Noname manuscript No.
(will be inserted by the editor)

Implementation Strategies for Efficient Media Fragment Retrieval

Davy Van Deursen · Wim Van Lancker · Erik Mannens · Rik Van de Walle

Received: date / Accepted: date

Abstract The current Web specifications such as HTML still treat video and audio resources as 'foreign' objects on the Web, especially lacking a transparent integration with current Web content. The Media Fragments URI specification is part of various efforts at W3C trying to make media a "first class citizen" on the Web. More specifically, with a Media Fragment URI, one can point to a media fragment by means of a URI, enabling people to identify, share, link, and consume media fragments in a standardized way. In this paper, we propose and evaluate a number of implementation strategies for Media Fragments. Additionally, we present two optimized implementation strategies: a Media Fragment Translation Service allowing to keep existing Web infrastructure such as Web servers and proxies and a fully integrated Media Fragments URI server that is independent of underlying media formats. Finally, we show how multiple bit rate media delivery can be deployed in a Media Fragments aware environment, using our Media Fragments URI server.

Keywords Format-independent · Implementation · Media Fragments URI · NinSuna

1 Introduction

Today, audio and video resources are omnipresent within the World Wide Web (WWW) and are used for various applications such as advertising, enterprise collaboration, entertainment, and product reviews. However, considering the current Web specifications such as HTML, video and audio resources are still treated as 'foreign' objects on the Web, especially lacking a transparent integration with current Web content. Therefore, various efforts at W3C, such as HTML5[1] and the Video in the Web activity[2], try

D. Van Deursen, W. Van Lancker, E. Mannens, R. Van de Walle
Ghent University – IBBT
Department of Electronics and Information Systems – Multimedia Lab
Gaston Crommenlaan 8, bus 201, 9050 Ledeberg-Ghent, Belgium
Tel.: +32-9-3314893
Fax: +32-9-3314896
E-mail: davy.vandeursen@ugent.be

[1] http://www.w3.org/TR/html5/
[2] http://www.w3.org/2008/WebVideo/Activity.html

to make media a "first class citizen" on the Web, enabling people to create, identify, navigate, search, link, consume, and distribute media resources [12].

In this paper, we focus on a specific part of W3C's Video in the Web activity: Media Fragment URIs. The mission of the W3C Media Fragments Working Group[3] (MFWG) is to address media fragments on the Web using Uniform Resource Identifiers (URIs) [4]. Following a requirement phase [15], three different axes have been identified for media fragments: temporal (i.e. a time range), spatial (i.e. a spatial region), and track (i.e. a track contained in the media resource) [16]. Furthermore, media fragments can be identified by name, which is a semantic replacement for addressing any range along the aforementioned three axes.

In this paper, we present a number of implementation strategies for the efficient retrieval of Media Fragments, based on the protocols described in the Media Fragments 1.0 specification [16]. For each strategy, we provide implementation details and elaborate on the pro's and cons of each approach. Further, we present two optimized approaches to serve and retrieve Media Fragments. More specifically, an approach is presented where the existing Web infrastructure is kept as much as possible to handle Media Fragments, as well as a fully integrated Media Fragments server that is independent of underlying media fragments. Additionally, we elaborate on an application scenario where Media Fragments URIs are meaningful: multiple bit rate media delivery in a Media Fragments aware environment.

2 Media Fragments 1.0 Specification

In this section, we present a brief overview of the Media Fragments 1.0 specification. The specification falls apart into two big parts: the specification of the URI syntax and the protocol for the retrieval of Media Fragment URIs.

2.1 URI Syntax

As introduced above, three fragment axes exist: temporal (i.e., a time range), spatial (i.e., a spatial region), and track (i.e., one or more specific tracks). These axes are also illustrated in Fig. 1. Note that the three fragment axes can also be combined (e.g., a spatial region over a time range for a given track). The specification defines the syntax for $mediafrag$ within the URI $protocol://path/mediafile\#mediafrag$. For brevity, we give for each axis an example; the full syntactical details can be found in the specification [16].

- temporal: `http://example.com/media.mp4#t=10,30` identifies the time range [10s,30s[of media.mp4;
- spatial: `http://example.com/media.mp4#xywh=10,10,50,50` identifies a spatial region of media.mp4 with as size 50x50 pixels and with as upper left coordinate (10px,10px);
- track: `http://example.com/media.mp4#track=video&track=audio_fr` identifies the video and french audio track of media.mp4.

[3] `http://www.w3.org/2008/WebVideo/Fragments/`

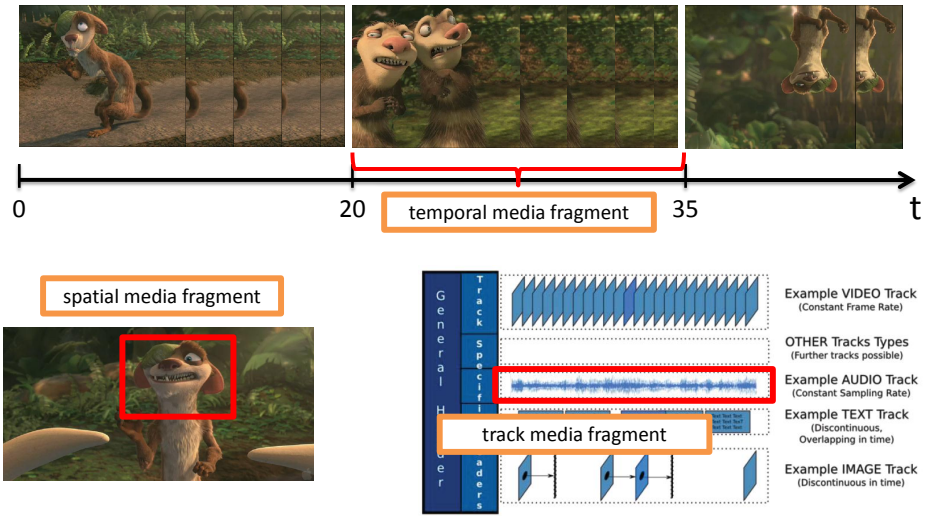

Fig. 1 Visualization of media fragments.

2.2 Media Fragment Retrieval

HTML fragments are typically intepreted by the User Agent (UA) after downloading the complete HTML document. However, applying the same approach for media fragments would not be efficient in terms of bandwidth because media resources are characterized by a large size (compared to HTML documents). Therefore, when retrieving media fragments, our goal is to retrieve only the bytes that correspond to the requested media fragment. For example, if we request the last minute of a 3 hours video, then we only need to download the bytes corresponding to the last minute.

2.2.1 URI Fragments vs. URI Queries

For media fragment addressing, both approaches (URI query and URI fragment) are useful. URI fragments ('#') point at so-called 'secondary' resources according to the URI specification [2]. Per definition, such a secondary resource is a sub-part of the primary resource. From a physical point of view, this means that media fragments (when extracted from their primary resource) should be expressible in terms of byte ranges. In other words, there must exist a mapping between a media fragment and one or multiple byte ranges in the primary resource.

On the other hand, using URI queries ('?') to identify a fragment results in a completely new resource, which means that there is no notion of a primary resource. Since URI queries result in new resources, there is also no restriction regarding the bytes used to represent the fragment. Thus, in case a media fragment needs to be extracted/re-encoded but it cannot be expressed in terms of byte ranges pointing to the primary resource, one could use a URI query instead of a URI fragment. For example, spatial media fragments are very hard to extract given the limitations of the current coding formats (see further).

In the remainder of this paper, the term 'media fragment' corresponds to a media fragment identified by a URI fragment.

2.2.2 The Role of Media Formats

The extraction of media fragments is dependent on the underlying media format. Moreover, media formats introduce restrictions regarding possibilities to extract media fragments along different axes. Dependent on the fragment axis, there are different requirements regarding the extraction of media fragments (i.e., expressing media fragments in terms of byte ranges and thus obtaining them without transcoding operations).

- Temporal: random access points need to occur in a regular way. Random access refers to the ability of a decoder to start decoding at a point in a media resource other than at the beginning and to recover an exact or approximate representation of the decoded resource [3]. In case of video, random access points typically correspond to intra-coded pictures.
- Spatial: independently coded spatial regions are required. More specifically, (mostly rectangular) regions of the picture need to be coded independently from each other. For instance, Regions Of Interest (ROIs) and a background are coded independently, which results in the possibility to extract these ROIs.
- Track: the way tracks are extracted from a media resource is dependent on the container format. In contrast to temporal and spatial fragment extraction, tracks are not 'encoded' but 'encapsulated' within a container format and can thus always be extracted without low-level transcoding operations. Based on the headers of the container format, it is possible to locate the proper byte ranges corresponding to the desired track. However, since tracks within a media resource are usually interleaved (e.g., interleaved audio and video, typically with intervals of 0.5s-1s), the number of byte ranges corresponding to one track becomes very high (e.g., a 30 minutes video track interleaved with a 1s interval is represented by 1800 byte ranges).

The MFWG made an overview table[4] indicating which media formats were compliant with the above requirements. Overall, we can state that current media format combinations (i.e., container + coding format) comply to the temporal and track fragment restrictions, but not to the spatial fragment restrictions. Therefore, it was decided not to retrieve spatial media fragments, but to always download the full resource and to interpret the spatial fragment locally. Also, in case of track fragments, special arrangements need to be made due to the possible occurrence of a huge amount of byte ranges.

3 Efficient Media Fragment Retrieval over HTTP

The current Web infrastructure, based on the HTTP protocol, is not aware of addressing methods others than bytes to point to a portion of a media resource. Therefore, in order to implement and deploy a system able to deal with Media Fragment URIs, the key requirement is to have a module that is able to translate media fragments (i.e., expressed in time or tracks) into fragments expressed in terms of bytes (i.e., byte

[4] `http://www.w3.org/2008/WebVideo/Fragments/WD-media-fragments-reqs/#fitness-table`

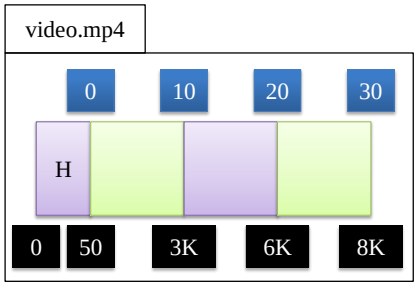

Fig. 2 Schematic representation of a media resource. The numbers at the top and bottom correspond to the display time and byte offset respectively. The 'H' represents the bytes corresponding to codec setup information.

ranges) [17]. As we will elaborate on in the next subsections, the main difference between the different media fragment retrieval strategies is the location of this translation module (i.e., at the UA or at the server).

In Fig. 2, a schematic representation of media resource is given, indicating the link between its byte offsets and the temporal fragment axis (expressed in terms of seconds). As discussed in Sect. 2.2.2, random access points occur in a regular way (i.e., at 0, 10, and 20 seconds). It is important to note that the link between byte offsets and media fragment axes is dependent on the coding and container formats. The latter implies that fragment translation modules are dependent on media formats. In Sect. 5, we will show how this format dependency can be solved when the translation module is located at the server.

As specified in [2], fragment identifiers are separated from the rest of the URI prior to a dereference. In other words, they are not sent to the server and thus the identifying information within a fragment needs to be interpreted by the UA. Applying this to Media Fragment URIs, UAs must be able to parse and interpret media fragment identifiers. Also, if UAs need help to perform the mapping between media fragments and byte ranges, the media fragment identifiers need to be communicated in some way to a media fragments aware server or service in the network. Therefore, the MFWG recommends a protocol for retrieval of media fragments over HTTP. More specifically, a number of new HTTP headers was developed, allowing to provide the media fragment information within an HTTP request. The details of the exact syntax can be found in the specification [16]; a number of examples of these new headers are provided in the next subsections.

The media resource depicted in Fig. 2 will be used as example during the elaboration on the different media fragment retrieval strategies. Given this media resource, we illustrate the retrieval of the Media Fragment URI `http://foo.com/media.mp4#t=11,19` for each strategy and discuss their pro's and cons.

3.1 Retrieval without Fragment-to-Byte Range Mapping

The most trivial way to retrieve media fragments is by downloading the full media resource and do the fragment interpretation locally at the UA. Thus, retrieving seconds 11 until 19 from media.mp4 (Fig. 2) consists of the following steps:

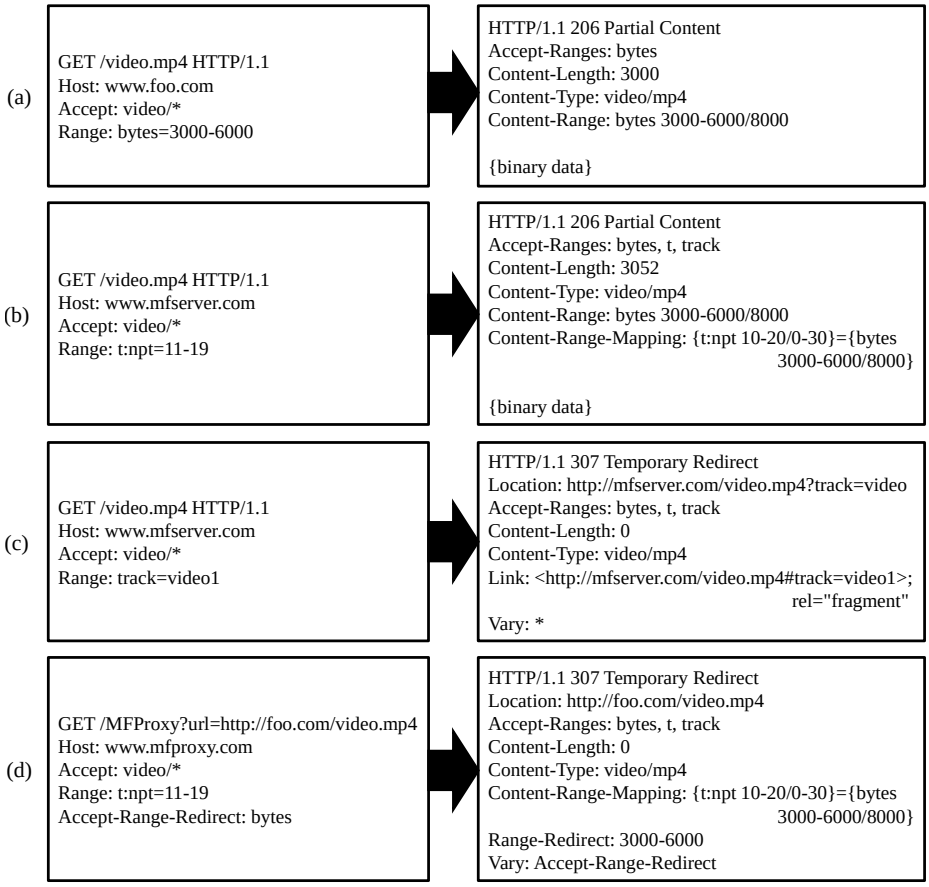

Fig. 3 Different HTTP request and response scenarios.

1. the UA downloads the full media resource (media.mp4) and initializes the decoding pipeline in the media player;
2. the UA interprets the media fragment identifier, seeks to the position of 11s, and starts the playback of the media fragment.

Note that this approach is comparable to the way fragment identifiers work with HTML documents. The advantages of this approach can be formulated as follows.

- The UA does not need the mapping between media fragments and byte ranges since it requests the full media resource.
- The media resource can be served on any regular HTTP Web server.

However, disadvantages of retrieving the full media resource are as follows:

- this approach is highly inefficient in terms of bandwidth cost, especially because media resources are typically represented by a large amount of bytes;
- since the full resource needs to be downloaded, the UA can only render the media fragment when all the bytes before the start of the fragment are downloaded.

Overall, this approach is very simple to implement, but becomes unfeasible when dealing with large media resources. To illustrated this simple strategy, we created a Media Fragments player[5], implementing the retrieval and rendering of Media Fragment URIs.

3.2 Fragment-to-Byte Range Mapping Calculation at the UA

If the UA is able to perform the translation between the media fragment identifier and byte ranges, it can optimize the retrieval of media fragments by only downloading the bytes corresponding to the fragment identifier. The latter can be achieved by performing an HTTP byte range request, as illustrated in Fig. 3(a). As already mentioned above, the translation between media fragments and their corresponding byte ranges is dependent on the media format. Generally, two approaches exist to perform a 'remote' translation, dependent on the organization of the container format. Note that 'remote' indicates that the mapping is calculated without having the full media resource at our disposal.

- *Index interpretation*: when the underlying container format of the media resource supports a full index providing a complete mapping of time and byte-offsets, then only the first couple of bytes corresponding to the index need to be downloaded. Subsequently, the index is interpreted in order to calculate the mapping between media fragments and byte ranges. The latter is dependent of the container format since different container formats use different structures to represent the index. Examples of container formats providing support for such a full index are MP4 and AVI.
- *Bisectional search over HTTP*: when no full index is provided at the beginning of the media resource, the proper byte positions need to found for a given media fragment identifier. This is obtained by applying a bisectional search over HTTP. More specifically, the UA starts by guessing which byte position corresponds to a given temporal position. Subsequently, these bytes are retrieved and interpreted. If the byte position is too high/low, another guess is made in the right direction until the correct byte offset is found. It is clear that this method is less efficient in terms of HTTP round-trips than the first method; but when no index is available, this method is still more efficient than downloading the full resource (Sect. 3.1). For instance, for old Ogg files, this approach is followed by applying a bisection search algorithm over the Ogg pages.

Note that, in case the mapping calculation fails (e.g., the UA is not aware of the underlying container format) or the mapping results in too many byte ranges (e.g., in case of interleaved tracks as discussed in Sect. 2.2.2), the UA can decide to download the full media resource and thus falls back to the approach discussed in Sect. 3.1.

When the UA is able to perform the mapping calculation, the retrieval of `media.mp4#t=11,19` corresponds to the following steps:

1. the UA interprets the media fragment identifier and performs the mapping calculation (by MP4-specific index interpretation in this example);
2. since the temporal range corresponds to the byte range 3000-6000, the UA sends an HTTP byte range request to retrieve the corresponding bytes (see Fig. 3(a));

[5] Available at `http://ninsuna.elis.ugent.be/MediaFragmentsPlayer`.

3. the server returns the requested bytes and the UA can start playing the media fragment.

The advantages of this approach are as follows:

- only the bytes necessary to play the media fragment are retrieved;
- an HTTP 1.1-compliant Web server supporting byte range requests (e.g., Apache) is enough to serve media fragments;
- media fragments can be cached by existing HTTP caches, since they are retrieved through regular HTTP byte range requests.

The disadvantage of this approach is that remote fragment-to-byte range mapping is dependent on the underlying media formats. This means that the UA has to be extended when new media formats need to be supported. Further, this approach requires at least two HTTP round-trips for obtaining the media fragment.

Currently, there are no Media Fragments 1.0 compliant UAs implementing this approach. However, there are media players that follow the same principle without having explicit support for the Media Fragments URI syntax. For instance, the Apple QuickTime player implements the index interpretation algorithm as explained above for MP4 media resources. More specifically, when a user seeks to another temporal position during playback of a media resource served through HTTP, the QuickTime player requests the proper bytes corresponding to the new temporal position, based on the index present in the MP4 header.

3.3 Fragment-to-Byte Range Mapping Calculation at the Server

When the server is enhanced with a translation module able to perform the fragment-to-byte range mapping calculation, it can answer media fragment requests from UAs in terms of seconds. Therefore, the HTTP Range request header needs to be extended with new units, i.e., time and track. This way, the UA can directly submit the media fragment identifier information within an HTTP request. Further, a new HTTP response header is necessary (i.e., Content-Range-Mapping), indicating how accurate the server was able to extract the requested media fragment [16]. For instance, temporal fragments are extracted based on the random access point borders (see Sect. 2.2.2), which do not necessarily correspond to the requested time points.

The retrieval of `media.mp4#t=11,19` corresponds then to the following steps.

1. The UA interprets the media fragment identifier and constructs an HTTP request containing a Range header with a time unit (see Fig. 3(b)).
2. The server performs the mapping calculation between the requested temporal range (i.e., 11-19 seconds) and byte ranges. Since the random access points of the media resource media.mp4 are located at 10s and 20s (see Fig. 2), the corresponding byte range is 3000-6000 bytes. Thus, the server creates an HTTP response message with in the body bytes 3000-6000 of media.mp4. Additionally, the server indicates by means of the Content-Range-Mapping header that the returned bytes do not correspond with the requested time range (i.e., 11-19s), but to the time range 10-20s (see Fig. 3(b)).
3. the UA interprets the HTTP response message, loads the received bytes, and plays the media fragments.

If the server is unable to perform the mapping calculation (e.g., the server does not support the underlying container format), it can return the full media resource. Additionally, if the mapping calculation results in too many byte ranges (e.g., interleaved tracks), the server can decide to redirect the UA to another resource that represents the requested media fragment (by means of a URI query). For instance, such a resource can be the same Media Fragments URI, but instead of using the '#' symbol, the '?' symbol is used. This way, the server does not have to communicate the list of byte ranges corresponding to the media fragment. An example of such a scenario is depicted in Fig. 3(c).
The advantages of this approach can be formulated as follows.

- the UA only needs to be able to interpret media fragments, there is no need to perform a mapping calculation between media fragments and byte ranges. Thus, only a small number of modifications are necessary for existing media players in order to support this approach.
- a media fragment is always retrieved within one HTTP round-trip.
- the mapping module is less complex than the module needed at the UA (see previous subsection) because the server has the full media resource at its disposal (i.e., no remote mapping calculation is necessary).

On the other hand, there are also a number of disadvantages.

- A compliant HTTP 1.1 Web server is not enough anymore to serve media fragments. More specifically, the server needs to be able to interpret the Media Fragment specific HTTP extensions (i.e., new Range header units and HTTP response headers). Additionally, the server needs to dispose of a fragment translation module. The latter is format-specific which means that the server can only serve media fragments compliant to its supported media formats.
- Existing HTTP caches are not able to cache media fragments with this approach, since they are unaware of the new HTTP Range header units. Only specialized media caches, aware of these new units, are able to cache media fragments retrieved using this approach.

We implemented a Media Fragments-aware server, which will be discussed in detail in Sect. 5.

3.4 Evaluation

An overview of the three different approaches for media fragment retrieval over HTTP is shown in Table 1. In this table, the three approaches are compared to each other according to five criteria. As already discussed above, the approach where the full media resource is retrieved is not feasible for large media resources (i.e., the bandwidth cost and latency are too high), despite the minimal extensions needed within the current Web infrastructure.

Comparing the mapping calculation at the UA and at the server, both approaches have their own pro's and cons. Server-side mapping calculation is the most efficient approach in terms of bandwidth cost and latency, but requires extensions for existing Web caches and servers. UA-side mapping calculation requires a large implementation cost at the UA, and the mapping calculation itself may introduce an additional latency.

Table 1 Evaluation of the different media fragment retrieval strategies.

	Full resource retrieval	Mapping calculation at UA	Mapping calculation at server
Bandwidth cost	full resource	media fragment + portions needed for mapping	media fragment
Latency	time to download bytes before start of the fragment	time needed for the mapping calculation	no extra latency introduced
UA extensions	MF interpretation and rendering	MF interpretation, remote mapping module, and rendering	MF interpretation and rendering
Cache extensions	none	none	new Range units
Server extensions	none	none	MF request, mapping, module, and response

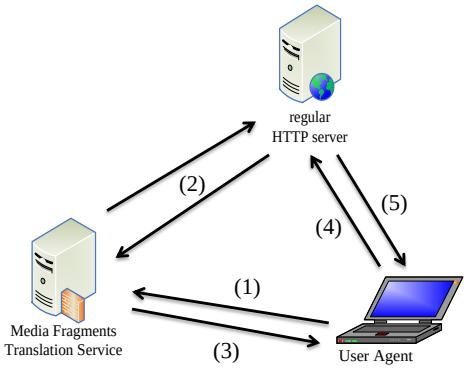

Fig. 4 Retrieving media fragments with the help from a Media Fragments Translation Service.

In the next sections, we propose two optimized approaches (i.e., one for UA-side and one for server-side mapping calculation). More specifically, we present the following (orthogonal) contributions:

- *Media Fragments translation service*: a service assisting UAs to perform their mapping calculation (see Sect. 4);
- *format-independent Media Fragments server*: a Media Fragments-aware server with a mapping calculation module that is independent of media formats (see Sect. 5).

4 Media Fragments Translation Service

The main reason why we introduce a Media Fragments Translation Service (MFTS) is to preserve as much as possible the existing Web infrastructure. More specifically, our goal is that regular HTTP servers can serve media fragments, existing HTTP caches are able to cache media fragments, and UAs only require a minimal extension for media fragments retrieval (i.e., the minimal extension comes down to just parsing and interpretation of Media Fragments URIs).

The role of the MFTS is illustrated in Fig. 4. Suppose we want to retrieve the media fragment `media.mp4#t=11,19` (see Fig. 2), the following steps are taken:

(1) The UA parses and interprets the Media Fragment URI. Since the media is served by a regular HTTP server, fragments can only be retrieved in terms of byte ranges. However, there is no media fragments translation module available at the UA. Therefore, the UA makes use of the MFTS by asking which byte ranges correspond to the temporal range 11-19 seconds. The corresponding HTTP request message is depicted in Fig. 3(d).
(2) The MFTS interprets the request from the UA and calculates the mapping between the media fragment and its byte ranges. The latter is achieved in the same way as discussed in Sect. 3.2, using so-called remote mapping calculation algorithms.
(3) When the MFTS has found the mapping between the media fragment and its byte ranges (i.e., 3000-6000 bytes), it constructs an HTTP redirect response message (see Fig. 3(d)) indicating the relation between the media fragment the UA requested and its location in terms of byte ranges.
(4) Since the UA now does know which byte ranges correspond to the media fragment, it can retrieve the media fragment using a regular HTTP byte range request (see Fig. 3(a)).
(5) Finally, the Web server returns the requested bytes, which correspond to the media fragment. The UA loads these bytes and can start playing the media fragment.

When the MFTS is unable to calculate the mapping (e.g., the MFTS is unaware of the underlying container format) or the mapping results in too many byte ranges (e.g., in case of interleaved tracks as discussed in Sect. 2.2.2), the MFTS redirects the UA to the full media resource.

We implemented a MFTS in order to demonstrate the feasibility of this approach. The MFTS is available at `http://ninsuna.elis.ugent.be/MFProxy` and can be used as follows:

```
GET /MFProxy?url=<MediaResourceURI>
Host: ninsuna.elis.ugent.be
Accept: video/*
Range: <Range>
Accept-Range-Redirect: bytes
```

Both time and track units are supported in the HTTP Range header. Note that for the moment, the MFTS only supports MP4 [5] and Ogg [11] media resources. The response times of the MFTS heavily depend on

- the connection between the MFTS and the HTTP Web server: a slow link will naturally cause a higher response time;
- the remote mapping calculation algorithm (dependent on the underlying media format): index interpretation (e.g., in case of MP4) is more efficient than bisectional search over HTTP (e.g., in case of Ogg) (see Sect. 3.2).

Given such a MFTS, every media resource on the Web can be served through a Media Fragment URIs, on condition that the MFTS supports the underlying container format. Moreover, the MFTS could also be used as a kind of media resource information service, not only providing the mapping between media fragments and byte ranges, but also other information such as available tracks, duration, or bit rate.

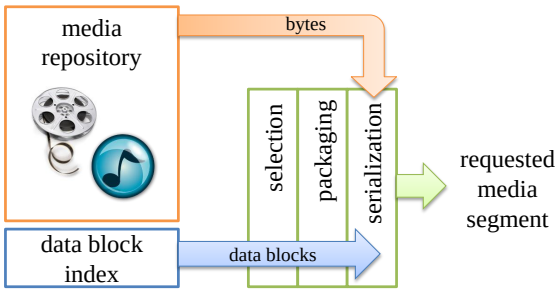

Fig. 5 High-level overview of media segment extraction inside NinSuna.

5 NinSuna: a Format-independent Media Fragments Server

In the previous section, our goal was to preserve as much as possible the existing Web infrastructure with the introduction of media fragments. In this section however, we elaborate on the architecture of specialized media fragment servers. Moreover, we propose an architecture that is independent of the underlying media formats and that is able to efficiently interpret standardized Media Fragment requests.

The NinSuna platform, which the authors introduced in [19], is a format-independent media delivery platform that is able to perform high-level selection operations such as fragment extraction. Moreover, as elaborated on in [20], NinSuna makes use of a format-independent packaging technique (i.e., to encapsulate media into a container format), independent of the high-level selection operations. In the next subsections, we show how Media Fragments can be served through NinSuna, with at its core a format-independent selection and packaging engine.

5.1 Functioning Overview

A high-level overview of the functioning of NinSuna[6] is depicted in Fig. 5. One of the keys in the design of our NinSuna platform is the datablock-based index for the media repository. This index is based on a model for media resources describing the high-level structure of media resources in terms of Random Access Units (RAUs)[7] and corresponding data blocks [18]. More specifically, for each track of the media resource, we describe the list of RAUs. Subsequently, we describe each RAU in terms of data blocks, containing information such as display time and byte range. For a video track, a data block typically corresponds to a video frame or slice. By creating a format-independent index for every media resource in the repository, we can obtain a format-independent selection and packaging core.

The datablock-based index is based on Semantic Web technologies. More specifically, our model for media resources is implemented in the Web Ontology Language (OWL, [9]); thus the instances of the model (i.e., the index) are represented in the Resource Description Format (RDF, [8]). Additionally, data blocks are retrieved by means of SPARQL Protocol And Query Language (SPARQL, [13]) queries [18].

[6] `http://ninsuna.elis.ugent.be`

[7] Each RAU starts with a random access point and ends just before the next one.

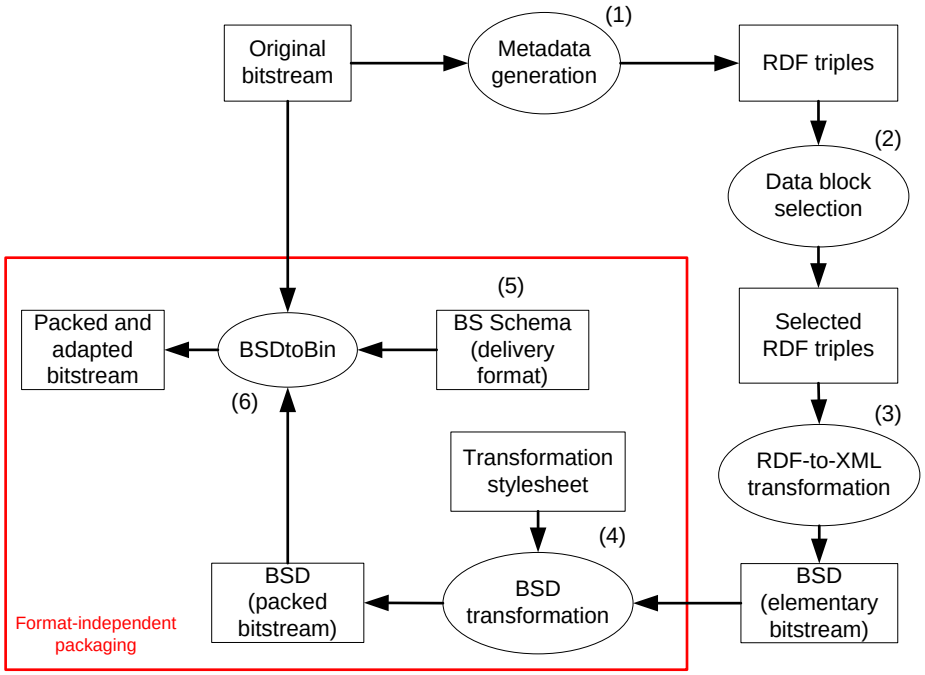

Fig. 6 The general workflow within NinSuna.

The general workflow of NinSuna is depicted in Fig. 6. If media resources need to be selected and packaged with our proposed method, metadata instances compliant to our model need to be generated during the metadata generation step (1); this way, the datablock-based index is created in the form of RDF triples. The requested parts of the media streams are obtained during the data block selection step (2), where RDF graphs describing data blocks are queried using SPARQL. Based on the selected data blocks, a simple RDF-to-XML transformation is performed (3). The result of this transformation is an XML description of the selected datablocks, called a Bitstream Syntax Description (BSD). The latter can be used to create a packaged version of the adapted media bitstream. The classes and properties defined in our model, needed for the packaging process, are mapped to XML elements and attributes respectively.

The actual packaging process starts with the transformation of the BSD representing (part of) the elementary media bitstream (4). The resulting BSD represents an adapted and packaged media bitstream. The obtained BSD is compliant with MPEG-B BSDL [7], which implies that the BSDL framework can be used for further processing. The BSD transformation can be implemented using XSLT or STX, which enables the use of a format-independent transformation engine. Additionally, a Bitstream Syntax Schema (BS Schema, [7]) needs to be created, describing the high-level structures and syntax elements of the packaging format (5). Finally, the adapted and packaged media bitstream is created using BSDL's format-independent BSDtoBin parser [7], based on the BSD representing the adapted and packaged media bitstream, the BS Schema describing the delivery format, and the original media bitstream (6).

Thus, given a media repository where each media resource is indexed according to the datablock-based index, creating a media segment within NinSuna requires the following steps (see also Fig. 5).

1. For each requested track and possibly given time range, the proper data blocks are selected. Note that the first returned data block always needs to correspond to a random access point.
2. The selected data blocks are packaged into a requested container format (e.g., MP4). For each container format, a packaging filter exists (i.e., represented by XSLT or STX), taking as input data blocks and producing an XML description corresponding to a packaged version of the selected data blocks (for instance, put a header and syntax structures around the data blocks) [20]. An example of such a filter will be shown further in this paper.
3. Finally, the selected and packaged data blocks are serialized. As already mentioned, the latter can be realized by making use of MPEG-B BSDL: based on the original media resource and an XML description of the selected and packaged data blocks, the desired media segment can be generated [1,7].

As described above, NinSuna enables the generation of a media resource corresponding to a segment of another media resource. Since the result of this operation is a new and playable media resource, we cannot call this a *media fragment*, as discussed in Sect. 2.2.1. However, we can use URI queries as an interface for the NinSuna platform. For example, `http://ninsuna/media.mp4?t=10,20` corresponds to a new media resource (with a new header) containing the frames between 10 and 20 seconds of media.mp4. Hence, in order to provide support for Media Fragment URIs in a format-independent way, additional provisions are necessary.

5.2 Extending NinSuna for Media Fragments Support

Extending a Web server with Media Fragments support is relatively easy. More specifically, if the server contains a fragment-to-byte range translation module for the requested media format, it can perform the mapping calculation and serve the corresponding bytes. However, in this scenario, the tranlation module is format-dependent: the server needs a different translation module for each media format that is being served.

Alternatively, NinSuna is designed to select and deliver media resources independent of the underlying coding formats. The latter is obtained by maintaining a generic index structure for each media resource; selection and packaging filters are based on this generic index structure (see Sect. 5.1). As a consequence, each served media resource is selected and packaged on-the-fly, which means that requested media resources are only available at runtime. Therefore, mapping media fragments to byte ranges is more complex within NinSuna in comparison to regular Web servers, since the exact byte ranges are only available at runtime. Additionally, as a matter of optimization, we do not want to generate the full media resource, if only a media fragment of that resource is requested.

In order to cope with these problems, we extended NinSuna so that it is able to efficiently interpret a media fragment request. Suppose we want to retrieve the media fragment `media.mp4#t=11,19` (see Fig. 2), the following steps are taken.

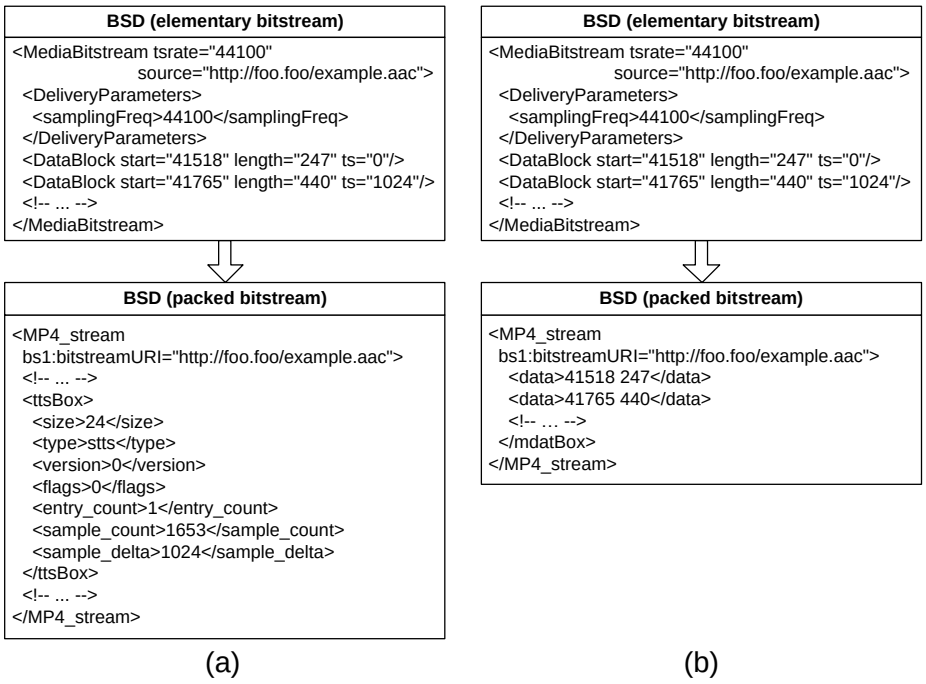

Fig. 7 Illustrating header and data mode in the MP4 packaging filter.

1. *Header generation*: the header (in this case the MP4 header) of the full media resource (i.e., media.mp4) is generated. More specifically, each packaging filter has two different modes: header and data. In header mode, the packaging filter only sends header XML structures to the serialization module. An example is depicted in Fig. 7(a), where only the header structures of an MP4 file are created. Additionally, the packaging header generates a map containing the relation between media fragments (i.e., tracks and time) and their byte offsets within the full media resource.
2. *Payload generation*: the data blocks corresponding to the requested temporal range (i.e., 11-19 seconds) are selected, packaged, and serialized. This time, the packaging filter operates in data mode and produces no header XML structures. For example, in Fig. 7(b), no header structures are generated for the MP4 file, only pointers to the data are generated.
3. *Response generation*: based on the output of the serialization module and the generated fragment-to-byte range mapping, an HTTP response is created (see Fig. 3(b)).

For efficiency reasons, the generated header and fragment-to-byte range mapping are temporally cached. Further, NinSuna allows the combination of URI queries and URI fragments. For example, consider the media fragment `media.mp4?t=10,20#t=2,4`. Retrieving the latter from NinSuna would result in an MP4 header corresponding to the media resource `media.mp4?t=10,20` and payload data corresponding to 2-4 seconds (or 12-14 seconds in media.mp4). An implementation of Media Fragment support for NinSuna and additional examples are available at `http://ninsuna.elis.ugent.be/MediaFragmentsServer`.

5.3 Application: Multiple Bit Rate Delivery

One of the scenarios where we deployed NinSuna and its server-side Media Fragments URI implementation is multiple bit rate delivery. Examples of multiple bit rate delivery on the Web are YouTube[8] or Dailymotion[9]. Typically, multiple bit rate versions of the same visual content are represented by different media resources. On the other hand, a media resource can also be characterized by a number of tracks, each combination representing the same audio-visual content but having a different bit rate. For example, consider the media resource media.mp4 having the following tracks:

1. high quality H.264/AVC video (1000 kbit/s)
2. medium quality H.264/AVC video (500 kbit/s)
3. low quality H.264/AVC video (200 kbit/s)
4. high quality AAC audio (192 kbit/s)
5. low quality AAC audio (96 kbit/s)

Each combination of one audio and one video track results in the same audio-visual representation. However, different combinations will result in different bit rates/qualities. Having multiple bit rates at our disposal allows us to deliver media to a wide range of devices (from High Definition TVs to mobile phones) and to anticipate differing network conditions [22]. The following two subsections illustrate two approaches to deliver different versions towards end-users, using Media Fragment URIs.

5.3.1 HTTP Download

In the context of the IBBT Gr@sp project[10], NinSuna was used to deploy multiple bit rate delivery over HTTP. Therefore, we ingested Apple's iTunes Movie Trailers archive[11] into our NinSuna platform so that multiple versions of each movie trailer were available for the end-user[12].

The different versions of a media resource are represented by Media Fragment URIs. More specifically, URI queries are used to indicate the desired tracks (i.e., desired bit rate). Note that we use URI queries for track selection since track extraction from a track-interleaved media resource results in a huge amount of byte ranges (see Sect. 2.2.2). Thus, the following combinations of media.mp4 could be created:

- `http://ninsuna/media.mp4?track=1;4`: high quality audio and video;
- `http://ninsuna/media.mp4?track=2;5`: medium quality audio and video;
- `http://ninsuna/media.mp4?track=3;5`: low quality audio and video;

5.3.2 HTTP Live Streaming

The downside of the approach presented in the previous section is that the quality of the media resource need to be known beforehand and cannot be changed during the media delivery. Dynamic switching of tracks is interesting when the available bandwidth

[8] `http://www.youtube.com/`
[9] `http://www.dailymotion.com/`
[10] `https://projects.ibbt.be/grasp/`
[11] `http://trailers.apple.com/`
[12] The trailers can be accessed at `http://ninsuna.elis.ugent.be/DownloadServlet/apple/grasp.html`

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=1192
http://ninsuna/LiveStreamingServlet/media.m3u8?track=1;4
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=596
http://ninsuna/LiveStreamingServlet/media.m3u8?track=2;5
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=296
http://ninsuna/LiveStreamingServlet/media.m3u8?track=3;5
#EXT-X-ENDLIST
```

Listing 1 Describing alternative versions of a media resource within an M3U8 index file.

is rapidly changing. Recently, Apple proposed *HTTP Live Streaming*, which is a new open standard for live video streaming over HTTP, currently submitted as an IETF Internet-Draft [10]. It is able to send live or prerecorded media to iPhones (and other devices). Any ordinary HTTP Web server can be used; for clients however, currently only QuickTime X (or later) and the player that comes with the iPhone OS 3.0 support HTTP Live Streaming.

In order to serve a media resource as a HTTP Live Streamable resource, the media resource needs to be segmented and stored as separate segments on the server. During this segmentation phase, an index file is created, containing a list of these media segment files and additional metadata. The format of the index file is an extension of the format used for MP3 playlists (M3U[13]), i.e., M3U8. The workflow for delivering media using HTTP Live Streaming is then as follows:

1. the UA fetches the index file of the desired media resource from the server;
2. based on the index file, the UA knows the location of the available media segment files;
3. the UA starts downloading each available media segment file in sequence.

One of the features that comes with HTTP Live streaming is dynamic stream switching. To realize this, the server maintains multiple versions of the same multimedia content. Since the UA requests small segments, the version of the multimedia content can be switched during the consumption of a particular media resource (e.g., due to varying network conditions). More specifically, the UA can decide to request the next segment from an alternative version (which can for instance be characterized by a lower bit rate).

We implemented HTTP Live streaming, inclusive dynamic stream switching, within NinSuna, using Media Fragment URIs. It is important to notice that we did not cut the media resources into different segments (as done in regular HTTP Live streaming setups). Moreover, we use Media Fragment URIs to point to these segments. Hence, it is not necessary to physically separate these segments [21]. In particular, the following extensions into NinSuna were necessary:

- MPEG-2 Transport Stream (MPEG-2 TS, [6]) packaging filter: currently, practical implementations of HTTP Live streaming UAs only support MPEG-2 TS as container format;
- M3U8 generation service: based on a requested media resource, a corresponding index file need to be generated containing pointers to the different segments.

For our media.mp4 example, the generated M3U8 index file is depicted in Listing 1. As one can see, three alternative versions are specified for the media resource, each

[13] `http://en.wikipedia.org/wiki/M3U`

```
#EXTM3U
#EXT-X-TARGETDURATION:12.0
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:11,
http://ninsuna/DownloadServlet/media.ts?track=1;4#t=0,11.3
#EXTINF:11,
http://ninsuna/DownloadServlet/media.ts?track=1;4#t=11.3,22.6
#EXTINF:12,
http://ninsuna/DownloadServlet/media.ts?track=1;4#t=22.6,34.3
#EXTINF:10,
http://ninsuna/DownloadServlet/media.ts?track=1;4#t=34.3,44.6
#EXT-X-ENDLIST
```

Listing 2 Dividing a media resource into packets by means of Media Fragment URIs.

containing a different bit rate. As discussed above, the alternative versions can be represented by means of Media Fragment URIs, by making use of URI query parameters to specify the correct tracks. However, note that we do not point to a version of the media resource itself, but to the M3U8 index file of that media resource (e.g., `media.m3u8?track=1;4`).

Given an M3U8 index file providing a number of alternative versions for a particular media resource, the UA decides which version has to be retrieved, given the constraints of its usage environment (e.g., available bandwidth, available battery power). For the chosen version, the UA retrieves the corresponding M3U8 index file; an example of the high quality version for our media.mp4 example is depicted in Listing 2. In the latter M3U8 index file, we see that the media resource is divided in temporal segments by means of Media Fragment URIs using the query parameter to indicate the correct tracks and using a fragment identifier to indicate the temporal range. More specifically, the media resource `media.ts?track=1;4` is divided in temporal ranges where each range is approximatly 12 s. Note that the boundaries of the media fragments occuring in the M3U8 index file are dependent on the random access points that occur in the media resource, which is the reason that the actual duration of each fragment differs from the target duration. The media fragments located in the M3U8 index file point to media resources packaged in an MPEG-2 Transport Stream container. These fragments can be requested and downloaded from our NinSuna server, as explained in Sect. 5.2.

6 Conclusions and Future Work

The Media Fragment Working Group contributes to making audio-visual content a first citizen on the Web by specifying a mechanism to address media fragments using URIs. In this paper, we gave an overview of the Media Fragments URI specification and provided a number of implementation strategies. We identified the key component within a Media Fragment URI implementation, i.e., a module able to translate media fragments into byte ranges. We presented solutions where this translation module was implemented at the UA or at the server. Further, we presented two optimized approaches to implement Media Fragments URIs. Firstly, we proposed the Media Fragments Translation Service, enabling to keep the existing Web infrastructure as much as possible when implementing Media Fragments URIs. Secondly, a format-independent Media Fragments server was presented. The latter implements the full server-side Me-

dia Fragments URI specification, while relying on an architecture that is independent of underlying media formats.

Future work consists of considering other protocols than HTTP for Media Fragments URI retrieval. More specifically, we need to investigate how media streaming protocols such as the Real Time Streaming Protocol (RTSP, [14]) can be mapped onto the current Media Fragments URI specification. Also, the relationship between metadata information located in the container formats and the track names used in Media Fragment URIs needs to be further investigated. For example, what are the possibilities for UAs to discover the track names and their corresponding track properties. We illustrated one approach using M3U8 index files to solve this issue, but a more general approach is more desirable.

Acknowledgements The research activities as described in this paper were funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), and the European Union.

References

1. M. Amielh and S. Devillers. Multimedia Content Adaptation with XML. In *Proceedings of 8th International Conference on Multimedia Modeling*, pages 127–145, Amsterdam, The Netherlands, November 2001.
2. T. Berners-Lee, R. Fielding, and L. Masinter. IETF RFC 3986: Uniform Resource Identifier (URI) – Generic Syntax, January 2005. Available at `http://tools.ietf.org/html/rfc3986`.
3. M. M. Hannuksela, Y.-K. Wang, and M. Gabbouj. Isolated Regions in Video Coding. *IEEE Transactions on Multimedia*, 6:259–267, April 2004.
4. M. Hausenblas, R. Troncy, Y. Raimond, and T. Bürger. Interlinking Multimedia: How to Apply Linked Data Principles to Multimedia Fragments. In *2nd Workshop on Linked Data on the Web (LDOW'09)*, Madrid, Spain, 2009.
5. ISO/IEC. Information technology – Coding of Audio, Picture, Multimedia and Hypermedia Information – Part 14: MP4 file format. ISO/IEC 14496-14:2003, December 2003.
6. ISO/IEC. Information technology – Generic coding of moving pictures and associated audio information: Systems. ISO/IEC 13818-1:2007, October 2007.
7. ISO/IEC. Information technology – MPEG systems technologies – Part 5: Bitstream Syntax Description Language. ISO/IEC 23001-5:2008, February 2008.
8. G. Klyne and J. J. Carroll, editors. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation. World Wide Web Consortium, February 2004. Available on `http://www.w3.org/TR/rdf-concepts/`.
9. D. McGuinness and F. van Harmelen, editors. *OWL Web Ontology Language: Overview*. W3C Recommendation. World Wide Web Consortium, February 2004. Available on `http://www.w3.org/TR/owl-features/`.
10. R. Pantos. HTTP Live Streaming. Available on `http://tools.ietf.org/html/draft-pantos-http-live-streaming-01`.
11. S. Pfeiffer. RFC 3533: "The Ogg Encapsulation Format Version 0," Available on `http://www.ietf.org/rfc/rfc3533.txt`.
12. S. Pfeiffer. Architecture of a Video Web - Experience with Annodex. W3C Video on the Web Workshop, 2007.
13. E. Prud'hommeaux and A. Seaborne, editors. *SPARQL Query Language for RDF*. W3C Recommendation. World Wide Web Consortium, November 2007. Available on `http://www.w3.org/TR/rdf-sparql-query/`.
14. H. Schulzrinne, A. Rao, and R. Lanphier. RFC 2326: "Real Time Streaming Protocol," Available on `http://www.ietf.org/rfc/rfc2326.txt`.
15. R. Troncy and E. Mannens, editors. *Use cases and requirements for Media Fragments*. W3C Working Draft. World Wide Web Consortium, December 2009.

16. R. Troncy, E. Mannens, S. Pfeiffer, and D. Van Deursen, editors. *Media Fragments URI 1.0*. W3C Working Draft. World Wide Web Consortium, June 2010.
17. D. Van Deursen, R. Troncy, E. Mannens, S. Pfeiffer, Y. Lafon, and R. Van de Walle. Implementing the media fragments uri specification. In *Proceedings of the 19th International World Wide Web Conference*, pages 1361–1364, Raleigh, NC, United States, April 2010.
18. D. Van Deursen, W. Van Lancker, S. De Bruyne, W. De Neve, E. Mannens, and R. Van de Walle. Format-independent and Metadata-driven Media Resource Adaptation using Semantic Web Technologies. *Multimedia Systems*, 16(2):85–104, 2010.
19. D. Van Deursen, W. Van Lancker, W. De Neve, T. Paridaens, E. Mannens, and R. Van de Walle. NinSuna: a Fully Integrated Platform for Format-independent Multimedia Content Adaptation and Delivery based on Semantic Web Technologies. *Multimedia Tools and Applications – Special Issue on Data Semantics for Multimedia Systems*, 46(2-3):371–398, January 2010.
20. D. Van Deursen, W. Van Lancker, P. Debevere, and R. Van de Walle. Format-independent Media Delivery, Applied to RTP, MP4, and Ogg. In *Proceedings of the 4^{th} International Conference on Multimedia and Ubiquitous Engineering*, Cebu, Philippines, August 2010.
21. D. Van Deursen, W. Van Lancker, and R. Van de Walle. On Media Delivery Protocols in the Web. In *Proceedings of the IEEE International Conference on Multimedia and Expo 2010*, pages 1028–1033, Singapore, July 2010.
22. A. Vetro, C. Christopoulos, and T. Ebrahimi. Universal Multimedia Access. *IEEE Signal Processing Magazine*, 20(2):16, March 2003.