncate this to an integer if it can't handle fractional point sizes.

  `Weight` *(int)*
  : The type wight to use. This may be 0 for normal or 1 for boldface.

  `Slant` *(int)*
  : The type slant to use. This may be 0 for Roman (normal) or 1 for Italic (slanted).

`Anchor` *(int)*
: Where the reference point is considered to be relative to the text:

  0 Center

  1 North

  2 South

  3 East

  4 West

  5 Northeast

  6 Northwest

  7 Southwest

  8 Southeast

`LS-TILE`
: Draws an image tile on the canvas. The JSON payload has the following fields in addition to those common to all map elements:

  `Image` The image name as known to the mapper.

  `BBHeight` *(float)*
  : The bounding-box height for the region bordering the image.

  `BBWidth` *(float)*
  : The bounding-box width for the region bordering the image.

`MARCO` Status check. If received, reply with a `POLO` command. Clients MUST NOT send this command.

N.B. It is important that your client not ignore these occasional ping messages. For example, if your client is too slow receiving messages such that the server needs to expend extra work to queue them up for you, it will be willing to do so if you have been at least responding to `MARCO` messages. If you haven't, the server will suspect your client has locked up or is not going to be able to catch up with the data being sent to it, and may decide to terminate the client's connection.

`MARK` Make a brief animated marker at the specified coordinates to draw attention to that space. The JSON payload contains the following fields:

`X` *(float)*
: The *x* coordinate of the marker.

`Y` *(float)*
: The *y* coordinate of the marker.

`OA` Updates attributes of a specified map object. Any attributes not listed in the command MUST remain as-is. The JSON payload includes the following fields:

`ObjID` The unique object identifier for the object to be modified. Alternatively, it may be in the form `@`*name* where *name* is the creature's name in any of the forms allowed for the `AC` command.

`NewAttrs` *(object)*
: The set of new attributes and their values, expressed as an object with field names matching the object attributes to be changed and their associated values being the new values for those attributes.

Example: `OA {"ObjID":"a984a3","NewAttrs":{"X":12,"Y":44.5}}`

Note that there was an implicit assumption in the past that the `Name` attribute of a creature would not change and it could be used as an immutable identifier within a map client for a creature token. However, this is not the case and in fact GMA now includes explicit features to change this very attribute. Clients must be prepared to deal with the consequences of a change to any attribute, including `Name`. The `mapper` client itself does this correctly starting with version 3.39.

`OA+` Adds one or more values to a single attribute of an object known to the mapper. The payload is a JSON object with the following fields:

`ObjID` The unique object identifier, specified as for the `OA` command.

`AttrName`
: The name of the attribute to be modified. This MUST be an attribute whose value is a list of strings. (E.g., the `StatusList` attribute of creature objects.)

`Values` *(list of strings)*
: A list of string values to be added to those already in the named attribute.

`OA-` Removes one or more values from a single attribute of an object known to the mapper. The payload is a JSON object with the following fields:

`ObjID` The unique object identifier, specified as for the `OA` command.

`AttrName`
: The name of the attribute to be modified. This MUST be an attribute whose value is a list of strings. (E.g., the `StatusList` attribute of creature objects.)

`Values` *(list of strings)*
: A list of string values to be removed from those already in the named attribute. It is not an error if the value wasn't there to begin with.

`OK` The server sends this to the client when it is ready for the client to start sending commands to it. The accompanying JSON payload includes the following fields:

`Protocol` *(int)*
: The protocol version used by the server. Map clients which do not support that protocol SHOULD warn their users and SHOULD disconnect since they can't guarantee they can actually communicate with the server.

`Challenge` *(base64-encoded bytes)*
: If present, the `Challenge` value is a base-64-encoded authentication challenge. A client must successfully respond with a valid `AUTH` command before any of its commands to the server will be accepted. The server will also refuse to send the client any map updates until it has successfully authenticated.

`Iterations` *(int)*
: The number of rounds of hashing to apply to generate the challenge response. This value SHOULD be in the range [64,4095].

`ServerStarted` *(RFC 3339-formatted datetime string)*
: The absolute wall-clock time that the server was started.

`ServerActive` *(RFC 3339-formatted datetime string)*
: The abolute wall-clock time of the server's last `MARCO` ping signal. If this was long ago (more than a minute or two with out-of-the-box server configuration), it is an indicator that the server's main event loop is deadlocked.

`ServerTime` *(RFC 3339-formatted datetime string)*
: The current wall-clock time on the server. This provides an accurate point of reference to determine how long ago the other server time values occurred.

`ServerVersion`
: The server's version number. (Added in protocol 405.)

`PRIV` Notice from the server that a command sent by the client is denied due to insufficient privilege. Clients MUST NOT send this command. The JSON payload includes the following field:

`Command`
: The command the client was attempting.

`Reason`
: The reason that command was denied.

`POLO` No-op. Clients MUST ignore if received. Clients SHOULD send this in response to a `MARCO` command from the server.

`PROGRESS`
: The server sends this to the client to indicate progress of a long-running operation such as downloading files. (Formerly this was sent as a comment prior to protocol version 400.) The payload is a JSON object with the following fields:

`OperationID`
: A unique identifier for the operation we're reporting progress for.

`Targets` *(list)*
: If this progress bar is associated with a specific character or characters, they are listed by name here.

`IsTimer` *(bool)*
: If true, this is showing the status of a running timer instead of an operation in progress.

`Title` The description of the operation in progress. This is suitable for display to the user.

`Value` *(int)*
: The current value of the progress meter. Units are arbitrary. The value may fluctuate up or down over the course of the updates that are received.

It may also go negative. It is up to the client to display negative values appropriately based on the type of progress meter or other context. For example, timers generated by GMA will run negative for a number of updates to show that the timer has expired before sending a final update with `IsDone` true to indicate that the timer should be removed from view.

`MaxValue` *(int)*
: The maximum value expected for the progress meter when it is finished. If this is 0, we don't yet know what the maximum will be, which SHOULD cause the client to use a progress meter style that indicates that it is not possible to estimate remaining time to completion. Also note that this value MAY change over the course of the progress report if the server becomes aware that it has a better maximum value available to it at that point.

`IsDone` *(bool)*
: A boolean indication that the operation is completed. The client SHOULD remove the progress meter from display when the operation is completed.

Clients SHOULD NOT send these; they are for the server to notify the client about progress. If a client does send progress update messages to other peers, it should be clear that what is being tracked is a process that the other clients are interested in.

As a special case, the server may send this message with `OperationID=*` and `IsDone=true`. This is a signal to clients that all timers should be cancelled and dismissed from view. If a subsequent `PROGRESS` message is sent with the `OperationID` of a timer that was dismissed in this way, clients should treat it like a new timer being created at that point.

`PS`
: Place a creature token on the map. This may be used to define a new creature object if no object with the given *ID* already exists, or replaces an existing token with the (possibly different) parameters given. The JSON payload includes the following fields:

  `ID`
  : The internal ID by which this creature is to be known. This must be unique. The client which creates the character token locally should create a unique ID, which is then broadcast via this command to the other clients, which use the same ID.

  `Name`
  : The name as displayed on the map. Must be unique among all creatures currently displayed.

  `Health` *(object)*
  : If not `null`, this gives the current health status of the creature. It is an object with the following fields:

    `MaxHP` *(int)*
    : The creature's maximum number of hit points.

    `LethalDamage` *(int)*
    : The amount of lethal damage sustained.

    `NonLethalDamage` *(int)*
    : The amount of non-lethal damage sustained.

    `Con` *(int)*
    : The grace amount of hit points the creature may sustain over their maximum before they are considered dead.

    `IsFlatFooted` *(bool)*
    : If true, the creature is flat-footed.

`IsStable` *(bool)*
: If true, the creature has been stabilized to prevent death while critically wounded. Creatures which are in non-critical states of health don't have this attribute set even though technically (in a sense) they are "stable".

`Condition`
: A custom condition status to display on the map, if you wish to override the map's calculation.

`HPBlur` *(int)*
: If 0, the creature's health is displayed accurately. Otherwise, this gives the percentage by which to "blur" the hit points as seen by the players. For example, if `HPBlur` is 10, then hit points are displayed only in 10% increments.

`Hidden` *(bool)*
: If true, this creature token is hidden from player view, although the GM can still see it. Added in protocol 404.

`Gx` *(float)*
: The grid *x* coordinate for the creature's reference point. Note that this is in grids, not pixel units.

`Gy` *(float)*
: The grid *y* coordinate for the creature's reference point. Note that this is in grids, not pixel units.

`PolyGM` *(bool)*
: If true, only the GM may access the polymorph capabilities of this creature. Map clients not logged in as GM MUST not display any information indicating that there are different skins available, nor allow the user to select among them.

`Skin` *(int)*
: If 0, show the default appearance on the creature's token. Otherwise, show one of the alternative images defined for that creature.

`SkinSize` *(list of strings)*
: This gives a list of values to use for the `Size` attribute corresponding to the `Skin` number. For example, if there are 3 skins defined for a shape-changing creature, and the first two are medium-size but the third is large, then `SkinSize` would have a value like `["M:base","M:disguise","L:giant"]`.

`Elev` *(int)*
: The creature's elevation in feet relative to the "floor" level.

`Color` The color code used to draw the creature's threat zone.

`Note` A note to attach to the creature token to indicate special conditions or status.

`Size` The tactical size category of the creature. This has the general form

category[*natural*][`->`*extended*][`=`*space*][`:`[`*`]*comment*]

where *category* is a single-letter category code, *natural* is a custom natural reach distance in feet (if different than the standard for that size category), *extended* is a custom extended reach distance in feet (if different than the standard for that size category), and *space* is a custom creature space diameter in feet (if different than the standard for that size category). (Changed in protocol 406.)

This is now DEPRECATED. If present and the `SkinSize` field is also present, then the value of `Size` MUST equal the value of the first element of `SkinSize`. The preferred usage going forward is to not set the `Size` field at all, and just use `SkinSize` alone. If there is only one skin for the creature, this value will have only a single value.

`DispSize`
: If non-empty, this indicates that the creature is temporarily resized to the given size category and should be displayed as such. The value is otherwise identical to `Size`. (Added in 406.)

`StatusList` *(list of strings)*
: A list of condition codes which apply to the character. These are defined by the `DSM` command (q.v.).

`AoE` *(object)*
: If not `null`, a spell area of effect should be drawn around the creature. The value is an object with these fields:

  `Radius` *(float)*
  : The distance in standard map pixel units away from the creature's center to the perimeter of the area of effect.

  `Color` The color with which to draw the spell area.

`MoveMode` *(int)*
: The mode of locomotion currently employed by the creature:

  0 Land (walking)

  1 Burrowing

  2 Climbing

  3 Flying

  4 Swimming

`Reach` *(int)*
: Indicates if the creature is wielding a reach weapon and thus has an expanded threat zone. If 0, the threat zone is normal for the creature. If 1, it cannot attack adjacent foes but has a wider threat space. If 2, it has both threat zones active at the same time.

`Killed` *(bool)*
: If true, the creature is dead.

`Dim` *(bool)*
: If true, the creature does not have initiative now, and their token should be de-emphasized compared to the one with initiative.

`CreatureType` *(int)*
: The specific type of creature. This may be 0 if the type is unknown, 1 for monsters, or 2 for players. Clients MUST NOT set this field to a value other than 1 or 2.

`CustomReach` *(object)*
: Specify a custom reach distance for this creature, if different than the standard for a creature of its size category. This contains the following fields:

  `Enabled` *(bool)*
  : If true, there is a custom reach distance as described in the other fields. if false, the other fields should be ignored.

`Natural` *(int)*
: The number of feet from the perimeter of the creature token to which the creature's natural reach extends.

`Extended` *(int)*
: As `Natural` but indicates the distance for reach weapons.

(Added in protocol 406.)

`READY` The server sends this to the client to indicate that all preliminary data has been sent, authentication (if applicable) has been successful, and the client may proceed to carry out normal operations now.

`REDIRECT`
: Instructs the client to use a different server for this session. This is used when the GM wants to use an alternate server temporarily without requiring the players to reconfigure all their clients. The server SHOULD send this during the initial stage of client interaction to avoid authenticating to a server you won't be using and then authenticating to the other one, but MAY be sent at any point. The client SHOULD disconnect immediately upon receiving this command, so the rest of the server negotiation may not even happen at all. The JSON payload includes the following values:

  `Host` The host name or IP address of the server to connect to for this session.

  `Port` *(int)*
  : The TCP port number on which to connect to the server.

  `Reason`
  : The reason for the redirection to a new server.

  This command SHOULD appear before any commands which would alter the client's internal state such as `DSM` or `AC`, because we cannot guarantee that the client will undo those settings before connecting to the new server which will send its own set of those commands. Ideally, when redirecting clients to a different server, `REDIRECT` should be the first item in the server's init file.

`ROLL` Reports the results of a die roll initiated by the `D` command. Clients MUST NOT send this command. The JSON payload includes the following fields:

  `Sender`
  : The name of the user who rolled the dice.

  `Recipients` *(list of strings)*
  : The names of the people the message was explicitly addressed to. For global messages, this should be `null`.

  `MessageID` *(int)*
  : The unique number assigned by the server for this chat message. (Die roll results are essentially a kind of chat message.)

  `MoreResults` *(bool)*
  : If true, there will be more `ROLL` messages yet to come which are part of the result set for the same die roll request.

  `RequestID`
  : The `RequestID` string passed by the client when requesting this die roll, or the empty string if they didn't.

  `Sent` *(RFC 3339-formatted datetime string)*
  : The date and time the message was sent. Clients SHOULD NOT set a nonempty value for this field. It will be filled in by the server when sending messages to clients. If this field is missing or empty when received

from the server, it means the date was not available to report for that message. Note that the special value "0001-01-01T00:00:00Z" is the "zero" value for dates and indicates that the date is not known (as is the case if this field is omitted entirely or is the empty string).

ToAll *(bool)*
: If true, this is a global message sent to all clients. The contents of the Recipients field are ignored; it should be omitted or set to null.

ToGM *(bool)*
: If true, this message will be sent only to the GM. The contents of the Recipients field are ignored; it should be omitted or set to null.

Title
: The title describing the purpose of the die-roll as set by the user, if any. The title string may include special formatting codes which will be used by clients supporting the DICE-COLOR-BOXES optional feature (see the ALLOW command for details). Clients which did not request this feature will not see the special formatting codes in the title string.

Result *(object)*
: The result of the die roll. This is an object with the following fields:

  Result *(int)*
  : The final numerical result of the die-roll.

  InvalidRequest *(bool)*
  : If true, the die roll never happened because the request could not be understood. In this case, the details about the request and reason for failure are in the Details array. The Result integer field should be disregarded since there was no result generated.

  ResultSuppressed *(bool)*
  : If true, this result message describes a die roll request without disclosing the results of the roll. This is used, for example, when sending a die roll for the GM to see privately (not even showing the requester the result). In this case, the value of the Result field should be ignored.

  Details *(list of objects)*
  : The details behind how that result was generated are given as a list of objects with the following fields:

    Type
    : A text label describing what the value means in the context of the die-roll result.

    Value
    : The value for this part of the result (as a string).

SYNC
: This command MUST be ignored if received by a client. A client sends this to the server to request a replay of map commands which would be necessary to catch it up to the state the other maps are currently in. A server may be configured to perform a sync operation upon client connection (after, if required, authentication) without the client explicitly sending a SYNC command to it.

SYNC-CHAT
: This command MUST be ignored if received by a client. Supersedes the SYNC command with CHAT argument in protocol versions prior to 400. A client sends this to the server to request a replay of chat messages (including die-roll result notices) according to the JSON payload, which may include this field:

  Target *(int)*
  : If missing or 0, all chat messages in the server's history are sent to the client. Otherwise, if Target is positive, all messages with

messageIDs greater than Target. If Target is negative, its absolute value gives the number of recent messages to send to the client (e.g., if Target is −50, then the most recent 50 messages are re-sent to the client).

TB Controls the display of the toolbar in the mapper client. The JSON payload is an object with the following field:

Enabled *(bool)*
: If true, the client SHOULD display its toolbar. If false, the client SHOULD NOT display it. The client may override this request based on the user's preferences.

TO Send chat message to a set of recipients. The payload is a JSON object with the following fields:

Sender
: The name of the user sending the message. Clients SHOULD NOT set this field; it is set by the server when sending these messages to peers.

Recipients *(list of strings)*
: The names of the people the message was explicitly addressed to. For global messages, this should be null.

MessageID *(int)*
: The unique number assigned by the server for this chat message. Clients SHOULD NOT set this field.

Sent *(RFC 3339-formatted datetime string)*
: The date and time the message was sent. Clients SHOULD NOT set a nonempty value for this field. It will be filled in by the server when sending messages to clients. If this field is missing or empty when received from the server, it means the date was not available to report for that message. Note that the special value "0001-01-01T00:00:00Z" is the "zero" value for dates and indicates that the date is not known (as is the case if this field is omitted entirely or is the empty string).

ToAll *(bool)*
: If true, this is a global message sent to all clients. The contents of the Recipients field are ignored; it should be omitted or set to null.

ToGM *(bool)*
: If true, this message will be sent only to the GM. The contents of the Recipients field are ignored; it should be omitted or set to null.

Text The text of the message to be sent.

/CONN This command causes the server to send a summary of the connections it is currently serving. These are sent back to the client via the CONN command documented above.

UPDATES
: This provides the latest available versions of various programs. A client MAY alert the user or initiate an automatic upgrade if it recognizes that it or one of its dependencies is out of date. Clients MUST NOT send this command. (Formerly this was sent as a comment prior to protocol version 400.) The payload contains the following field:

Packages *(list of objects)*
: Each element in this list is a JSON object with the following fields:

Name The name of the software package, such as "mapper", "go-gma", or "core".

`MinimumVersion`
: If a server wishes to limit clients from this package to only those with a minimum version number (as self-reported by the client in its `AUTH` message), then a `MinimumVersion` and `VersionPattern` field MUST be added to that package's information here. The `MinimumVersion` field is a string with the minimum client version allowed to be used, as a semantic version string such as "`1.2`", "`1.7.3`", "`1.6-alpha.1`", etc. This will be matched against the value captured from the client's version number via the `VersionPattern` field. *This field should be used only in server init files. Clients SHOULD NOT use this field, nor expect it to be sent by the server.*

`VersionPattern`
: This gives a regular expression which is matched against the `Client` field sent by the client as part of its `AUTH` message when signing on to the server. This expression MUST contain a single capturing group which yields the client's version number to be compared against the `MinimumVersion` field described above. *This field should be used only in server init files. Clients SHOULD NOT use this field, nor expect it to be sent by the server.*

`Instances` *(list of objects)*
: A list of all available instances of the named package. Each is a JSON object with the following fields:

  `OS` The name of the operating system platform for which this instance of the package is available (e.g., "freebsd", "linux", "darwin", "windows", etc.). If this is empty, then this instance is platform-independent.

  `Arch` The name of the hardware architecture for which this instance of the package is compiled (e.g., "amd64", etc.) If this is empty, then this instance is architecture-independent.

  `Version`
  : The semantic version number that users should be using. This is set by the GM (or designated server administrator—henceforth we will simply use the term "GM" for this role) for their game. It may not necessarily be the latest version available generally. The version string MAY omit the third version number (e.g., "`4.2`" instead of the more standards-compliant "`4.2.0`").

  `Token` The download token used for obtaining a copy of the package. This is entirely under the GM's control and refers to a file the GM placed on their game server for their players to access. The *token* value is the non-static part of the download URL from their game server.

`WORLD` This command provides various campaign-world information useful for the client to know. Clients MUST NOT send this command. The payload includes the following parameters:

`Calendar`
: The name of the calendar system in play, as known to the GMA game clock. (Formerly this was sent as a comment in the form `// CALENDAR`

// *system* prior to protocol version 400.)

Declares which calendaring system the various dates and times are based upon.

`ClientSettings` *(object)*
Any of the following settings will override the client's own settings. Clients SHOULD notify users that this has happened. This allows the GM to make global changes to things like where images are stored without requiring players to adjust all their settings.

`MkdirPath`
The path to the `mkdir` program on the server (used for GM uploads of mapper content to the server).

`ImageBaseURL`
The base URL from which the client will retrieve map and image files.

`ModuleCode`
The current module's ID code.

`SCPDestination`
The directory where GM uploads of mapper content should be sent to.

`ServerHostname`
The hostname (and optionally username in the form *name*`@`*host*) for the GM to upload mapper content to the server.

Example: `WORLD {"Calendar":"golarion"}`

SEE ALSO

`mapper`(5), `gma`(6), and `server`(6).

AUTHORS

Steve Willoughby / steve@madscience.zone; John Mechalas (elevation and movement modes).

HISTORY

This document defines map protocol version 415, which added the `Iterations` parameter to the `OK` message.

Version 414 changed the definition of the `DENIED` message to allow it to be sent at any time during the conversation.

Protocol 413 added speculative definitions for the `CORE`, `CORE/`, and `CORE=` commands, and extensions to the `PROGRESS` command.

Protocol 412 added support for sharing die-roll presets.

Protocol 411 added timestamps to chat messages and die-roll results.

Protocol 410 added the `REDIRECT` command and the `ClientSettings` field of the `WORLD` command.

Protocol version 409 added the `Stipple` pattern fill attribute.

Protocol version 408 introduced the `PolyGM` creature attribute.

Protocol version 407 added the image animation features.

Protocol version 406 dropped the `Area` attribute from creatures and added the `CustomReach` attribute and the expanded syntax for the `Size` attribute, as well as the new `DispSize` attribute.

Protocol version 405 introduced the `ServerVersion` field to the server's greeting.

Protocol 404 introduced the `Hidden` attribute for creature tokens, the transparent image type, and the

`Transparent` attribute for status markers.

Protocol version 403 introduced the `AI/` server message and additional fields to the `OK` message.

Protocol version 402 improved the `ROLL` server message by adding support for blind rolls to the GM.

Protocol version 401 introduced the `Grid` parameter to the `AV` command.

Protocol version 400 changed the server communication protocol to use JSON instead of TCL lists. It also introduced the `PROGRESS`, `PROTOCOL`, `READY`, `UPDATES`, and `WORLD` commands.

As of the beta-3 release, the `RequestID` and `MoreResults` fields were added to die roll requests and results messages.

Version 332 added the `GRANTED` server response, guaranteeing that `AUTH` will be followed by a response in any case.

Version 331 changed the way persistent chat messages are managed, altering the `ROLL`, `TO`, `SYNC`, and `CC` commands.

Version 329 added the `CC` command and the extended form of `SYNC`.

Previously, version 328 introduced the `ACCEPT` command.

Version 327 added the `DENIED` and `PRIV` response commands.

Version 326 added support for chat channels and die rolling, and changed the response from the `/CONN` command to be a structured sequence of data records rather than sending the reply in comments.

Protocol 325 added connection debugging support by changing the `AUTH` command and adding `/CONN`.

The version 324 protocol added the `DSM`, `OA+`, and `OA-` protocol commands. This version of the protocol also assumes mapper file format 15 is being used.

Previously, protocol version 323 added support for server-side persistent state and the introduction of the `SYNC` protocol command.

Protocol version 322 added comments to version 321.

Protocol version 321 added authentication support to the capabilities of protocol 320.

Map protocol version 320 has the same commands as 319, but the `HEALTH` attribute of creatures changed from map version 12 to 13, so we are advancing the protocol version at the same time since maps released for protocol 319 were expecting the older map format.

Map protocol version 319 differs from version 318 (the first explicitly numbered version) by the addition of the `CLR@`, `M?`, and `M@` commands.

COPYRIGHT