Wubloader v3

Background

How HLS works

In HLS streaming, a stream consists of:

- The **master playlist** file, which describes the available streams of the content (eg. alternate languages, camera angles, or most commonly alternate quality/resolutions). It links to media playlists.
- Several **media playlist** files, which give a list of segment links as well as metadata on them such as a timestamp (actual time, not time since stream start), length, and whether you've reached end-of-stream. In a live stream, the media playlist changes over time, with old segments removed from the list and newly-broadcast segments added.
- Many **segment** files, which are individual short sections (eg. 5-10sec) of video or audio. This is the actual streamed content.

So a HLS client works by doing the following:

1. It fetches the master playlist and selects a stream (eg. "source")
2. It fetches the selected media playlist
3. It downloads all available segments in that playlist and displays them in order
4. While that content is playing, it re-fetches the media playlist again in a loop
5. As each new segment appears, it adds it to the queue to be displayed after the previous segment.

HLS and Twitch

On twitch, you use twitch's api to retrieve the master playlist. The URLs for the media playlists (one per stream quality) are signed by the original API server (so twitch knows who you are) and expire after 24 hours. The URLs in the media playlist for the segments again are signed but expire after a very short time (approx 1 minute). The media playlist file only shows the most recent minute or so of segments.
Depending on your geographical location (as determined by your IP) twitch will point you to a different edge node (eg. in San Francisco you'll be directed to a San Jose datacenter, from Germany you'll be directed to Amsterdam).
Despite the fact that the urls and media playlists will be different from location to location, we have confirmed that:

- The video is always split into exactly the same segments, byte for byte
- The segments have timestamps attached which are the time of broadcast
- If you fetch segments with the same timestamp from two different transcoded quality streams, the content will line up exactly.

Problems with existing system

- If one node has an issue recording the stream, it has a permanent hole which will show up in any video cut from that node
    - Our current workaround to this is to re-cut the video on another node and hope it doesn't also have a hole within the same timespan
    - Since the stream goes into one big video, such issues are difficult to identify or isolate.
- Uses google sheets as the primary database for coordination
    - Adds to sheet usage, number of columns, and complexity, which causes client-side lag for users
        - This may be unrelated and solvable without needing a change in bot behaviour.
    - Lack of proper concurrent operations leads to race conditions
        - This was observed in practice during DB2018, where both nodes would attempt to cut the same row and overwrite each other's results.
- Media playlist returns 403 Forbidden after 24h of streaming to file
    - Fix exists but we'd need to run a custom version of streamlink
    - This caused our one example during DB2018 of both wubloaders missing the same timespan of video and we had to resort to cutting a video by hand from VODs.
- Stream is stored as a monolithic video file (new file every time stream is interrupted)
    - This means more video to read when cutting, so longer cut times
    - Since timestamps are node-local and represent time-of-record, not time-of-broadcast, they vary slightly due to clock drift and stream delay. This means timestamps in the video file are only valid for the same node.
        - This in turn means we must always cut on the same node where editing is done, which is generally fine but not ideal, as it prevents us from load-sharing as effectively.
        - This also prevents us from using the same timestamps across different transcodings (ie. different quality), which creates the requirement for "draft videos" to be cut.
    - An operator has no easy way of watching a small section of the large video file without downloading the whole thing.
        - Currently we work around this using the Chunk Sheet, which is an extra complication that could be done away with if this was possible.

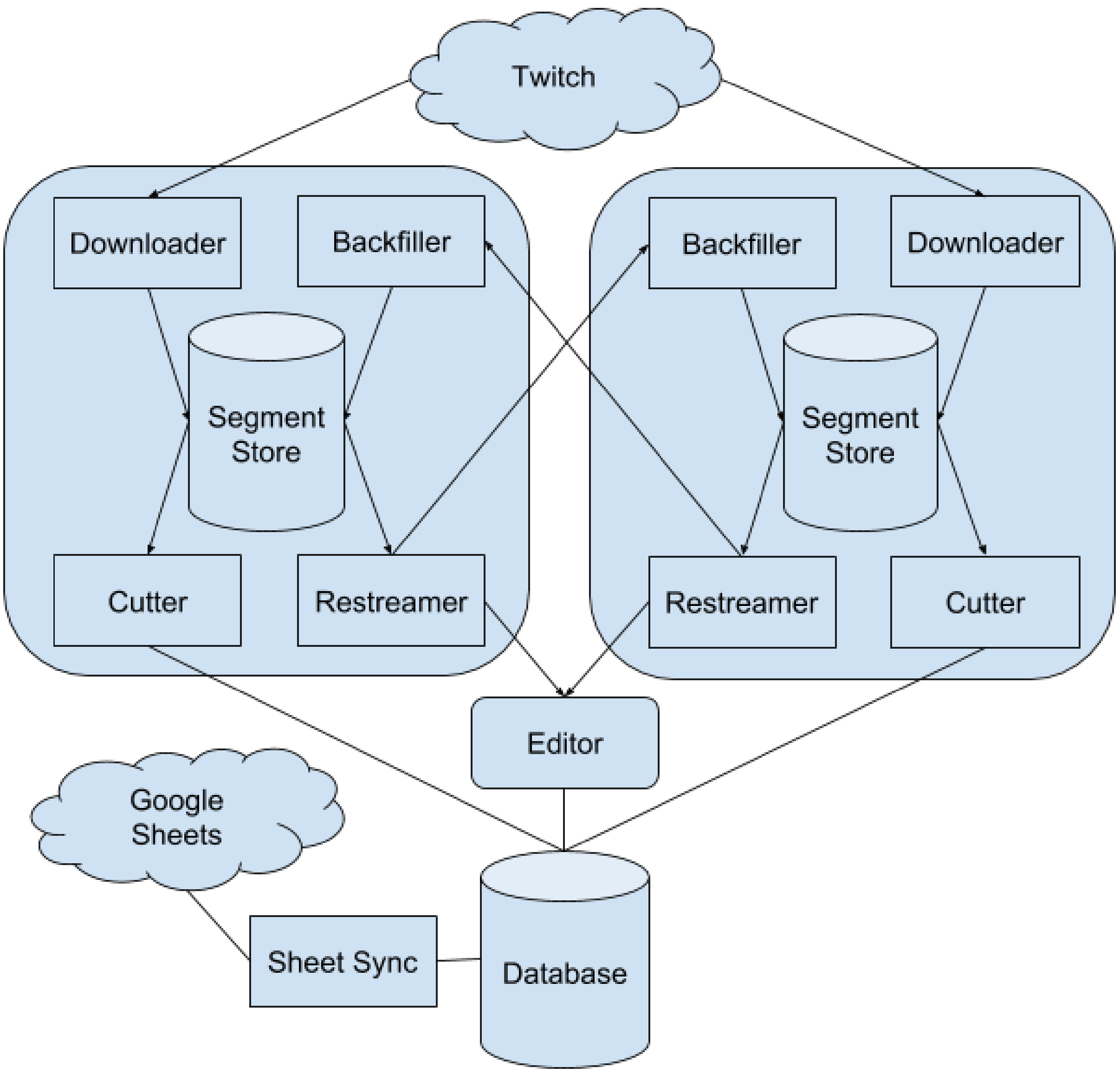
The new system

While the actual implementation may not be broken up like this, for purposes of explanation I will divide the implementation into various areas of responsibility:

- **Segment Store**: Stores all the segments along with their timespan and hash
- **Downloader**: Downloads segments from twitch (aka. what streamlink currently does)
- **Backfiller**: Downloads segments from other nodes, to catch stuff that node missed
- **Cutter**: Does the actual cutting and uploading

- **Restreamer**: Serves the saved segments to the editor and backfiller
- **Editor** (aka Thrimbletrimmer): Watches the saved segments and submits trim times
- **Database**: Saves trim times and other info, coordinates who is uploading what
- **Sheet Sync**: Copies inputs from Google Sheets and displays outputs on it

A diagram of which parts interact with which follows.



In the above, Editor runs inside a user's browser (though it wouldn't talk directly to the database, it would go via some simple api service, not pictured for clarity). The large encapsulating boxes represent an individual node. Sheet Sync and Database may run on one of these nodes, or elsewhere, it doesn't matter. Pictured is two nodes but you can scale out to any number. Additional nodes simply add more capacity to cut more videos at once, and greater redundancy.

The segment store

Central to all this is the segment store, which I will discuss first.

The segment store contains all the segments that make up a stream, indexed by their timestamp and stored along with their length and hash. This doesn't require anything fancy and is probably best implemented simply as a file structure, eg.

```
        1080p/31/41:59-6.0-419cdb633906313c3087d0950651a321.ts
```

would correspond to a segment of the 1080p stream starting at bustime 31:41:59, that is 6 seconds long and has hash 419cdb633906313c3087d0950651a321.
Splitting into a directory per hour keeps the number of files per directory reasonable (600 within each hour assuming a segment length of 6s, and less than 200 in the top-level hours directory), and lets us look up what exists for a given time range easily (by listing the directory for that hour). By naming each segment by its hash, we can also easily detect and warn if multiple segments exist for the same time range (which shouldn't happen, but it's entirely possible twitch could do something weird so better to be able to handle it).
We can avoid issues with partially written segments by writing to a tempfile first, then doing an atomic rename.

What if it goes down?

If the disk of a single node fails, we lose all data on that node. However, since the backfillers run often, the only potentially irreplaceable data would be video segments that:

- Can no longer be fetched (dropped off the end of the media playlist)
- No other node fetched from twitch due to other issues
- No other node fetched from this node via the backfiller already

Since the first point requires a segment to be several minutes old, and the third point requires a segment to (in normal conditions) be under a minute or so old, and the second point is expected to be rare, it would take a very strange set of conditions for a single node's loss to result in irrecoverable data loss.
Note we could bring a new node up at any time and it would simply backfill (up to some configurable time back) from existing nodes.

The downloader

This part's nice and simple. All it does is stream from twitch and put things into the segment store. We can also augment it with more stubborn retry logic than streamlink, refresh the master playlist regularly, etc.
We capture both the source quality and 480p versions of the stream.

What if it goes down?

Until it is started again, the backfiller will fetch any missed segments from other nodes.
This is only a problem if the downloaders on all nodes are down at the same time, for a period longer than a minute or so (the time it takes for a brand new segment to appear then drop off the back of the media playlist).

The cutter

This operates similarly to the existing wubloader, except that it reads from the database instead of a google sheet. It claims a cut job (without race conditions due to SQL concurrency semantics), performs the cut, then uploads the result to youtube. Note that

since it’s cutting from the segment store instead of a single video, we can selectively only include the files we actually need, which will cut down on how much processing ffmpeg needs to do. I’ll have to confirm but I believe the segment files might even be formatted such that directly concatenating them byte-for-byte will produce a correctly encoded video, which means we’d only actually need to trim the first and last segments, so cutting could be extremely quick, perhaps a few seconds.
If the cutter detects holes, they might be fixable if they exist on another node, and the replication process is just being slow/having issues. Because of this, the cutter should return an error saying that the video has holes, and require an operator to manually confirm that that’s ok and the holes are unfixable before the cutter will allow it.

What if it goes down?

We can’t use the node to cut new videos until it comes back up. Any in-progress videos would be restarted.

The restreamer

This simply serves files and directory listings directly from the segment store, as well as having the ability to generate a master and media playlist for the editor to stream from. Apart from the playlist generation, this could literally just be nginx.

What if it goes down?

We can’t download segments from this node until it comes back up. Editors would need to use other nodes. Any segments which only exist on that node would not be accessible until it comes back up.

The backfiller

The backfiller’s job is to catch anything the downloader missed, that one of the other nodes caught. It periodically calls the restreamer on other nodes and asks them for a list of all their segments for a time range. If any of those segments don’t already exist locally, it downloads them to the segment store. This way as long as at least one node captures a segment, it will be replicated everywhere.

What if it goes down?

We can’t backfill segments that this node missed until it comes back up. Since these segments should be rare, the impact should be minimal.

The editor

The editor allows a user to trim the video by using HLS to stream the segments from the restreamer. Note that since the video is already nicely cut up into segments, no further processing is required for us to stream it to the editor client. The client can optionally stream either the source quality or 480p version of the stream. All timestamps will be based off the time-of-broadcast timestamps in twitch’s original media playlist, so which one you use doesn’t matter. The resulting cut times, title, etc are written to the database.

This will require changes to Thrimbletrimmer but it should hopefully not be too hard to find an existing javascript HLS client.
Note that the exact same interface can be used in place of a chunk sheet, simply by requesting the server stream a given half-hour section of the source segments.

What if it goes down?

Operators will need to use another node as the source of streamed video for editing. The node can still be used for cutting.

The database

The database is used to tell the editor what time range to stream / the original description, and then to list upload jobs that the nodes can claim, cut, upload, then write back the video URL to.
We could also use this for any other minor bits of coordination information that crop up, such as telling the backfillers what other nodes exist.
My current plan is to use Postgres here, but I’m open to any alternative that fulfils the requirements.

What if it goes down?

We cannot edit or upload videos. Videos that were in the process of being cut when the database failed would still upload, and care must be taken to not accidentally re-upload the same video when the database returns.
We would still be able to capture and replicate video between machines.

Sheet Sync

This component reads certain columns from the google sheet designated as “inputs”, and writes them to the database. It also reads certain database columns and writes them to certain google sheet columns designated “outputs”.
This strict input/output split has the very nice property of being idempotent - running the same sync operation a second time will update everything to the correct value again, even if google sheets weirdness causes the wrong value to be written in some previous iteration. In this way the sync is self-healing. It can also do sanity checking and parsing of text fields into native formats.
The input columns allow operators to give input to the editor and cutters about descriptions, times, etc. The output columns allow operators to observe the state of the system, eg. when an upload is in progress or what the final video link is.

What if it goes down?

The sheet’s outputs will appear to freeze, and the inputs will no longer have any effect. The sheet can continue to be written and events noted down, and editing can continue to take place, but no new rows can be edited (since their start/end times won’t exist) and modifications to previous ones will have no effect.

Google Sheets

You know what this is.

What if it goes down?

We can no longer record what happens (and when), though I daresay at least one enterprising VST member would start writing things down on the nearest text editor / piece of paper.
Other than that, effects would be similar to the sheet sync going down.

Risks

- An unexpected failure mode could occur that brings down the downloader on all nodes at the same time
    - We can mitigate this to a certain extent by heavy testing on other major, long streams eg. SGDQ which is about 4 months out from next year's run.
    - We can also run a Wubloader v2 capture independently of all the v3 nodes. We could then convert the sheet back to v2 and use that system as a last resort.
- Twitch could change some detail of their implementation that invalidates the replication concept
    - This would be quickly detected by the way segments are laid out on disk
    - We can still fall back to not replicating, and we're no worse off than the old system
- Something goes wrong, and the only person who understands things is asleep
    - I'd argue we already have that to a certain extent, and we can hopefully mitigate this by involving more people in development / purposely avoiding having one person do most of it.