1 MPEG-21 standard and XML

1.1 Overview

MPEG-21 is a framework for exchanging digital resources and meta data. Basically, it is a way to combine arbitrary binary and textual resources in a structure, and then exchanging that structure as a single unit. For example, a CD would be one such unit, with audio tracks, covers and lyrics combined in a meaningful way.

The standard is almost entirely formulated in XML - it has nothing to do with video codecs. At present time there are about 18 different parts to it, most of which have their own XML namespace. Each namespace can be considered as a 'tool' one can choose whether or not to support.

1.2 Scope

This document is meant to be an introduction to MPEG-21 for designers of XML parsers.

1.3 MPEG-21

Choosing to formulate a standard in XML can help in more ways that one. For MPEG-21 however, several shortcomings have been identified, for example speed. The big question is whether these shortcomings are unique to MPEG-21, and therefore should be addressed by MPEG, or if solutions should be sought together with the XML community.

2 MPEG-21 and XML technology

Generally one would like to preserve the XML Information Sets plus

- Rapid document traversal
- Mapping of strings to integers - working with integers is faster than strings
- Good information theoretical properties, preferly
  - instantaneous decoding
  - scalable performance
  - only compress the XML structure, model actual contents as white noise

The different parts of MPEG-21 represents different kinds of functionality, here is a walk through.

2.1 Part 2: Digital Item Declaration (DID)

Part two makes up the root of any MPEG-21 document. The 'atom' of the structure is the *'Digital' Item* element. As the first example illustrates, the DID schema allows for recursive structures, so the entire structure is of unknown depth and height.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <DIDL>
3
4    <Item>
5      <!-- add children here-->
6
7      <!-- recursive structure of unknown depth-->
8      <Item>
9        <Item>
10           ...
11       </Item>
```

```
12        <Item>
13            ...
14        </Item>
15      </Item>
16
17      <!-- add more children here-->
18    </Item>
19 </DIDL>
```

The second example shows how metadata may be included via the *Descriptor* element. Its children are typically *Statement* for text and *Component/Resource* for binary meta data.

```
1 <Descriptor>
2   <Statement mimeType="text/plain">Plain text metadata</Statement>
3 </Descriptor>
4 <Descriptor>
5   <Component>
6     <Resource mimeType="image/png">Embedded CDATA metadata</Resource>
7   </Component>
8 </Descriptor>
```

XML is also text, so XML of *any* namespace is permitted via the Statement element. In other words, MPEG-21 systems will encounter both known and unknown namespaces:

```
1 <Descriptor>
2   <Statement mimeType="text/xml">
3     <Known-XML-namespace>
4       <!-- known namespace data-->
5     </Known-XML-namespace>
6   </Statement>
7 </Descriptor>
8
9 <Descriptor>
10  <Statement mimeType="text/xml">
11    <Unknown-XML-namespace>
12      <!-- unknown namespace data-->
13    </Unknown-XML-namespace>
14  </Statement>
15 </Descriptor>
```

In addition, one can freely add new attributes to all MPEG-21 elements. In all cases meta data can be referred to instead of embedded, by applying a 'ref' attribute to the Statement and Resource elements.

```
1 <Resource mimeType="image/png" ref="http://the.url.to.the.image"/>
2 <Statement mimeType="text/plain" ref="http://the.url.to.the.text"/>
```

Requirements:

- Namespaces
- Handle unknown namespaces
  - an universal compression algorithm
  - skipping of subthrees (what cannot be understood has zero information value)
- Embedding of binary resources

- no network overhead, setup time or non-ideal protocol state
- transferring the resource is not optional
- in environments with constraints on the number of open connections one need not wait for a free connection

A very simple example of an almost valid document is given below.

```
<DIDL>

  <!-- collecting of binary resources+metadata wrapper - AKA Digital Item-->
  <Item>

    <!-- some metadata-->
    <Descriptor>
      <Statement mimeType="text/plain">
        Human readable description of grandparent element
      </Statement>
    </Descriptor>
    <Descriptor>
      <Statement mimeType="text/xml">

        <!-- known namespace data-->
        <Known-XML-namespace/>
      </Statement>
    </Descriptor>
    <Descriptor>
      <Statement mimeType="text/xml">

        <!-- unknown namespace data-->
        <Unknown-XML-namespace/>
      </Statement>
    </Descriptor>

    <!-- assotiated binary resources -->
    <Component>
      <Descriptor>
        <Statement mimeType="text/plain">Audio track 1</Statement>
      </Descriptor>

      <!-- binary resource included by URL -->
      <Resource mimeType="audio/mp3" ref="http://myURL/track1.mp3"/>
    </Component>
    <Component>
      <Descriptor>
        <Statement mimeType="text/plain">Audio track 2</Statement>
      </Descriptor>

      <!-- binary resource included by embedding as complex data -->
      <Resource mimeType="audio/mp3">Embedded CDATA A</Resource>
    </Component>

    <Component>
      <Descriptor>
        <Statement mimeType="text/plain">Album cover</Statement>
```

```
      </Descriptor>

      <!-- binary resource included by embedding as complex data -->
      <Resource mimeType="image/png">Embedded CDATA B</Resource>
    </Component>
  </Item>
</DIDL>
```

DID also defines a way to select (read: exclude) different parts of a document by the means of a Boolean algebra mechanism. The tags involved in this are Choice, Selection and Condition.

```
<!-- boolean selection mechanism
- select no more than one and no less than one (= radio buttons) -->
<Choice choice_id="UniqueIdentifier" minSelection="1" maxSelections="1">
  <Descriptor>
    <Statement mimeType="text/plain">
      A Boolean algebra mechanism for selecting (read:excluding) parts of a document
    </Statement>
  </Descriptor>

  <!-- alternative one -->
  <Selection select_id="Alternative one unique identifier">
    <Descriptor>
      <Statement mimeType="text/plain">I want track 1</Statement>
    </Descriptor>
  </Selection>

  <!-- alternative two -->
  <Selection select_id="Alternative two unique identifier">
    <Descriptor>
      <Statement mimeType="text/plain">I want track 2</Statement>
    </Descriptor>
  </Selection>
</Choice>

<!-- parts of document subject to the selection mechanism -->
<Component>

  <!-- which selections must be selected to include this part of the document?' -->
  <Condition require="Alternative one unique identifier"/>
  <Descriptor>
    <Statement mimeType="text/plain">
      Audio track 1 with imposed restriction
    </Statement>
  </Descriptor>
  <Resource mimeType="audio/mp3" ref="http://myURL/track1.mp3"/>
</Component>
<Component>

  <!-- which selections must be selected to include this part of the document?' -->
  <Condition require="Alternative two unique identifier"/>
  <Descriptor>
    <Statement mimeType="text/plain">
      Audio track 2 with imposed restriction
```

```
    </Statement>
  </Descriptor>
  <Resource mimeType="audio/mp3">Embedded CDATA A</Resource>
</Component>

<!-- parts of document NOT subject to the selection mechanism -->
<Component>
  <Descriptor>
    <Statement mimeType="text/plain">Album cover</Statement>
  </Descriptor>
  <Resource mimeType="image/png">Embedded CDATA B</Resource>
</Component>
```

This means that parts of the document will potentially never be used. Although a server would have to store the whole document, a client would only need parts of a document at one time, since the logical procedure is to first select something and then receive it. Additional requirements:

- Repeated document traversal / two way navigation
  - efficient representation / storage
  - document metrics (position and level)
- Skipping of sibling nodes (skip to 'current level - 1')
- On-demand delivery, based on some API that
  - uses Selection/Condtion rules, and/or
  - supported namespaces

In addition, part 2 of MPEG-21 requires XInclude support, but only on certain elements which have the 'id' attribute. The following example illustrates roughly how XInclude works:

```
<?xml version="1.0" encoding="UTF-8"?>
<DIDL>

  <Item id='MyCD'>

    <Item>
      <Item>
          ...
      </Item>

      <!-- circle reference to Item with id='MyCD', endpoint must have the 'id' attribute -->
      <xi:include xpointer="element(MyCD)"/>
    </Item>

    <!-- include some external document via XInclude, target must have id=" attribute-->
    <xi:include href="http://someSite.com/somedocument" xpointer="element(SomeID)">
      <xi:fallback>
        <!-- some XML that is only used if the xinclude href is not resolved! -->
      </xi:fallback>
    </xi:include>

    <!-- add more children here-->
  </Item>
</DIDL>
```

Notice that referring to external documents is allowed, and that if that document cannot be obtained, it is possible to define what should be inserted instead via the *fallback* tag. XInclude strengthens the demand for document metrics plus rapid and multi-directional traversal. It also means the system would need multiple parsers (and implementations of such) at once, as one not necessarily have control over XInclude'd documents.

2.2 Parts 4, 5 and 6: Intellectual Property Management and Protection (IPMP)

This part of the standard requires that parts of a document are protected (encrypted) and that certain access control mechanisms exist. Encryption goes hand in hand with digital fingerprinting - embedding all resources in a single XML document could be desirable for safety reasons (during transport, not neccerarily in storage). Encrypted elements would probably contain XML in normal text format that cannot be optimized.

```
1 <!-- parts of the document may be protected by encryption-->
2 <ipmpItem>
3   <SomeNormalNodes>Plain XML</SomeNormalNodes>
4
5   <SomeEncryptedNodes>Encrypted XML</SomeEncryptedNodes>
6 </ipmpItem>
```

This again means that we will need multiple parsers. Also, using document metrics one could possibly allow some kind of IPMP logic to check whether nodes within some restricted areas of the document are accessed.

2.3 Part 7: Digital Item Adaptation (DIA)

The MPEG-21 basically scripts any given resource on an atomic level, describing the resource in XML. For example, one would attach meta data in the form 'I-frame from byte 34312 of length 2300' to a video resource.

This description then undergoes a transform according to some quality of service parameters and is finally used to do the necessary modifications (e.g. trans coding), creating the desired permutation. In other words, part 7 introduces monster big nodes that in many cases are not used or the use of must be cached/looked up. Parser requirements:

- Really efficient skipping of subtrees
- Data representation ($\neq$ compression) which allows for efficient processing

XPath/XQuery support is also required.

2.4 Part 10: Digital Item Processing (DIP)

DIP's view of an MPEG-21 document is based on types, by something called an Object Map. It also has ECMAScript style scripts. All its XML requirements but one is already listed above;

- Building DOM level 3 nodes based on XPath/XQuery expressions
  - So nextEvent(), nextEvent(), getElement() ?

Use of DOM by default is in my opinion a bit overkill. It is only needed when there is an explicit need to manipulate an XML structure. For example [2],

```
function foo(arg1)
{

    choice = didDocument.getElementByID( "bitrate_choice" );
    DID. configureChoice( choice );

}
```

illustrates when DOM is NOT needed. At the very least the configureChoice method should be overloaded,

```
function foo(arg1)
{

    DID. configureChoice( "bitrate_choice" );

}
```

DOM is basically very memory expensive, and how often would one actually need to manipulate a document? All other operations, like for example saving a track from a DID, should be possible with other XML APIs. Every MPEG-21 client might as well have an MPEG-21 editor built in.

2.5 Part XX: Digital Item Streaming

MPEG is trying to define how to stream digital items. However it is not clear at this point how this will actually work.

2.5.1 Streaming

Streaming of temporal metadata makes sense, but the streaming parameters will depend completely on the metadata - it will be application spesific just like supporting streaming of for example 3GPP video. One should perhaps make a generic multichannel streaming container - DIA is supposed to include all relevant parameters. Requirements:

- Possibly base our on-demand API also on metadata temporal nature

2.5.2 Live items

Updating items in time should be possible, but raises some questions. How is the user supposed to relate to changes? What if a choice I have already answered is changed? What if the objectmap is changed? How does this work technically, do one use push or pull, do we want to update parts of an recursive structure, or simply reload? Do we known which nodes that may change?

Perhaps this issue is best handeled by DIP, possibly be defining some document or node change listener mechanism.

2.5.3 Sequenced excution of tools and incremental delivery

If we have a big Choice structure that will not fit in memory of a client, do we

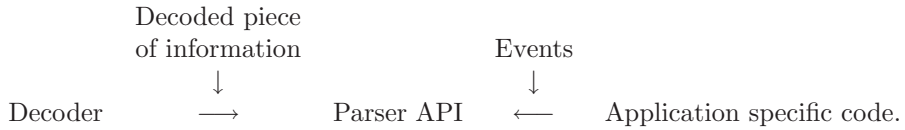
1. send the client the (at all times legal) choices (fragments) in a step by step procedure, or
2. send the client a sceleton of the whole item including components/resources, and let it request fragments as needed?

Things don't excactly get easier as a DIP can request nodes by id (id, choice_id or select_id) either. Should fragments be well-formed or not? In the first case yes, in the second probably not. This document does not relate to the first alternative, only the second.

A key question is also how big Digital Item structures actually should be, or to what extent choices should act as browsing.

2.6 Parser APIs

What functionality should be inside an API and what should not? That question is not that obious, as factory classes can have multiple implementations of the same API. The key question is perhaps, using what abstract model will people be writing code. Generally we have



At this point, it is clear that a pull-parser API utilizing some 'while' hasNext() getNext() paradigm would work very well as the number of nodes sometimes is unknown. 'for' loops like in kXML Document Object Model (DOM) are not ideal. Using pure DOM would require way too much memory, so we are left with APIs like StAX.

This means we can pause parsing at any time (hopefully without blocking the IO), and that we will be passing the parser itself as an argument in our code.

2.7 Handeling unknown namespaces

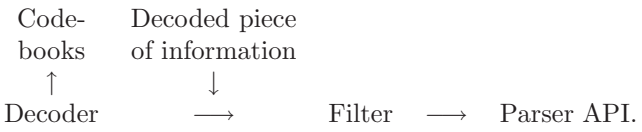
The overhead involved with handeling unknown (and therefore potentially useless) namespaces should be minimized. Say we have the following system architecture:

$$\text{Client} \longleftrightarrow \text{Application server} \longleftrightarrow \text{Database}.$$

The application server typically loads an XML document from the database and caches it in memory. It does not know which namespaces the clients supports, so it must preserve all namespaces. However, when running application specific code (as long as it is not serializing the document) unknown namespaces could be filtered out. In the client, unknown namespaces can be filtered out from the start.

2.7.1 Filtering, codebooks and integer mappings

We wish to filter out unknown tags, namespaces and attributes, which of course means one has to introduce another node in the above parser structure. If we want to stop unknown events from reaching the client code, the best way to do that is to stop decoded information from becoming events in the first place;



This would require some kind of actual filter API.

Decoding cannot work without codebooks. Thus MPEG-21 documents (due to unknown namespaces) require codebooks that are transmitted alongside the actual data, or preferly, generated dynamically. How codebooks should be organized? One could

- have codebooks in a per-namespace basis, possibly adding small namespaces to the same codebook
  - having an 'exclusive' mode in codebooks for known one-namespaced subtrees could improve speed
- have one codebook each for namespace, element and attribute tags.

We could easily construct codebooks for known namespaces, but theese would generally be based on statistics (from some document instance or schema) not neccessarily reflecting the true statistical properties of a document instance, and they would be inflexible to changes in the standard, so that will have to be discussed further.

Filtering of decoded information can be done implicity whilst generating codebooks. Simply do not allow unknown codewords (compare strings) in the codebooks and ignore non-declared codewords plus skip children if possible.

String-to-integer mappings could easily (by a some pointer array) be done at the same time.

2.8 Layered functionality which exchanges one event for another

IPMP node encryption and XInclude support means that some nodes must be processed before others. The two basically exchanges one event for a chain of others. Note that inserting logic that influences the parser is not much different from what one would do anyways when passing the parser as argument. We'd need

- More filters, but at a higher level than the previous
  - support for XPath/XQuery lookup on 'id' attributes
    - 'id' owners would be mapped on document traversal and/or looked up on demand
    - skipping may get more complicated, one in some cases still needs to scan for 'id' attributes
  - handle IPMP nodes
- Some kind of multi-parser hierarchy. For example, the root parser could have a push and pop stack of parsers.

This subsection can be realized either using some parser wrapper or some 'parser plugins', and should be seen in relationship with the next subsection.

2.9 Fragments, skipping and access to elements

Any application code would have to 'see' the entire document at once, that is, ignoring whether it is fragmented or not. Reassembly of the document (adding fragments) should be 'event-on-demand', that is, when the application code wants access to that spesific event, the underlying logic must provide just that. Note that skipping of nodes does not count as access, and that skipping can be done much more efficiently, by directly skipping in the stream, in an binary XML format that the traditional text XML.

There is no point of limiting the on-demand aspect of fragmented nodes to actual fragments. The same on-demand aspect should also be applied to any nodes that may be predicted to be skipped: Logical parts of the documents, and namespaces which may or may not be known to a client.

Note that the decoder itself must be able to cope with skipping in the stream - can encoder/decoder and fragmentation be seperated? Idealy, fragmentation would not require more than one canonical (encoded/compressed) representation of the document on the application server.

Interpreting events is just one of the first steps in producing some result. The on-demand aspect of a MPEG-21 document allows us to take that process a step further. The 'distance' to the desired end result of on-demand elements could be increased or reduced in an ideal state manner. It would for example be possible to assossiate a ready-to-use object (graphics, resources, anything) to an on-demand element. Or, going the other way, one could start moving children of such elements from memory to persistent storage. For instance, performance could be scaled as a function of available memory (there is bound to be at least some free processing time). Additionally requirements:

- A priori knowledge of document analyzis, intelligent serialization on the encoder side
- Another layer in our parser structure, to handle fragment requests
  - must know when there is a missing fragment in order to work
  - simply supply fragments when encountered

- Assign one or more objects to specific nodes (the obious candidate beeing the parser of a fragment)
  - insert events into the stream during document traversal (typically DOM),
  - add information to a already inserted slot in the event stream (typically non-DOM), or
  - some other solution

Skipping in the stream is clearly a low-level operation that cannot be implemented by some XML namespace - it will affect the decoder. At the same time we will need 'hooks' in the stream for our objects (fragments) - and those hooks will need to slip by the encoder. So fragmentation and encoding/decoding can be seperated if one introduces two new basic operations, skip(metric) and hook(size). For fragments one would also need a namespace in order to communicate how to actually obtain fragments.

2.10 Example

Here is a simple example based on one of the above XML. Note that more object hooks may be inserted all over the place. The extra Descriptor after point 3 and 5 could act as 'terminators' - this could possibly be handeled in an more elegant way.

```
<?xml version="1.0" encoding="UTF-8"?>
<DIDL>
  <Item>
    <Descriptor>
      <Statement mimeType="text\plain">
        Human readable description of grandparent element
      </Statement>
    </Descriptor>
    <Descriptor>
      <Statement mimeType="text\xml">
        <Known-XML-namespace/>
      </Statement>
    </Descriptor>
    <Descriptor>
      <Statement mimeType="text\xml">
        <Unknown-XML-namespace><!-- skip to point #1 at current level minus 3-->
          ....
        </Unknown-XML-namespace>
      </Statement>
    </Descriptor>
    <!--point #1-->
    <Choice choice_id="UniqueIdentifier" minSelection="1" maxSelections="1">
      <!-- skip to point #2 at current level minus 1-->
      <Descriptor>
        <Statement mimeType="text\plain">
          A boolean algebra mechanism for selecting (read:excluding) parts of a document
        </Statement>
      </Descriptor>
      <Selection select_id="Alternative one unique identifier">
        <Descriptor>
          <Statement mimeType="text\plain">I want track 1</Statement>
        </Descriptor>
      </Selection>
      <Selection select_id="Alternative two unique identifier">
        <Descriptor>
          <Statement mimeType="text\plain">I want track 2</Statement>
        </Descriptor>
```

```
      </Selection>
    </Choice>
    <!--point #2-->
    <Component>
      <!-- skip to point #3 at current level-->
      <Condition require="Alternative one unique identifier"/>
      <!--point #3-->
      <Descriptor>
      <!-- skip to point #4 at current level minus 1-->
      <FragmentNamespace someReassemblyMechanism="#1"/>
      <!--object hook-->
    </Component>
    <!-- point #4-->
    <Component>
      <!-- skip to point #5 at current level-->
      <Condition require="Alternative two unique identifier"/>
      <!--point #5-->
      <Descriptor>
      <!-- skip to point #6 at current level minus 1-->
      <FragmentNamespace someReassemblyMechanism="#2"/>
      <!--object hook-->
    </Component>
    <!--point #6-->
    <Component>
      <Descriptor>
        <Statement mimeType="text\plain">Album cover</Statement>
      </Descriptor>
      <Resource mimeType="image/png">Embedded CDATA B</Resource>
    </Component>
  </Item>
</DIDL>
```

Fragment number one

```
  <Statement mimeType="text\plain">Audio track 1 with imposed restriction</Statement>
</Descriptor>
<Resource mimeType="audio\mp3" ref="http://myURL/track1.mp3"/>
```

and two.

```
  <Statement mimeType="text\plain">Audio track 2 with imposed restriction</Statement>
</Descriptor>
<Resource mimeType="audio\mp3">Embedded CDATA A</Resource-->
```

3 Conclution/Summary

- We need an encoder/decoder solution that can cope with skipping in the stream at certain points, and skipping methods needs to be exposed to the application code through the parser API. Skipping is of course always possible, but is much more efficient when 'skips' are inserted in the event stream
- We need an universial compression algoritm of XML. In the interest of standards, that should be developed together with the XML community.
- Fragments should always be preceeded by a skipping point and references to fragments should be directly embedded in the event stream.
- Embedding fragments references is not much different from embedding object references, so we might as well do that.
- Use free CPU time to refine/degrade parts of the document (for instance, download fragments) and insert references to objects in the event stream. Ideally, have object hooks also 'cover' logical parts of the document.
- Performance will depend on how intelligent this can be done. Encoding is much harder that decoding, but encoding should only be done once.
- Use a multilayered structure.
- This is applicable where the XML document 'is the program' - not when XML is used for mere data-to-object mapping

4 Future work

- divide and conquer
- discussion of alternatives/existing technologies and user scenarioes, identify weaknesses
- propose parser, codebook, fragmentation API and examples
- new encoder/decoder based on [1]
- possibly seperate reading (buffering) from decoding, multiparser(multithreaded) access to the same data

References

[1] "Fast Infoset Project", https://fi.dev.java.net

[2] "Digital Item Processing", ISO/IEC FDIS 21000-10:2005(E)