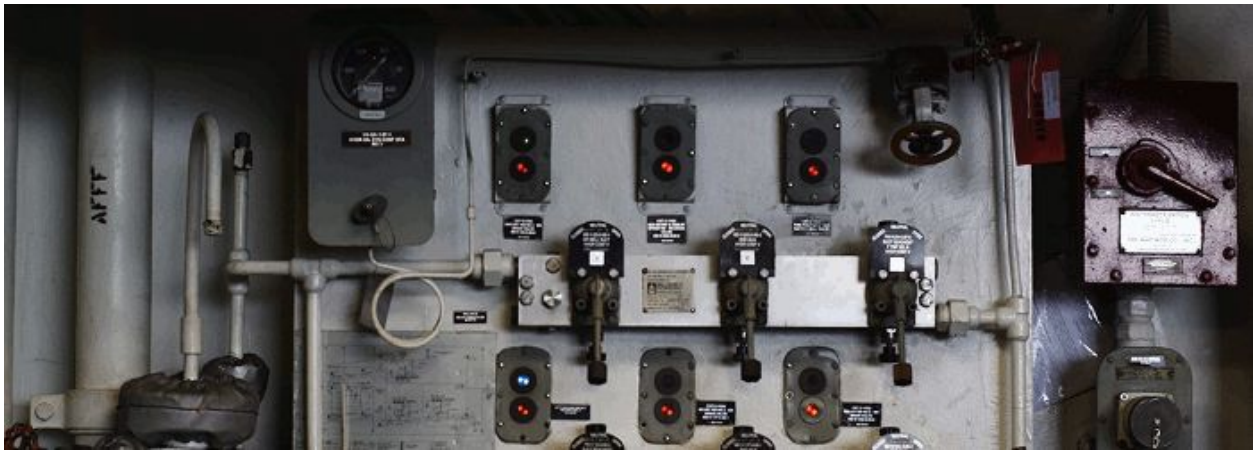
Buffer subsystem (bufd) explained



Introduction

This document explains how data-caching (for non-segmented streams) works in Gigapxy/GigA+. Relevant parameter names (in `gigapxy.conf` or `gigaplus.conf`) under `ng.bufd` are always highlighted as **this**.

In Gigapxy, gng is responsible for streaming. Each gng maintains a cache for channels (streams), a portion of the stream in saved in the cache and then is discarded as the stream moves on. Each gng process maintains its own cache and does not share it with any other processes.

Common pool and buffer chains

A cache is made up of N buffers. A gng can allocate up to **max_unit_count** buffers, each of **max_unit_size** bytes, forming a common pool. Initially, gng allocates **prealloc_count** buffers, which is **max_unit_count / 4** by default. All channels can 'borrow' buffers from the common pool as they need them.

Buffers for each channel form a chain: from the **tail** (the oldest buffer) to the **head** (the freshest data). In `gng.log` buffers are referred to as **STG**n, where **n** is the buffer's sequence ID (1...max_unit_count).

This is what a channel's buffer chain looks line:

Parameter **max_units_per_channel** defines how many buffers can be in a single channel's chain. Normally, this number should not be too high (between 3 and 10): if it is, and a chains get too long, the common pool may get exhausted, disrupting the work of the whole gng, which then would be unable to create any more channels or even cache more data for the existing ones.

Example:

```
max_unit_count          = 64;
max_unit_size           = 67108864;
max_unit_duration_sec   = 600;
max_units_per_channel   = 8;
```

Parameters **max_unit_size** and **max_unit_duration_sec** define how much data can be stored in a single buffer: **max_unit_size** defines the size of the buffer in bytes, **max_unit_duration_sec** limits the size to the number of seconds' worth of streaming. For instance, if the buffer size is 64 MB but it is limited to 10 seconds and 32 MB happen to hold 10 seconds of that particular stream, only 32 MB out of 64 MB would be utilized. If one wants the buffers to be used at the full capacity, it would suffice to set **max_unit_duration_sec** to high enough value to overflow the buffer size.

Buffer material: file or RAM

Buffers can be represented by files, with or without memory mapping - or by same-sized chunks of RAM. For smoother, faster operations most people prefer memory. Files would allow one to save memory and run Gigapxy on a memory-constrained device. Unless there is a reason to economize, I would suggest using in-memory buffers (**mmap_anon = true**).

Example:

```
data_dir                = "/opt/gigapxy";
mmap_files              = false;
mmap_anon               = true;
```

If using files, **data_dir** will be the path to the directory where all buffers will be created and pre-allocated as files at gng's start-up (for a large number of big files on a slow FS, be prepared to wait). If **keep_files** is set to true, files would be visible in **data_dir**, otherwise (by default) not.

Parameter **mmap_files** will define whether bufd should `mmap(2)` each buffer it (actively) uses and work with it as with an in-memory buffer. If **mmap_files** is set to false, all buffer-bound I/O will be file-based. NB: Whenever possible (and would make sense), bufd will try to use `sendfile(2)` in place of a plain `write(2)`.

If **mmap_anon** is true, files are not used at all, making the above parameters useless. gng then uses memory for all the buffers, no file mapping performed. Note: if **ng.use_sendfile** is set to true (not within bufd!), gng would still try to use `sendfile(2)` to send the data out to clients. (This was implemented as an experimental feature, no significant gain noticed.)

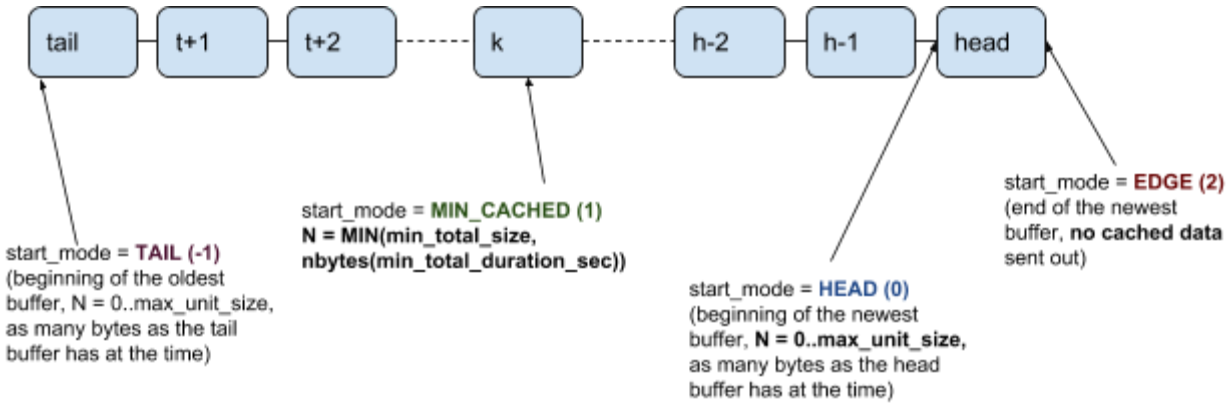
Start modes

The **start_mode** parameter defines where in the channel's buffer chain a client starts, which signifies how much cached data would become immediately available to the client before it has to wait for 'live' datagrams. This parameter will, in turn, greatly affect how the client-side media player would behave when a channel switch occurs.

When a new client is given to a gng it finds the **start position** and tries to send out as much data as possible from that position to the client's socket. The maximum amount of data to send out is the full size of a buffer (**max_unit_size**), but the *actual* number of bytes sent at once would depend on many parameters (such as, for instance, socket buffer size and state).

The important points are: 1) a data segment sent out cannot exceed the size of a buffer; 2) all starting positions are *buffer-aligned*: they are either at the start or at the end of a buffer. After the initial write, the client's position advances by the number of bytes actually sent (i.e. the value `write`(2) returned).

The available start modes are: **TAIL (-1)**, **HEAD (0)**, **MIN_CACHED (1)** and **EDGE (2)**. The picture below presents a brief summary of what they mean:



- TAIL (-1) - beginning of the oldest buffer; this mode is rarely used (by clients) but it almost always guarantees that there'll be at least one buffer to send;
- HEAD (0) - beginning of the newest buffer: the new buffer could be almost full or half empty or have just 1K of data, it all depends on the moment we land there. But we know that we are would always send somewhat less that one buffer's worth;
- MIN_CACHED (1) - spans as many buffers as it takes to exceed **min_total_size** or **min_duration_sec**. gng will start with the **head** buffer and step one buffer back until the cumulative number of bytes exceeds **min_total_size** or the number of seconds goes beyond **min_duration_sec**. This guarantees that at least **N** bytes (which could be more than 1 buffer) would be sent from cache;
- EDGE (2) - end of the newest buffer, we wait for the fresh data to send out the first portion.

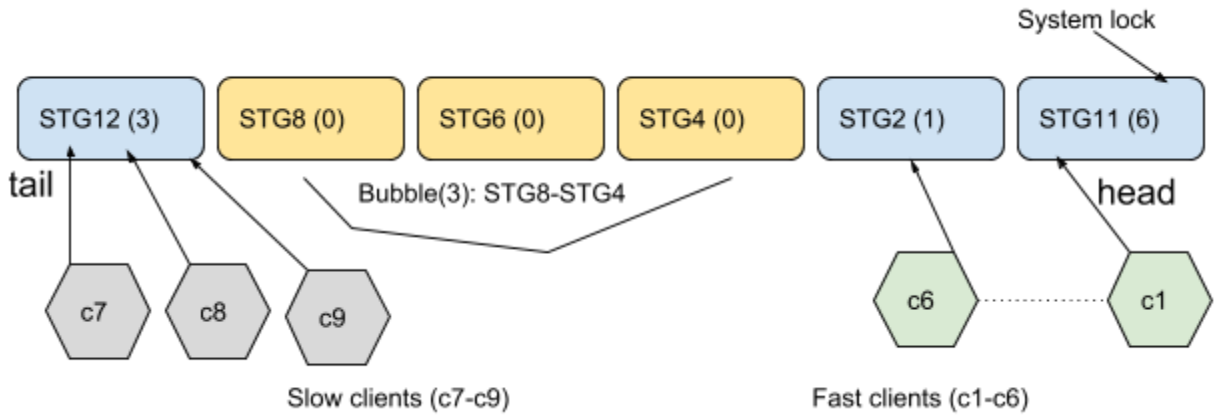
Example:

```
start_mode              = 0;
min_total_size          = 1048576;
min_total_duration_sec  = 20;
```

Locks, clients and bubbles

A newly-assigned client begins streaming channel content from the *start position* (as per selected start mode) in the buffer it got assigned to. As it 'lands' on a new buffer, it puts a *lock* on it, signifying that it intends to perform I/O with this particular buffer, now and/or in the future. The client will read from the buffer as soon as the (destination) socket would allow to be written to. (And until then the client would I/O-wait.) When the destination socket allows more data to be written, the client writes as much data as it can using a single `write`(2) call: from its current position till the end of the buffer. When the end of the buffer is reached, the client removes the lock from the buffer and requests a new one from `bufd`. The new buffer now gets locked, which literally just increments a counter, showing how many clients are 'interested' in this particular buffer.

With many clients on the same channel, we could see a series of counters, each corresponding to a buffer. The illustration below presents a chain of buffers with the lock counter in round brackets. Let's take a closer look, from head to tail.



The head (`STG11`) has six locks and five associated clients. The extra lock is from bufd, signifying that bufd keeps writing new data into the head buffer. As bufd fills up the head, it requests a new buffer and moves the lock to it. Clients C1-C5 are reading from the head, C6 is reading from the next buffer - `STG2`.

No clients read from the next three buffers: STG4, STG6 and STG8. But the tail buffer (STG12) has three clients. A sequence of more than two unlocked buffers is a *bubble*. Why do bubbles happen? The most popular cause happens to be *slow clients*. A client advances within its channel's buffer chain only as

fast as connection would allow data to be sent (remember that a client I/O-waits to *write* before it can read any data). If the STB (or any other device or software) on the far end of the connection cannot process stream data fast enough, it will communicate such inability via TCP and the output will be delayed. Meanwhile, the chain will grow and the faster clients would gravitate toward the head, leaving behind a gap (or a *bubble*) of unlocked buffers.

Example:

```
# Defines bubble-burst modes:
#
#   0 = NONE        - no bubble detection or bursts.
#   1 = SCAN        - scan for bubbles, warn but do not burst.
#   2 = BURST       - scan, warn and burst.

burst_mode  = 1;
```

Gigapxy (gng) knows how to detect and 'burst' bubbles. Firstly, a careful measurement is made to determine if a bubble should be dealt with at all. Secondly, bursting is not disabled by default, but detection is performed and a warning is issued every time a bubble is found. If, however, a bubble must be 'burst', it is done by simply removing the client lock(s) from the leftmost buffer (the one closer to the tail). After the 'burst', gng detaches the unlocked buffers from the channel and sends them back to the common pool to be 'recycled'. It also resets essential attributes within the buffer, thus letting the clients that still reference it that the buffer is no longer valid for their I/O. (And the associated clients react by disconnecting.)

Recycling policy

A client relinquishes its lock every time it moves to another buffer, you could even say it *moves* its lock to another buffer. The counter may go to zero but that does not necessarily indicate that the buffer can be disposed of - there could be clients *behind* this buffer. A recycling subroutine is executed every **recycle_timeout_sec** seconds by `bufd` in order to clean up channels' buffer chains. It starts from the tail and keeps removing unlocked buffers until it finds one.

Example:

```
recycle_timeout_sec     = 20;
```

Emergency recycling

This is the procedure of forcibly detaching the tail buffer from a channel (disconnecting all clients locked on the tail) and then attaching the buffer as the head. This is done when **allow_emergency_recycle** is set to true and when a channel is unable to allocate a new buffer from the common pool. This is truly an **emergency** procedure often preceding a DOS (Denial of Service) state of a `gng`.

Example:

```
allow_emergency_recycle = true;
```

If a buffer has been emergency-recycled (grep gng log for 'emergency'), a change in configuration (of gng) is most definitely due. Do not continue as-is.

Debug logging (bufd)

One can turn on debug logging in `gng` just for `bufd` in the configuration. It will make gng logs somewhat bigger but not excessively large. It makes sense to run with bufd debug logging when you're in the process of tuning up bufd settings. Once you've found the configuration that works for you, restrict `gng` to either info or normal logging.

Example:

```
log: {
     level_bufd   = "debug";
};
```

Summary and considerations for creating a bufd config

1. Calculate **how many channels** you will have *in total* and how many you could count on having active clients for *at the same time*; use these two numbers to estimate **how many buffers** you need;
2. Estimate how long you want a single buffer to play back for and see how big your average buffer should be. See how much data you want cached. Use the results to define the **size of a buffer**.
3. Most likely you'd want to use RAM-based buffers, but do you **have enough RAM**? If not, try memory-mapped files on a fast FS. If using files, consider a very fast HDD/SSD that will sustain heavy usage (multiple small writes).
4. Anticipating *slow clients*? Enable bubble bursts (**burst_mode = 2**) and debug-logging for bufd. Do not allow long chains, limit **max_units_per_channel** to prevent starvation (when one long chain may exhaust the pool).

Example of a real-life bufd configuration

This configuration has been given as part of a public post, only the bufd portion is presented here:

```
bufd = {
    data_dir              = "/giga_bufs";

    mmap_files            = true;
    mmap_anon             = false;

    start_mode            = 1;

    max_unit_count        = 300;
    max_unit_size         = 33554432;
    prealloc_count        = 150;

    max_unit_duration_sec   = 1200;
    min_total_duration_sec  = 20;

    min_total_size        = 1048576;

    recycle_timeout_sec     = 20;
    allow_emergency_recycle = true;
  };
```

Enjoy Gigapxy/GigA+!