

ARCHIVING WEB VIDEO

Radu POP

European Archive
Montreuil, France

Gabriel VASILE

European Archive
Montreuil, France

Julien MASANES

European Archive
Montreuil, France

ABSTRACT

Web archivists have a difficult time gathering web video that are, more often than not served with non-standard tools and protocols. This paper offers a survey of the state of the art in this domain.

Based on an experience of several years gathering web video content, we present detailed examples to help understand the issues and solution to capture web video content.

We also present a architectural framework for scaling web video content capture developed as part of the EU research project LiWA¹.

1.INTRODUCTION

Video has become a key part of the web today. Technology developed to serve video, have always being dominated by the need of the media industry, specifically when it comes to avoiding direct access to the files by the users. As a side effect, it has made web archivist's task of gathering content (hence files) much more difficult, requiring the development of specific approaches and tools.

The objective of this paper is to survey the main difficulties of archiving video on the web. Based on experience gained at the European Archive Foundation working on a variety of cases in the last years, we grouped the different problems in video capture in two main categories and we illustrate several technical solutions using some eloquent examples.

The first category includes Web sites that use standard HTTP protocol to deliver video content. Their difficulty consists in the various techniques used to obfuscate links to video files (e.g.: 2 or 3 hops and redirects). A representative example is the YouTube Web site.

The second category of problems are represented by Web sites that use transport protocols other than HTTP. From the various streaming protocols currently used on the Internet we selected as example the most recent and difficult one, RTMP streaming protocol. This section is illustrated by the swr.de site.

It is important noting that the technologies used to serve video on the web evolve extremely fast, and the case presented here are likely to change quickly in their details. However, the intention is to give enough details even if they might age rapidly, to understand the logic of these technics.

A consequence of what is today a cat-and-mouse game, it that our tools need to be continuously improved and updated. This is true both at the capture and access level.

In the second part of the paper we propose an architectural design to adrese the problem and enable rapid adaptation as well as scalability based on decoupling the processes in charge of video download from the crawler. This design improves the efficiency of both, in terms of scalability as well as flexibility as downloading tools developed by multimedia experts are easier to integrate and update. Finally, working asynchronously with the crawler offers a better support for error handling and process management.

2.VIDEO CAPTURE BY EXAMPLES

The different levels of complexity in capturing video content is illustrated in more details by on the following two examples.

2.1.HTTP redirects on YouTube.com

Each YouTube video is uniquely identified by a hashed identifier (a string of 11 characters) and can be generally accessed in an HTML page with an URL similar to:
<http://www.youtube.com/watch?v=uniqueID>

Probably the most difficult challenge in harvesting YouTube videos is represented by the frequent updates of the mechanisms provided by YouTube to access the videos. This represents a continuous effort to hide the direct URLs of the video files.

Using the classic harvesting methods, the crawler has to follow 5 distinct steps of direct links or redirects in order to actually access the video content. The general patterns of the intermediary URLs are summarised in the following table.

#	URL	mime-type	HTTP response
1	http://www.youtube.com/watch?v=foobar	text / html	200
2	http://s.ytimg.com/yt/swf/watch-vfl118818.swf	application / x-shockwave-flash	200
3	http://www.youtube.com/get_video?video_id=foobar&t=...	text/html	204 (OK no content)

¹ <http://liwa-project.eu/>

4	http://v1.lscache4.c.youtube.com/videoplayback?ip=0.0.0.0&sparams=id...	Redirect	302
5	http://v1.cache2.c.youtube.com/videoplayback?ip=0.0.0.0&sparams=id...	video / x-flv	200

Table 1. YouTube steps to access the video files.

Given a video identifier (foobar) and the URL of the YouTube page (#1), the crawler firstly discovers the URL of the Flash player (#2) used in the page. From the list of parameters passed to the player, the crawler may identify the HTML query (#3) to use when asking for the video content. Before actually obtaining the URL of the video file (#5), the crawler has to follow an intermediary redirect (#4) containing encoded tokens, like the IP address of the host and the time-stamp of the request.

The URL of the video (#5) is integrated into the Flash parameters that are dynamically generated when loading the page (#1). The problem encountered by the crawler consists in correctly identifying this URL, since it contains different escaped characters and it is concatenated with different other parameters interpreted by the Flash object. For instance, in the current version of YouTube pages, the “flashvars” parameter is a string of 6374 characters long, whereas the URL to be matched contains 392 characters.

In a closer analysis of the page (#1), the URL of the video (#5) may be also identified in the JavaScript fragment that generates the Flash parameters. In this recent version of YouTube pages, the text line to parse contains a “PLAYER_CONFIG” string, followed by a list of URLs in a random order. In between two pipe (“|”) characters one can extract the URL of the video, which already includes the computed tokens, based on the IP address and the time-stamp.

As one can notice, the path to follow in order to harvest the video files is twisted and the URLs of the video resources are obfuscated by redirects and addition of temporary tokens. In terms of crawling parameters, this can be translated into a 5 levels depth crawl of the YouTube page. Moreover, the URLs in #2, #4 and #5 point out to different sub-domains, therefore they need to be explicitly added to the scope of the crawl.

Finally, a major problem generated by the redirect URLs is related to the archive's access tools. Even if each video file is uniquely identified on YouTube, a different URL is dynamically generated at each download (due to the time-stamps). For this reason, an explicit reference needs to be added to the URLs index, to keep the trace between the original page (in #1) and the URL of the video file (in #5).

In practice, there are two possible approaches to video capture on YouTube: (i) an on-line video capture and (ii) an off-line technique for video download.

- With regards to the crawling process, in the **on-line approach** the video files are downloaded at crawl time, using an additional processor to Heritrix crawler. The beanShell script written

by Adam Taylor ¹, for instance, injects all the intermediary URLs (in #2-5) into the frontier, since the intermediary hops are generally out-of-scope for the current crawl (e.g. s.ytimg.com, v1.cache2.c.youtube.com). The video files are added to the same WARC files created by the crawler.

- The **off-line approach** downloads the video files in the post processing phase, based on the URLs of the YouTube pages. It uses an external downloader, as the one developed by Ricardo Garcia Gonzales ², which captures the video content and dumps in into flv files. Using the WARC Tools ³, the flv files are packed into distinct WARC files.

Both methods need to generate a link from the original URL of the video (as it appears in the Web page) and the name of the file or the new URL pointing to the video content (as it is stored in the archive, following the redirect hops).

The advantage of the **on-line approach** is that the download of the video files is done by the crawler itself and there are no other external tools to monitor and synchronize. Moreover, all the HTTP headers are stored in the archive, following the conversation with the YouTube servers. The drawback of this method is that the final URL of the video content (as in #5) does not any-longer contain the original identifier of the video (as in #1). Tracking back the video identifier becomes difficult to manage. The archive is also polluted with all the redirect URLs (they are no longer valid URLs, since the download tokens have limited validity).

On the other hand, the **off-line approach** offers more flexibility in monitoring and managing the external downloaders (e.g. for error handling). The video files keep the name of their original identifiers and the redirect URLs are not stored in the archive. A "fake" HTTP header needs to be inserted in the WARC files for each flv file, since the external downloader does not keep the server responses.

2.2. RTMP streaming on SWR.de

2.2.1. Overview of the Streaming Protocols

Streaming, that corresponds to Internet Engineering Task Force (IETF) standards, allows the server to control the transmission and is heavily optimized to keep things running in real-time. Clients don't have to download potentially huge files and this approach is particularly well-suited to live broadcasts. In fact, streaming usually uses two types of real-time streaming protocols: Real-time Transport Protocol (RTP) [rfc3550] to send packets of media data and Real Time Streaming Protocol (RTSP) [rfc2326] for control information. RTP uses the potentially lossy UDP, which does not attempt

¹<http://webarchive.jira.com/wiki/display/Heritrix/BeanShell+Script+For+Downloading+Video>

²<http://bitbucket.org/rg3/youtube-dl/wiki/Home>

³<http://code.google.com/p/warc-tools>

to retransmit lost packets, so all parties are designed to tolerate the dropping of packets during a transmission. This means that clients have to gracefully handle not getting all of the data for a video frame or an audio sample. This is preferable to a TCP/IP-based approach, which could make an indeterminate number of retries (and thus take an indeterminate amount of time) to get the missed packet.

Real Time Streaming Protocol (RTSP) is a network control protocol for use in entertainment and communications systems to control streaming media servers. The protocol is used to establish and control media sessions between end points. Clients of media servers issue VCR-like commands, such as play and pause, to facilitate real-time control of playback of media files from the server.

The RTSP protocol has similarities to HTTP, but RTSP adds new requests. While HTTP is stateless, RTSP is a stateful protocol. A session identifier is used to keep track of sessions when needed. Thus, no permanent TCP connection is needed. RTSP messages are sent from client to server, although some exceptions exist where the server will send to the client.

Multimedia Messaging Service (MMS) is a telecommunications standard for sending messages with multimedia objects (images, audio, video, rich text). MMS is an extension of the SMS standard, allowing longer message lengths and using WAP to display the content. MMS messages are delivered in a fashion almost identical to SMS, but any multimedia content is first encoded and inserted into a text message in a fashion similar to sending a MIME e-mail.

Real Time Messaging Protocol (RTMP) is a proprietary protocol developed by Adobe Systems for streaming audio, video and data over the Internet, between a Flash player and a server. To guarantee smooth delivery of video and audio streams, while still maintaining the ability to transmit bigger chunks of information, the protocol may split video and data into fragments. The size of the fragments used can be negotiated dynamically between the client and server, and even disabled completely if desired. Fragments from different streams may then be interleaved and multiplexed over a single connection. Adobe opened the specification for the RTMP protocol on June 15, 2009¹, but this specification appears to leave out many details of the protocol implementation.

2.2.2. Capturing streaming video

From a recent crawl performed by European Archive for capturing streaming videos we present in this section several technical details, based on the example of SWR.de Web site.

Video page layout

The structure of the Web page displaying the videos follows a similar pattern as presented in the previous example on YouTube.com. The HTML page contains a

main video panel including the original video player (i.e. the open-source JW Flash Video Player² from LongTail Video). The content of each HTML page is dynamically generated by the server and stored in specific HTML elements, like:

```
<div class="container">
  <noscript>
    <p class="flashError">
      JavaScript not active !
    </p>
  </noscript>

  <script type="text/javascript">
    ...
    function createVideoPlayer() {
      rtmpStreamer =
        "rtmp://fc-ondemand.swr.de/a4332/e6/";
      file =
        "rtmp://fc-ondemand.swr.de/a4332/e6/
        foobar-video.flv";
      ...
    }
  </script>

  <object width="520" height="344"
    type="application/x-shockwave-flash"
    id="player46"
    data="http://www.swr.de/static/
    jwplayer/player46.swf">
    <param name="flashvars" value="...">
    ...

  </object>
</div>
```

The Flash player is embedded into an <object> element and loaded from the Web server. All the parameters needed by the player are prepared by a JavaScript function (createVideoPlayer) and passed over to the Flash object (using the flashvars parameters). Starting from this point, the entire conversation with the streaming server is directly handled by the Flash player.

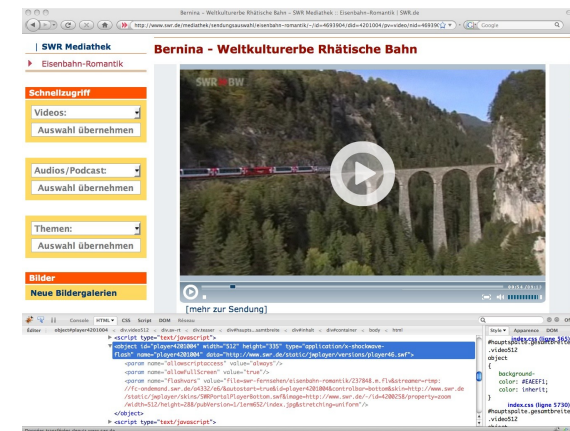


Figure 1. Live Web site – RTMP video.

Downloading the streaming video

In this particular example, the URL of the video file is written in clear inside the JavaScript fragment. From the crawler point of view, this represents a favorable case, since the JavaScript extractor is able to identify and extract the URL of the video file. But this URL does not represent a valid URL for the crawler, since it does not support protocol schemes other than HTTP/HTTPS. The

¹<http://www.adobe.com/devnet/rtmp/>

²<http://www.longtailvideo.com/players/jw-flv-player/>

RTMP URL will therefore be reported as a non-valid URL in the crawl's errors log.

In order to effectively download the video files (rtmp://.../foobar-video.flv), the RTMP URLs are filtered out from the errors log and passed to an external downloader (i.e. FLVStreamer). The RTMP downloader dumps the video content in flv files which are packed afterwards in WARC files.

Accessing the archived video content

From the access point of view, the fundamental difference between the live Web page and the archived one is represented by the transport protocol used to deliver the video content to the player. Deploying a streaming server on the archive's infrastructure involves lots of effort and it represents a costly solution to develop as well as to maintain. It would also require to run different servers for different protocols. In order to be generic, we provide access to the archived video content through HTTP protocol and the flv files are directly served from the WARC files together with the HTML pages and other resources. For the moment, streaming specific functionalities are lost (such as fast-forward or incremental download), but the basic access to the content is assured and can be adapted to different cases without overhead.

An important drawback of replacing streaming with plain HTTP becomes visible for large videos. If no adaptation is made, the player will need to load the entire flv file before starting playing the video. We have adapted our access tools at EA to alleviate this problem by incrementally loading chunks of the video files from the archive. The size of the chunks is subject of optimization, but there is a trade-off to be done between a faster access to playing the videos and the complexity of the buffering methods.

At access time, the main adjustment to be done is replacing the original flash player. Since the archived video file is no longer streamed, the original player in the page cannot be used on the archived version. The HTML container needs to be updated accordingly:

```
<div class="container">
  <noscript>
    <p class="flashError">
      JavaScript not active !
    </p>
  </noscript>

  <embed width="520" height="344"
    flashvars="file=http://collection.europarch
ive.org/swr/20100601084708/rtmp://fc-
ondemand.swr.de/a4332/e6/foobar-video.flv
src="http://collections.europarchiv
e.org/media/player.swf"
type="application/x-shockwave-flash"
id="video6455688_plr"/>
</div>
```

Compared to the live page, in the archived version we replace the <script> and <object> elements with a specific <embed> element that contains an archive-specific player:

“http://collections.europarchiv.org/media/player.swf” and the URL of the archived video:

“http://collection.europarchiv.org/swr/20100601084708/rtmp://.../foobar-video.flv”

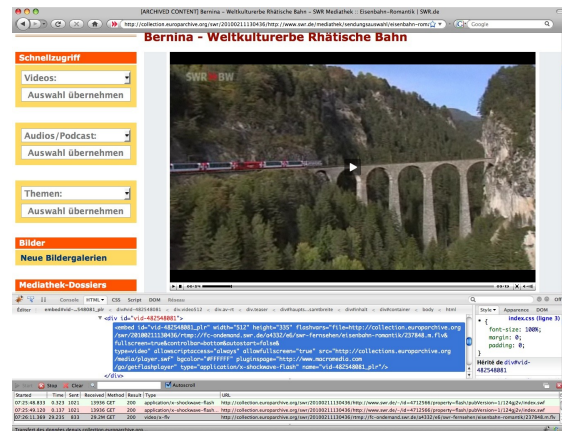


Figure 2. Archived Web site – HTTP video.

Notice that the URL of the archived video file was transformed into an HTTP URL pointing to a flv file in the archive.

The replacement of the HTML elements is done on the fly, by specific methods implemented in the access code on the server. The WARC files always keep the original version of the pages. Finding a common pattern to correctly identify and replace the player containers is still challenging, since the structure and the attributes of the <object> elements could differ from one Web site to another.

In this section we focused on the example of capturing the RTMP streaming videos, because the downloading process involves a specific streaming protocol. Nevertheless, the player replacement techniques implemented in the access code are used in the same manner also for the HTTP-downloaded videos.

3.VIDEO CAPTURE USING EXTERNAL DOWNLOADERS

As part of the new technologies for Web archiving developed in the LiWA project¹, a specific module was designed to enhance the capturing capabilities of the crawler, with regards to different multimedia content types (for an early attempt on this topic at EA see [Baly 2006]). The current version of Heritrix is mainly based on the HTTP/HTTPS protocol and it cannot treat other content transfer protocols widely used for the multimedia content (such as streaming).

The *LiWA Rich Media Capture* module² delegates the multimedia content retrieval to an external application (such as MPlayer³ or FLVStreamer⁴) that is able to handle a larger spectrum of transfer protocols.

The module is constructed as an external plugin for Heritrix. Using this approach, the identification and retrieval of streams is completely de-coupled, allowing the use of more efficient tools to analyze video and

¹http://www.liwa-project.eu/

²http://code.google.com/p/liwa-technologies/source/browse/rich-media-capture

³http://www.mplayerhq.hu

⁴http://savannah.nongnu.org/projects/flvstreamer/

audio content. At the same time, using the external tools helps in reducing the burden on the crawling process.

3.1. Architecture

The module is composed of several sub-components that communicate through messages. We use an open standard communication protocol called *Advanced Message Queuing Protocol (AMQP)*⁵.

The integration of the Rich Media Capture module with the crawler is shown in the Figure 3 and the workflow of the messages can be summarized as follows.

The plugin connected to Heritrix detects the URLs referencing streaming resources and it constructs for each one of them an AMQP message. This message is passed to a central Messaging Server. The role of the Messaging Server is to de-couple Heritrix from the clustered streaming downloaders (i.e. the external downloading tools). The Messaging Server stores the URLs in queues and when one of the streaming downloaders is available, it sends the next URL for processing.

In the software architecture of the module we identify three distinct sub modules:

- a first control module responsible for accessing the Messaging Server, starting new jobs, stopping them and sending alerts;
- a second module used for stream identification and download (here an external tool is used, such as the MPlayer);
- a third module which repacks the downloaded stream into a format recognized by the access tools (WARC writer).

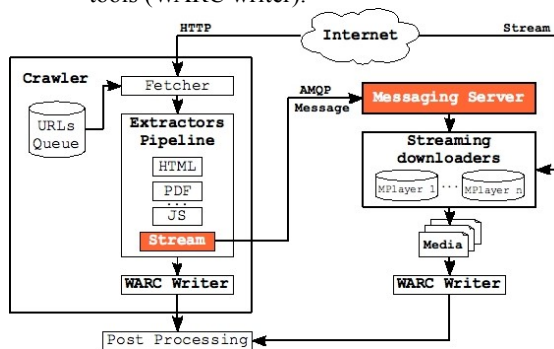


Figure 3. Streaming capture module interacting with the crawler.

When available, a streaming downloader connects to the Messaging Server to request a new streaming URL to capture. Upon receiving the new URL, an initial analysis is done in order to detect some parameters, among others the type and the duration of the stream. Of course, if the stream is live, a fixed configurable duration may be chosen.

After a successful identification the actual download starts. The control module generates a job which is passed to the MPlayer along with safeguards to ensure that the download will not take longer than the initial estimation.

After a successful capture, the last step consists in wrapping the captured stream into a WARC file, which is moved afterwards to the final storage.

3.2. Optimizations

The main issues emerging from the initial tests were related to the synchronization between the crawler and the external capture module. In the case of a large video collection hosted on the Web site, a sequential download of each video would definitely take longer than the crawling process of the text pages. The crawler would therefore have to wait for the external module to finish the video download.

A speed-up of the video capture process can be indeed obtained by multiplying the number of downloaders. On the other hand, parallelizing this process would be limited by the maximum bandwidth available at the streaming server.

An other solution for managing video downloaders is to completely de-couple the video capture module from the crawler and launch it in the post processing phase. That implies the replacement of the crawler plugin with a log reader and an independent manager for the video downloaders.

The advantages of this approach (used at EA for instance) are:

- a global view on the total number of video URIs
- a better management of the resources (number of video downloaders sharing the bandwidth)

The main drawback of this method is related to the incoherencies that might appear between the crawl time of the Web site and the video capture in the post processing phase:

- some video content might disappear (during one or two days delay)
- the video download is blocked waiting for the end of the crawl

Therefore, there is a trade-off to be done when managing the video downloading, between: shortening the time for the complete download, error handling (for video contents served by slow servers), and optimizing the total bandwidth used by multiple downloaders.

4. CONCLUSION AND PERSPECTIVES

As can be seen, it is difficult to design a general solution for dealing with all the Web sites hosting video content. Based on the general methods presented in this paper, the harvesting technique should be adapted to each particular case. The crawl engineering effort needed to adapt the tools is generally dependent on the complexity of the Web site.

The remaining work in this area folds under three main areas:

- Scaling video capture, likely by decoupling it from the crawl and better handling of the numerous errors and interruption that video servers in general, and streaming servers in particular, generate.
- Improving automatic detection of obfuscated links, and following them with specific rules

⁵<http://www.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol>

(allowing off-domains, detection in other file formats than html etc.)

- Developing a generic access and presentation method. This entails, detecting players characteristics automatically to replace them in a generic manner, managing better access and serving options for large files etc.

5.ACKNOWLEDGEMENT

This work is partly funded by the European Commission under LiWA (IST 216267)

6.REFERENCES

[Baly 2006] Baly, N., & Sauvin, F. (2006). Archiving Streaming Media on the Web, Proof of Concept and First Results. International Web Archiving Workshop (IWA 06), Alicante, Spain.