NAME

Gigapxy − an inter-protocol data stream relay and proxy.

DESCRIPTION

Gigapxy pipes data *channels* to corresponding *clients*; either of the two endpoints may be a network socket or a file.

Basic terminology and use cases

Gigapxy uses the term *channel* for a data source and *client* for a destination. This terminology has roots in IPTV (IP television) operations: an IPTV provider gives its service subscribers (clients) access to TV/video channels via an IP-based network.

A common scenario using **Gigapxy** would be feeding (UDP) data from *M* multicast channels to *N* (TCP) clients/subscribers: media players, tools such as wget, curl, etc. A client, in fact, could be any application issuing an appropriate HTTP request.

Gigapxy is designed to serve many clients per channel, efficiently and economically. A built-in *caching mechanism* allows new clients to start reading cached (channel) data at once, minimizing the delay associated with making a new connection or a multicast group subscription. For the end user that means that changing IPTV channels becomes very fast.

Application modules

Gigapxy is a server application; its services include relaying data to to clients and performing administrative tasks, such as reporting application statistics in various formats.

The two core modules of Gigapxy are:

gws (Gigapxy Web Service)
: processes, validates and dispatches user requests, handles administrative tasks;

gng (Gigapxy Engine)
: serves data to clients.

The two modules run as separate processes, a single instance of **gws**(1) controls a number (N >= 1) of **gng**(1) processes.

gws(1) processes and validates a user request for data, sets up input and output ends of associated data streams, then dispatches the request to the appropriate **gng**(1) instance to handle data transfer. If **gws** receives an administrative request, it may service it locally or relay to an appropriate **gng**.

A **gng**(1) , on its end, is fully dedicated to channel−to−client data transfer; it is not to be affected by delays associated with HTTP request processing or even a crash of the controlling **gws**(1) instance.

gng reports to **gws** the events for the clients and channels that it is handling; the reports let **gws** track the data streams and load−balance requests between multiple **gng** instances.

gng also (if configured) regularly updates **traffic performance statistics (TPS)** that **gws** needs to produce traffic reports.

SETTING UP

Gigapxy is built as a single executable binary (named *gigapxy*), with two soft links to it set up by the installation process, the links denote the modules: **gws** and **gng**.

To see an overview of command-line parameters accepted by a module, run it with one of the following

command−line parameters: **−h, −?, −−help** or **−−options**

As one might expect, a command-line parameter always overrides the corresponding setting in the configuration.

Please note that running a module without any option is **NOT** equivalent to requesting help summary; it will just run the module in default configuration.

Most of Gigapxy's parameters should be specified in module−designated config files. The following are the default locations for configuration files: Gigapxy will look for either *gws.conf* or *gng.conf* (depending on the module being launched) and then (if neither could be opened) for *gigapxy.conf* in each of those locations unless a full path is specified at command line.

(current directory)

/etc

/usr/local/etc

If configuration file path is specified at command line, the module will only try to open that a file at that particular path.

The installation provides '/etc/gigapxy.conf' as the default configuration file containing sections for both **gws** and **gng**. However, each module's section could be put into a separate file and passed to the module via the '-C|--config' command-line parameter.

The documentation includes a fully annotated configuration file, with every possible option specified and commented on, at:

/usr/share/doc/gigapxy/examples/gigapxy-commented.conf on *Linux*, or at the corresponding **/usr/local** location on *FreeBSD*.

PREPARING TO RUN

Gigapxy can run in a terminal or as a daemon. To run as a daemon, it must be started with root privileges. Root privileges are not required after a short period of initialization; therefore it is suggested that the module run in a non−privileged mode, under a non−root user. The default configuration has application modules (started as root) switch to non-privileged *gigapxy* account, which is automatically created at installation point with the home directory of */var/run/gigapxy*. The user is**not** remo ved at de−installation for safety reasons.

Log−file directory */var/log/gigapxy* is automatically created at the installation point. No module will start without being able to write into a log file.

NB: It makes sense to have your logs reside in a **designated partition** that is not shared with your system's root directory. Setting up for **log−rotation** and **archival** are two related tasks that must not be overlooked.

It is suggested that all module instances write **their own log**, although writing into a shared log is also possible.

System log is automatically updated when a module runs as a daemon.

RUNNING

The very base topology of Gigapxy is running one controlling **gws** hooked up with a single **gng**. This would utilize only two CPUs/cores, so you might want to add more **gng** instances to spread channels across available cores. Mind that all clients of a single channel go to the one designated (by their controlling **gws**) **gng**. If you want to balance clients of the same channel across multiple **gng**'s, you would have to introduce more **gws** instances, each of them handling its portion of the channel's load.

The only rule to follow starting up modules is that a **gws** process should always start before any **gng** instances it controls. (You could test-run your topology in separate terminal windows to see how it works.) An example control script has been provided at:

/usr/share/gigapxy/scripts/gigapxy.sh on *Linux* or at the corresponding **/usr/local** location on *FreeBSD*

Before the configuration is finalized and the process is not (yet) fully automated, the two command-line options: **−T** and **−v** may be quite helpful. (See **command-line options**.)

The **−v** option works cummulatively: it allows for up to **−vvvv** to specify the deepest (debug) level of verbosity in the log output. If you are testing a particular feature or trying to reproduce a bug, this is the way to run for the log to be most helpful to the support team.

NB: Debug logs grow VERY large very fast so please make sure you have enough space in your **dedicated log partition** and **log rotation** set up. **gng** is especially verbose in its debug output so take extra caution there: provide both the space and the log storage fast enbough to handle a lot of writing without serious performance degradation.

For the **−T** option, bear in mind, that invoking it will disable switching to an alternate (non−privileged) user from root.

Running **Gigapxy** is trivial once the configuration has been properly set up: launch the modules individually or via a control script.

Gigapxy can be considered running and fully functional when a **gws**(1) is running with at least one **gng**(1) controlled by it. A **gws**(1) may run on its own without a single **gng**(1) attached but it will not be fully functional: it will **NOT** be accepting user requests for data until a **gng** connects. You could still check on it by requesting a status report via admin port.

A **gws**(1) shutting down gracefully (after one of the *quit*$_{P}$ *signals:* **TERM, QUIT or INT** will also shut down all its engines. This behavior could be overridden in the configuration to safeguard against abnormal situations or bugs causing a graceful **exit**(3) instead of a crash. Please refer to the **gws.conf**(5) for details.

If a **gws**(1) crashes, the subservient engines will not shut down at once but after **N** attempts to re−connect with a **gws**(1) at the same (socket) path. The associated parameters are also configurable.

Requesting data (user requests)

gws(1) has listeners on two ports for user and admin HTTP requests. The user−request formats are:

URIs for multicast sources support SMM (**source−specific multicast**) via {source−addr}:{mcast−addr}:{mcast−port} specifier.

a) *http://{addr}:{gws_port}/{cmd}/{mcast−addr}:{mcast−port}* OR *http://{addr}:{gws_port}/{cmd}/{src−addr}@{mcast−addr}:{mcast−port}*

WHERE
{addr}:{gws_port} ::= IPv4/6 address of the user-request listener;
{cmd} ::= udp;
{mcast−addr}:{mcast−port} ::= IPv4/6 address of the mulitcast group;
{src−addr} ::= source address for (SSM).

NB: IPv6 addresses are always specified as [{addr}]:port, as in [ff18::1]:5056.

This (udpxy−style) type of request specifies multicast group as the data source and the requesting HTTP connection as the destination.

b) *http://{addr}:{gws_port}/src/{channel−uri}/dst/{client−uri}*

WHERE
{addr}:{gws_port} ::= IPv4/6 address of the user−request listener;
{channel−uri} ::= URI for the channel (see format below);
{client−uri} ::= URI for the client (see format below);

URI format: {protocol}://{path}?{query}

c) *http://{addr}:{gws_port}/${alias}*

This type of request uses a *channel alias*: a dollar−sign prefixed name that resolves to a URL for a channel within a group. Refer to **channels.conf**(5) for details on configuring channels using aliased groups.

Supported protocols are: *FILE*, *TCP*, *UDP*, *HTTP*. Below are a few examples of requests using different protocols and formats:

a) *http://acme.com:8080/src/file:///opt/data/somefile.dat/dst/?a=bb&c=dd*

gws(1) is listening on port 8080 at acme.com

Channel is a file with the full path: */opt/data/somefile.dat*

The request has an associated query 'a=bb&c=dd' which could be used to specify additional parameters for the session.

Client (dst) is not specified, which defaults to the connection of the HTTP request.

The contents of */opt/data/somefile.dat* will be sent to the client; at EOF point the engine will wait (in a non-blocking manner) for the file to expand (be appended with more data) and, if the file gets expanded, will send the new data to the client. If the file does not expand within a certain (configurable) time period, the channel will time out and the clients' sessions will be terminated.

b) *http://acme.com:8080/src/udp://[ff18::1]:5056/dst/file:///opt/data/somefile.dat*

Channel is a multicast group with IPv6 address ff18::1, port 5056

Client is a file with the path: */opt/data/somefile.dat*

The engine will write any data arriving for the channel (multicast group) into the named file. The

channel may time out if no data arrive within a certain time period, in which case the session will be closed. If there's an error writing to the destination file, the session will also end.

c) *http://acme.com:8080/src/udp://[ff18::1]:5056/dst/*

d) *http://acme.com:8080/udp/[ff18:1]:5056*

The two requests above are equivalent (just stated in two different formats).

Both specify channel as the multicast group [ff18:1]:5056 and the (requesting) HTTP connection as the client. A timeout may occur on either of the network connections here, either of the two connections could also be broken by the peer, thus terminating the session.

e)
http://acme.com:8080/src/http://10.0.1.12:4056/udp/224.0.2.26:4033?kk=yy/dst/tcp://192.168.12.10:5051?mm=ff

specifies that channel data comes as a response to the HTTP GET /udp/224.0.2.26:4033?kk=yy request sent to http://10.0.1.12:4056. Whatever application handles HTTP requests at that address is expected to reply with a data stream destined to a TCP socket connected to the address: 192.168.12.10:5051. This session also has an associated query: ´mm=ff´, which could have a meaning in the context of the given session.

This request underlines Gigapxy's capability to cascade or 'daisy-chain' requests, and, therefore, link its instances or itself up with other applications compliant with either of the two request formats ('udp-channel' and 'src-dst pair'). A chain, such as, for instance, udpxy -> gigapxy -> udpxy -> media player, is made possible by this functionality.

f) *http://acme.com:8080/$TV9*

requests to use an **aliased channel** TV9 as the source, the destination defaulting to the requesting connection.

g) *http://acme.com:8080/src/$TV9?key=BF094744c5/dst*

requests the same aliased channel in gigapxy format and appends the *key* parameter to the URL the alias resolves to.

h) *http://acme.com:8080/udp/10.0.11.26@224.0.2.26:5050* OR
http://acme.com:8080/src/udp://10.0.11.26@224.0.2.26:5050/dst/

request (via **SSM**) a multicast channel at 224.0.2.26:5050 coming from 10.0.11.26, taking advantage of **IGMPv3**.

For further details on aliased channels one should refer to **channels.conf**(5)

HTTP URL re−direction

A client could be re−directed to an alternate source if the requested channel happens to be unavailable at the time. gws would reply with **HTTP 302** (Moved Temporarily) in the hope that the client software recognizes the code and would follow the re−direction link. **gws** performs a basic comparison check to ensure that there's no re−direction loop, yet the responsibility (re−direction loop detection & prevention) lies on the client side.

HTTP HEAD support

HTTP HEAD requests can be used to check for channel availability. **gws** treats HTTP HEAD in the same manner as it would treat a GET, with the exception that it would not send back any channel data; neither would it forward any information to a **gng**. Re−direction, however, is still performed as appropriate.

Administrative requests

Gigapxy listens on a dedicated TCP port for administrative requests. The request types are as below:

a) reports: *http://{addr}:{port}/report?type={type}&format={format}&cached={0|1}*

WHERE:

{type} ::= traffic|tps

{format} ::= html|web|xml

Gigapxy supports the following types of reports:

TPS (traffic, tps) - throughput statistics on active channels and clients.

The available report-output formats are:

HTML (html, web) - output as an HTML/web page.

XML (xml) - output as an XML page.

Other popular formats, such as *json* are also planned for the future.

Note: generation of throughput statistics should be enabled in appropriate config settings for TPS reports to work.

Caching: gws(1) may cache its reports for a certain time period, defined as **ws.report.cache_timeout_ms** in **gws.conf**(5) The request URL may request invalidation of the cache by using *cached=0* parameter. **NB:** this is to be used when getting the most actual data is critical. In all other cases, using cached reports would be a wiser choice, saving CPU resources when many report requests come in close proximity.

b) drop/disconnect a channel or a client: *http://{addr}:{port}/drop?channel={channel_tag}&client={client_tag}*

WHERE:

{channel_tag} is the name tag for the channel;

{client_tag} is the name tag for the client (within the given channel). If **client** parameter is missing, then **channel**={channel_tag} with **all its clients** will be disconnected.

Both channel and client must be specified **exactly** as TPS reports display them. For instance, for a multicast channel tagged as UDP://224.0.12.15:7010 (please do mind that URI parameters, such as authorization credentials etc., are *not* included) and a client tagged as TCP://192.168.10.15:50905, with **gws** listening for *admin requests* on 127.0.0.1:4047, the request:

http://127.0.0.1:4047/drop?channel=UDP://224.0.2.15:7010&client=TCP://192.168.10.15:50905 will drop (disconnect) **only the client**, leaving the channel up and running, whereas

http://127.0.0.1:4047/drop?channel=UDP://224.0.2.15:7010 would drop (disconnect) **all clients** within the channel and cancel/disconnect the channel's inbound data stream.

gws, upon receiving a 'drop' request, looks up the *channel* record (but not the client), locates the appropriate **gng** and relays the request to it. It is **not** the responsibility of **gws** to fulfill the request (since **gng** handles it from there), so **gws** would report success (HTTP 200 OK) as soon as the request is sent to**gng**. If the client in the request is invalid, the error will only be discovered by **gng** which sends no feedback to the request's origin. Should the request be successfully fulfilled by **gng**, it will report client/channel drops to **gws**, resulting in appropriate entries added to the access log (see **gws.conf**(5) for more info on gws logs).

c) ping/status of the service: *http://{addr}:{port}/ping* or *http://{addr}:{port}/status*; **status** keyword is supported to comply with the *udpxy* status command (which, in effect, resulted in a status report), which is **NOT** the equivalent ping, nevertheless, was used to check if the service is up; the preferred keyword for gigapxy is, of course, **ping**. **gws** returns **HTTP 200** whenever it receives the command.

d) disconnect all clients and channels: *http://{addr}:{port}/reset* - this will have **gws** send **SIGUSR2** to all attached **gng** instances. **SIGUSR2** directs a **gng** to drop all its channels and clients.

AUTHORIZATION

Gigapxy utilizes *authorization helpers* − user−supplied components − communicating with **gws**(1) via **STDIN** and **STDOUT**. With authorization enabled (via config), each user request results in an authorization request sent to a vacant *auth helper*. An illustrative example of a helper is prodvided at:

/usr/share/gigapxy/scripts/gauth.sh

An authorization request is a text string terminated by *CR/LF, with the following fields separated by whitespace:*

[ID] [peer] [source] [destination] [CRLF]

[ID] is A{num}, where {num} is a sequence number generated by **gws**; *Example:* A3404;

[peer] is combined IP address and port of the remote host requesting access; *Example:* 104.12.33.67:12301;

[source]: URI of the channel being requested and the *authorization token*; $*Example:* udp://224.0.2.12:5011?auth=ef031204ba0c.

NB: The format of the authorization token is not dictated in any way by **gws**: it's a mere convention between the client requesting access and the (user−defined) authorization logic embedded in the helper. **gws** passes what it recognizes as source to auth helper as is.

[destination] is URI for the destination or '−' for destination being the requesting TCP connection; *Example: −*;

[CRLF] is a sequence of two symbols with ASCII codes 0x0d and 0x0a.

The example request will be as below:

A3404 104.12.33.67:12301 udp://224.0.2.12:5011?auth=ef031204ba0c -

The helper validates the request and responds in the following format:

[ID] [result] [CRLF]

[ID] is the request ID, i.e. **A3404** in our case.

[result] is a numeric value that **gws** recognizes as an **approval** code **if 0 (zero)** and as **denial otherwise**.

Therefore, an approval for the request above should look as:

A3404 0

NB: A denial code could be arbitrary as long as it is non−zero; **gws** logic recognizes no difference between **1** and **210045**, they both indicate denial of access and result in the **403 Forbidden** HTTP response being forwarded to the client; then the client session ends.

Since **gws** does not have any guarantee that a helper would not block on a request, it **times out** auth requests using the applicable settings for user requests (please see **gws.conf**(5) for the particular settings). If a client/user request times out on handling an authorization task, the engaged auth helper gets **kill**(2) −ed.

Do make sure your time−out settings for user requests are well−balanced to allow ample time for auth requests to complete gracefully. Also, ensure that enough auth helpers are running to distribute requests to. **gws**(1) issues warnings about a slow auth helper when it detects one (at a time−out), a sequence of such warnings would indicate a mis−comfiguration issue.

Expiration date for trial versions of gigapxy

Please not that all **beta** versions come with an expiration date that is displayed in round brackets in application info. Running a gigapxy module (gws or gng) with -V option will display the application info line. Gigapxy will not run past the expiration date or if it cannot reliably tell what time it is, by contacting an NTP service over the internet. This feature is **not** applicable to non-trial (commercially licensed) versions of gigapxy.

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gws(1),**gng**(1),**gws.conf**(5),**gng.conf**(5),**channels.conf**(5),**gigapxy.auth**(5)

NAME

gws − Gigapxy web service daemon.

SYNOPSIS

gws [−h?TvVqkU] [−C *config_file*] [−l *logfile}] [−p pidfile]*

DESCRIPTION

gws is the front−end module of Gigapxy. It handles user and administrative requests submitted via HTTP protocol. The format of requests is described in the **gigapxy**(1) manpage.

gws dispatches user requests to Gigapxy engines, instances of a **gng**(1) daemon. At least one **gng** should be running for Gigapxy to accept requests for data. A **gws** can control up to 64 engines.

gws takes its parameters from a *configuration* file, which is either *gws.conf* or *gigapxy.conf* by default and can contain sections for any or all gigapxy modules. **gws** will look for the default configuration in a) *current directory*; b) */etc*; c) */usr/local/etc*. Path to a specific configuration file could be given at command−line (see **OPTIONS**). Configuration options for**gws** are described in detail in **gws.conf**(5) manpage.

gws rereads its configuration in response to SIGHUP. **gws** will force−rotate its log in response to SIGUSR1.

OPTIONS

gws accepts the following options:

−h, −−help, −?, −−options
: output brief option guide. This is **NOT** the behavior when run without parameters.

−C, −−config *path*
: specify configuration file.

−l, −−logfile *path*
: specify log file.

−p, −−pidfile *path*
: specify pid file.

−T, −−term
: run as a terminal (non-daemon) application. This is the default behavior when **gws** is run by a non−privileged user. −T could be specified when run as root in order **NOT** to become a daemon, for instance, for debugging purposes.

−v, −−verbose
: set the level of verbosity in the output. This option could be repeated to get to the desired level, which is 0, unless the option is used at least once. *Level 0* will reduce output to the very essential log entries of **NRM (normal)** priority; *level 1* will set verbosity to output to **INF (info)**: suitable for monitoring but not debugging; *level 2* will enable **DBG (debug)** level for *most (but not all)* application modules; *level 3* will set **DBG (debug)** for all modules. This switch has a rather inflexible nature, for more precise setting of log levels please use config settings alone.

−V, −−version
: output application's version and quit.

−q,−−quiet
: send no output to terminal. This is to supress any output normally sent to standard output or error streams. Unless specified, when run from a non−privileged account, **gws** will **mirror** diagnostic

messages sent to the log (as specified with the −l option) to standard output.

−k,−−oldmcast
use legacy multicast API. **gws** uses newer protocol−agnostic API by default, some (older) systems may not fully support it or exhibit erroneous behavior when using it. Enabling this option will have **gws** use the older protocol−specific multicast API.

−U,−−unauth
Disable authorization (if configured). This option allows a quick command−line override to disable whatever authorization method has been configured.

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gigapxy(1),**gng**(1),**gws.conf**(5),**gng.conf**(5)

NAME

gws.conf − Gigapxy web service daemon configuration file.

DESCRIPTION

gws(1) is the **gigapxy**(1) web service daemon, responsible for processing incoming requests. The configuration file contains the parameters read by the daemon at launch. The config file is human−readable and is in **libconfig** format. An example of a **gws.conf** is provided with the installation; please refer to it or the libconfig manual.

Once the parameters are read by **gws**, the daemon operates with those values util the configuration is *re−loaded* in response to **SIGHUP**.

All **gws**(1) settings beginning with the **ws.** prefix, as in *ws.section.param*. A configuration file could contain other (non-gws) settings too; **gws** will simply disregard those.

The configuration settings are given below. The default value for a setting is given in square brackets as **[default]**. Parameters without default values are **mandatory**.

ws.ng.socket_path = *path* **[/var/run/gpx-ngcomm.socket]**
is the domain socket path for communications between **gws** and the attached **gng**'s.

ws.ng.force_shutdown = true | false [true]
If true, **gws** will attempt to shut down (kill −SIGTERM) all attached **ng**'s on shutdown.

ws.ng.pick_method = *method* **[round-robin]**
gng selection method, using one of the following criteria: **round-robin** - next engine from the (circular) list; **min-channels** - engine with the minimum channels; **min-clients** - engine with the minimum clients.

ws.ng.accept_min_attached = *num* **[1]**
The number of NGs that should be **attached** to this **gws** before it can accept user requests.

ws.split_channels = true | false [false]
When set to true, **gws** chooses a **gng** for every new client before anything else, using *ws.ng.pick_method*. This allows to load-balance a single channel to multiple gng-s/cores. The default method (with this setting **off**) matches one channel to a particular **gng**: all clients for that channel get handled by the initially-assigned **gng**.

ws.log.*
Below are the settings pertaining to different modules within **gws**(1)**.** Setting verbosity for one of those allows to variate debug log detailization for specific modules within the program. Not every module though has a specific level attributed to it; most default to the non-specific **common** level.

The follow settings are for the application (debug) log. Application log captures various actions as they happen without any specific focus.

ws.log.level_default = err| crit| warn| norm| info| debug [info]
Defines the level of verbosity for the log across all modules.

ws.log.level_common = err| crit| warn| norm| info| debug [info]
Sets the level of verbosity for non-specific modules. **NB:** setting this le vel to **debug** will result in a VERY verbose output.

ws.log.level_syscall = err| crit| warn| norm| info| debug [info]
Sets the level of verbosity for system call and libc wrappers.

ws.log.level_bufd = err| crit| warn| norm| info| debug [info]
Sets the level of verbosity for (stream) buffer management operations.

ws.log.level_tput = err| crit| warn| norm| info| debug [info]
Sets the level of verbosity for operations on throughput statistics.

ws.log.file = *path*
Full path to debug log.

ws.log.max_size_mb = *num* **[16]**
Maximum file size (in Mb, i.e. 1048576−byte chunks). Log is rotated when this size is exceeded. **gws** will force−rotate its current log in response to **SIGUSR1**.

ws.log.max_files = *num* **[16]**
Maximum number of files to rotate to. The next rotation after this limit removes the oldest rotated log.

ws.log.time_format = local| gmt| raw| raw_mono| no_time| [local]
Sets format to display timestamps for log entries. *local* will loglocal−timezone specific time in YYYY-MM-DD HH24:MI TZ format. *gmt* will log GMT time in the same human−readable format as *local*; *raw* logs high−resolution time as the number of seconds.nanoseconds since the Epoch (1970-01-01 00:00:00 UTC); *raw−mono* logs system−specific **monotonic** time (used for timespan measurement, not correlated to clock time). *no_time* logs no time at all.

ws.log.show_pid = *true*|*false* **[true]**
Display PID as a log entry field.

ws.log.enable_syslog = *true*|*false* **[true]**
Write errors, warnings and critical messages to syslog(2).

ws.access_log.*
The following settings are for **gws** access log, serving a specific purpose of capturing channel and client session statistics. Access log is updated every time a new data stream is opened or closed. The entry types are:

OPEN_CHANNEL *channel_address*
gws opens a connection to the given channel. Data starts flowing from the channel (specified by *channel_address*) into internal storage and on to channel subscribers.

CLOSE_CHANNEL *channel_address num_users*
gws closes a connection to the given channel (specified by *channel_address*). *num_users* were subscribed to the channel at the point of closure.

OPEN_CLIENT *client_address channel_address*
A client at *client_address* successfully subscribes to channel at *channel_address*. This is prior to the moment when the first chunk of data gets sent to the client (by designated **gng**).

CLOSE_CLIENT *client_address channel_address num_users uptime nbytes npkts*
Client session ends; summary statistics showing: number of subscribers *num_users* left for the given channel; session *uptime* shown as **seconds.nanoseconds**; total bytes (*nbytes*) transferred; total packets/chunks (*npkts*) transferred.

NG_ATTACH/DETACH/QUIT *pid index fd*
New **gng**(1) attached/detached/quit to/from **gws**(1). Shown are: *gng* pid, internal index and connection fd. **NG_QUIT** means that gng may have sent no CLOSE_xx messages prior to its exit.

AUTH_START/EXIT *pid*
Authorization helper started/exited. Shown is the helper's pid.

ws.access_log.file = *path*
Full path to access log.

ws.channel_groups = *path* **[]**
Full path to aliased channel-group configuration file (if any). If empty, no channel groups will be defined. See details on aliased channel groups in **channels.conf**(5)

ws.channel_group_refresh = um [0]
Check every *N* seconds if channel-group config file changed, re-load and apply new channel-group settings if it did.

ws.access_log.max_size_mb = *num* **[16]**
Maximum file size (in Mb, i.e. 1048576−byte chunks). Access log is rotated when this size is exceeded.

ws.access_log.max_files = *num* **[16]**
Maximum number of files to rotate to. The next rotation after this limit removes the oldest rotated access log.

ws.access_log.time_format = local| gmt| raw| raw_mono| no_time| [local]
Sets format to display timestamps for log entries. See *ws.log.time_format* for details.

ws.access_log.show_pid = *true*|*false* **[true]**
Display PID as a log entry field.

ws.listener.*
The following are the settings equally applying to listeners of the two types of requests (admin and user) handled by the application. You can define up to 32 user and 4 admin listeners. See *gigapxy−commented.conf* for an example of multiple−listener config.

ws.listener.*.alias = *unique-alias* **[{ifc}:{port}]**
Unique human-readable identifier for the given listener. Populated by default by interface name and port (see below) separated by colon. For ifc=eth0 and port=3030, the alias, unless specified otherwise, would be set to *eth0:3030*.

ws.listener.*.ifc = *interface* **[any]**
Name or the address of the network interface for the listener of requests. **any , all** signifies the 'anonymous' interface with the address of 0, which means that the first eligible network interface will be picked by your OS.

ws.listener.*.port = *number*
Port number for the listener.

ws.listener.*.default_af = inet | inet6 [inet]
is the address family to be used when an interface cannot be uniquely linked to a family. For instance, an interface could have both IPv4 and IPv6 addresses associated with it.

ws.listener.*.is_safe = *true|false* **[false]**
Perform no authorization checks on user requests from this listener (allow all).

ws.pidfile.directory = *dirname* **[/var/run/gigapxy]**
Directory for the pidfile (must be writable by *ws.run_as_user*).

ws.pidfile.name = *filename* **[gws−*{user_port}*.pid]**
Name (w/o directory part of the path) of the pidfile, the default value uses the user−request listener port number.

ws.idle_clk_ms = *milliseconds* **[−1]**
Time (ms) to wait before doing any idle-time tasks, −1 = no limit. This sets the resolution (or granularity) for the timeouts or any other tasks done in idle time. The default value will have it perform idle tasks only when an actual event (connection, signal, etc.) interrupts the wait loop.

ws.max_sockets_to_accept = *num* **[127]**
Max number of sockets to accept in one event. When an incoming connection breaks the event loop, the module will try to **accept**(2) up to this limit of new sockets.

ws.multicast_ifc = *name* **[any]**
Default interface to use for sourcing multicast data.

ws.rcv_low_watermark = *num* **[16]**
Do not trigger a socket READ event unless at least *num* bytes have been received.

ws.run_as_user = *username* **[]**
Run as this user when running as a daemon (if empty, do not switch).

ws.run_as_uid = *uid* **[−1]**
Run as the given user (uid) when running as a daemon (if −1, do not switch). If gid is not specified, then gid = uid. uid > 0 will override *run_as_user*.

ws.run_as_gid = *gid* **[−1]**
Run in the given group (gid) when running as a daemon (if −1, gid = uid).

ws.tcp_no_delay = true | false [true]
Set TCP_NODELAY option for each accepted socket.

ws.use_http10_get = true | false [false]
Use HTTP/1.0 in channel (GET) requests for data. This is to prohibit the server to use **chunked** transfer encoding in response. **nginx**, often used as a proxy layer, has chunked encoding enabled by default and may send video stream wrapped as HTTP chunks. For now, **gigapxy** does NOT support parsing HTTP chunks in video streams.

ws.user_ping = true | false [false]
Allow '*ping*' or '*status*' requests on user−request listeners. NB: this feature is provided solely to maintain compatibility with **udpxy** which has no dedicated admin listeners. User−side pings are disabled by default, **DO NOT ENABLE** unless absolutely necessary, it is considered a safer practice to **use admin listeners** for all admin requests.

ws.legacy_multicast_api = true | false [false]
Use older (family-specific) API to manage multicast subscriptions.

ws.non_daemon = true | false [false]
If started as root, become a daemon if **true**.

ws.enforce_core_dumps = true | false [false]
When set to **true**, the process invokes the necessary syscalls to make itself *core-dumpable* and set core limit to *unlimited*. The default value of **false** leaves it to the shell defaults. **NB:** Under certain *Linux* versions, UID−changing daemons become non−core−dumpable (see **/proc/sys/fs/suid_dumpable** and **prctl**(2) for details).

ws.quiet = true | false [false]
No output to stdout/stderr if true.

ws.process_limits.*
This section allows to impose limits on the running process via **setrlimit(2)** syscall. Memory limits are specified as strings containing numerals and an optional denominator suffix, such as *Kb*, *Mb or Gb*. The number can have a fraction, so "1.5Kb" evaluates to 1024 + 512 = 1536 − the value to be submitted as a limit. "0" value or omission of a limit parameter leaves current (system−imposed) limit unchanged.

ws.process_limits.rss = {N}{suffix} ["0"]
Resident memory cap: a process cannot exceed this amount in resident memory, memory allocation call(s) should fail. **NB:** This limit cannot be enforced under **Linux**, where it would be replaced by **RLIMIT_AS** (virtual memory cap). If both RSS and VMEM are to be limited under Linux, the smaller value is used with RLIMIT_AS. Under **FreeBSD**, RSS limit is fully supported.

ws.process_limits.vmem = {N}{suffix} ["0"]
Virtual memory cap = **RLIMIT_AS**. Used in place of *RSS* cap under Linux. Both Linux and FreeBSD fully support it.

ws.http_read_timeout_ms = *milliseconds* **[200]**
Timeout (in milliseconds) to read an HTTP−message portion.

ws.user_request_timeout_ms = *milliseconds* **[500]**
Timeout (in milliseconds) for a user request to be processed.

ws.admin_request_timeout_ms = *milliseconds* **[300]**
Timeout (in milliseconds) for an admin request to be processed.

ws.module_request_timeout_ms = *milliseconds* **[100]**
Timeout (in milliseconds) for a module request to be processed. Module requests are those that go between *gws* and *gng*.

ws.http_data_content_type = *type_specifier* **[application/octet-stream]**
HTTP Content-Type for data payload.

ws.channel_sample_timeout_ms = *milliseconds* **[−1]**
Pre-sample each new channel trying to read from it with the given timeout; unless −1 == timeout, then do **NOT** pre-sample channels. **NB:** channels will be pre-sampled by **gws**, which will therefore **wait** and suffer the associated latency penalty.

USE WITH DISCRETION.

ws.tput_stats.*
The following section specifies the parameters needed for engines to report **traffic throughput statistics**, queried using **report** admin request. See **gigapxy**(1) for details on reports and admin request particulars.

ws.tput_stats.enabled = true | false [true]
Do not provide channel/client statistics unless true. Please note that engines will use additional CPU cycles to gather and calculate relevant statistics.

ws.tput_stats.channel_path = *posix_shmem_path* **[/gxy-cha.shm]**
POSIX shared memory path for channel statistics (<= 32 characters).

ws.tput_stats.client_path = *posix_shmem_path* **[/gxy-cli.shm]**
POSIX shared memory path for client statistics (<= 32 characters).

ws.tput_stats.max_channel_records = *num* **[250]**
Max number of records (across all engines) in channel statistics. This should be no less than the maximum number of channels to be handled at once.

ws.tput_stats.max_client_records = *num* **[1000]**
Max number of records (across all engines) in client statistics storage. This should be no less than the maximum number of clients to be handled at once by all engines.

ws.tput_stats.max_speed_delta = *num* **[8]**
Max difference (in Kb) between channel and client speeds. Speed delta is visible in TPS reports and will be highlighted if delta gets exceeded.

ws.report.*
The following section specifies the parameters needed to support generation of various reports.

ws.report.default.type = *name* **[traffic]**
Default report type to use (with a URL not specifying one).

ws.report.default.format = *name* **[html]**
Default report format to use (with a URL not specifying one).

ws.report.memory.min = *bytes* **[524288]**
Initial memory for the spool buffer (to contain full report text prior to the output).

ws.report.memory.max = *bytes* **[16777216]**
Maximum memory for the spool buffer (to contain full report text prior to the output).

ws.report.max_send_attempts = *num* **[16]**
Max number of transfer/send/output attempts to take if cannot output all at once.

ws.report.cache_timeout_ms = *num* **[500]**
Reports will be cached and served to subsequent requests within this timespan (ms), or NOT cached at all if the value <= 0 (a fresh report will be generated for each request).

ws.report.backup_file = *filepath* **[]**
File to save each report into (overwriting the previous one). If empty, do NOT save.

ws.sync.regular_timeout_ms = *ms* **[500]**
After a GNG attaches, synchronize (retrieve) channel/client stats from TPS cache in N ms after the attach. Enabled only if TPS (*ws.tput_stats.enabled* is **true**).

ws.sync.forced_timeout_ms = *ms* **[10000]**
If at least one GNG is attached, synchronize (retrieve) channel/client stats from TPS cache every N ms. Enabled only if TPS (*ws.tput_stats.enabled* is **true**).

ws.redirect.err_channel = *channel_URL* **[]**
Redirect client (via **HTTP 302**) to *channel_address* if requested channel is unavailable (for any reason other than an error in an internal component of gigapxy). Channel URL must be a full HTTP URL that will be returned to client via HTTP 302 response.

ws.redirect.no_access = *channel_URL* **[]**
Redirect client (via **HTTP 302**) to *channel_address* if access to the requested channel has been denied (by an authorization helper). Channel URL must be a full HTTP URL that will be returned to client via HTTP 302 response.

ws.psensors.*
Performance sensors allow to measure resource utilization between two specific points within the application, using the metrics provided by **utime**(2) **utime**(2) call at each end of the sensor. All sensor data will be printed out at the application exit in the format similar to the output of **time(1)** utility.

Performance sensors are a debugging/profiling facility and incur additional load on the system.

USE WITH DISCRETION.

Defined sensors:

app = application runtime; **ev_loop** = event processing (all events); **ev_read** = reading/processing inbound data; **ev_write** = writing/processing outbound data; **ev_err** = processing error events; **ev_pp** = post−processing events; **ws_userq** = processing user requests; **ws_admrq** = processing administrative requests (reports, etc.).

ws.psensors.enable_all = *true|false* **[false]**

Enables all sensors if true, disables all otherwise. This is to initialize the set of enabled−sensor flags to either all ones (if enabled) or all zeros. This setting is to be used in combination with **ws.psensors.except**.

ws.psensors.except = *sensor_list* **[]**

Enables sensors in the list if **ws.psensors.enable_all** is *true*, or disables those sensors if *false*. This way *enable_all* is used to initialize the set of sensors while *except* narrows it down by enabling/disabling its specific elements.

EXAMPLE A:

ws.psensors.enable_all = true; # Enable all sensors.

ws.psensors.except = ["ev_read", "ev_write"]; # Disable those listed herein.

Enables all sensors except *ev_read* and *ev_write*.

EXAMPLE B:

ws.psensors.enable_all = false; # Disable all sensors.

ws.psensors.except = ["ev_read", "ev_write"]; # Enable those listed herein.

Enables *ev_read* and *ev_write* sensors, all others are disabled.

ws.auth.*

Authorization helpers are user-defined applications (plug−ins) used by **gws** to screen user requests, based on request−specific data, such as user address, request URI, etc. **gws** starts one or several helpers and communicates with them via pipes connected to helpers' *STDIN* and *STDOUT* streams. Example helper scripts (*a1p-auth.sh*, *b2p-auth.sh*) for two supported protocols are provided in */usr/share/gigapxy/scrpts* under Linux (*/usr/local/share/*.. under FreeBSD).

ws.auth.enabled = *true|false* **[false]**

Enable helpers unless false.

ws.auth.helper_protocol = *"A1P"|"B2P"* **["A1P"]**

Defines the communication protocol between **gws** and *auth helpers*. A1P is the older/simpler protocol, please see details in **gigapxy.auth(5)**

ws.auth.b_fields = *fields* **["USDP"]**

This **B2P-specific** setting defines the fields (and their order) to be sent to *auth helpers* for evaluation. "USDP" stands for *URL*, *Source*, *Destination* and *Peer* - they will be sent to helpers in that order. Full list of protocol-supported fields can be found in **gigapxy.auth(5)**

ws.auth.exec = *exec_path_with_params* **[]**
Specify full path to the helper executable with all command−line parameters. This constitutes a complete absolute−path to the helper binary **with** all required command−line options and parameters. **NB:** all helpers will be launched under user/group specified in *ws.run_as** settings.

ws.auth.min_helpers = *count* **[1]**
Number of helpers to start with and always keep running.

ws.auth.max_helpers = *count* **[1]**
Maximum number of helpers to run.

ws.auth.deny_no_auth = *true|false* **[false]**
Deny access to URI/resource if authorization cannot be performed (due to an internal error). Allow by default so that authorization framework failure would not result in denial of service.

ws.auth.no_spawn_tmout = *ms* **[5000]**
Maximum time (ms) to disallow launching helpers after suspected cascading crashes. When a helper crashes shortly after being launched, **gws** disables further helper launches for the configured time period.

ws.auth.aux_params = *list_of_params* **[]**
Additional **A1P-specific** parameters passed to auth helpers. The available parameters are:

listener-alias = alias for the originating listener

ws.auth.can_rewrite_endpoints = *true|false* **[false]**
Instructs **gws** using **B2P protocol** to be ready to re-write *Source* or *Destination* endpoints if specified in *auth helper* response message.

ws.auth.allow_custom_urls = *true|false* **[false]**
Instructs **gws** using **B2P protocol** to allow URLs that do not follow the two **gigapxy-oriented** patterns (*udp/address:port* or *src/s_url/dst/d_url*). This setting should be **true** if *auth helpers* were to match custom URLs to custom *Source/Destination*.

ws.auth.cache.*
Negative authentication responses can be cached by gws. This allows for much faster response when helpers' time is at the premium and may better chances in case of a DOS attack. The cache's eviction method is **LRU** (least recently used) and each entry (source URL) has a time-out.

ws.auth.cache.enabled = *true|false* **[false]**
Enable response cache if set to *true*.

ws.auth.cache.max_records = *num* **[5000]**
Set the maximum number of items in cache. If the number goes higher, extra items will be LRU−evicted.

ws.auth.cache.expiry_sec = *num* **[300]**
Set the lifespan of a cache item, in seconds.

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gigapxy(1),**gws**(1),**gng**(1),**gng.conf**(5),**channels.conf**(5),**gigapxy.auth**(5)

NAME

channels.conf − Gigapxy channel−group configuration file.

DESCRIPTION

gws(1) uses channel−group configuration to define channel sources that could be referenced not by absolute address but via an **alias**. An alias is a name prepended by a dollar−sign character. **gws**, as it processes a URL, recognizes an alias and translates it to an absolute−address URL to be used as a source.

An alias creates a name−to−URL mapping for user requests.

An example channel−group configuration is provided with the installation at */usr/share/doc/gigapxy* under Linux or */usr/local/share/doc/gigapxy* under BSD. **channels** is the top−level section, under which **channel groups** are listed/defined. The parameters used in configuring a single channel group are as below:

alias

This is the name to be used in URLs with the dollar−sign prefix. The name/alias will be translated into one of the URLs from the set defined for the given group.

urls

The URL to resolve the alias to. A URL may contain an alias but only to be resolved remotely (by the **gigapxy** daisy−chained to the current one). In the future, more than one URL (with a load−balancing option) may be supported for this setting.

EXAMPLE

This is what contents of a **channels.conf** file may look like:

```
channels = (
  {  alias = "TV5"; urls = ["file:///opt/prog/tv5/channel-down.ts"]; },
  {  alias = "NightLife"; urls = ["udp://10.0.24.16:5054"]; } );
```

Aliases are used with a dollar-sign prefix. A request to TV5 channel thus may look as:

a) http://acme.com:8080/$TV9

Or, in src/dst format, with a custom **key** parameter:

b) http://acme.com:8080/src/$TV9?key=BF094744c5/dst

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gigapxy(1),**gws**(1),**gws.conf**(5)

NAME

gigapxy.auth − Gigapxy authentication manual.

DESCRIPTION

gws employs *helpers* − custom scripts − to authenticate and authorize incoming user requests. Any application reading from **STDIN** and responding via **STDOUT** could serve as a *helper* as long as it 'speaks' one of the two communication protocols: **A1P** or **B2P**. This page is dedicated to giving the insight into *auth helpers*, employed protocols and associated capabilities.

Common features:

Both protocols have things in common. Firstly, they are textual and line-oriented: a message is a text string ending with **CRLF** ASCII sequence (single **LF** symbol under Linux and FreeBSD). **gws** writes messages/lines to *helpers* via unnamed pipes connecting to the helpers' **STDIN**.

Message example: **B10 P 134.12.12.50:5050**

Messages contain *fields* separated by whitespace. An empty (blank) value is always specified as − (dash). Some fields are common for both protocols, the first field is always the same: *Session ID*.

Session-ID = A|B{1 .. 2147483647} [examples: A100, A1, B150433]

Identifies the request. The first symbol is *protocol_id*: 'A' for **A1P** and 'B' for **B2P**. The rest of the field is **session_number** - non-zero 32-bit unsigned decimal integer; *session ID* in the incoming message should match the one in the response.

Result-Code = {0 .. 2147483647} [examples: 0, 1, 111]

32-bit unsigned decimal integer, specifies the result of the evaluation. Only **0 (zero)** code is treated as **APPROVE** response, all others currently signify authorization failure.

A1P request:

A1P uses pre-defined sequence of mandatory and optional fields in each request/response message.

The request fields are: *Session-ID Peer Source Destination* [*Listener*] (the last field is optional and is added only if **ws.auth.aux_params** value contains **listener-alias**).

A1P request example: **A102 10.0.1.15:30403 udp://224.0.2.25:3030 - bb1**

Peer = address:port

Address/port of the client (that sent the original request to **gws**)

Source = channel URL

URL of the requested channel, as specified either in **udp** or **src** section of the request URL.

Destination = client URL

URL for the destination. For most requests, destination is the socket/connection that started the request (i.e. peer), empty value (dash) is used to specify it.

Listener = alias

Alias of the listener that accepted the request. Do mind that when using this option, **alias** must be specified for each listener in **gws.conf**.

A1P response:
A1P response example: **A102 0**

B2P protocol
B2P is an extension of A1P protocol that mainly addresses the **inflexibility** of A1P (fixed number of fields come in and come out). The core features that drove towards creating a new protocol were: a) *custom URLs* and b) endpoint (source/destination) *re-write capability*. B2P accommodates both of these features and provides future expansion of functionality. B2P adds one **mandatory** field to *Session-ID*, the *Field-Mask*.

Field-Mask = [a-z][A-Z]{16} [example: USDP]
Specifies the fields (up to 16) that will follow (in the order they will appear). A single symbol is designated to each of the recognized fields, the mask is, in effect, a sequence of field identifiers.

Field identifiers:
U = Request-URL - the B2P field that holds URL for the HTTP request, the way it was in the header. Example: /udp/224.0.4.56:4504

S = Source

D = Destination

P = Peer

A = User-Agent

L = Listener

r = Result-Code

Field-Mask 'USDP' means that the message, besides the mandatory two fields, must have four fields of the corresponding types. **gws.conf** pro vides *ws.auth.b_fields* setting to specify what information gws will send to auth helpers with every B2P message.

B2P request:
Example B2P request: **B102 UPL /udp/224.0.2.26:5034?auth=0x93fb0ad 10.0.3.14:40987 bb1**

Some fields (**r**) don't make much sense in the request and will be rejected by **gws** if specified.

B2P response:
It's up to the helper implementation what set of fields would be returned, but at least one field should be. Absense of **Result-Code** is assumed as **APPROVE** as long as other fields are present in the response. With all the flexibility, only certain fields will be accepted in by **gws** in the response message.

Response-approved fields:
S = source will be re-written to the returned value

D = destination will be re-written to the returned value

r = APPROVE if 0, DENY otherwise.

A typical B2P **denial** response would be: **B102 r 111** (Don't you worry about 111, any non-zero number would do).

Custom URLs and source re-write
B2P (and appropriate settings in **ws.auth** config section) allows completely opaque URLs to be converted to gigapxy-compliant source/destination pairs. **Request-URL** field matched to helper-specific endpoints

allows to reply with the appropriate **Source** (and **Destination** is needed) and let **gws** know what the endpoints are.

Here's an example scenario:

GET /dc03d03332f09a is the original HTTP request as read by **gws**.

The auth config specifies:

```
auth: {
    enabled = true;
    helper_protocol = "B2P";
    b_fields = "USP";
    exec = "/usr/local/bin/b2p-auth.sh /var/log/gigapxy/auth.log";
    deny_no_auth = true;
    can_rewrite_endpoints = true;
    allow_custom_urls = true;
};
```

allow_custom_urls lets **gws** ignore that the URL could not be parsed into gigapxy endpoints, so both *Source* and *Destination* remain empty after request has been parsed.

gws sends a B2P request: **B1 USP /dc03d03332f09a - 10.0.14.26:40987**

Please note that **Source** is empty in the request and could be omitted if we know it's never needed by the helper. The helper translates the data (using its own logic) into the following response:

B1 S udp://226.0.3.14:6060

gws reads the response and assumes the request is APPROVED (no **r** field but another field present). It then takes **udp://226.0.3.14:6060** as the source endpoint, directing to read from the given multicast channel.

Where do I begin?

Having decided which features you'd need and thus which protocol to select, make a copy of the corresponding *example helper* in **/usr/share/gigapxy/scrpts** under Linux (**/usr/local/share/..** under FreeBSD). If you understand the logic, but dislike /bin/sh, use any other language. Once your helper (kind of) works, make a text file (requests.txt) with sample requests (the kind you'd be most likely processing) and run:

cat requests.txt | auth-helper /var/log/helper.log

The output will be the response messages. If somethig does not quite work, the log (where **your** script writes) should help.

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gigapxy(1),**gws**(1),**gng**(1),**gng.conf**(5),**gws.conf**(5)

NAME

gng − Gigapxy engine daemon.

SYNOPSIS

gng [−h?TvVq] [−C *config_file*] [−l *logfile}] [−p pidfile] [−w path] [−P cpunum]*

DESCRIPTION

gng is the Gigapxy **engine** module performing I/O on behalf of data requests submitted to **gws**. The format of **gws** requests is described in the **gigapxy**(1) manpage.

gng *attaches* to the specified **gws** upon start−up; up to 64 engines may attach to a single **gws**. The controlling **gws** relays (pre−processed) data requests to the attached engines for execution.

gng takes its parameters from a *configuration* file, which is either *gng.conf* or *gigapxy.conf* by default and can contain sections for any or all gigapxy modules. **gng** will look for the default configuration in a) *current directory*; b) */etc*; c) */usr/local/etc*. Path to a specific configuration file could be given at command−line (see **OPTIONS**). Configuration options for**gws** are described in detail in **gng .conf**(5) manpage.

gng re−reads its configuration in response to SIGHUP. **gng** will force−rotate its log in response to SIGUSR1.

OPTIONS

gng accepts the following options:

−h, −−help, −?, −−options
output brief option guide. This is **NOT** the behavior when run without parameters.

−C, −−config *path*
specify configuration file.

−l, −−logfile *path*
specify log file.

−p, −−pidfile *path*
specify pid file.

−w, −−gws *path*
specify path to the controlling **gws** (domain socket).

−T, −−term
run as a terminal (non-daemon) application. This is the default behavior when **gws** is run by a non−privileged user. −T could be specified when run as root in order **NOT** to become a daemon, for instance, for debugging purposes.

−v, −−verbose
set the level of verbosity in the output. This option could be repeated to get to the desired level, which is 0, unless the option is used at least once. *Level 0* will reduce output to the very essential log entries of **NRM (normal)** priority; *level 1* will set verbosity to output to **INF (info)**: suitable for monitoring but not debugging; *level 2* will enable **DBG (debug)** level for *most (but not all)* application modules (please mind that **bufd** is **NOT** at debug at level 2); *level 3* will set **DBG (debug)** for additional modules, including **bufd**; *level 4* will set all modules to debug. This switch has a rather inflexible nature, for more precise setting of log levels please use config settings alone.

−V, −−version
output application's version and quit.

−q,−−quiet
send no output to terminal. This is to supress any output normally sent to standard output or error streams. Unless specified, when run from a non−privileged account, **gws** will **mirror** diagnostic messages sent to the log (as specified with the −l option) to standard output.

−P,−−cpu
Set CPU affinity for the main process. This option allows to restrict the main **gng** process to the given CPU/core (numbered from *0* **to** *N*−**1).**

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gigapxy(1),**gws**(1),**gws.conf**(5),**gng.conf**(5)

NAME

gng.conf − Gigapxy engine configuration file.

gng(1) is the **gigapxy**(1) data processing daemon, responsible for the I/O on behalf of user requests relayed to it by **gws**(1). The configuration file contains the parameters read by the daemon at the launch. The file is in easy−to−read **libconfig** format. An example of a **gng.conf** is provided with the installation.

Once the parameters are read by **gng**, the daemon operates with those values util the configuration is *re−loaded* in response to **SIGHUP**.

All **gng**(1) settings begin with the **ng.** prefix, as in *ng.section.param*. A configuration file may also contain settings intended for other modules; **gng** would ignore all non−gng settings.

The configuration settings are listed below. The default value for a setting is given in square brackets as **[default]**. Parameters without default values are **mandatory**.

ng.ng_socket_path = *path* **[/var/run/gpx-ngcomm.socket]**
is the domain socket path for communications between **gws** and the attached **gng**'s.

ng.log.level_default = err| crit| warn| norm| info| debug [info]
Defines the level of verbosity for the log.

ng.log.file = *path*
Full path to log.

ng.log.max_size_mb = *num* **[16]**
Maximum file size (in Mb, i.e. 1048576−byte chunks). Log is rotated when this size is exceeded. **gng** will force−rotate its current log in response to **SIGUSR1**.

ng.log.max_files = *num* **[16]**
Maximum number of files to rotate to. The next rotation after this limit removes the oldest rotated log.

ng.log.time_format = local| gmt| raw| raw_mono| no_time| [local]
Sets format to display timestamps for log entries. *local* will loglocal−timezone specific time in YYYY-MM-DD HH24:MI TZ format. *gmt* will log GMT time in the same human−readable format as *local*; *raw* logs high−resolution time as the number of seconds.nanoseconds since the Epoch (1970-01-01 00:00:00 UTC); *raw−mono* logs system−specific **monotonic** time (used for timespan measurement, not correlated to clock time). *no_time* logs no time at all.

ng.log.show_pid = *true*|*false* **[true]**
Display PID as a log entry field.

ng.log.enable_syslog = *true*|*false* **[true]**
Write errors, warnings and critical messages to syslog(2).

ng.pidfile.directory = *dirname* **[/var/run/gigapxy]**
Directory for the pidfile (must be writable by *ng.run_as_user*).

ng.pidfile.name = *filename* **[gng−***{user_port}***.pid]**
Name (w/o directory part of the path) of the pidfile, the default value uses the user−request listener port number.

ng.idle_clk_ms = *milliseconds* **[−1]**
Time (ms) to wait before doing any idle-time tasks, −1 = no limit. This sets the resolution (or granularity) for the timeouts or any other tasks done in idle time. The default value (−1) will have it perform idle tasks only when an actual event (connection, signal, etc.) interrupts the wait loop. **gng** will set the idle clock to the minimum value of a channel/client timeout.

ng.run_as_user = *username* **[]**
Run as this user when running as a daemon (if empty, do not switch).

ng.run_as_uid = *uid* **[−1]**
Run as the given user (uid) when running as a daemon (if −1, do not switch). If gid is not specified, then gid = uid. uid > 0 will override *run_as_user*.

ng.run_as_gid = *gid* **[−1]**
Run in the given group (gid) when running as a daemon (if −1, gid = uid).

ng.non_daemon = true | false [false]
If started as root, become a daemon if **true**.

ng.enforce_core_dumps = true | false [false]
When set to **true**, the process invokes the necessary syscalls to make itself *core-dumpable* and set core limit to *unlimited*. The default value of **false** leaves it to the shell defaults. **NB:** Under certain *Linux* versions, UID−changing daemons become non−core−dumpable (see **/proc/sys/fs/suid_dumpable** and **prctl**(2) for details).

ng.no_rtp_strip = true | false [false]
When set to **true**, the engine does **not** attempt to convert RTP-over-TS into plain TS datagrams (enabled by default). When 'stripping' is disabled, **gng** would consider RTP packets as non-TS and relay them AS-IS.

IMPORTANT: data buffers MUST be memory−mapped (mmap_files or mmap_anon) for gng to
perform RTP stripping. Disable RTP stripping (ng.no_rtp_strip = true) if using non−memory buffers.

ng.use_sendfile = true | false [true on FreeBSD otherwise false]
Prefer to use sendfile(2) to send out data. This makes a big difference on FreeBSD, which implements zero-copy through this syscall. Setting this to **true** on Linux may or may not improve performance (so it's false by defaylt under Linux).

ng.quiet = true | false [false]
No output to stdout/stderr if true.

ng.cpunum = -1 | 0 .. N [-1]
Set affinity to CPU #N (zero-based) for this process, unless -1 (or <0).

ng.process_limits.rss = {N}{suffix} ["0"]
Resident memory cap: a process cannot exceed this amount in resident memory, memory allocation call(s) should fail. **NB:** This limit cannot be enforced under **Linux**, where it would be replaced by **RLIMIT_AS**

(virtual memory cap). If both RSS and VMEM are to be limited under Linux, the smaller value is used with RLIMIT_AS. Under **FreeBSD**, RSS limit is fully supported.

ng.process_limits.vmem = {N}{suffix} ["0"]
Virtual memory cap = **RLIMIT_AS**. Used in place of *RSS* cap under Linux. Both Linux and FreeBSD fully support it.

ng.max_channels = *num* **[200]**
Maximum number of channels allowed (per engine).

ng.max_channel_clients = *num* **[500]**
Maximum number of clients per single channel.

ng.channel_io_timeout_sec = *seconds* **[5]**
Maximum time (in seconds) to wait on I/O for a channel.

ng.client_io_timeout_sec = *seconds* **[5]**
Maximum time to wait on I/O for a client.

ng.client_busy_timeout_sec = *seconds* **[86400 = 24 hours]**
Maximum time for a client session.

ng.can_extend_clients = *true*|*false* **[false]**
If a client times out, check if there's pending (channel) data and the client is writable. If writable, extend its wait period (just this one time) by client_io_timeout_sec.

ng.client_socket_sndbuf_size = *bytes* **[system default]**
Client (sending) socket send buffer size (bytes).

ng.channel_socket_rcvbuf_size = *bytes* **[system default]**
Channel (receiving) socket buffer size (bytes).

ng.channel_lo_wmark = *bytes*, **0** = **none [0]**
Low watermark for channel sockets.

ng.client_tcp_cork = *true*|*false* **[false]**
Use Linux TCP_CORK socket option to aggregate client packets. Linux only.

ng.client_tcp_nopush = *true*|*false* **[false]**
Use BSD TCP_NOPUSH socket option to aggregate client packets. BSD only.

ng.multicast_ttl = *hops* **[2]**
Multicast TTL value set for the outgoing mulitcast traffic.

ng.tput_stats.*
The following section specifies the parameters needed for engines to report **traffic throughput statistics**, queried using **report** admin request. See **gigapxy**(1) for details on reports and admin request particulars.

ng.tput_stats.enabled = true | false [true]
Do not provide channel/client storage unless true. Please note that engines will use additional CPU cycles to gather and calculate relevant statistics.

ng.tput_stats.channel_path = *posix_shmem_path* **[/gxy-cha.shm]**
POSIX shared memory path for channel storage (<= 32 characters). **Note:** should match the corresponding **gws** setting.

ng.tput_stats.client_path = *posix_shmem_path* **[/gxy-cli.shm]**
POSIX shared memory path for client storage (<= 32 characters). **Note:** should match the corresponding **gws** setting.

ng.tput_stats.channel_report_ms = *milliseconds* **[5000]**
Report channel throughput every N milliseconds. (Will save the statistics in shared memory.)

ng.tput_stats.client_report_ms = *milliseconds* **[5000]**
Report channel throughput every N milliseconds. (Will save the statistics in shared memory.)

ng.tput_stats.max_packet_delta = *bytes* **[−1]**
Warn if two consecutive packets differ by more that N bytes, −1 = ignore. This setting allows to watch out for inconsistencies in the UDP streams, where all messages are supposed to be of the same size.

USE WITH DISCRETION.

ng.ws.max_reconnects = *num* **[10]**
Attempt N reconnects with **gws**, unlimited if −1, none if 0. If the controlling **gws** crashes, **gng** makes a number of attemtps, separated by pauses, to re−attach to it. Therefore, if a monitor on the crashed **gws** restarts it successfully, the formerly−attached **gng**'s may re−attach.

ng.ws.reconnect_delay = *milliseconds* **[500]**
Delay (in milliseconds) between reconnect attempts.

ng.http_data_content_type = *type_specifier* **[application/octet-stream]**
HTTP Content-Type for data payload.

ng.bufd.*
The following section specifies the parameters for the internal cache used by **gng** to multiplex access to channel data. For each channel (that needs to be cached) **gng** maintains a chain of buffers, representing consecutive segments for traffic data.

ng.bufd.keep_files = true | false [false]
Do not unlink(2) bufd files (make them visible). This is a debugging option.

USE WITH DISCRETION.

ng.bufd.mmap_files = true | false [true]
Map bufd files into memory. Results in faster access to cache but may exhaust host memory.

ng.bufd.mmap_anon = true | false [false]
Allocate buffers in memory w/o using any filesystem space (i.e. buffers are not backed up by files). This option provides the fastest access to cache but is limited by process's memory constraints. It also overrides mmap_files.

MUST USE mmap_anon OR mmap_files if you intend to strip RTP datagrams.
gng can only strip RTP datagrams in memory, so memory mapping is a MUST if you're handling RTP traffic. If you're NOT handling RTP and economize on RAM using file-backed buffers, then please **disable** RTP stripping by setting **ng.no_rtp_strip = true**

ng.bufd.mlock = true | false [false]
mlock(2) data buffers into physical memory. Make sure your system parameters allow this, for reference see mlock(2) manpage.

ng.bufd.data_dir = *pathname* **[/tmp]**
Directory to place bufd files into.

The following three settings affect the way **gng** caches data. There is a certain amount that can be kept per channel to ensure that new clients can start receiving data without delay. The settings below regulate that amount and set the point (in the cache) from which data gets served to a new client.

ng.bufd.min_total_duration_sec = *seconds* **[5]**
Minimum of data cached for a channel, measured in time it took to receive it. No channel buffers get recycled until this much data has been saved. The exact amount preserved in cache could be a above but **never below** the imposed threshold; **gng** would recycle a buffer only if, after its removal, the cache would still have >= *seconds* worth of data.

ng.bufd.min_total_size = *bytes* **[1048576]**
Minimum of data cached for a channel, in bytes. No channel buffers get recycled until this much data has been saved.

The two settings above work in tandem, each of them setting a threshold. **gng** will consider that enough data has been cached as soon as either or both of those thresholds have been reached: if, for instance, the first setting is **5 seconds** and the second one is **10485760 bytes (10 Mb)**, then **enough** is as soon as we've cached 10Mb or accumulated more than 5 seconds worth of data (if the channel is slow, it may be less than 10Mb).

Channel data is stored as a **sequence of buffers**, from the most−recently−received one − the **HEAD**, to the oldest one − the **TAIL**.

A newly−joined client/subscriber needs the first data buffer to start with. The setting below defines the algorithm **gng** would use to pick one.

ng.bufd.start_mode = 0 | 1 | −1 | 2 [1]
Defines the method to pick the initial buffer for a client:
0 (HEAD) picks the most−recently−received buffer, offset: 0 (cached: all of the most recent buffer).

2 (EDGE) picks the most−recently−received buffer, offset: END−OF−DATA (*no data cached*).

1 (MIN_CACHED) picks the buffer that allows to transfer N seconds or M bytes of data, offset: 0 (cached: N seconds/M bytes).

−1 (TAIL) picks the oldest buffer in cache, offset: 0 (cached: all current data for the channel).

All the above methods, except **EDGE** (2) position a new client at zero offset in the selected buffer. This way there is always some data to send to the client right away, without I/O wait. **EDGE** ensures a **no-cache** policy: only the data received during the client's lifespan gets relayed to the destination.

MIN_CACHED (1) uses *ng.bufd.min_total_duration_sec* and *ng.bufd.min_total_size* to determine which buffer to pick. The algorithm starts at the HEAD and moves towards the tail accumulating the volume and time for each buffer it lands on; as soon as either of the thresholds is reached, the buffer is selected as the one to start at.

ng.bufd.burst_mode = 0 (none) | 1 (scan) | 2 (burst) [1]
Defines the method used to prevent excessive growth of buffer chains using a 'bubble-burst' technique. A 'bubble' is a (long) sequence of unused buffers (U) squeezed in between a small number of active buffers (A). A slow client may claim (lock on) a buffer and then slow down, while other clients go ahead. The tail-side buffer would be still locked while the buffers towards the head get used and un−locked: AA*UUUUUU-UUUUU*A. The U-sequence is the 'bubble'. In mode **1 (scan)** gng scans a channel looking for a bubble (and warns if it finds one), in mode **2 2 (burst)** it also tries to invalidate the rightmost A−buffer and release the underlying U−sequence (the bubble).

ng.bufd.max_unit_count = *num* **[128]**
Maximum number of buffers for all channels within the given **gng** instance. Not all buffers, as a rule, get allocated at once. This sets the limit to the number of buffers across all channels.

ng.bufd.prealloc_count = *buffers* **[max_unit_count / 4]**
Number of shared buffers to pre-allocate at the engine's start.

ng.bufd.max_units_per_channel = *num* **[1/3 of max_unit_count, yet within 4..12 range]**
Maximum numbers of buffers per channel. Setting this to a well-balanced value will provide for fair distrubution of buffers across channels and will prevent hogging buffer space by channels with slow clients.

ng.bufd.max_unit_size = *bytes* **[16777216]**
Maximum number of bytes in a single buffer. When this threshold is hit, another buffer is added to cache.

ng.bufd.max_dgram_size = *bytes* **[1500]**
Maximum size of a (UDP) datagram expected (should not exceed MTU). **NB:** must be adjusted if using *jumbo frames*.

ng.bufd.max_unit_duration_sec = *seconds* **[30]**
Maximum duration (seconds) of a single buffer. When this threshold is hit, another buffer is added to cache.

ng.bufd.allow_emergency_recycle = true | false [false]
Allow to force−recycle the oldest buffer when cannot allocate a new one.

ng.bufd.client_catch_up_ms = *ms* **[0]**
Advance to the most recent buffer if no data gets sent out within the given time period. Once a client times out, its output marker is shifted to 'catch up' with the source data flow. For start_mode = **EDGE (2)**, the

offset would be at the end of the head buffer (edge), for other modes - byte 0 of the head buffer.

associated with a channel to track incoming datagrams. Certain features rely heavily on such tracking. A datagram index holds a number of datagram sequences. Each sequence accounts for a number of consecutive datagrams *of the same size*.

ng.bufd.stale_dgram_ms = *ms* **[0]**
If non−zero, the value sets expiry period for inbound data in UDP channels. If a client finds itself at the point of needing to transmit a stale datagram, its output offset gets shifted to byte 0 of the most−recently−received datagram.

ng.bufd.max_dseq_ms = *ms* **[8 * stale_dgram_ms]**
Maximum time in milliseconds (worth of data) to aggregate in a single datagram sequence. This setting could be made smaller to increase precision of time measurement for *stale_dgram_ms* or made larger to aggregate more datagrams in a single sequence (in order to save space).

USE WITH DISCRETION.

ng.bufd.recycle_timeout_sec = *seconds* **[30]**
Time period to check for stale channel buffers that could be recycled (−1|0 = never).

ng.transfer_buffer_size = *bytes* **[1048576]**
Size of the intermediate buffer used to facilitate I/O. This setting is **unused** when cache buffers are memory−mapped.

ng.cli_write_delay.*
This section regulates delaying data output to reduce the number of **write(2)** syscalls. Data is accumulated until the saved portion is large enough.

ng.cli_write_delay.enabled = *true*|*false* **[true]**
Write delays are enabled if set to **true**.

ng.cli_write_delay.timeout_ms = *delay_ms* **[100]**
Delay for no more than *N* milliseconds.

ng.cli_write_delay.max_buffered = *max_bytes* **[1048576]**
Delay up to *max_bytes* of data, disregard if 0.

ng.psensors.*
Performance sensors allow to measure resource utilization between two specific points within the application, using the metrics provided by **utime**(2) **utime**(2) call at each end of the sensor. All sensor data will be printed out at the application exit in the format similar to the output of **time(1)** utility.

Performance sensors are a debugging/profiling facility and incur additional load on the system.

USE WITH DISCRETION.

Defined sensors:
app = application runtime; **ev_loop** = event processing (all events); **ev_read** = reading/processing inbound data; **ev_write** = writing/processing outbound data; **ev_err** = processing error events; **ev_pp** = post−processing events; **ng_chaio** = channel data I/O; **ng_clio** = client data I/O.

ng.psensors.enable_all = *true*|*false* **[false]**
Enables all sensors if true, disables all otherwise. This is to initialize the set of enabled−sensor flags to either all ones (if enabled) or all zeros. This setting is to be used in combination with **ng.psensors.except**.

ng.psensors.except = *sensor_list* **[]**
Enables sensors in the list if **ng.psensors.enable_all** is *true*, or disables those sensors if *false*. This way *enable_all* is used to initialize the set of sensors while *except* narrows it down by enabling/disabling its specific elements.

EXAMPLE A:

ng.psensors.enable_all = true; # Enable all sensors.

ws.psensors.except = ["ev_read", "ev_write"]; # Disable those listed herein.

Enables all sensors except *ev_read* and *ev_write*.

EXAMPLE B:

ng.psensors.enable_all = false; # Disable all sensors.

ng.psensors.except = ["ev_read", "ev_write"]; # Enable those listed herein.

Enables *ev_read* and *ev_write* sensors, all others are disabled.

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gigapxy(1),**gws**(1),**gng**(1),**gws.conf**(5),**channels.conf**(5),**gigapxy.auth**(5),**mlockall**(2)