smeb - A simple WebM streaming server

Stephan Soller, Computer Science and Media
Stuttgart Media University
ss312@hdm-stuttgart.de

Abstract

Video live streaming has become common place in the Internet. Usually one source stream (e.g. of a conference) is broadcasted to many viewers. This paper describes the design and implementation of an WebM live streaming server. How it relates to other existing streaming servers and what design decisions have been taken. The way how WebM live streams are broadcasted is explained in more detail as well as are some refinements required to support browser clients.

1. Introduction

smeb is a simple WebM streaming server. It receives a video stream from one source (the producer) and allows multiple viewers to watch this stream. In short the video stream received from the producer is copied to all viewers of that stream.

This is similar to Icecast v2.4 [1], stream-m [2] or ffserve [3] but with a different feature set:

- smeb can create streams on the fly as soon as a someone sends data. When the producer disconnects a stream is automatically deleted after a timeout (stream-m and ffserve need preconfigured streams).
- smeb offers a very simple machine-friendly JSON interface to query all available streams and their status (not machine-friendly with ffserve).
- The interface to send video streams to smeb is very simple (a HTTP POST request). It doesn't require any additional protocols (Icecast and ffserve require custom protocols).

So smeb covers a feature space not quite covered by any of the above streaming servers. It doesn't require preconfigured streams (that eliminates stream-m and ffserve) and it's easy to send video streams to smeb (close to undocumented for Icecast v2.4 when smeb was conceived). It is not designed to be "the" streaming server but rather a building block for live streaming systems that needs a minimum of configuration and maintenance.

2. Basic design decisions

This chapter shortly describes the core concepts of smeb.

The basic principle of smeb is WebM live streaming based on the Matroska container as explained in detail in the next chapter. New viewers only need to receive a short header when connecting to a stream. After that the video stream from the producer can be more or less copied to all connected viewers. For clients this is a normal not seekable video (the user can't use the timeline to jump ahead and back). Thanks to this all browsers and media players that support WebM playback can be used to watch such a live stream (Chrome, Firefox, Opera, VLC, ffplay, etc.).

The second defining design choice of smeb is to use HTTP as transport protocol. This protocol is native to browsers and supported by many media players. HTTP usually works even behind large cooperate firewalls. Thus allowing employees there to watch the video streams. HTTP is also quite a bit simpler and easier to implement than alternative protocol stacks for video streaming.

With HTTP the server interface remains quite simple:

- The producer sends a WebM video stream to the server via a POST request.
- Viewers can watch the video with a GET request to the same resource.
- A GET request to index.json returns a list of all currently available streams and their status.

In case the producer dies unexpectedly the clients are kept connected for some time. When the producer resumes sending the clients resume playing without any user interaction. To them it's just a long network stall. So there is no need for users to refresh the browser when the stream is interrupted. Older events have shown that when the viewers are disconnected on stream interrup-

tions up to halve of them do not reconnect to continue watching.

The server can also handle multiple streams at the same time. In case multiple events take place at the same time, one event needs multiple streams (e.g. one for each room) or to implement simple quality switching (send each quality as an independent stream).

smeb does *not* do any video encoding. It merely distributes the received data to all connected viewers. This keeps smeb itself much simpler and almost configuration free. Any complex encoding as well as encoding performance optimization are done outside of smeb. Established tools like ffmpeg offer many options for that. Putting such an encoder into smeb would require to replicate all these options.

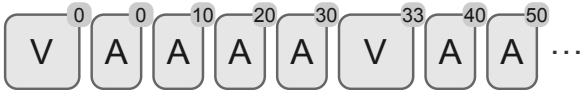
Implementation wise smeb is build with a `poll()` based event loop. To exclude any 3rd party error sources, get absolute control over the mainloop and all error handling smeb was written in C, directly using the `poll()` system call.

3. WebM live streaming

There are many approaches to video live streaming. This chapter provides some insight into the approach taken by smeb [4].

A video file stores several audio and video tracks. A video player then reads the file and plays the audio and video tracks synchronously. These tracks are usually stored in an interleaved fashion and each data block (e.g. video frame) of a track is accompanied by a timestamp. The timestamps specify at what moment a video frame should be shown or an audio block should be played.

Figure 1: Data blocks of an video (V) and audio (A) stream. Stored in one file by interleaving. The small numbers are the timestamps of the blocks (in milliseconds).



How this interleaving and timestamping is done is defined by the *container format*. The container format used for WebM videos is a simplified subset of the Matroska container format. All of the techniques described here are actually based on the Matroska specification but also apply to WebM.

WebM stores data as a tree of elements, each element containing some basic data (integer, string, etc.) or child elements. This structure is very similar to XML and in fact has been extracted into it's own data format: EBML [5]. Matroska defines EBML elements and their meaning and WebM restricts which of these elements can be used in WebM files. This is similar to how SVG defines XML elements and their meanings and SVG Tiny restricts what elements can be used on handheld devices.

Figure 2: XML representation of a simplified element tree of an WebM video.

```
<EBML>...</EBML>      // EBML and Doctype information
<Segment>
    <Info>...</Info>  // Video and Audio track format
                         and information
    <Cluster>...</Cluster>  // Video and audio data
                               for about 1 sec.
    <Cluster>...</Cluster>  // Video and audio data
                               for about 1 sec.
    ...               // Many more Cluster elements
</Segment>
```

Figure 2 shows an simplified element tree of a WebM video. The XML representation is only used to clarify the structure. EBML encodes elements with an integer element ID, an integer specifying the size of the payload followed by the payload data itself.

The basic idea behind smebs live streaming is to split an incoming WebM video into a "header" part and cluster elements. The header part contains the *EBML* element, the start of the *Segment* element and the *Info* element. And with them all the information about the video a browser or media player needs to know (how many tracks are in there, what video codec, what audio codec, etc.). The *Cluster* elements then each contain the data of all tracks for about one second.

Figure 3: XML representation of the header part and two separate cluster elements

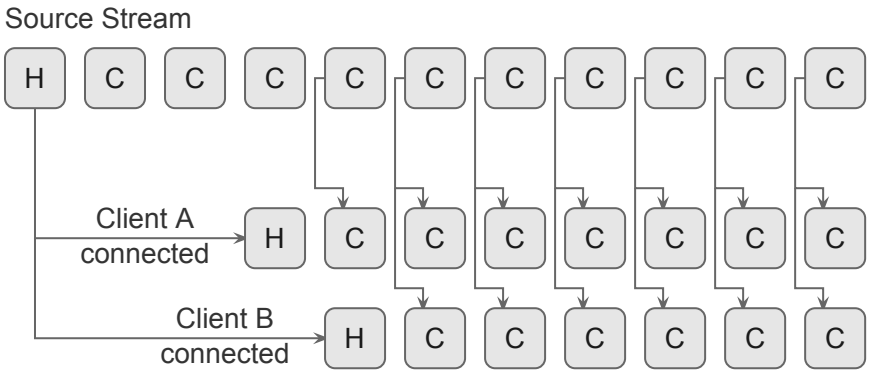
```
<EBML>...</EBML>
<Segment>
    <Info>...</Info>
```

```
<Cluster>...</Cluster>
```

```
<Cluster>...</Cluster>
```

When a producer begins to sends a WebM stream smeb first extracts the header part (everything up to and including the *Info* element) and stores it. Then when a viewer connects to the same WebM stream smeb first sends the header part. Every Cluster element then received from the producer is broadcasted to every viewer that already received the header part. If no one watches a stream all Cluster elements smeb receives are discarded. See figure 4.

Figure 4: Diagram of how smeb handles new viewers. H is the header part, C are individual Cluster elements.



An important detail of this scheme is that viewing clients never see the end of the *Segment* element. Since smeb processes live streams we don't know how long the entire video will be. It's unknown how many *Cluster* elements will be in the *Segment* element or how large all these *Cluster* elements will be. Usually the ID and size encoding of EBML elements would require us to specify an exact size of the *Segment* element. But:

“ There is only one reserved word for Element Size encoding, which is an Element Size encoded to all 1's. Such a coding indicates that the size of the Element is unknown, which is a special case that we believe will be useful for live streaming purposes. However, avoid using this reserved word unnecessarily, because it makes parsing slower and more difficult to implement.” [5]

This exception makes it possible to set the size of the Segment element to "unknown". Allowing smeb to send out as many Cluster elements as it receives. The clients continue to play the video until the data stream ends (the connection is closed). This small detail of the Matroska/EBML specification is an absolutely necessary underpinning of smebs live streaming approach. Container formats that lack such a way to specify an "unknown size" (e.g. MP4) can't be streamed with this approach.

4. Further refinements

The basic approach could be quickly verified to work with the ffmpeg sample video player ffplay. Browsers, however, require some additional constraints to prevent them from simply disconnecting or playing only partial streams (only audio and no video for example). This chapter roughly describes the quirks discovered by manual testing and reverse engineering.

Self contained first Cluster

The first Cluster element received by browsers must be self contained. It must not reference data the browser didn't get because it just connected and missed any previous data. Otherwise browsers simply disconnect immediately or only play the audio track and display no video.

This constraint is broken by the VP8 video codec. As many modern video codecs it mainly encodes differences from one frame to the next (or prediction errors of these differences to be more precise). To display one frame the video player has to know the previous frame. This "has to know" chain is broken by occasional "keyframes". They contain a complete frame and don't depend on any previous frames. This makes the video more error resilient and allows to easily resume playback from each of these keyframes.

When a new viewer connects to a stream the first frame in the first cluster has to be such a keyframe. Otherwise the first frame would require knowledge of the previous frames the viewer never received. Unfortunately testing with ffmpeg generated WebM streams showed most Cluster elements don't start with a keyframe. While ffplay ignores frames it can't decode browsers mostly disconnect when they encounter such frames.

To solve this problem smeb builds a special "intro" cluster for each stream. This intro cluster starts with the last encountered keyframe and contains all video and audio data since then. As soon as a new keyframe is encountered the intro cluster is reset and refilled with new data. Now when viewers connect to a stream smeb sends the header part, the current intro cluster and then starts to pass unmodified Cluster elements to that client.

Timestamp patching

One design goal of smeb was to keep clients connected even when the producer stream breaks down. As soon as the producer resumes sending new data the clients should automatically resume playback.

smeb achieves this by simply stopping to send Cluster elements when the producer breaks down. But the watching clients are not disconnected, they simply receive no more data. When a new producer connects and sends more data smeb simply continues to broadcast the new Cluster elements to all watchers. To watching clients this only looks like a long network stall.

When implementing this approach a problem became apparent: Each block of video and audio data in a Cluster element contains a timestamp. This is necessary so that video players can synchronize the audio and video tracks. These timestamps usually start at zero and have to increase as time goes by (see figure 1).

For example when a producer died the highest timestamp reached was 2400000 ms (40 minutes since stream start). Shortly after that a new producer connects and starts sending a WebM stream. Unfortunately this new producer restarts the timestamps at 0. So watching clients connected to smeb would observed normal Cluster elements with timestamps up to 2400000, then a long network stall and after that new Cluster elements with timestamps suddenly restarting at 0.

Some media players can cope with such sudden timestamp jumps. Browsers do not and simply disconnect on these occasions. To solve this problem smeb inspects and patches the timestamps of the received WebM streams. The patched timestamps are continuously increasing so clients don't see a sudden jump in the timestamps. Even when a producer dies and a new one sends timestamps starting with 0 again.

5. Further work

While a first prototype of smeb is finished and operational there is still much to be done:

Basic HTTP authentication should be required to send a stream. So only users with the correct password can actually broadcast streams.

Improve handling of clients that are to far behind. Either because of a slow connection or because they don't consume data (e.g. "hanging" browsers). Right now they're disconnected from the server if they lag to far behind. But the might be ways to bring these clients up to speed, e.g. by only sending halve the video frames until the client catches up.

Currently smeb doesn't support producers that *send* their WebM stream via HTTP chunked encoding. Implementing that would allow to receive streams from encoding software that doesn't have options to disable chunked encoding.

smeb doesn't respect the HTTP version a watching client requests. smeb always *answers* watching clients with HTTP/1.1 chunked encoding. This causes problems with old HTTP proxy servers that only understand HTTP/1.0. This could be solved by using a simple HTTP/1.0 connection without chunked encoding for those clients.

Creating a live JPEG preview of the last keyframe would allow proper previews of currently live streams on websites.

Under some circumstances randomly received data can crash the semb server. This happens for example when automated attacks hit the server. This can cause smeb to receive malicious PHP code that it doesn't understand but tries to parse as a WebM stream. As the code is currently build for experimentation it isn't hardened in the way production ready code would be.

smeb also served as test bed to explore the limits of implementing goto based state machines in C. Because of that large parts of the code are in need of proper refactoring.

Since smeb is a prototype it also suffers occasional memory leaks. When broadcasting several streams for half a day these can add up to several hundred megabytes of lost memory. This should be solved by proper management of received and send buffers.

6. Conclusion

Currently smeb is a fully functioning prototype. It confirmed the feasibility of the basic Matroska live streaming approach with browsers as clients. smeb also allows resuming of died streams without disconnecting all watching clients.

Despite being fully functional several important features (e.g. HTTP basic authentication) are still missing. Also the code quality and robustness is just that of a prototype. So smeb is not yet production ready.

7. References

[1] Icecast.org
http://www.icecast.org/
Retrieved 2014-07-14

[2] stream.m a WebM live streaming server
http://code.google.com/p/stream-m/
Retrieved 2014-07-14

[3] ffserver Documentation
https://www.ffmpeg.org/ffserver.html
Retrieved 2014-07-14

[4] Matroska Streaming
http://matroska.org/technical/streaming/index.html
Retrieved 2014-07-14

[5] EBML principle (Extensible Binary Meta Language)
http://matroska.org/technical/specs/index.html#EBML_ex
Retrieved 2014-07-14