NAME

server – GMA battle grid map server (Go version)

SYNOPSIS

[`gma go`] `server` [`-cpuprofile` *path*] [`-debug` *flags*] [`-endpoint` [*hostname*]`:` *port*] [`-help`] [`-init-file` *path*] [`-log-file` *path*] [`-password-file` *path*] `-sqlite` *path* [`-telemetry-log` *path*] [`-telemetry-name` *string*]

DESCRIPTION

The individual `mapper`(6) clients used by players in a game may keep in contact with one another so that they all display the same contents. A change made on one client (moving a creature token or adding a room, say) appears on all the others. This is accomplished by starting a `server` process and having all of the `mapper` clients connect to it via their `--host` and `--port` options.

Once connected, the server will send an initial greeting that may define a list of player character tokens to appear on the `mapper` context menus, or any other useful information the clients need to have at startup time. It may, at the GM's option, even initialize the client to show the full current game state.

From that point forward, the server relays traffic between the clients, so they communicate with each other via the service. The server also tracks the commands it sees, so that it maintains a notion of the current state of the game. Clients may re-sync with the server in case they restart or otherwise miss any updates so they match the server's state. The server may respond directly to some client queries if it knows the answer rather than referring the query to the other clients.

To guard against nuisance or malicious port scans and other superfluous connections, the server will automatically drop any clients which don't authenticate within a short time. (In actual production use, we have observed some automated agents which connected and then sat idle for hours, if we didn't terminate their connections. This prevents that.)

OPTIONS

The command-line options described below may be introduced with either one or two hyphens (e.g., `-endpoint` or `--endpoint`). Options which take parameter values may have the value separated from the option name by a space or an equals sign (e.g., `-endpoint=:2323` or `-endpoint :2323`).

`-cpuprofile` *path*
: Enable CPU profiling via Go's pprof tool. Sample data will be saved to the named *path*.

`-debug` *flags*
: Add debugging information to the server log. The *flags* value is a comma-separated list of debugging options. The following options are available:

`all`	All possible debugging information.
`none`	No debugging information (this cancels any previously-specified debug flags, but more may be added after this).
`auth`	Authentication operations.
`db`	Database operations.
`events`	Background events.
`i/o`	Input/output operations.
`init`	Client initialization.
`messages`	Message traffic between the server and clients.
`misc`	Miscellaneous debugging.
`qos`	Quality of service stats (even if no action was taken).
`state`	Changes to the game state.

`-endpoint` [*hostname*]**:** *port*
: Accept incoming client connections on the specified *hostname* and TCP *port* number. If *hostname* is omitted, connections are accepted on all the machine's network interfaces. If the entire option is omitted, it defaults to port 2323.

`-h`, `-help`
: Print a usage summary and exit.

`-init-file` *path*
: The contents of this file are used to initialize the clients every time they connect. See the **CLIENT INITIALIZATION** section below for details.

`-log-file` *path*
: Append a record of server actions to the specified file. If debugging is enabled, those messages will go to the log file as well. By default, the log is printed to the standard output, which may also be explicitly specified by a single hyphen ("−") as *path*.

`-password-file` *path*
: This enables client authentication. By default, the server will allow any client to connect and immediately interact with it. However, if this option is given, the server will require a valid user credential before allowing the client to operate. The contents of the password file are stored in plaintext, one password per line.

  The first line is the general player password. Any client connecting with this credential will be admitted with any username they request other than "`GM`".

  The next line, if present, gives the privileged GM password. Any client connecting with this credential will be granted with game master privileges under the username "`GM`".

  Any subsequent lines have the format "*username***:** *password*". This assigns a specific *password* for the given *username*, such that any client wishing to sign on with that specific username must present this specific credential.

  N.B. This is an extremely trivial challenge-response authentication mechanism used solely to protect a game server from revealing in-game spoilers and rejecting nuisance connections. It should **not** be relied upon to secure any sensitive information. No passwords used here should be the same as passwords used for anything else of consequence.

`-sqlite` *path*
: Specifies the filename of a sqlite database the server will use to maintain persistent state. This includes such things as stored die-roll presets, known image locations, and the chat history. If *path* does not exist, a new empty database will automatically be created by the server.

`-telemetry-log` *path*
: If the server is configured to send telemetry metrics, this provides the name of a file into which to write telemetry debugging information. The default is not to write debugging information at all. If *path* is a single hyphen ("−"), then the log will go to the standard output.

`-telemetry-name` *string*
: If the server is configured to send telemetry metrics, this provides the name of the application for purposes of identifying this running instance of the server. Defaults to "`gma-server`".

CLIENT INITIALIZATION

When a client connects, the server begins by sending a number of messages up front, before and/or after successfully authenticating. The initial negotiation with the client goes through the following stages:

Connection The server declares the protocol version it is using.

Preamble The server sends a number of messages to the client, which may include comments, declaring party members, notice of software updates, and campaign information. We recommend limiting the preamble to comments.

Authentication If configured to do so, the server will demand a valid credential from the client before proceeding any further.

Post-auth After successful authentication (or unconditionally if no authentication is required), the server sends a number of messages just as described for the preamble stage.

Ready The server then signals to the client that the negotiation is completed and the client is then free to issue any commands to the server, and may receive any messages from the server. (Before this point, the server won't even consider receiving commands from the client that aren't part of this negotiation, and won't be sending normal traffic to the client yet.)

Sync Finally, the server may send additional data to the client (typically this is synchronization data to catch the client up to the server's current notion of the game state).

By default, the preamble, post-auth, and sync stages are effectively nil. However, the presence of a client initialization file via the `-init-file` option specifies what to send to the client during negotiation.

Each line of the file is a server message to be sent to the client, formatted as documented in the server protocol specification. (I.e., a command word followed by a space and a JSON parameter object.) Long commands may be continued over multiple lines of the file, as long as the brace ("`{`") that begins the JSON data appears on the line with the command name, and all subsequent lines are indented by any amount of whitespace. The final brace ("`}`") that ends the JSON data may appear at the end of the last line or on a line by itself (in which case it need not be indented itself).

The commands which may appear in the initialization file include the following:

`//` This line is transmitted AS-IS to the client. This command does not require JSON data to follow it. All text from the "`//`" to the end of the line are considered a comment and are sent verbatim. This is useful to provide a human-readable message to anyone connecting to the game port.

Clients may interpret what they see in comment messages from the server but are not under any obligation to do so. Currently, the following special comment is recognized by the `mapper`(6) client (at least):

`// notice:` *message*
: If the comment begins with the string "`notice:`" (not counting whitespace), then the *message* following it will be shown to the user. In this way, the GM or other server administrator may communicate urgent notices to all the users of their game server. This notice comment must appear before the `READY` command in the server's init file.

`AC` Add a character to the client's quick-access context menu. Typically this is the party of player characters. Any JSON parameters accepted by the server `AC` message may be given, but for the purposes of the client initialization, the important ones are `ID`, `Name`, `Color`, `Size`, `Area`, and `CreatureType`, providing a unique ID for the character, their name as it appears on the map, the color of their threat zone, creature size category, threatened area size category, and creature type (1 for monsters or 2 for players).

`DSM` Defines a condition status marker that may be placed on creature tokens. This will update an existing marker already known to the mapper, or add new ones to the set of condition markers. The parameters are

`Condition`
: The name of the condition. While this is arbitrary, it should be short, preferably a single word. It should not begin with an underscore to avoid conflicts with internal names used by the GMA software.

`Shape` Describes the shape of the marker drawn over the token. See the protocol documentation in `mapper-protocol`(7).

`Color` The color of the marker.

`Transparent`
: If present and true, this means to use a semi-transparent creature token when this condition is in effect.

`Description`
: A sentence or paragraph describing the effects of that condition.

`QOS` Sets quality of service limits in the server. If a client session violates any of these limits, its session will be terminated immediately. Make sure that whatever values you configure here are far enough out of bounds to justify ejecting the offending client. The value for this item is a JSON object where each entry is a QoS rule to enforce, and the corresponding value for the rule is a set of fields as described below. If any of these rules are not included in the `QOS` payload, that rule will not be enforced at all.

`QueryImage` Reject clients which excessively ask for the same images after being informed of where to find them by the server. There should be a little allowance for the client to take the time to obtain the image, so a small number of repeated requests is ok, but a properly functioning client should stop asking for the same image right away. This rule's value is a JSON object with these fields:

`Count` *(int)*
: The maximum number of `AI?` requests a client can send for the same image after it's already been answered by the server.

`Window` *(duration)*
: If this field is omitted or is blank, the client will be ejected if it ever exceeds `Count` requests for the same image after the server has answered it. Otherwise, this specifies a duration in a form such as "`15m`" or "`1h30m`" which indicates that in order to trigger the rule, `Count` repeated requests must arrive during this period of time. (The server will reset the counters every time this much time elapses.)

`MessageRate` Reject clients which send more than a certain number of requests during a given window of time. This rule's value is a JSON object with these fields:

`Count` *(int)*
: The maximum number of messages allowed for the client to send during a time `Window` before the rule is triggered.

`Window` *(duration)*
: Just as with the `QueryImage` rule, this specifies the time frame in which the threshold number of messages isn't allowed to be exceeded.

`Log` Enables a periodic logfile record of QoS metrics. If QoS debugging is enabled (`-debug qos`) or (`-debug all`), details about the data collected for each of the enabled rules is logged. Otherwise a single line is logged per client connection, in the form "`QoS img=`*icount*`, rate=`*rcount*`/`*rmax*", where *icount* is the number of different image requests currently being tracked, *rcount* is the number of packets received so far in the current time window, and *rmax* is the threshold number of packets allowed before the client is rejected.

`Window` *(duration)*
: The time interval after which to print each log message.

`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 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`
An explanation of why the redirect is being performed (optional).

The server will disconnect from the client immediately after issuing the `REDIRECT` command to it.

`UPDATES` Advertises to the client the version of each software package you recommend for them to use. The JSON data has a single parameter called `Packages` which is a list of objects with the following parameters:

`Name` The name of the 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.

`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.

See the `sample.init` file shipped with the go-gma source code for an example of this, or note that the regular expression to match the `mapper`(6) client is `"^\\s*mapper\\s+(\\S+)"`.

Note that backslashes in the regular expression need to be escaped with another backslash (i.e., `\\`) to satisfy the encoding requirements for JSON.

`Instances`
A list of available versions of the package. If multiple versions are listed here, they should each be for a different platform. Each instance value is an object with the following fields:

`OS` The target operating system for this version of the package. If omitted or blank, it is OS-independent. Values are `freebsd`, `linux`, `darwin`, `windows`, etc.

`Arch` The target hardware architecture for this version. Values are `amd64`, etc.

`Version`
The recommended version you want players to use.

`Token` If you provide a downloadable copy of the software on your server for players to get, specify the download token here. The mapper tool currently has the capability to self-upgrade based on this token. The mapper is configured with the option `--update-url=`*base* which is combined with the *token* value to get the filename to be downloaded from your server. The URLs retrieved will be *base*`/`*token*`.tar.gz` and

base/*token*`.tar.gz.sig`.

`WORLD` Sends campaign information. The fields of the JSON payload include

`Calendar`
: Names the calendar in play.

`ClientSettings`
: Overrides some of the server- and game-specific client preference settings. The value is a JSON object with the following fields:

  `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.

`AUTH` This command word (without JSON data) in the initialization file causes the server to perform the authentication step before continuing. Thus, it marks the end of the preamble stage. Following lines will be sent as part of the post-auth stage.

`READY` This command word (without JSON data) in the initialization file causes the server to signal to the client that the negotiation is complete and normal client/server interaction may begin. Thus it marks the end of the post-auth stage. Anything after this point is for the sync stage.

In this final part of the file (after the `READY` command), any of the following server messages may be included to be sent to the client: `//`, `AC`, `AI`, `AI?`, `AV`, `CC`, `CLR`, `CLR@`, `CO`, `CS`, `DD=`, `DSM`, `I`, `IL`, `L`, `LS-ARC`, `LS-CIRC`, `LS-LINE`, `LS-POLY`, `LS-RECT`, `LS-SAOE`, `LS-TEXT`, `LS-TILE`, `MARK`, `OA`, `OA+`, `OA-`, `PROGRESS`, `PS`, `REDIRECT`, `ROLL`, `TB`, `TO`, `UPDATES`, or `WORLD`. (Technically, any of these commands can appear anywhere in the initialization file, but we strongly recommend limiting commands to `//`, `AC`, `DSM`, `REDIRECT`, `UPDATES`, and `WORLD` in all stages except the final (sync) stage.)

`SYNC` This command word (without JSON data) in the initialization file will cause the server to behave as if the client sent a `SYNC` command to it after the negotiation is complete. This sends the full game state to the client, so that a newly connected mapper will display the current map contents the other players see.

SECURITY

The authentication system employed here is simplistic and not ideal for general use, but is considered to be good enough for our purposes here, since the stakes are so low. It is intended just to discourage cheating at the game by looking at spoilers or direct messages intended for other users, not for any more rigorous protection.

The challenge/response system employed by the server is designed to resist replay attacks since it does not divulge the actual password in the clear over the network, although other attacks such as man-in-the-middle remain possible. This authentication mechanism is used because at this point the server and clients do not support encrypted communications. (If this becomes supported in the future, a more robust authentication mechanism will be possible which does not have the weaknesses documented here.)

The main weakness of the system is that passwords are stored in plaintext on the server and on each client, which means it is critical to secure the password file and the system itself. Caution your players to use a

password for the mapper that is different from any other passwords they use (which should be the password practice people observe anyway). A breach that reveals passwords from the server's file, or the client configuration files where passwords are stored, would then only allow an imposter to log in to your map service, which admittedly is more of an inconvenience than a serious security issue, assuming you use your map server just for playing a game and not for the communication of any sensitive information.

Don't use the GMA mapper server for the communication of sensitive information. It's part of a game. Just play a game with it.

SIGNALS

The map service responds to the following signals while running. These actions may not be taken immediately but should happen within a few seconds.

`HUP` This signal terminates all existing client connections but leaves the server up and ready to accept new incoming connections.

`INT` Gracefully shuts down the server.

`USR1` Causes the server to re-read its initialization file. Clients which connect after this will see the new initialization information. This also jumps the next message ID for chat messages and die roll results to most likely be a larger ID than other servers (it sets the next ID to the current UNIX timestamp value, just as the server does when it starts; this will make it ahead of other servers on the assumption that server clocks are correct and no server will sustain a message rate of >= one message per second since it was started).

`USR2` This signal causes the server to dump a human-readable description of the current game state database to the log file.

SEE ALSO

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

The server communications protocol is definitively documented in the `mapper`(6) manpage which comes with the GMA-Mapper package.

AUTHOR

Steve Willoughby / steve@madscience.zone.

BUGS

If the server is not configured to require authentication, that means it won't drop nuisance connections either, since it's accepting all connections as valid, even if it never sends any valid data to the server.

COPYRIGHT