NAME

GigA+ − IPTV−stream delivery software.

DESCRIPTION

GigA+ (giga−plus) is a software to deliver video streams from content−providers to clients (STBs, smart TVs, video players, etc.). GigA+ can deliver video as linear streams − via *HTTP*, or as segments − via *HLS*. For **HLS**, a third−party HTTP server (such as *NginX*) may be used, otherwise it's done by **gxng**(1) module.

GigA+ is a descendant of **Gigapxy**, which is centered on *linear* delivery and has two modules: **gws** and **gng**. **GigA+** includes similar modules, named **gxws** and **gxng**.

Basic terminology and use cases

GigA+ uses the term *channel* for a data source and *client* for a destination.

GigA+ feeds data from *M* multicast channels to *N* clients (where N >= M). A client is an application that issued an appropriate HTTP request to a GigA+ module.

GigA+ is designed to serve as many clients, as possible, efficiently and economically. For *linear* delivery, a built-in *cache* allows new clients to start reading cached (channel) data at once, minimizing the delay associated with switching between IPTV programs. For the end user this means that changing IPTV channels is very fast.

In case of **HLS** delivery, **GigA+** allows to distribute the load to multiple servers using a load−balancing module. **DVR** could be set up for any of HLS−enabled channels to allow delayed playback.

APPLICATION MODULES

GigA+ modules could be divided into three categories: **preparation**, **delivery** and **load−balancing**. For *linear* streaming, all modules are about **delivery**. For *HLS* delivery, a linear source stream must be first *prepared*: divided into segments and segments mapped to playlist items. Load−balancing modules ensure that data segments are available on multiple hosts and that requests get distributed between those hosts.

gxws (Web service)

processes and validates a user request, sets up input and output ends of the associated data streams, then dispatches the request to the appropriate engine: a **gxng** process. In case of **HLS**, the request is either for a *playlist* or for a *data segment*. For a *playlist*, **gxws** reaches out to the *playlist manager* module (**gxpm**) and passes the playlist to the client. For a *data segment*, **gxws** passes the request to a **gxng** engine, to delver the content. **gxws** controls *N* **gxng** instances, where *N* <= *number of cores*. This allows to distribute CPU load evenly.

gxng (Delivery engine)

processes requests from **gxws** and delivers data streams to clients. Each **gxng** has a controlling **gxws** to which it *attaches* via a designated UNIX socket. Most of the CPU load falls on **gxng**s, so there are usually multiple instances attached to the controlling **gxws**, pinned to different CPU cores. **gxng** reports client−related events it is handling; the reports allow **gxws** to track data streams and load−balance between multiple engines. **gxng** re gularly updates **traffic performance statistics (TPS)** for **gxws** traffic reports.

vsm (HLS Stream Manager)

is a stream−preparation module for **HLS** channels. It takes care of partitioning the input (linear) stream into segments and notifying all other modules of its actions. Each **vsm** instance handles a

distinct channel, all required parameters **vsm** obtains from a *channel spec* − a human−readable configuration file. **vsm**, however, is just a front: it invokes two other video−preparation modules to perform its function. **vsm** also takes care of restarting relevant modules and ensuring that feed interruptions and stream errors are handled correctly.

gxseg (HLS Stream Segmenter)
splits input stream into segments. It notifies the *playlist manager* on every split, to account for every segment. This module is launched and run by **vsm** and is never called directly.

wux (Message proxy)
facilitates message passing from **gxseg** to the *playlist manager*. **wux** maintains a *segment manifest* to prevent data loss in case of a crash. This module is launched and run by **vsm** and is never called directly.

gxpm (HLS playlist manager)
is responsible for generation of playlists. **vsm** produces meta−data for **gxpm**: it supplies playlist−item info as items get generated by **gxseg**. **gxws** relays playlists from **gxpm** to the clients. For load−balancing, **gxpm** also sends notifications (one per item) to **dwg**(1) *download agents*.

dwg (Download agent)
listens for *segment−ready* notifications from **gxpm** and fetches segments from remote severs.

flb (FastCGI load balancer)
is a web−server module to distribute segment requests across a set of hosts. It could use *round−robin* or *min/max* criteria for distribution. **flb** redirects requests via HTTP 302 to the servers picked by the distribution algorithm.

STREAMING MODES

GigA+ supports two streaming modes: *linear* (source stream is served is a continuous manner) and *HLS* (stream is served as a sequence of data segments, compliant with Apple *HLS* protocol).

Linear streaming

gxws and **gxng** are the key modules when a request for linear streaming is made. A request is initially received by **gxws**, which (after authentication) connects to the source stream and then delegates further I/O to one of the *attached* **gxng** engines.

A request for linear streaming may look like the following:

http://acme.com:8080/src/udp://224.0.2.26:5959/dst/?key=493f064567

For a detailed look at *request formats* please refer to **gxws**(1) manpage.

In this case the **gxws** parses the above request and subscribes (unless already subscribed) to the requested multicast channel 224.0.2.26:5959. Then it passes both connections (source **A = 224.0.2.26:5959**, destination **B = request socket**) to the selected **gxng** instance. **gxng** proceeds with relaying data from A to B until either connection is shut down.

Single−server HLS streaming
is the mode when content is delivered via **HLS**, with all data segments on a *single server*. An*HLS−playlist* request comes to **gxws**. Such a request may look like the following:

http://data-host1:4046/hls-m3u/showtime/playlist.m3u8

For a detailed look at *request formats* please refer to **gxws**(1) manpage.

In this case **gxws** identifies this as a request for a **LIVE** playlist. **gxws** requests the playlist from **gxpm** using **showtime** as the channel identifier. For **gxpm** to be able to produce a LIVE playlist for **showtime**, the channel's **vsm** should be up and running, supplying **gxpm** with metadata on generated segments. **gxpm** responds to the request with the HLS playlist containing links (URLs) to data segments (in the **LIVE**−feed window). A data−segment URL looks like the following:

http://data-host1:4046/hls-fra/showtime/23192767.ts

This links back to the **gxws** (using the same port and *hls−fra* prefix), so the segment (23192767.ts) is to be fetched via **gxng**. Once **gxws** recveives the request, it passes it on to a **gxng** engine to deliver the segment to the client. In another scenario, there could be a different link for this segment:

http://data-host1:8086/gpx-seg/showtime/23192767.ts

In this case, a third−party web server has been set up to listen on *port 8086* and serve files from */gpx−seg*. That web server is an independent component responsible for segment delivery. No **gxng** is needed and **gxws** could operate without a single **gxng** attached (if only HLS requests were to be served).

Multi−server HLS streaming
is the mode when data segments are replicated across N > 1 participating servers. In this scenario,**gxpm** sends segment−ready notifications to a multicast channel and **dwg**(1) download agent reads the notifications and downloads the respective segments.

gxpm is set up to generate segment URLs suitable for load balancing. The balancing is done using **flb**(1) module. **flb**, paired with a (third−party) web server, load−balances requests by picking servers from the pool and re−directing inbound data−segment requests accordingly, via HTTP 302. A data−segment URL may look like the following:

http://lb-host:5089/gpx-seg/showtime/23192767.ts

flb listens on **lb-host:5089**, with hosts in the pool ranging from *data-host1* to *data-host8*. It picks *data-host4* for this particular request and re-direct to:

http://data-host4:8086/gpx-seg/showtime/23192767.ts

where a third−party webserver (or **gxws**) services it. **flb** is NO T a stand−alone server but a FastCGI module. For now, **flb** has been tested and is guaranteed to function currectly with *NginX* 1.4+.

For advanced load balancing (beyond round−robin), *Redis* NoSQL database must run on one of the servers. The database consolidates various metrics across the server pool, allowing **flb** to use *mini−max* algorithms based on custom *load sensors*. For further details on setting up load balancing and configuring mini−max,

one should refer to **gxa-lb-setup**(5) manpage.

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gxws(1),**gxng**(1),**flb**(1),**dwg**(1),**gxpm**(1),**gxseg**(1),**wux**(1),**vsm**(1),**gxa-lb-setup**(7)

NAME

gxws − GigA+ web service daemon.

SYNOPSIS

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

DESCRIPTION

gxws is the front−end module of GigA+. It handles user and administrative requests submitted via HTTP protocol. The format of requests is described in the **USER REQUESTS** and **ADMIN REQUESTS** sections of this page.

gxws dispatches (certain) user requests to GigA+ engines, instances of the **gxng**(1) daemon. At least one **gxng** instance should be running for GigA+ to accept requests for data (except for HLS−segment requests when a third−party web server delivers the segments). A single**gxws** instance can control up to 64 engines.

gxws takes its parameters from a *configuration* file, which is either *gxws.conf* or *gigaplus.conf* by default, and can contain sections for any or all GigA+ modules. **gxws** 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**gxws** are described in detail in the **CONFIGURATION** section of this page.

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

OPTIONS

gxws 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 **gxws** 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, **gxws** will **mirror** diagnostic messages sent to the log (as specified with the −l option) to standard output.

−k,−−oldmcast
use legacy multicast API. **gxws** 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 **gxws** 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.

−K, −−syskey
generate system key (to use in licensing) and exit.

USER REQUESTS

gxws(1) supports two categories of streaming: *linear* and *HLS*, and has distinct formats for either category.

Linear−streaming requests

The formats for linear streaming are:

a) *http://{addr}:{gxws_port}/{cmd}/{mcast−addr}:{mcast−port}*

WHERE
{addr}:{gxws_port} ::= IPv4/6 address of the user-request listener;
{cmd} ::= udp;
{mcast-addr}:{mcast-port} ::= IPv4/6 address of the mulitcast group;

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}:{gxws_port}/src/{channel−uri}/dst/{client−uri}*

WHERE
{addr}:{gxws_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}:{gxws_port}/${alias}*

This type of request uses a *channel alias*: a dollar−sign prefixed name that resolves to a URL for a channel. 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*

gxws(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 GigA+'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 -> GigA+ -> 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 src−dst format and appends the *key* parameter to the URL the alias resolves to.

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

HLS−streaming requests

Subdivide into two types:

a) *Playlist requests*

provide clients with the meta−data on the **data segments** to play back. The format of a playlist (M3U8) request is as below:

http://{addr}:{port}/hls-m3u/${channel−tag}/playlist.m3u8[?utime={unix−time}]

WHERE:
hls−m3u is the identifying tag/keyword for the request type;
channel−tag is a symbolic tag, identifying the channel (**not** an alias from *channels.conf*);
unix−time (optional) is a UNIX timestamp for the start of the transmission (**DVR** mode).

Example (LIVE): http://acme.tv:5046/hls-m3u/tv5monde/playlist.m3u8

Example (DVR): http://acme.tv:5046/hls-m3u/tv5monde/playlist.m3u8?utime=1499069128

For *DVR* user supplies UNIX time, if data goes back as far as specified, the transmission will begin from that moment (approximately).

b) *Segment requests*

provide actual data segments to the clients. **gxpm(1)** must be configured to pro vide the URLs referencing **gxws**. The format of the URLs is as below:

http://{addr}:{port}/hls-fra/${channel−path}/{anyname}.ts

WHERE:
hls−fra is the identifying tag/keyword for the request type;
channel−path is a directory path, relative to *hls.data_root* (see **CONFIGURATION** section);
anyname is the file name (w/o the *.ts* extension), which could be of any format compatible with filename guidelines for the OS.

Example: http://acme.tv:5046/hls-fra/vol3/2017-07-03/60706956.ts

Please note that the particular structure of the *channel−path* is heavily dependent on the settings in the *channel specification* for **vsm**(1) and the settings for the **gxpm**(1) module.

On receipt of a *segment request*, **gxws** parses, validates it (assuring, for instance, that the referenced file exists) and relays the request to one of the attached **gxng** instances, using the same load−balancing method it would use for any other type of requests. **gxng** transmits the contents of the file to the client using the original connection (passed to it by **gxws**).

NOTE also that a channel could be set up to deliver segments via a *third−party* web server. In that case, no requests for segments arrive at **gxws** on behalf of that channel.

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. gxws would reply with **HTTP 302** (Moved Temporarily) in the hope that the client software recognizes the code and would follow the re−direction link. **gxws** 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. **gxws** 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 **gxng**. Re−direction, however, is still performed as appropriate.

ADMIN REQUESTS

gxws(1) listens on dedicated TCP ports for administrative requests. The request types are as below:

TPS reports

TPS (traffic, tps) - throughput statistics on active channels and clients. The request format is as below:

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

WHERE:
{type} ::= traffic|tps
{format} ::= html|web|xml

The output formats are:
HTML (html, web) - output as an HTML/web page.
XML (xml) - output as an XML page.

The default format is *html*. **Note:** throughput statistics should be enabled in appropriate config settings.

Example: http://acme.tv:4047/report?type=tps

Report caching

gxws(1) may cache a report for a certain time period, defined as **ws.report.cache_timeout_ms** in the configuration. The request may request invalidation of the cache by using *cached=0* parameter in the URL. **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.

Example: http://acme.tv:4047/report?type=tps&format=xml&cached=1

Drop channel/client

is to 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 channel).

NB: 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 **gxws** 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.

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

Ping/status

is to ping or get status: *http://{addr}:{port}/ping* or *http://{addr}:{port}/status*; **status** keyword is supported to comply with the *udpxy* status command, which is **NOT** equivalent to **ping**. Nevertheless, udpxy users used to issue a **status** request to check if the service was up. For GigA+ one should use **ping**. **gxws** returns **HTTP 200** whenever it receives either of the two commands.

Disconnect all (reset)

disconnects all clients and channels: *http://{addr}:{port}/reset* − this will have **gxws** send **SIGUSR2** to all attached **gxng** instances. **SIGUSR2** directs a **gxng** to drop all its channels and clients.

AUTHORIZATION

GigA+ utilizes *authorization helpers* − user−supplied components − communicating with **gxws**(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 (for FreeBSD use */usr/local* prefix):

/usr/share/gigaplus/scripts/gauth.sh

An authorization request is a text string terminated by *CR/LF*.

Example:

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

Since **gxws** does not have any guarantee that a helper would not block on a request, it **times out** auth requests (please see **CONFIGURATION** section for particular settings). If a request times out on an authorization task, the respective 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. **gxws**(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.

For further details on authorization protocols and other relevant information, please refer to **gxws.auth**(5) page.

CONFIGURATION

gxws(1) reads configuration from a file, either gxws.conf or gigaplus.conf, by default. After reading and validating the config it applies whatever changes come from the command−line.

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

All **gxws**(1) settings begin with the **ws.** prefix, as in *ws.section.param*. Therefore, what's referenced below as, for instance *aa.bb*, should be *ws.aa.bb* in the config file. A configuration file could contain non−ws settings too; **gxws** will ignore 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**.

ng.*
is the section defining communications between **gxws**(1) and **gxng**(1)

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

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

ng.pick_method = *method* **[round-robin]**
gxng 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.

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

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

log.*
Below are the settings pertaining to different modules within **gxws**(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.

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

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

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

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

log.time_format = local| gmt| raw| raw_mono| no_time| [local]
Sets format to display timestamps for log entries. *local* will log local−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.

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

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

access_log.*
The following settings are for **gxws** 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*
gxws 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*
gxws 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 **gxng**).

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 **gxng**(1) attached/detached/quit to/from **gxws**(1). Shown are: *gxng* pid, internal index and connection fd. **NG_QUIT** means that gxng 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.

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

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.

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.

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

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

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)

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.

listener.*
The following are the settings equally applying to [up to 16] listeners of the two types of requests (admin and user) handled by the application. See *gigaplus−commented.conf* for an example of multiple−listener config.

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*.

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.

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

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.

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

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

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

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.

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.

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

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

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

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*.

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

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

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, **gigaplus** does NOT support parsing HTTP chunks in video streams.

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.

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

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

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).

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

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.

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.

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.

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

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

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

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

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

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 **gxws**, which will therefore **wait** and suffer the associated latency penalty.

USE WITH DISCRETION.

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

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.

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

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

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.

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.

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.

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

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

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

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

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

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

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).

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

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 (*tput_stats.enabled* is **true**).

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 (*tput_stats.enabled* is **true**).

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 gigaplus). Channel URL must be a full HTTP URL that will be returned to client via HTTP 302 response.

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.

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.).

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 **psensors.except**.

psensors.except = *sensor_list* **[]**
Enables sensors in the list if **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:

psensors.enable_all = true; # Enable all sensors.

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

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

EXAMPLE B:

psensors.enable_all = false; # Disable all sensors.

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

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

auth.*

Authorization helpers are user-defined applications (plug−ins) used by **gxws** to screen user requests, based on request−specific data, such as user address, request URI, etc. **gxws** 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/gigaplus/scrpts* under Linux (*/usr/local/share/*.. under FreeBSD).

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

Enable helpers unless false.

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

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

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 **gxauth**(5)

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 *run_as** settings.

auth.min_helpers = *count* **[1]**

Number of helpers to start with and always keep running.

auth.max_helpers = *count* **[1]**

Maximum number of helpers to run.

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.

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, **gxws** disables further helper launches for the configured time period.

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

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

auth.allow_custom_urls = *true|false* **[false]**
Instructs **gxws** using **B2P protocol** to allow URLs that do not follow the two **gigaplus-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*.

auth.cache.*
Negative authentication responses can be cached by gx 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.

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

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.

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

hls.*
section is responsible for HLS−specific parameters.

hls.enabled = true|false [false]
allows HLS streaming requests, if set to true. Otherwise, further parameters in this section are ignored.

hls.data_root = *path* **[]**
absolute path to the root directory for segment data files. **gxws**(1) uses this path in conjunction with the *channel−path* in the *segment* (i.e. **hls−fra**) requests to compile full path to the requested segment file. For instance, if the request is

http://acme.tv:5046/hls-fra/vol3/2017-07-03/60706956.ts then, with *hls.data_root* = /opt/data, the full path to 60706956.ts would be: /opt/data/vol3/2017-07-03/60706956.ts. **gxws** uses the path to check whether the file is accessible.

hls.request_timeout_ms = *num* **[1000]**
maximum time for **gxws** to process an HLS request. If I/O for such a request exceeds the limit, the request is terminated.

HLS requests may need a different timeout since their control flow differs. The default value is assumed to be sufficient, but may be adjusted according to the actual pattern. If set to 0, the timeout for HLS does not differ from the one for linear−streaming requests (user_request_timeout_ms).

hls.m3u8_tmout_sec = *num* **[30]**
time (in seconds), **gxws** keeps a cached *channel record* for an HLS playlist (M3U8) request. When a client requests a LIVE playlist, it usually means that the request is to be repeated when all playlist's items have been played back. For that purpose **gxws** caches the record created for the initial request for *num* seconds, after that the record gets deleted. (See *hls.gpm_socket*.) If playlists normally last longer/shorted than the default time, you might consider altering the value.

hls.fast_urq_len = *N−bytes* **[0]**
number of bytes in a user request after which **gxws** should attempt to process the received data, even if the request is incomplete.

Some HLS clients (**vlc** one of them) send HLS M3U8 requests in little data portions making significant pauses in between (beats me **why** they do it, but the fact remains). Often the most significant part, containing the requested URL, is received first, with other (less important) headers to follow. If **gxws** is to wait for the full request (i.e. till CRLF,CRLF), it would often take too long, while the needed part is there from the very beginning. This setting lets the logic push its luck and try to process the data as soon as we have *N* bytes. With **vlc**, in my experience, this noticeably expedites the process. Yet, **USE AT YOUR OWN RISK.**

hls.gpm_socket = *path* **[]**
full path to the user−request socket/listener of **gxpm**(1)

Example: hls.gpm_socket = "/tmp/gpm-Y.sock"

NOTE: **gxws**(1) requests a playlist from **gxpm**(1) every time a new HLS *channel* is created. It the stores the path to the playlist in the channel record and services further requests for the same playlist from the given file (which it expects **gxpm** to update). **gxws** also updates a special *lock file* (see lid.* section in **gxpm** config) for **gxpm** to know that the playlist is being used.

hls.snooze_http11 = true|false [false]
permits to 'snooze' an HLS−client connection after a successful request, instead of terminating.

If **true**, HTTP/1.1 keep−alive connections are not closed on **gxws**'s end; **gxws** waits (for N ms) for another request to arrive or the connection to be terminated by the peer.

hls.conn_reuse_sec = *N−sec* **[20]**
number of seconds gxws(1) would wait for a 'snoozed' HTTP/1.1 (idle) connection to be re-used (i.e. for a new request on the socket).

hls.force_close = true|false [false]
close **all** HTTP/1.1 connections at the end of a (user) request, regardless of the *keep−alive* HTTP setting.

AUTHORS
Pavel V. Cherenkov

SEE ALSO
gigaplus(1),**gxng**(1),**gxpm**(1),**vsm**(1),**gxws.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

Below is an example of a channel configuration (supplied as a file in the package):

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

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gxws(1)

NAME

gxws.auth − GigA+ authentication manual.

DESCRIPTION

gxws 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). **gxws** 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 **gxws**)

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 **gxws.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. **gxws.conf** pro vides *ws.auth.b_fields* setting to specify what information gxws 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 **gxws** 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 **gxws** 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 **gxws** know what the endpoints are.

Here's an example scenario:

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

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

gxws 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

gxws 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

gigaplus(1),**gxws**(1),**gxng**(1)

NAME

gxng − GigA+ streaming engine daemon.

SYNOPSIS

gxng [−h?TvVq] OPTIONS

DESCRIPTION

gxng is the GigA+ **engine** module performing I/O on behalf of data requests submitted to **gxws**. The format of **gxws** requests is described in the **gxws**(1) manpage.

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

gxng takes its parameters from a *configuration* file, which is either *gxconf* or *gigaplus.conf* by default and can contain sections for any or all GigA+ modules. **gxng** 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**gxng** are described in detail in the **CONFIGURATION** section of this page.

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

OPTIONS

gxng 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, −−gxws *path*
specify path to the controlling **gxws** (domain socket).

−T, −−term
run as a terminal (non-daemon) application. This is the default behavior when **gxws** 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, **gxng** 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 **gxng** process to the given CPU/core (numbered from *0* **to** *N*−**1).**

−K, −−syskey
generate system key (to use in licensing) and exit.

CONFIGURATION

gxng(1) reads configuration from either gxng.conf or gigaplus.conf (in current directory, /etc or /usr/local/etc), unless −C option was used. After reading and validating the config it applies whatever changes come from the command−line.

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

All **gxng**(1) settings begin with the **ng.** prefix, as in *section.param*. Therefore, what's referenced below as, for instance *aa.bb*, should be *ng.aa.bb* in the config file. A configuration file could contain non−ng settings too; **gxng** will ignore 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**.

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

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

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

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

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

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.

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

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

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

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

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. **gxng** will set the idle clock to the minimum value of a channel/client timeout.

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

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*.

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

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

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).

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, **gxng** would consider RTP packets as non-TS and relay them AS-IS.

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).

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

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

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.

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.

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

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

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

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

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

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.

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

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

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

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

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

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

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

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.

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

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

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

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

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.

ws.*
section decribes the parameters of communication between **gxng** and **gxws**.

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

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

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

consecutive segments for traffic data.

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

USE WITH DISCRETION.

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

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.

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.

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

The following three settings affect the way **gxng** 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.

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; **gxng** would recycle a buffer only if, after its removal, the cache would still have >= *seconds* worth of data.

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. **gxng** 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 **gxng** would use to pick one.

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 *bufd.min_total_duration_sec* and *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.

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)** gxng scans a channel looking for a bubble (and warns if it finds one), in mode **2 (burst)** it also tries to invalidate the leftmost A−buffer and release the underlying U−sequence (the bubble).

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

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

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.

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.

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*.

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.

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

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.

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.

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

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

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

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.

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 **psensors.except**.

psensors.except = *sensor_list* **[]**
Enables sensors in the list if **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:

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:

psensors.enable_all = false; # Disable all sensors.

psensors.except = ["ev_read", "ev_write"];

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

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gigaplus(1),**gxws**(1)

NAME

gxpm is a playlist manager for **GigA+**.

SYNOPSIS

gxpm −Y ipath −S spath −C config [OPTIONS]

DESCRIPTION

gxpm(1) compiles playlists (in **M3U8** format) in response to requests by **gxws** instances for various *channels* (sources). Channel meta−data arrives from **vsm** (via **gxseg**) to the **gxpm**'s *source* socket (*spath*). From the *item* socket (*ipath*) **gxpm(1)** responds to requests from **gxws**: assembles and relays playlists.

PARAMETERS

The following parameters are accepted:

−Y, −−items *ipath*
specifies path to the **item** UNIX socket, servicing **gxws** requests.

−S, −−sources *spath*
specifies path to the **source** UNIX socket, ingesting channel meta−data from **vsm** instances.

−C, −−config *path*
configuration file (*gxpm.conf* by default).

OPTIONS

The following options are accepted:

−D, −−appdata *path*
path to playlist data (*/tmp* by default). This is the root for item data, one level above the directory for the particular channel. **gxpm** uses this path as a base for channel−related data. So, if **vsm**s use a common data root for all channels, this would be it. Ignored if *channel−specific* directories are reported by **vsm** (that is, if **cha_ROOT** is in the *channel spec*). Please see **vsm**(1) for details.

−F, −−prefix *URL*
URL prefix for all playlist items, unless a channel specifies custom prefix by *cha_PFX* in the *spec*. Please refer to**vsm**(1) for details.

−l, −−logfile *path*
path to log file. Please do mind access permissons for the directory and kindly take −−runas option (see below) into account.

−L, −−level *crit|err|norm|warn|info|debug*
log level. The default level is *info*, unless specified otherwise in config.

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

−u, −−runas *username*
user to run as if started as a daemon. Starts as *root* (in daemon mode) by default. **NB**: kindly **avoid** running **gxpm(1)** as root.

−T, −−term
run as a terminal (non-daemon) application. This is the default behavior when 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, −−version
output application's version and quit.

−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. **NB**: this option is **depr ecated**, please use **−l, −−logfile** *path* instead.

−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, the module will **mirror** diagnostic messages sent to the log (as specified with the −l option) to standard output.

−h, −−help, −?, −−options
output brief option guide. This is output when run without parameters.

−K, −−syskey
generate system key (to use in licensing) and exit.

CONFIGURATION

gxpm(1) uses a configuration file for most settings, the path to the file must be specified at command line. The module starts by initializing config setting to default values, then loading the settings it finds in the config file (thus, overriding the defaults) and then overriding config values with those specified from the command line. An example *gpm.conf* is supplied with the package.

All config settings have **gpm.** prefix (no *x*), therefore, for instance, setting *A* fully reads in config as *gpm.A*. The configuration settings are as below:

src_socket = *path* **[]**
path to the **source** UNIX socket, ingesting channel meta−data from **vsm** instances.

pl_socket = *path* **[]**
path to the **item** UNIX socket, servicing **gxws** requests.

log.*
log file settings:

file path to the log file.

level log level: *crit*|*err*|*norm*|*warn*|*info*|*debug*

max_size_mb = *N*
maximum log size in MB (1MB = 1048576). Every time a log file reaches above this size it is renamed as an archive and a fresh log is started.

max_files = *n*
maximum number of log files before rotation. Every time the number of archive logs exceeds this number, the oldest archive is removed.

runtime.*
settings regulating how the application starts/runs.

pidfile []
path to the pidfile (no pidfile unless specified).

run_as_user *username* **[]**
run as *username* if started as a daemon. Running as root by default.

non_daemon = true|false [false]
run as a terminal (non-daemon) application. By default, an instance started under root runs as a daemon.

data_dir = *directory* **[/tmp]**
path to playlist data (*/tmp* by default). This is the root for item data, one level above the directory for the particular channel. See −D option (above) for details.

item_url_prefix = *URL* **[]**
string to prepend every playlist item URL with. This could be used if all sources use a common data root and, therefore, setting *cha_PFX* for each channel becomes redundant.

max_sources = *N* **[256]**
how many channels/sources shall one instance handle at the max? [1..1024]

max_src_tasks = *N* **[256]**
how many playlists/tasks can there be per single channel/source? [1..512] It actually depends. You need only **one** task for **all** live−playlist requests, but every *DVR* request might as well be unique. Use your own judgement.

max_playback_time = *sec* **[604800] = 1 week**
how many **seconds** worth of data should be stored per source/channel? [600..5184000] Usually, it also depends on a channel and could be set via *cha_CAPACITY* in **vsm** channel spec.

max_live_items = *N* **[6]**
maximum items per LIVE playlist. [4..64]

src_tmout_sec = *sec [30]*
time out sources (with no tasks/playlists) after N seconds of no input. [1..300]

task_tmout_sec = *sec [20]*
discard playlist/task if untouched for N seconds [1..100]. **gxpm** creates an *access−tracking* (.alk) file per playlist/task that it expects **gxws** to modify each time it reads the particular playlist. **gxpm** checks the modification timestamp regularly and **expires** the playlist if last modification time is more thant *N* seconds behind.

gzip_playlists = true|false [true]
generate compressed LIVE playlists (EVENT, VOD are always gzipped) **gzip**(1)

verify_item_filesize = true|false [true]
verify that reported segment files exist and have the declared size. **vsm** reports both paths to segments and their size to **gxpm**. This option makes sure the information is correct.

use_sid_playlist_dir = true|false [false]
place generated playlists into a subdirectory, named after the *cha_ID* of the source. **vsm** passes *cha_ID* and *cha_ROOT* from the channel spec. If the above parameter is **true**, then the playlist path uses *cha_ID* as a subdirectory under the root and places all channel's playlists there. Otherwise, playlists are placed into the *cha_ROOT* and **prefixed** with *cha_ID*. For instance, let *cha_ID* = 'CNN' and *cha_ROOT* = '/opt/channel'. Then, with **true**, a path to a playlists for the channel may be: */opt/channel/CNN/playlist.m3u8*; otherwise it would be */opt/channel/CNN-playlist.m3u8*.

refresh_playlists = true|false [false]
refresh each dynamic playlist (LIVE, EVENT) on every change affecting it. If set to false (default), a playlist is refreshed only when user requests a corresponding task.

md_report.*
settings to report segment meta−data to **dwg**(1) via multicast. **gxpm** publishes segment meta−data messages to a designated multicast group. Subscribers within the (multicast) network can extract segment URL's from these messages and download segments.

enabled = true|false [false]
no meta−data sent (and further parameters in this section ignored) unless **true**.

pub_url = *multicast−URL* **[]**
URL of the multicast group to publish to. Example: udp://227.3.2.160:2020

lid.* = { *task−lock parameters* **}**
defines parameters for a *task lock* created by a recepient for each processed message. **dwg** needs the lock to avoid redundant downloads. If N > 1**d wg** instances run on a host, each of them subscribes to *pub_url* group, but only instance per task is required. **gxpm** encapsulates a *task−lock ID* into each message. Each **dwg** instance, upon receiving a message, tries to (atomically) create a *new* lock file. If the file is already there, the task is already taken and the message is skipped. The parameters are: **prefix** = *symbolic prefix*, **min** = *N* (N > 0), **max** = *M* (M > N). The number rolls over to *min* after reaching the *max*. Example: lid: { prefix = "h1"; min = 100; max = 299; };

src_base = *URL*
is the URL prefix to use for downloading. If a web server has been set up so that the data root directory for the segments is mapped to *http://acme.tv:8080/seg*, then this would be the value to use. A segment with the relative path of *CNN/20170627-11/6298994298.ts* should then map to a resolvable URL: *http://acme.tv:8080/seg/CNN/20170627-11/6298994298.ts*.

mcast_ifc = *interface*
interface name for the multicast network. Example: eth0

AUTHORS

Pavel V. Cherenkov

SEE ALSO

vsm(1),**gxseg**(1),**gxws**(1),**dwg**(1)

NAME

gxseg is a stream segmentation utility within **GigA+**.

SYNOPSIS

gxseg -i infile [OPTIONS]

DESCRIPTION

gxseg(1) is the tool for slicing a source stream into segments. **vsm**(1) calls it with the options defined by the *channel spec*. For historical reasons, it uses no config file, all options are selected at the command line. **gxseg** uses *libav* libraries from **ffmpeg**(1) and therefore inherits some of its traits. This tool is not to be called directly, **vsm**(1) calls it when needed. This page is strictly for reference and to promote understanding of all the tools within **GigA+**.

OUTPUT STREAM

gxseg outputs important metadata into *STDOUT*, all diagnostic and debugging messages go to *STDERR*. This is done so that the utility could be pipelined to other modules. In **GigA+** the module down the pipeline is **wux**(1) relaying STDOUT from **gxseg** to **gpm**(1) in an orderly fashion. For details on the output metadata, please refer to the **STANDARD OUTPUT FORMAT** section below.

PARAMETERS

The following mandatory parameters are accepted:

−i|−−source *infile*
URL for the source in **ffmpeg**(1) URL format.

−N|−−channelid *string*
unique identifier for the source channel. **gxseg** uses it to form the full path to the channel's data directory.

OPTIONS

The following options are accepted:

−o|−−outfile *path*|**copy []**
path to M3U8 playlist, <infile>.m3u8 if 'copy'. If *outfile* is specified, **gxseg** creates a **LIVE** M3U8 playlist at the specified path, refreshing it with every added item. If *copy* is given as the path, extension of the *infile* is replaced with **m3u8** and forms the resulting path for the playlist. If the parameter is omitted playlist generation is skipped.

−3|−−m3upfx *prefix*
URL prefix for M3U8−playlist paths. This is the prefix to put in front of every playlist item's path. For instance, with prefix=*http://acme.tv:4040/seg*, for *data/CNN/t34932001.ts* the playlist URL would be *http://acme.tv:4040/seg/data/CNN/t34932001.ts*.

−1|−−logstdout *path*
mirror *STDOUT* to the designated file.

−1|−−logstderr *path*
mirror *STDERR* to the designated file.

−S|−−storage *dirpath* **[.]**
base directory for segment files, current directory by default.

−R|−−relpaths *p1:p2:..:pK*
comma−separated paths to storage shards. For more detailed info on shards in GigA+, please refer to the **cha_SHARDS** parameter of the *channel spec* in **vsm**(1) documentation.

−G|−−granularity *mask*
granularity time-mask, in the **strftime**(2) format. For details, please see **cha_GRAN_MASK** parameter of the *channel spec* in the **vsm**(1) documentation.

−U|−−duration *sec* **[5]**
segment duration in seconds.

−M|−−mux *content−mask* **[vas]**
types of context to multiplex (include), where v=video, a=audio, s=subtitles.

−P|−−profile *name* **[]**
channel profile/quality tag. Presently unused.

−l|−−loglevel *num* **[1]**
ffmpeg(1) log level (for libav libs).

−m|−−maxseg *M* **[100]**
max number of segments per LIVE playlist *num=1..100*. After M is hit, the app removes the oldest item.

−J|−−joint *N* **[0]**
compile joint segments out of N < M regular ones. Presently unused.

−F|−−falseroot *path* **[]**
initial base directory, until synced. See **cha_FALSE_ROOT** parameter of the *channel spec* in **vsm**(1) documentation.

−W|−−markerdir *path* **[/tmp]**
directory for PTS−marker files. To conduct a *roll−over*, **gxseg** runs for a while as a *backup instance* and generates files containing PTS for the last−generated segment. This parameter designates the directory for such files. For details on the *roll−over* process, please refer to **cha_ROLLOVER** parameter of the *channel spec* in the **vsm**(1) documentation.

−u|−−pidscrid [false]
use PID, not *channelid* as *sourceID*. **gxseg** begins work with outputting a source−identifying message for the channel that would be segmented by this instance. If this parameter is **true**, the app would use process ID (*pid*) as the unique identifier for the channel, not the value set with the −N|−−channelid parameter.

−a|−−maxstore *sec* **[0 = none]**
define storage quota for the channel. Setting this parameter >0 adds the max_sec=*sec* parameter to the source−identifying message (for **vsm** to consume). The parameter tells **vsm** how much data is to be stored for this channel. See **cha_CAPACITY** parameter in the **vsm**(1) documentation.

−p|−−pidfile *path* **[false]**
create pidfile, quit if file or process already exists.

−d|−−alang *l1,l2,..* **[false]**
order audio tracks per language list. Presently unused.

−A|−−intmout *sec* **[10]**
exit if no input within N seconds.

−E|−−encrypt *spec−path* **[]**
use spec to write encryption tasks to STDOUT. Presently unused.

−T|−−thumbnail *seconds* **[off]**
generate JPG thumbnail every N seconds.

−t|−−maxthumbs *N* **[0]**
maximum number {0..99} of thumbnails to keep.

−g|−−png [false]
generate PNG thumbnails (instead of JPEG).

−e|−−errignore [10]
maximum number {-1..MAXINT} of non-critical I/O errors in a row to ignore; -1 = all.

−H|−−errbadtime [0]
maximum number {-1..MAXINT} of invalid PTS/DTS errors in a row to ignore; -1 = all.

−s|−−suspend [false]
await SIGUSR2 after start.

−k|−−keepseg [false]
keep all segments (do not rotate).

−X|−−exitonerr [false]
exit(3) on critical errors. If not set, **gxseg** repeatedly tries to re−open source stream and re−initialize the segmentation process after a critical error. (**NB:**: every re−initialization **LEAKS MEMORY** from **libav**.)

−D|−−dummy [false]
output nothing to files. Presently unused.

−I|−−interleave [false]
use interleaved I/O (libav) if set.

−Z|−−gzipm3u8 [false]
make GZIP’ed playlists.

−h, −−help, −?, −−options
output brief option guide. This is output when run without parameters.

−q|−−quiet [false]
minimum logging.

−K, −−syskey
generate system key (to use in licensing) and exit.

STANDARD OUTPUT FORMAT

gxseg(1) outputs metadata as single−line messages of the following types/formats:

SOURCE

is the first message going to STDOUT, specifying to the consumer (any app down the pipeline) the stream that this instance would be segmenting. The format is as below:

S*streamID* **init [~]***channel−tag* **data_dir=***root−dir* **[max_sec=***sec***] [item_pfx=***prefix***]**

where
streamID: either*c hannel−tag* (−N) or the process's *pid* (see −u);
init: message type;
~(tilde): added if this instance started as a backup (before roll−over);
channel−tag: specified by −N;
root−dir:= as specified by −S;
sec:= as specified by −a;
item−fpx:= as specified by −3.

Example: Sprime1 init prime1 data_dir=data

SEGMENT

is the message providing segment metadata. The format is as below:

SEGMENT: *path timestamp duration size num−id PTS*

where
path: relative path to the file;
timestamp: UNIX time for the **start** of the segment;
duration: duration in seconds;
size: of the file;
num−id: numeric ID for the encryption job (currently unused);
PTS: PTS time for the **start** of the segment;

Example: SEGMENT: prime1/7918931084.ts 1498837025 4.800000 6566088 0 7918931084

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gpm(1),**wux**(1),**vsm**(1),**ffmpeg**(1)

NAME

wux − STDIN to UNIX socket copy utility.

SYNOPSIS

wux [−r] [−m *nrec:rsize*] *unix−socket−path* [*manifest−path*]

DESCRIPTION

wux reads data from **STDIN** and writes it into a designated UNIX (stream) socket. **wux** is expecting the source to be in text format, it reads and writes one line at a time. Optionally, every processed line is added to a (text−based) manifest file. The manifest holds text lines in records of a fixed size, with a pre−defined cap on the record count. When the maximum number of records is reached, wux 'rotates' the manifest contents, deleting the oldest record.

wux is an integral part of GigA+ architecture. In GigA+ **wux** is used by **vsm**(1) to relay messages between **gxseg**(1) and **gpm**(1) modules. GigA+ users are not supposed to run **wux** directly (as it is run by other modules) but may want to know how it operates.

wux uses no configuration file, relying on command−line parameters.

PARAMETERS

wux accepts one mandatory parameter: the path to the destination UNIX socket. All data read from **STDIN** is to be written to that socket.

Path to the manifest file. Unless the path is specified, no data is written to the manifest.

OPTIONS

The options may be given in any order, before or after filenames. Options without an argument can be combined after a single dash.

−r expect response from the peer socket. Unless this option is specified, the application would not read any (response) output from the destination socket.

−m *mrec:rsize*
specifies two comma−separated parameters: maximum number of records in the manifest and manifest−record size (fixed). Both parameters **must** be specified if using the mainfest. The minimum record size is **64**.

ENVIRONMENT

INPATH variable can be set to the text file to replace **STDIN**.

AUTHORS

Pavel V. Cherenkov

SEE ALSO

gigaplus(1),**gpm**(1),**vsm**(1),**gxseg**(1)

NAME

vsm is a video−stream feed manager for **GigA+**.

SYNOPSIS

vsm [−l logfile] specfile [gxseg−options]

DESCRIPTION

vsm facilitates preparation of the inbound stream for serving via HLS. This involves segmenting, feeding meta−data, handling various notifications, errors and abnormal situations. **vsm** packs quite a bit of business logic. The module is a script invoking a number of GigA+ (and some third−party) utilities (**gxseg**, **wux**, **hos**, **prbsm**) to perform its duties. For configuration it uses a plain UNIX shell script, hereinfater referred to as *channel spec* (or simply the *spec*), assigning values to reserved environment variables. Each **vsm** instance is responsible for one (distinct) channel.

PARAMETERS

specfile is one mandatory parameter: path to the *channel spec*. **NB:** Use caution with what you put into the spec, for it is, in fact, executed by **vsm** and may affect your system as any other shell script could. **NEVER** run vsm (or spec) as **root**. Because you don't need to.

logfile parameter is optional and specifies path to the log where all output would go; or to *STDERR* if skipped.

gxseg−options parameter is optional. It allows to supply custom command−line options to **gxseg** module. Reserved for advanced customization of **vsm** behavior. Do**NO T** use unless you absolutely MUST.

CHANNEL SPEC

Is what **vsm** uses for configuration. A verbosely−commented example is supplied with the distribution and gets installed to

/usr/share/doc/gigaplus/examples/vsm-channel.spec

(**FreeBSD** users should consider */usr/local/share/*.. instead). You may use this example as a template for your own specs.

The following sections will present the variables set within the spec, essentially those are the configuration parameters.

cha_ID

name/id for the channel. It will be used within directory paths so the recommendation is to keep it simple and compliant with OS guidelines for directory names.

Example: cha_ID="cinemax"

cha_GPM

path to the UNIX socket connecting to the *source−listener* of GigA+ playlist manager (**gxpm**), which receives notifications on generations of each data segment via UNIX−socket. **vsm** establishes a connection to **gxpm** via **gxseg** and **wux**.

Example: cha_GPM="/tmp/gpm-S.sock"

cha_URL

URL (in ffmpeg−compliant format) for the stream source.

Example: cha_URL="udp://224.0.2.26:5050"

cha_ROOT
top−level root directory. This path should **NOT** include the name/ID of the channel, since *cha_ID* is to be used for that automatically by **vsm**. For instance, if the *cha_ROOT*="/opt/data" would make the channel's directory /opt/data/${cha_ID}.

Example: cha_ROOT="/opt/data"

cha_PFX
custom URL prefix to be used by **gxpm** for this channel's playlist URLs. A channel can specify its "own" prefix or rely on the common one in the **gxpm**'s configuration.

Example: cha_PFX="http://myown.tv:8181/seg/"

cha_SHARDS
lists colon−separated partitions/shards used to store segment data in. This parameter is **optional**, no partitioning will be applied if skipped. If specified, data would be evenly spread across shards. For instance, with cha_ROOT=*/opt/data* and cha_ID=*tv5* and cha_SHARDS="*vol1:vol2*" segments would be put into */opt/data/vol1/tv5* and */opt/data/vol2/tv5* in a round−robin fashion.

Example: cha_SHARDS="vol1:vol2:vol3"

cha_CAPACITY
defines how many *seconds* worth of channel data to store. This is the core setting for *DVR*, defining the size of the "time window" to afford. **vsm** passes this setting on to **gxpm** which will remove the "expired" segments in real time. At the start though, if **cha_HOS** (see below) specified, **vsm** will invoke **hos** utility and let it purge all "expired" segments.

Example: cha_CAPACITY=14400

cha_DURATION
is channel−specific duration of a single segment in seconds. This parameter is **optional**, the default value is **5**.

Example: cha_DURATION=6

cha_GRAN_MASK
specifies time−specific mask for a subdirectory within the channel's data storage. The format of the mask is per **strftime**(3) specification. If specified, each data segment is put into a subdirectory that is created based on the said mask. For instance, with cha_GRAN_MASK="*%Y%m%d−%H*", a segment generated on July 6th, 2017 at 3:56 pm will be placed into a subdirectory named *20170706-15*. If cha_ROOT=*/opt/data*, with cha_ID=*tv5*, the full path would become: */opt/data/tv5/20170706-15* (no shards) and */opt/data/vol1/tv5/20170706-15* for the first shard if cha_SHARDS="*vol1:vol2*".

Example: cha_GRAN_MASK='%Y%m%d-%H'

cha_HOS
specifies path to the clean−up utility that **vsm** runs when started. GigA+ supplies its own *hos* script for the purpose, yet it's up to you to replace it with a custom app. The supplied **hos** is responsible for keeping a channel's storage from swelling over the reasonable size, removing "expired" segments, rotating and archiving logs, etc. If *cha_HOS* is omitted, **no such cleaning** is done.

Please see **vsm−scripts**(1) documentation for details.

Example: cha_HOS="hos"

cha_MITEMS

specifies the number of items to keep in the *mainfest* that **wux** will maintain for the channel. A *channel manifest* is just a list of segments recently added. **gxpm** is aware of manifests and tries to load one up when a channel joins in. This is done to "catch up" with the channel that had some down time or crashed and stayed offline for a while. **Manifest** is also the way for **gxpm** to become aware of channel segments that were generated while **gxpm** itself was down (for some reason). The parameter sets the number of items allowed in the manifest, which is rotated (by **wux**) on the FIFO basis. If *cha_MITEMS* is omitted, **there will be no manifest** for the channel.

Example: cha_MITEMS=2800

cha_MAX_CERR

specifies the number of (critical) stream errors to tolerate. Some streams exhibit anomalies, causing **libav** I/O routines to bail out (with EOD indication or else). This parameter allows to re−start the I/O loop *N* times before exiting the app. **vsm** sets this parameter to **5 (five)** by default. The value of **−1** means ignore all I/O errors.

NB: Ignoring I/O errors (with **libav** specifically) comes with a price: memory leaks. The more errors ignored, the more memory wasted. Use your judgement to set this parameter to an optimal value for your channel(s).

Example: cha_MAX_CERR=5

cha_ERR_BADTIME

specifies the maxumum number of **PTS/DTS**−specific errors tolerated by **gxseg**(1) while processing a stream. The essense or the error is that *libav* API receives **invalid** PTS/DTS (presentation/display time-stamps within an MPEG−TS packet). In the **gxseg** log it is manifested as:

pkt: rc=-28 stream0 buf=0x0/0 pts/dts=-9223372036854775808/-9223372036854775808

cha_LMARK_DISKSIZE

specifies channel size in MB (total size of all segment and meta−data files) at full capacity (after cha_CAPACITY seconds passed). **Note**: hos won't archive segments until this size is reached.

cha_HMARK_DISKSIZE

specifies maximum allowable size (in MB) for a channel. **hos** will try to stop **vsm** if this size is exceeded.

cha_ROLLOVER

sets the number of *seconds* between two consecutive *roll−overs*. One instance of**vsm** e xits and the control "rolls over" to another one (recently started) for this particular channel. This is done to abate the inner deficiency of ffmpeg's *libav* libraries that (historically, for some reason) just cannot read a stream w/o interruption 24/7 and begin to "choke" after a while. Thus, the "old" **vsm** exits (retires) and the new instance takes over the job. The parameter regulates how often that process repeats. If omitted, **no roll−overs** get performed on the channel.

An alternate way to do *roll−overs* is by (supplied) **force−rollover.sh** script that should end up in */usr/share/gigaplus/scripts* (for Linux, */usr/local/share/...* under FreeBSD). Schedule script runs via **crontab**(1) to get roll−overs happen when you wish and as often as needed.

Example: cha_ROLLOVER=7200

cha_FALSE_ROOT
specifies full path to the directory, where the "new" instance of **vsm** would store its data before the "roll−over". There's a certain time period between the launch of the new instance and the moment that new instance can take over the job of the "old" process. During that period, both **vsm** instances run, but only the "old" one supplies data segments (to **gxpm**). Until the "old" one is done, the new one has to store its segments somewhere too. The last segment within that period that the "new" instance generates in then symlinked to the regular channel−storage directory (implying that the cha_FALSE_ROOT directory cannot be quite simply purged after a "roll−over").

Example: cha_FALSE_ROOT=/opt/bvt-hls/fake

cha_PRB
specifies path to a stream probe utility that allows to tell whether there's any data flow from the source. **vsm** needs it to know when a channel goes offline, the transmission simply stops. It is important to tell the "offline" scenario from another issue with the channel that might cause **vsm** to exit returning an error code. GigA+ supplies **prbsm** of its own making but one is always welcome to replace it with a custom app/script. If *cha_PRB* is omitted, **vsm** exits when it "thinks" the channel has gone offline.

Example: cha_PRB=prbsm

cha_ADMIN_EMAIL
specifies administrator's email where **vsm** would send notifications of importance, such as: start, exit, going online and offline. If omitted, no notifications are sent.

Example: cha_ADMIN_EMAIL='admin@myown.tv'

cha_TRANSOPT
specifies options passed to **ffmpeg**(1) for trans−coding the source channel before it gets segmented. This is a simplistic approach to trans−coding content and should be used for *simple cases*, such as making sure audio streams within the channel are of the right codec(s). If the parameter is set, **vsm** runs **ffmpeg** on the source and pipes the output to **gxseg**. Use at your own risk/discretion and **do test** the options with **ffmpeg** before you put them into the spec.

Example: cha_TRANSOPT="-map 0:0 -map 0:1 -map 0:2 -map 0:3 -vcodec copy -c:a mp3 -c:s copy"

AUTHORS
Pavel V. Cherenkov

SEE ALSO
wux(1),**gxpm**(1),**vsm-scripts**(5),**ffmpeg**(1),**crontab**(1),**gxseg**(1)

SYNOPSIS

This page documents auxillary scripts for segmenting a video stream via **vsm**(1) − GigA+ video−stream manager. The scripts are: **hos**, **prbsm** and **force−rollover.sh**.

DESCRIPTION

vsm(1) encompasses a lot of business logic on its own, yet certain functionality is externalized and put into separate scripts. One of the reasons for that is that users could customize the abstracted components and use a different language for implementation. The components are:

hos

is the channel housekeeping utility, launched by **vsm** at the start of its work. The utility assures that the channel's data storage keeps only the necessary data that is no older than the archival period given in the **channel spec** (see *cha_CAPACITY* in **vsm**(1) documentation).

hos is abstracted from **vsm** so that it could be invoked either manually or by a **cron**(8) job.

SYNOPSIS

hos OPTIONS *channel−tag*

hos takes just one mandatory parameter:

channel−tag
: is the nametag used for the channel to attend to (as in *cable1*). **hos** will retrieve the channel's spec and read parameters from it.

OPTIONS

−−common
: do not work on channel data, only deal with *common* logs (i.e. gxpm.log, gxws.log, dwg.log, etc.) − for channel−agnostic modules.

−−term do NOT generate log, send output to terminal only. By default, **hos** will create a log in the channel's root directory and mirror all messages it sends to STDERR to it. This option disables logging.

−n simulation mode (like make −n), print commands but do nothing.

−L force **vsm** to rotate the log. Signals the channel−bound **vsm**(1) to rotate its log.

−−sysrep
: include *system report* at the beginning of each log. The report consists of a printout from **vmstat**(8) and a process table dump via **ps**(1) utility. This fattens up the log but may be quite useful to have if a server goes down (from being resource−strapped or else). Use at your discretion.

−−minfree *MB*
: alert if free (physical) RAM goes below this limit. This setting allows to set a threshold for the minimum of RAM available. It measures the *OS−unmanaged* free memory, so the system may actually run fine even with 0 value. Use at your discretion.

Example: hos --term --sysrep -L cable1

prbsm

checks whether the channels is sending data, i.e. is *ONLINE* or *OFFLINE*. **vsm**(1) needs the utility to check on a channel that crashed or stalled. If the channel went OFFLINE, **vsm** notifies the admin and waits for the channel to go back ONLINE.

SYNOPSIS

prbsm [−v] [−p sec] input−url [once|stabilize]

prbsm has one mandatory parameter:

input−url
is the **ffmpeg**(1) −compliant URL for the source stream.

OPTIONS

−v verbose output.

−p *sec* pause (in seconds) to take between checks.

once|stabilize
check *once* if specified, or wait for the channel to go ONLINE (check as long as it takes).

Example: prbsm -p 5 http://acme.tv:4040/udp/224.0.2.26:5050 stabilize

force−rollover.sh

signals **vsm** to initiate a roll−over process. The script was created so that a roll−over could be scheduled via **crontab**(1)

SYNOPSIS

force−rollover.sh *channel−root−dir* signals to roll over a channel at the given directory.

Example: force−rollover.sh /opt/channels/cnn

AUTHORS

Pavel V. Cherenkov

SEE ALSO

vsm(1),**gpm**(1),**crontab**(1),**ffmpeg**(1)