216217 P2P-Next
D5.4.1d
Report on developments in lightweight interactivity (LIMO)

Contractual Date of Delivery to the CEC:	31st December 2011
Actual Date of Delivery to the CEC:	31st December 2011
Author(s):	Andrew Nicolaou, Chris Needham, Dominic Tinley, Sean O'Halpin, Theo Jones (BBC), Ingar Arntzen (NORUT), William Bamford (ULANC)
Participant(s):	
Work package:	WP5.4
Est. person months:	
Security:	PU
Nature:	P
Version:	1.0
Total number of pages:	62

Abstract:
This document describes research and development into synchronising timed metadata with live streamed video over the web using lightweight interactive media objects (LIMO) and a modified browser supporting the HTML5 audio and video elements. It was completed for M48 of the P2P-Next project.

Keyword list:
HTML5, LIMO, P2P, streaming, video

Contents

Executive summary

This document

This document describes research and development carried out in 2011 to explore synchronising timed metadata with live streamed video over the web using lightweight interactive media objects (LIMO) and a modified browser supporting the HTML5 audio and video elements. It was completed as part of the P2P-Next integrated project.

This report begins with a brief summary of the work completed during the project as a whole and a rationale for exploring synchronisation with live video in this final year. It goes on to provide an analysis of different timed metadata synchronisation techniques and draws conclusions about which approach should be taken to the specific problem of synchronising metadata with live streams in a browser.

The document then describes work required to bring current browser versions in line with the latest iteration of the HTML5 specification which is work we have gone on to undertake for Firefox 8. By implementing the HTML5 media element `startOffsetTime` attribute we show that live synchronisation becomes possible, which validates the emerging HTML5 spec and opens up new possibilities for enriching live media with all kinds of related interactive events.

In the final sections we describe demonstration code in which the timed metadata are LIMO events, although the approach taken in this work to correctly implement the current HTML5 specification can in fact be extended to work with other forms of timed data.

Objectives of WP5.4

Embedding interactivity with content is a key research challenge when moving television from the current linear channel model towards an on-demand, personalised, and interactive medium.

The aim of WP5.4 has been to investigate, experiment, and implement a lightweight solution for interactivity which we've called LIMO. The plan was that this must be generic and support basic interactivity such as timed events (eg subtitles), actions (eg clickable chapter points), and links (eg web links for further information). Furthermore it must support the P2P architecture and other types of networks.

The work outlined in this document relates to T5.4.1 'LIMO architecture and reference implementation'.

Background

In 2008 the LIMO team explored the potential ways to provide interactivity alongside video with lower barriers to entry than existing technologies available at the time, culminating in a plan to develop a system using a combination of the emerging HTML5 standard and JavaScript.

In 2009 the team developed demonstrations of how subtitles, chapters and further information panels could be added to video on demand using a combination of HTML5, JavaScript and JSON developed specifically for the project.

In 2010 the team developed two new prototypes building on the existing LIMO event types to demonstrate additional interactive features in the related areas of quizzing and polling.

In 2011 reviewing the prototypes to date led to a realisation that a significant problem which needs solving, both for this project and others providing interactivity alongside video, is that of synchronising a live audio/video stream played within a browser to the timed metadata (eg LIMO events) streamed alongside it.

The team has used the final year of LIMO work within P2P-Next to explore solutions to an issue we believe has wide reaching impact and could create a significant legacy for the interactive media strand of the project.

We expect to continue work on this beyond the end of P2P-Next itself.

Changing context over four years

This section of the document describes in more detail how the focus of the work has changed in order to fit with the shifting interactive media landscape and ensure research and development work remains beyond state of the art.

2008

In 2008 the LIMO team explored the potential ways to provide interactivity alongside video with lower barriers to entry than existing technologies available at the time, culminating in a plan to develop a system using a combination of the emerging HTML5 standard and JavaScript.

The first conclusion was that work must be guided by four principles which were developed by considering potential users and the interactive features they would expect:

- Enable maximum flexibility
- Must work for consumers and professionals
- Lower barriers to entry
- Use existing standards wherever possible

The BBC's starting point for interactivity would have been MHEG as it had at the time a large team skilled in this area. However, this was ruled out as while the BBC had extensive experience with MHEG it is not a straightforward technology to use. It is also a technology developed for broadcast television so not necessarily optimal for content delivered over the web.

Building on the guiding principles we believed it should be possible for non-professionals to create interactive events and we didn't feel MHEG would open the platform up to a broad base of content creators, including novices, using consumer tools and web developers with strong HTML skills (but weak general programming skills).

For these reasons and others described in full in our first report D5.4.1a we believed HTML5 was the best technology to use for experimental interactive media alongside video. When work on LIMO began the HTML5 specification was in its infancy but we predicted that by year four of the project the industry would have adopted HTML5.

As it turns out, while the main browsers all now support most of the HTML5 specification, there are still parts of the spec which have not been fully implemented that are essential for accurate synchronisation between live media and related interactivity within a browser environment. We come back to this point in more detail in 2011.

2009

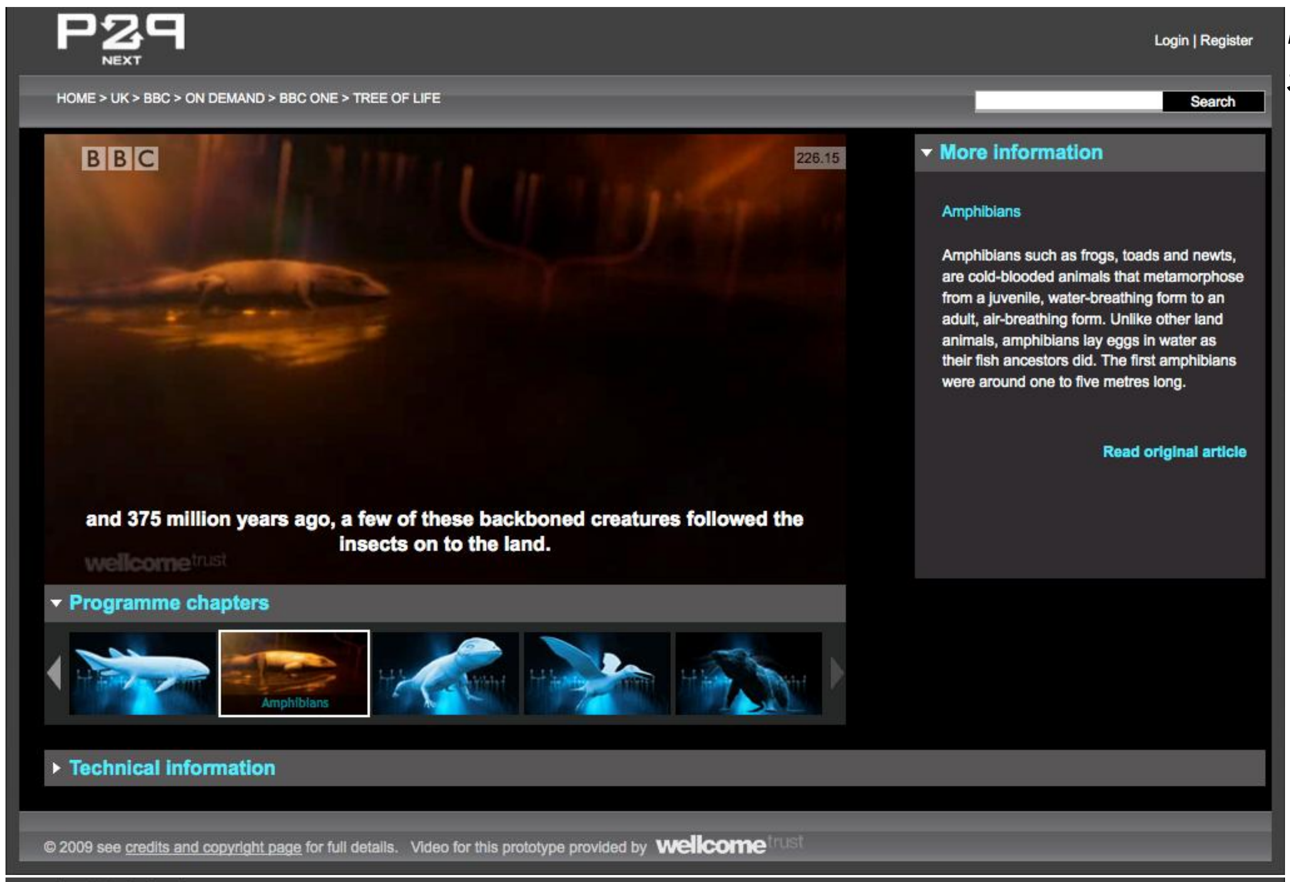
In 2009 the team developed demonstrations of how subtitles, chapters and further information panels could be added to video on demand using a combination of HTML5, JavaScript and JSON developed specifically for the project.

Prototypes used the nascent HTML5 `<video>` element to determine the time against which interactive elements (eg subtitles, chapters and information panels) should be triggered according to the in and out times recorded against them within a companion LIMO event file.

As described in full in our second report D5.4.1b the data for any LIMO presentation consists of an array of event objects each of which represents the state for a particular time eg the object below represents the subtitle to be displayed while the current time of a video is between 5.00 and 10.00 seconds:

```
{
  "start": 5.00,
  "end": 10.00,
  "sender": "video#myVideo",
  "event": "timeupdate",
  "receiver": "div#subtitle",
  "type": "HTML",
  "value": "Half past nine and we've just passed <a href='www.sheffield.com'>Sheffield</a>"
}
```

This approach can be seen in action through the Tree of Life and R&DTV demos:



Amphibians

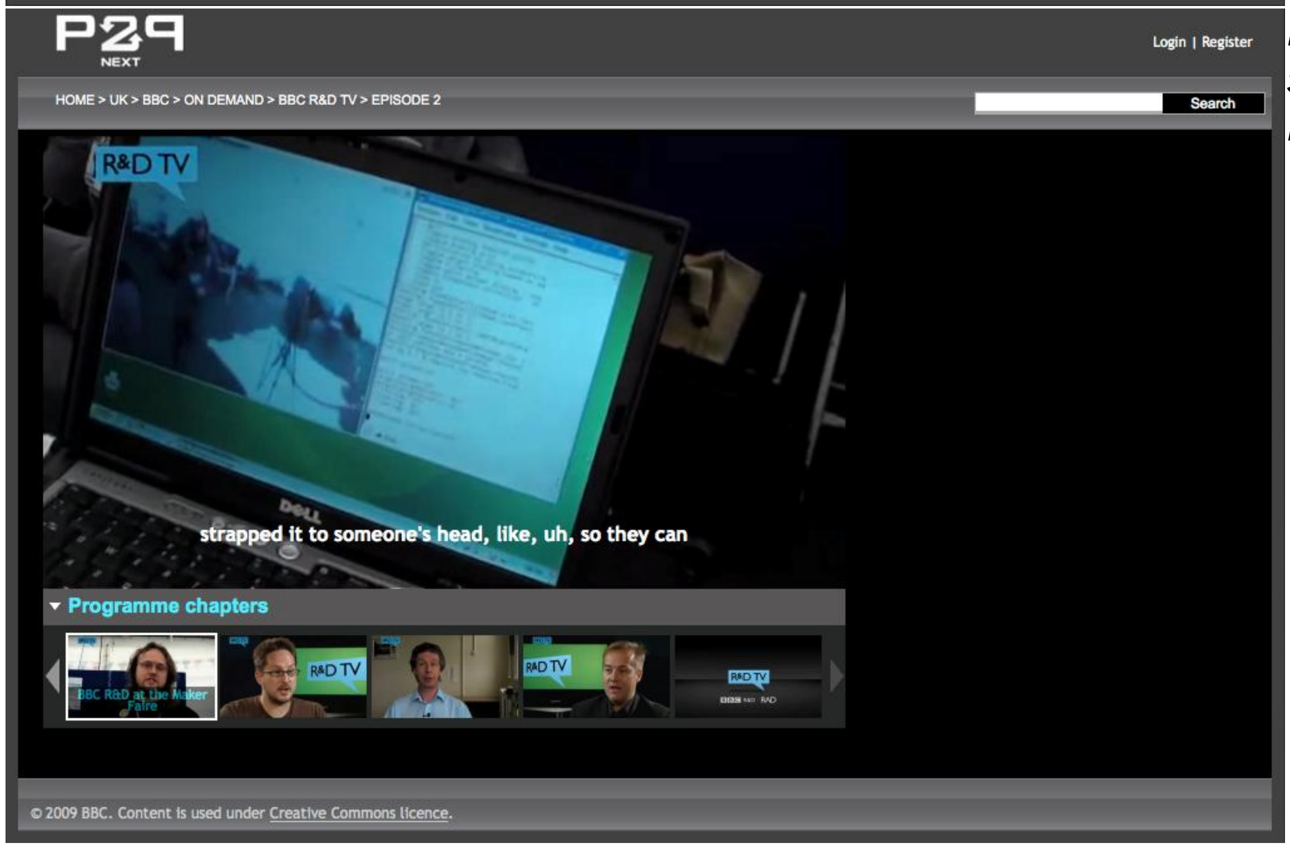
Amphibians such as frogs, toads and newts, are cold-blooded animals that metamorphose from a juvenile, water-breathing form to an adult, air-breathing form. Unlike other land animals, amphibians lay eggs in water as their fish ancestors did. The first amphibians were around one to five metres long.

Read original article

and 375 million years ago, a few of these backboned creatures followed the insects on to the land.

© 2009 see credits and copyright page for full details. Video for this prototype provided by wellcome trust

Fig 1 – Screenshot showing integrated Tree of Life demo


P2P NEXT
Login | Register
HOME > UK > BBC > ON DEMAND > BBC R&D TV > EPISODE 2
Search
R&D TV
Programme chapters
BBC R&D at the Maker Faire
R&D TV

strapped it to someone's head, like, uh, so they can

© 2009 BBC. Content is used under Creative Commons licence.

Fig 2 – Screenshot showing integrated R&DTV demo

```
Demo 1        http://limo.rad0.net/current_live/demo/treeOfLife/
Demo 2              http://limo.rad0.net/current_live/demo/rdtv/episode2/
Username            rad
Password            r4dl1mo
```

The R&DTV demo was picked up by the HTML5 community as a example of work which pushed the boundaries of interactive video on the web (eg web video specialist Silvia Pfeiffer blogged about it at http://blog.gingertech.net/2009/08/19/jumping-to-time-offsets-in-videos, while developer Remy Sharp used it as an example in presentations such as http://www.slideshare.net/remy.sharp/html5-friends and his book Introducing HTML5 http://www.amazon.co.uk/gp/product/0321687299).

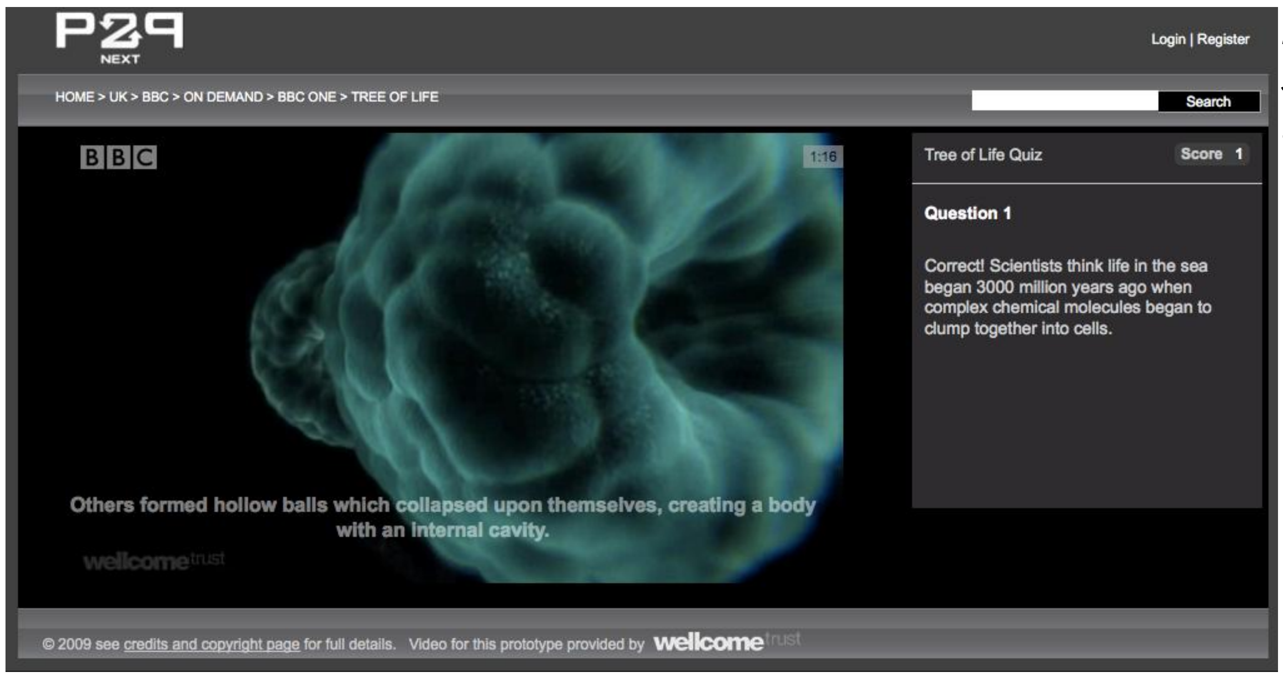
The demo led to email discussions with Silvia Pfeiffer in her role as invited expert on various W3C video-related working groups.

Overall we believe the demos had a positive impact on the further development of the HTML5 standard with regards to video.

2010

In 2010 the team developed two new prototypes building on the existing LIMO event types to demonstrate additional interactive features in the related areas of quizzing and polling.

The first of these added a timed quiz to the Tree of Life demo. A quiz panel is loaded to the right of the video which includes an introduction, questions, feedback and results. Questions are loaded when the video timeline reaches a defined point. The viewer must answer the question within a defined time as each must be answered before the next question appears.


P2P NEXT
Login | Register
HOME > UK > BBC > ON DEMAND > BBC ONE > TREE OF LIFE
Search
BBC
1:16
Tree of Life Quiz
Score 1
Others formed hollow balls which collapsed upon themselves, creating a body with an internal cavity.
wellcome trust
© 2009 see credits and copyright page for full details. Video for this prototype provided by wellcome trust

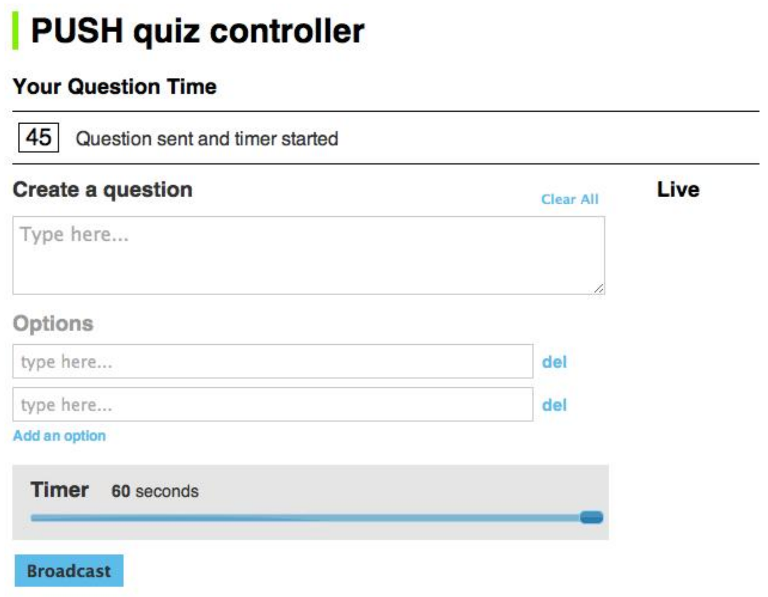
Question 1

Correct! Scientists think life in the sea began 3000 million years ago when complex chemical molecules began to clump together into cells.

Fig 3 – Screenshot showing Tree of Life timed quiz

Demo 3	http://limo.rad0.net/current_live/demo/quiz/treeOfLifeTimedQuiz/
Username	rad
Password	r4dl1mo

The second of these involved early investigations into second screen interactivity (which is now becoming more commonplace) and led directly to widespread trials of the technology alongside the BBC's Autumnwatch programme broadcast in October and November 2010.



PUSH quiz controller

Your Question Time

45 Question sent and timer started

Create a question Clear All **Live**

Type here...

Options

type here... del

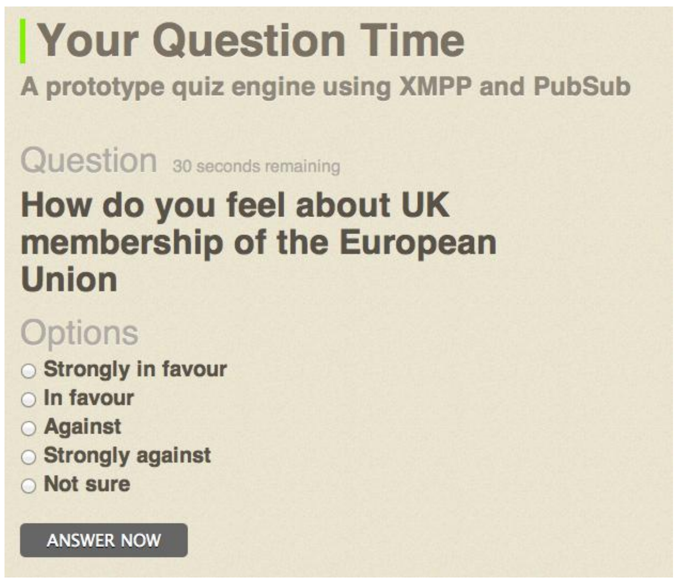
type here... del

Add an option

Timer 60 seconds

Broadcast

Fig 4a – Screenshot showing remote controlled poll admin view



Your Question Time

A prototype quiz engine using XMPP and PubSub

Question 30 seconds remaining

How do you feel about UK membership of the European Union

Options

○ Strongly in favour
○ In favour
○ Against
○ Strongly against
○ Not sure

ANSWER NOW

Fig 4b – Screenshot showing remote controlled poll user view (for second screen)

```
Demo 4 (admin) http://quiz.prototype0.net/admin/
Demo 4 (user)        http://quiz.prototype0.net/
Username             quiz
Password             quiz123.$
```

As described in full in our third report D5.4.1c the premise for a remote controlled poll is that a content creator wishes to enhance a live video by asking viewers to express their opinions during the programme. This is most relevant to live programmes where the content and timings are not known in advance so the content provider can create questions on the fly.

User testing for the project confirmed the idea of using 'second screen' devices for interactivity is popular although people felt some of the more complex ideas we had around user generated polls probably outweighed their benefit. The larger Autumnwatch trial also confirmed this is an area in which the public are interested.

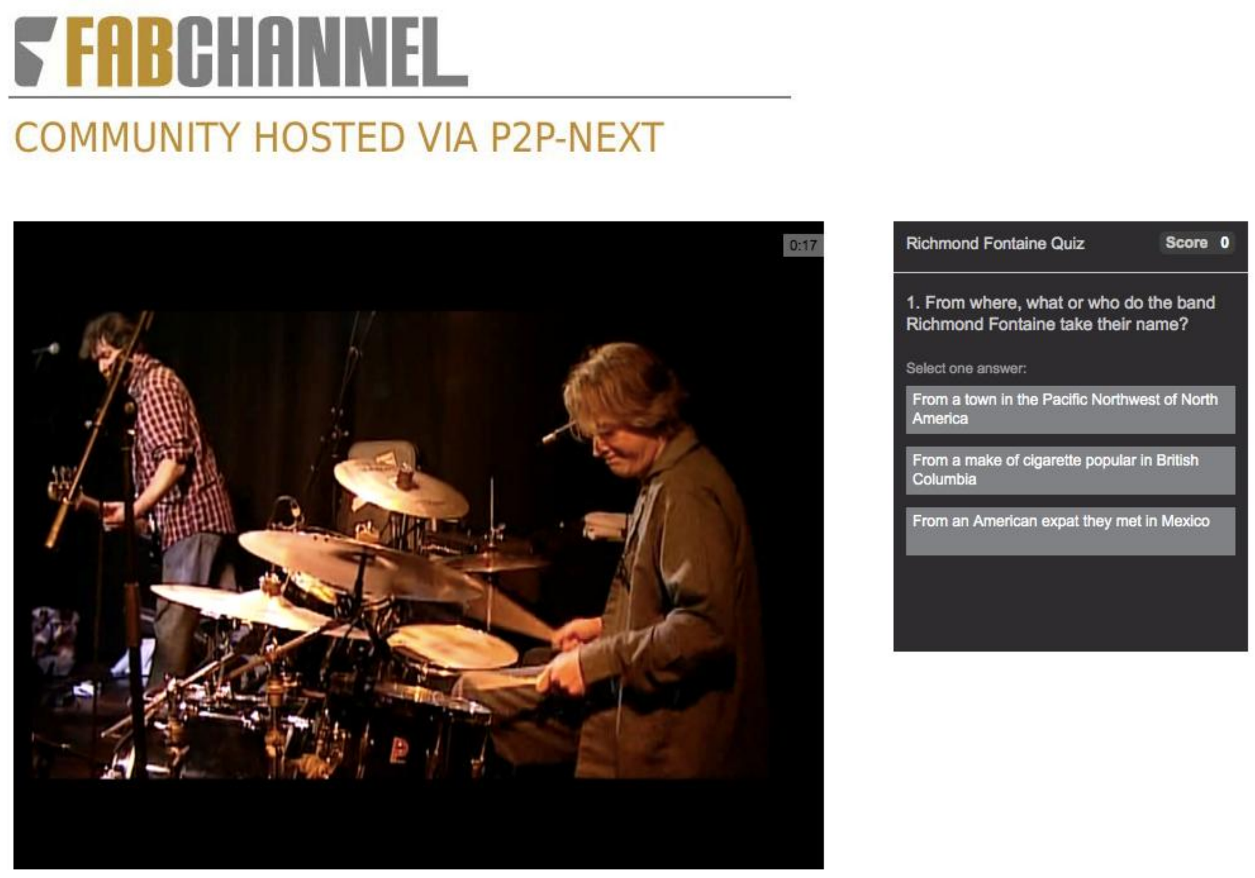
2011

In 2011 reviewing the prototypes to date led to a realisation that a significant problem which needs solving, both for this project and others providing interactivity alongside video, is that of synchronising a live audio/video stream played within a browser to the timed metadata (eg LIMO events) streamed alongside it.

The team has used the final year of LIMO work within P2P-Next to explore solutions to an issue we believe has wide reaching impact and could create a significant legacy for the interactive media strand of the project.

Before conducting further experiments with both second screen and live video it was necessary to refactor the LIMO code to make a clearer separation between the LIMO engine and the manifest and event files containing the timed data.

BBC, NORUT and ULANC redefined the LIMO standard as documented in Appendices 1 and 2 to make it possible for components to be swapped out. This means while the LIMO.js engine can be used on browser implementations, the engine can use native iOS code on the iPhone version. We created a several prototypes based on the refactored code.



Richmond Fontaine Quiz Score 0

1. From where, what or who do the band Richmond Fontaine take their name?

Select one answer:

From a town in the Pacific Northwest of North America

From a make of cigarette popular in British Columbia

From an American expat they met in Mexico

Fig 5 – Screenshot showing refactored quiz alongside FabChannel content

`Demo 5`	`http://limo.rad0.net/current_live/demo/quiz/limoEngineTreeOfLifeQuiz/`
`Username`	`rad`
`Password`	`r4dl1mo`

NORUT and ULANC went on to explore second screen scenarios in more detail while the BBC focused its attention on the issue of live streaming which we believed would keep the research ‘beyond state of the art’. We describe this particular aspect of the 2011 work in detail in the report sections that follow.

Influence map

Below is a visual summary of how LIMO work has evolved over the four years of the project indicating where the public outputs of the research and development have influenced other organisations and fed into emerging standards (with an approximate timeline on the right).

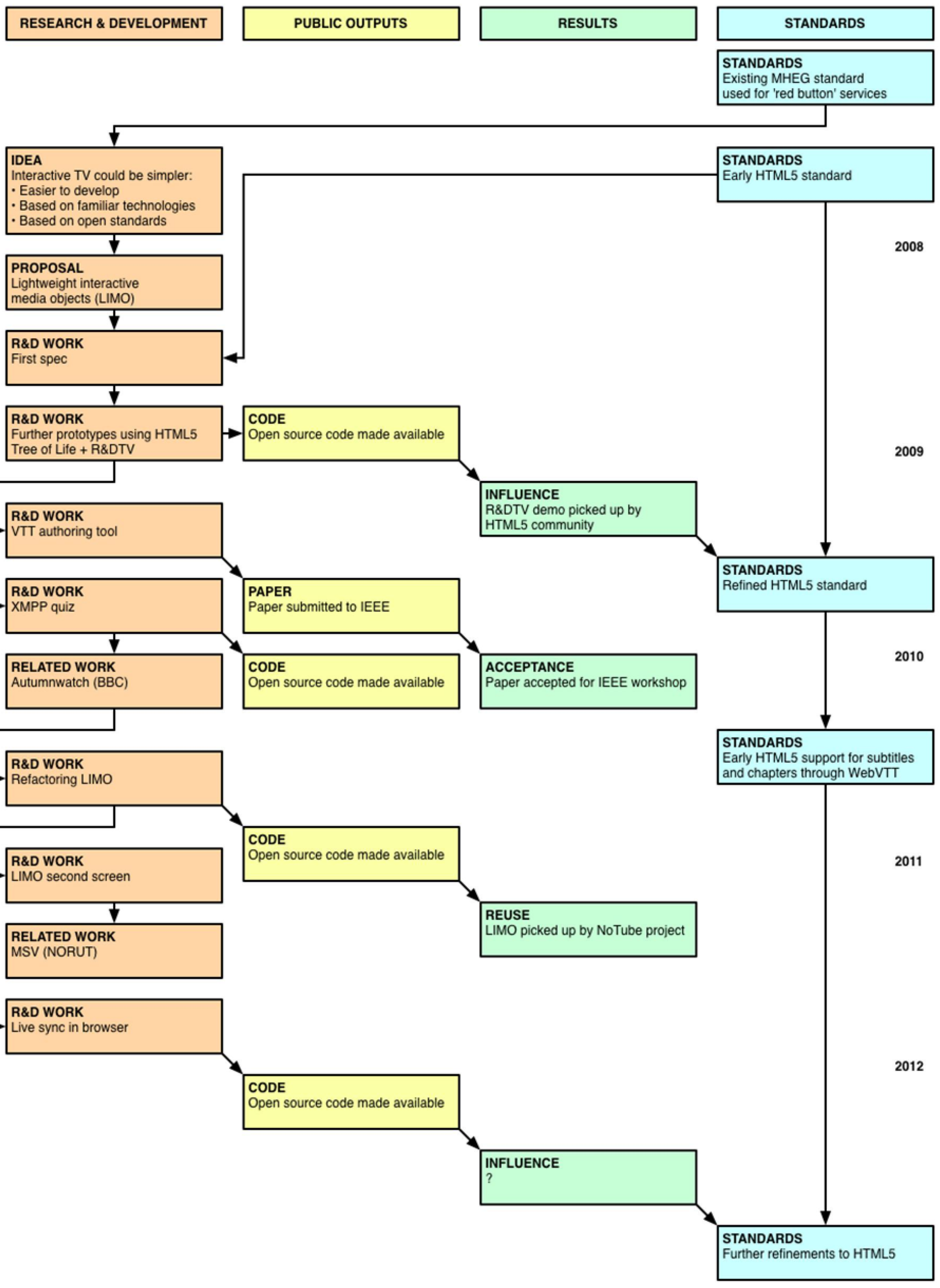

RESEARCH & DEVELOPMENT
PUBLIC OUTPUTS
RESULTS
STANDARDS
STANDARDS Existing MHEG standard used for 'red button' services
IDEA Interactive TV could be simpler: • Easier to develop • Based on familiar technologies • Based on open standards
STANDARDS Early HTML5 standard
2008
PROPOSAL Lightweight interactive media objects (LIMO)
R&D WORK First spec
R&D WORK Further prototypes using HTML5 Tree of Life + R&DTV
CODE Open source code made available
2009
INFLUENCE R&DTV demo picked up by HTML5 community
R&D WORK VTT authoring tool
STANDARDS Refined HTML5 standard
R&D WORK XMPP quiz
PAPER Paper submitted to IEEE
2010
RELATED WORK Autumnwatch (BBC)
CODE Open source code made available
ACCEPTANCE Paper accepted for IEEE workshop
STANDARDS Early HTML5 support for subtitles and chapters through WebVTT
R&D WORK Refactoring LIMO
CODE Open source code made available
2011
R&D WORK LIMO second screen
REUSE LIMO picked up by NoTube project
RELATED WORK MSV (NORUT)
R&D WORK Live sync in browser
2012
CODE Open source code made available
INFLUENCE ?
STANDARDS Further refinements to HTML5

Fig 6 – Influence map

Current focus on live synchronisation

Reasons to focus on live synchronisation

One of the key innovations of NextShare is the ability to create live (and progressive download) P2P streams, whereas existing P2P technologies such as BitTorrent require the user to download the full content before being able to start viewing.

One important potential application for broadcasters is for live events eg sports events such as the Olympic Games. Live video is also becoming more widely used by non-broadcaster with sites such as Ustream (http://www.ustream.tv) allowing anyone with a webcam to broadcast their own live video stream over the internet.

While there are new standards for interactivity alongside video that have developed since LIMO was first mooted (eg Popcorn on the web and Hybrid Broadcast Broadband TV for internet connected TVs) these haven't yet tackled the problem of true live synchronisation.

Some broadcasters have developed games that can be played on second screens alongside their more popular brands (eg in the UK, Channel4's Million Pound Drop, and BBC R&D's trial with Secret Fortune) these rely on the interactivity being triggered from a central source with all devices remaining in sync rather than the devices being synchronised to the video itself.

At IBC this year (the leading worldwide annual event for broadcast media) there were many examples of systems that supported second screen interactivity alongside video using approaches such as audio fingerprinting. However, we did not see any that allowed the interactive elements to be paused or recorded for later viewing.

The BBC has a particular interest in live events which are likely to be an area where national broadcasters maintain their unique role for some time to come. However the BBC also has a keen interest in on-demand media leading the UK market with the iPlayer so it seems natural to find solutions for seamless transition from one to the other which currently do not exist.

For all the reasons outlined above we decided to investigate issues surrounding live synchronisation with a goal to make LIMO work with live generated content, whether distributed via P2P or other streaming protocols. We believe this final piece of LIMO work within the P2P-Next project is the most likely way we can create a lasting legacy for all the research conducted to date.

Related technologies

When we started the project there were no other serious contender technologies for creating 'lightweight' interactive media alongside video.

Although one possible outcome of the LIMO work suggested in earlier deliverables was to define a new standard for timed interactive media we concluded there is little point doing this when there are other standards in existence which are not well used eg SMIL.

Having decided against pushing for a new standard we have made the goal of this year's work to explore issues and limitations affecting *all* interactive media and find solutions that could be adopted more widely within *existing* standards.

We have monitored related technologies throughout the project but in this section we take a snapshot of where they are now, particularly in relation to our interest in live synchronisation and their potential support for this.

Existing standards and technologies for synchronised multimedia

CMML

About:

- Continuous Media Markup Language (CMML) is an XML-based language for timed metadata.
- CMML is intended to be multiplexed with AV media within an Ogg container using Annodex, an encapsulation format based on Ogg, which interleaves time-continuous data with CMML markup in a streamable manner (see http://en.wikipedia.org/wiki/Annodex#Annodex_File_Format)
- CMML thus allows metadata description of 'clips' within a media stream with the specification including a description of how to encode in a bitstream.
- The full specifications of CMML and Annodex are available on the Annodex community website at http://www.annodex.net/TR/cmml.txt and http://www.annodex.net/TR/annodex.txt

Advantages:

- CMML allows arbitrary timed metadata, using <meta> elements inside <clip> elements which can be used for textual searches on otherwise binary video and audio files

Disadvantages:

- CMML only allows one image URL, web link URL, and description per time region (clip)
- The support for timed metadata is limited: <meta> elements can be used to encode an arbitrary set of key/value pairs, but don't allow structure (unless encoded into the key names)
- It is closely tied to its application within Annodex

SMIL

About:

- Synchronized Multimedia Integration Language (SMIL) is a W3C recommended XML markup language for describing multimedia presentations with similarities to HTML
- SMIL defines markup for timing, layout, animations, visual transitions, and media embedding, among other things
- It allows the presentation of media items such as text, images, video, and audio, as well as links to other SMIL presentations, and files from multiple web servers
- The full specification is available on the W3C website at http://www.w3.org/TR/smil/
- There are a variety of players and authoring tools available (see http://www.w3.org/AudioVideo/#SMIL)

Advantages:

- SMIL is a comprehensive language for describing presentations
- It allows synchronisation of subtitles with video

Disadvantages:

- SMIL is not a streamable format (and is not intended to be)
- It is not suitable for arbitrary metadata
- The specification is big and complex

TTML

About:

- Timed Text Markup Language (TTML) is a W3C specification that covers many aspects of timed text on the web
- The full specification is available on the W3C website at http://www.w3.org/TR/ttaf1-dfxp/

Advantages:

- TTML is a rich language for describing presentation aspects of timed text
- It is supported by authoring tools for both Adobe Flash and Microsoft Silverlight

Disadvantages:

- TTML is limited to timed text with display attributes (layout, styling)
- Metadata only exists at the document level (eg title, description, copyright) so does not support arbitrary timed metadata

WebVTT

About:

- WebVTT (initially titled WebSRT) is a synchronised media format for sequential timed text with styling, based on the SubRip subtitle format (http://en.wikipedia.org/wiki/SubRip)
- WebVTT is a ‘living standard’ that has been written into the ‘experimental’ HTML5 specification
- The ‘living standard’ is described in detail on the W3C website at http://dev.w3.org/html5/webvtt/

Advantages:

- WebVTT is being incorporated into the HTML5 standard via the <track> element
- WebVTT triggers JavaScript event handlers at start and end of each event, allowing custom code to be executed in response to events

Disadvantages:

- The primary application for WebVTT is timed text. The <track> element has a ‘kind’ attribute that can be set to ‘metadata’, but it is unclear from the specifications how to use this for arbitrary timed metadata (eg how to structure the metadata in a WebVTT file, or how to access from Javascript in the browser)
- WebVTT could be streamed, but it is unclear from the HTML5 specification whether this is supported
- The file format is based on SRT, not on existing structured data formats, eg JSON or XML, so specific parsers must be written for WebVTT

LIMO data format

About:

- The data format aspect of LIMO is a minimal data format for timed metadata
- LIMO is JSON-based for easy consumption in the browser
- The format defines some simple event types (eg chapter, subtitle, etc)
- A LIMO manifest allows clients to select appropriate sources of events

Advantages:

- LIMO is easily extensible as authors can define new event types as needed

Disadvantages:

- Existing LIMO event types are not as fully featured as other formats (eg compare LIMO subtitles with WebVTT or TTML)
- The data format and delivery protocol for streamed LIMO events has not been fully defined. The current implementation uses the Socket.IO JavaScript library, so the protocol is negotiated between the client and server based on the client's capabilities (HTML5 WebSocket, AJAX long polling, etc)

Multimedia and Rich Internet Application (RIA) environments

Adobe Flash

About:

- Adobe Flash is the predominant technology for video on the web which provides a rich programming environment, support for live and on-demand audio and video, and content protection

Advantages:

- Flash supports delivery of timed text and metadata synchronised to live streams (see www.adobe.com/devnet/flashmediaserver/articles/metadata_video_streaming.html)

Disadvantages:

- Flash is a proprietary technology that doesn't work on all platforms (eg no support for Apple iOS) and which relies on expensive authoring software with a steep learning curve

Conclusions:

- While it is possible to synchronise media with live streams within Flash it is not an open approach to take

Microsoft Silverlight

About:

- Microsoft Silverlight is an application framework for writing and running rich internet applications, with features and purposes similar to those of Adobe Flash. The run-time environment for Silverlight is available as a plug-in for web browsers running under Microsoft Windows and Mac OS X

Advantages:

- Silverlight includes a Text Stream Data Insertion Application which 'gives organizations a unique opportunity to create engaging, immersive rich media experiences, with external data seamlessly synchronized with video and audio content' (see http://bit.ly/vGp6rG)

Disadvantages:

- Like Flash, Silverlight is a proprietary technology that doesn't work on all platforms (eg again no support for Apple iOS) and which relies on authoring software with a steep learning curve

Conclusions:

- While it is possible to synchronise media with live streams within Silverlight it is not an open approach to take

Browser-based multimedia libraries

The following libraries are written in JavaScript, for execution within the web browser.

Popcorn

About:

- Popcorn is a set of tools and programs to help developers and authors create interactive pages that supplement video and audio with rich web content
- The project which is supported by Mozilla was recently given a new home at http://mozillapopcorn.org/welcome-to-the-new-popcorn/ to coincide with its version 1.0 launch which has attracted significant coverage from technical commentators (see http://blog.mozilla.com/blog/2011/11/05/popcorn-1-0-launch-and-world-premier-of-one-millionth-tower)
- Popcorn has two main components:
    - Popcorn.js is an event framework written in JavaScript which can be thought of as roughly equivalent to jQuery for video
    - Popcorn Maker is a browser-based graphical tool with templates for creating interactive media using familiar timeline controls. Completely free and open source, popcorn maker offers a collection of useful templates, but since it's built from standard HTML, you can customize it to your heart's content
- Popcorn does not define a markup language for timed metadata or presentations, instead presentations are developed by making calls to the Popcorn.js APIs

Advantages:

- Popcorn has high visibility, being supported by The Mozilla Foundation, and an active developer community working on the core library, plug-ins, and authoring tools
- Supports several existing timed text formats, such as TTML and WebVTT

Disadvantages:

- Popcorn does not support live synchronisation

Conclusions:

- Despite the emergence of Popcorn.js since the P2P-Next project began we chose to continue work on LIMO as to switch to a new framework would divert time which could be more usefully spent building tools that could be of benefit to both LIMO, Popcorn.js and other frameworks which may appear in future
- As we don't believe we should be *competing* with Popcorn.js and attempting to promote a rival framework we have used year four of our project to *complement* Popcorn on work that will be of benefit to anyone wishing to synchronise media within the browser

LIMO engine

About:

- Purpose-built execution engine for LIMO events
- Consumes events in the LIMO data format

Advantages:

- Works with on-demand video, where the events are retrieved in advance, and live video, where events are streamed to the browser alongside the audiovisual media
- Lightweight. The LIMO engine has no built-in rendering capability, so events must be handled and interpreted by custom JavaScript code in the browser

Disadvantages:

- Uses HTML5 `<video>` timeUpdate event to trigger LIMO events. This event has limited timing resolution, guaranteed by the HTML5 standard to be at least every 250 ms. In practice, we found the actual frequency of events to be browser-dependent, and much higher than the minimum specified. The LIMO engine could be redesigned to use JavaScript timers (eg setTimeout and setInterval functions) to get higher timing resolution (Popcorn.js has an option to do this)
- The LIMO engine supports streamed and AJAX event delivery, but these have not been fully integrated to support pausing and seeking within live streams
- For simple applications such as subtitles WebVTT may be more suitable, as rendering is done by the browser, and the content author does not need to implement custom JavaScript code
- Somewhat overtaken by Popcorn.js, which offers similar capabilities, but with a more powerful playback engine

Conclusions:

- Despite the emergence of Popcorn.js we have chosen to continue work on LIMO as to switch framework at this stage of the project would divert time which could be more usefully spent building tools that should be of benefit to both LIMO, Popcorn.js and other frameworks which may appear in future

Overall conclusions – what's missing

Synchronised live media is possible using Flash or Silverlight but these are proprietary systems on the streaming server and in the browser which raises the barriers of entry to producers and consumers.

Historically you have been able to create richer experiences through the proprietary platforms but with browsers becoming more powerful in themselves it should be possible to do live synchronisation within the browser. However, the specific elements required for this are not currently supported.

Given the current lack of support for live synchronisation we have set out to develop solutions that could potentially be adopted as an extension of these existing formats (although each of them may have other limitations that prevent live streaming which we've not explored in great detail).

What we mean by synchronisation with live streams

In this context, *live* does not mean *realtime*. Due to the delay inherent in the signal path from the live broadcast to the streaming server, due to encoding, routing, client-side buffering, decoding and potentially other issues the streamed version will typically appear in a browser a number of seconds after the corresponding broadcast version.

Considering how to marry up the media and event streams, we quickly came to the same conclusion as Ishibashi and Tasaka (http://bit.ly/t9A6Cc) that the 'timestamp … necessary for the intra-stream synchronization control' has to be in reference to a clock supplied in a master stream.

For our purposes, the media stream carries the master clock and the event stream is slaved to that master clock. We assume the existence of a shared reference clock across various media, eg if a subtitle is entered with clock time 17:15:00 then it will appear at 17:15:00 in terms of the program clock, even if, because of the propagation delay, a person watching on FreeSat does not actually see that until 17:15:04 according to UTC.

Some inputs provide a reference clock, such as the System Time Clock of a program stream carried in an MPEG-TS stream.

For others, we need an external clock reference. In the case of the demonstrator (described in detail below) this is the system time (in UTC) of the streaming server. In practice, for our purposes it is irrelevant where the master clock comes from as long as we can guarantee that events are timestamped in reference to it. Because streaming video across the internet is not instantaneous, this timestamp will be behind UTC on the client by the time the video frame is decoded and displayed.

Our demonstrator encodes the reference clock in the `dateUTC` field of the WebM header which is interpreted as the `startOffsetTime` of the HTML5 media element.

We interpret the `startOffsetTime` as the time of the first video frame in the stream.

Reasons for inserting reference clock in WebM header

Our primary example use case is providing accurately synchronised subtitles for a live broadcast streamed to an HTML5 compliant web browser.

Existing proposals for supporting subtitles alongside HTML5 audio/video, such as WebVTT (http://dev.w3.org/html5/webvtt), assume a static media resource of finite size and the existence of a corresponding static subtitle resource which can be synchronised using offsets from the start of the media and provided before publication.

In a live broadcast context, subtitles are typically entered by an operator during the broadcast and so are not available in a static resource that can be delivered beforehand. Therefore, to provide synchronised subtitles alongside a live broadcast displayed in an HTML5 `<video>` element, we need to provide a dynamically sourced stream of subtitle events to the browser.

Additionally, to ensure that these subtitle events are accurately synchronised to the broadcast, we need to use the appearance of the subtitle on the video frame as the synchronisation point. The software that enters subtitles (or derives them from the video frames) uses the DVB System Time Clock as its reference and so we need to do the same.

Issues with live synchronisation

For on-demand video it’s relatively easy to synchronise interactivity as the current playback time is defined in terms of an offset from the start of the media and the client can receive the whole package of timed events in one go.

For live streaming video there are a number of challenges to address:

1. A user can join the stream at any time so they won’t have received the history of events that have already taken place which may be critical to the display of the interactive media at that point

2. They may have connected to one of many stream servers which has started streaming at any time in the past

3. Most of the events you want to synchronise with the live media stream have not happened yet by the time the user joins the stream so you cannot provide them up front though you may want to provide expected events (eg programme changes) in advance

In the majority of cases users will always join part way through a stream as there will nearly always be a back history of events of some kind, eg a live programme social media commentary may begin long before the broadcast event begins.

For live streaming there is a similar issue when a user hits pause. A client device can be configured to record all events during the time a media stream is paused, but unless it has a way to resynchronise to the same clock used by that media stream it will have no way to resynchronise those events.

Furthermore, in a production environment, media streams will usually be served by multiple streaming servers which will have been started at different times. So each stream will need its own clock reference.

Potential solutions to live synchronisation

There are three basic approaches to synchronising with a live media stream:
1. Synchronising the client's clock to a remote master clock
2. Matching against the AV signal itself to derive a synchronisation signal on the client
3. Using metadata to share a reference clock with the client

Synchronising to a remote clock

The most widely used remote clock synchronisation method in use is NTP although there are others such as RADclock which itself uses NTP as a time source and BBC R&D's SynchTV project which uses a DVB time signal as its absolute time reference. In the discussion below, we look at NTP as an example of synchronising to a remote *absolute* reference clock. A different approach to synchronisation to a *relative* remote or local clock is developed in recent research by NORUT which proposes synchronising via a shared 'media state vector'.

NTP

About:

- NTP stands for Network Time Protocol, and it is an internet protocol used to synchronise the clocks of computers to some time reference
- NTP needs some reference clock that defines the true time to operate with all clocks are set towards that true time (ie it will not just make all systems agree on some time, but will make them agree upon the true time as defined by some standard)
- NTP uses UTC as its reference time.
- (Source: http://www.ntp.org/ntpfaq/NTP-s-def.htm)

Advantages:

- The best general solution for keeping computer clocks in time with UTC
- Fault tolerant, scalable, reliable, accurate and widely available

Disadvantages:

- It cannot work for us because synchronising to UTC or any specific remote clock cannot provide us with the clock reference we need because we are not synchronising to an absolute time frame but to time as expressed either implicitly or explicitly in the media stream

Conclusions:

- Synchronising to a remote clock that provides an absolute reference time is of no use to us in the context of synchronising timed metadata to streamed audio/video because the nature of streaming means that all times are inherently relative to that media stream rather than to the absolute clock
- Note client will still need something like NTP to prevent its own internal clock drifting

Media State Vector (MSV)

About:

- A more promising technique for synchronising to a remote clock is Media State Vector (MSV), an initiative of NORUT which proposes a 'theoretical foundation for synchronisation of media experiences spanning multiple devices, users and media sources'.

Advantages:

- Works for multiple devices
- Works for pause and rewind (or any kind of free navigation)
- Content-agnostic so may be used to synchronise content from multiple sources
- Supports synchronisation of LIMO-type media, without the requirement of there being a video element.
- Can use both a local clock source (such as a media stream) or a remote shared clock
- Can be used to synchronise videos across multiple screens

Disadvantages:

- When applied in remote mode MSV is subject to some of the same issues as NTP (network lag, jitter, skew, etc)
- Current implementation uses the script-tag hack to avoid the cross-domain restriction which is a security issue

Conclusions:

- MSV is an interesting approach to synchronising media across multiple devices in a way that's complementary to our own work. Using the clock we are providing in the media stream as the source for a local 'motion' (in MSV terms) would enable us to 'slave' second screen applications to the audio/video playing out on a main screen.

Synchronisation based on matching the audio/visual signal

There are currently two common ways to synchronise based on the content of the AV signal, differing in whether they modify the source signal itself or interpret it:

1. Embedding an explicit *watermark* or *barcode* into the AV signal
2. Deriving an AV signal's *fingerprint* on the broadcast side then applying the same transformation function on the client side to match

Watermarking

About:

- Digital watermarking is the process of embedding information into a digital signal which may be used to verify its authenticity or the identity of its owners, in the same manner as paper bearing a watermark for visible identification. In digital watermarking, the signal may be audio, pictures, or video. If the signal is copied, then the information also is carried in the copy. A signal may carry several different watermarks at the same time (Source: http://en.wikipedia.org/wiki/Digital_watermarking)
- Audio watermarking involves analysing the audio track to reveal positions in the signal where, considering the signal masking characteristics of the human audition, we can hide some digital codes without affecting the sound quality of the original (Source: http://blog.eltrovemo.com/529/synchronized-second-screen-technologies-panorama/)
- Watermarking is typically used to assert ownership for copy protection and to track provenance
- When the watermarking is imperceptible to the audience, it is a form of steganography
- Examples of visible watermarks are barcodes and QR codes

Advantages:

- Does not require a return path
- Can be added just prior to delivery to client
- Can be encrypted

Disadvantages:

- It is particularly hard to hide a watermark in a digital audio signal, especially during silence
- Watermark appears on client rendering and can be intrusive
- Need to modify the source AV signal
- Need to interpret the AV signal on the client
- Need to make sure that the watermark is preserved after being added in processing pipeline
- Need to frequently repeat the watermark or barcode to provide accurate clock signals
- Cannot provide accurate enough synchronisation for lip synching subtitles without severely distorting the input signal

Conclusions:

- Watermarking is an intrusive method in that it modifies the actual audio or video as received by the client – in particular, in many trials, many listeners have complained that audio watermarking is audible and annoying
- Because it relies on significant processing on both the server and client side to embed and extract the watermark, and because it cannot provide the timing resolution we needed for subtitles, we felt it was not a suitable technique for our purposes

Audio fingerprinting

About:

- An acoustic fingerprint is a condensed digital summary, deterministically generated from an audio signal, that can be used to identify an audio sample or locate similar items in an audio database (Source: http://en.wikipedia.org/wiki/Acoustic_fingerprint)
- The 'condensed digital summary', typically called a *signature*, is calculated on an ongoing basis on the server. The client on receiving the audio applies the deterministic function to a short segment of audio to derive a fingerprint. It then sends this fingerprint back to the server which looks it up in its database to find a match. For synchronisation, the server needs to be able to match not just what programme or track is being played out but at what second the fingerprint matches
- Robust acoustic fingerprint algorithms work even on compressed audio in a variety of encodings

Advantages:

- Works even when the binary representation is quite different
- Works for analogue
- Does not need direct access to the broadcast streams
- Can work for existing devices without modifying them
- There are currently many commercial implementations available
- You don't have to be the owner of the contents to analyze it, so this is a perfect way to build synchronized services if you are not a TV channel or a cinema studio

Disadvantages:

- For live need to calculate fingerprints on the fly
- Requires recording fingerprints on the server side which could potentially result in a large database to match against
- Requires processing the audio signal on the client to derive the fingerprint
- Hard to make robust across the range of possible playback devices
- Delay between acquiring the signal and determining the playback position
- To work for dubbed versions, need fingerprints of all soundtracks
- For accurate per-client synchronisation, needs a return path to the server to compare the client's fingerprint with that stored on the server
- Existing commercial implementations are expensive

Conclusions:

- Even though audio fingerprinting can be made to work after a fashion as a synchronisation mechanism, it is better used for identification rather than synchronisation. Even in the context of identification, audio fingerprinting is more suitable for identifying known frequently repeated segments of audio, eg music tracks, than novel live broadcast audio.
- Audio fingerprinting was used for synchronisation in a second screen trial conducted by BBC R&D for a programme of 'Secret Fortune':
  'Even just from this technical perspective, we learned a number of useful things from the pilot. For example, audio watermarks take a finite amount of time to detect, which

prevented us from implementing any watermark-driven synchronised behaviour at the very beginning of a programme (or programme segment). Also, while it is obvious that signalling events in programmes from a central server on the internet can only work for people watching the programme as it is broadcast, it is perhaps less obvious that people watching on different broadcast platforms (eg Freeview, Freesat or analogue TV) see a given part of the programme at slightly different times, which reduces the synchronisation accuracy accordingly.' (Source: http://www.bbc.co.uk/blogs/researchanddevelopment/2011/10/dual-screen-secret-fortune-sync-api.shtml)

- As there are many companies already providing sync via audio fingerprinting and as we believe it is not the best solution possible, we rejected this option

Synchronising using a reference clock supplied in metadata

Using metadata supplied with the stream is similar to the way that DVB works. A DVB stream carries a clock signal called the System Time Clock (STC) and each audio and video frame in that stream has a Presentation Time Stamp (PTS) which indicates, in reference to that STC, when the frame should be output (see http://en.wikipedia.org/wiki/Presentation_time_stamp)

In our context, the System Time Clock is provided to the browser by the combination of the `startOffsetTime` and the stream running time. The Presentation Time Stamp is the time encoded in the LIMO event.

Note that the two approaches described below, in-band metadata and using out-of-band metadata along with `startOffsetTime`, are not mutually exclusive and could usefully be employed in a complementary fashion, eg canonical or pre-prepared metadata could be provided via an in-band stream with additional/alternative events being supplied out-of-band.

In-band metadata

About:
- In-band metadata refers to timed metadata that has been multiplexed with the audio/video stream using the same reference clock as that audio and video
- A proof of concept build of Firefox that supports Kate subtitles in an Ogg container has demonstrated the viability of additional metadata codecs (see https://bugzilla.mozilla.org/show_bug.cgi?id=481529)
- MPEG-DASH has support for additional metadata streams.

Advantages:
- Enables reliable, accurate synchronisation
- Need only deliver a single multiplexed stream to the client
- Could be used as a clock source for out-of-band timed metadata

Disadvantages:
- Redundancy eg captions need to be for multiple languages even if the viewer needs only one, if any
- Increases bandwidth required for streaming
- Unless container designed to allow arbitrary timed metadata, can be difficult to extend, requiring changes to container format, muxer, demuxer, eg systems designed to carry only subtitles
- Requires multiplexing metadata into the broadcast stream
- Requires support for each supported codec in browsers

Conclusions:
- This is likely to be a serious contender for how timed metadata is delivered to web clients. However, as work has already been done in this area (although it has

apparently stalled) and we are concerned about the disadvantages noted above, we decided to look at the other promising possible solution (see next section).

Supplying a reference clock in media stream header (our solution)

About:

- This solution is dealt with in detail in the following sections but in short it uses the HTML5 media elements standard attribute `startOffsetTime` to provide a base reference time to the client to which LIMO events are synchronised
- It calculates the `startOffsetTime` for each media stream and supplies it to the browser via a WebM stream header

Advantages:

- Enables reliable, accurate synchronisation
- Does not require multiplexing metadata into the broadcast stream
- Implements a previously unimplemented attribute of the HTML5 media specification
- Implements a previously unimplemented feature of the WebM specification

Disadvantages:

- Requires changes to the way streams are encoded (to include the `startOffsetTime`)
- Requires changes to browser codecs to extract the `startOffsetTime` and provide it to user scripts
- Ogg explicitly excludes a metadata element corresponding to `startOffsetTime`

Conclusions:

- This is our preferred solution. We believe that by implementing the specified but missing functionality in the WebM and HTML5 media specifications we can achieve highly accurate synchronisation.

Conclusion

Synchronising a client's clock to a remote clock is not sufficient to provide synchronisation with a media stream as there is no guarantee that the media stream as received on the client is synchronised to that remote clock.

Fingerprinting and watermarking work around the lack of an accessible reference clock in the media stream in current AV consumer devices. Both solutions have been found to be difficult to make reliable in practice and fingerprinting in particular requires considerable processing resources on both the server and client sides.

Providing reference clock metadata in the media stream and making that externally accessible on the client-side is clearly the right way to share a reference clock with clients. Unfortunately, this solution has received the least attention - neither the WebM `dateUTC` field nor the HTML5 `startOffsetTime` attribute have been implemented in current codecs and browsers. However, this solution has the benefit of being based on existing agreed and documented standards and potentially being the most reliable and accurate way to provide synchronisation of out-of-band timed metadata with live media streams.

Functional design considerations for live LIMO

The LIMO data format and engine need to support a variety of event attributes for different production situations (eg alternative programme genres and formats) and different contexts of use (eg a user watching live or on-demand).

It's impossible to anticipate all the potential combinations. The key thing is the LIMO framework should be flexible enough to accommodate innovation from content providers without the need for the underlying engine to be rewritten each time.

The categories of variation within event types are described below. We have considered these issues in previous deliverables but here we are revisiting them from the perspective of live streams which have hitherto not been supported.

Streams

There are three main types of stream where events can live at the various stages of production:

1. Asset bin
    - ○ Unscheduled assets
    - ○ Running order – edited on the fly, in its simplest case a fully pre-prepared playlist of all the events a provider intends to insert into a live stream with all timings but could include any combination of the following:
    - ○ Fixed events – (around which other events will be slotted) eg commercial breaks
    - ○ Pre-prepared – events or packages of events (to be triggered manually at the appropriate time) eg interactivity to support pre-prepared VT for a live programme
    - ○ Ad hoc events – where the time and content is created on the fly (eg the name of a contributor on screen)
1. Live Stream
2. On-demand Stream

It should be possible to flag any event as live, on-demand or both (but not neither):

- Live – events only to be shown when the programme is live eg phone number to vote for a participant
- On-demand – events only to be shown when the programme is on demand eg alternative to the phone number described above, possibly message to indicate phone number mentioned by presenter should *not* be called
- Both – programmes to be shown both live and on-demand eg subtitles

The categories above can be further expanded to support some more complex use cases eg:

- Time-shifted – events to be shown while any part of the programme is still live but the user is not necessarily at the live playback point
- Timed – events to be shown only between certain start and end times eg phone numbers for voting where the phone line closes at a particular time which may not be at the same time the programme ends
- 'Seam carving' – where a programme is edited at a later stage leaving the interactivity pegged to the correct points in the shortened video or audio (much like the way seam carving a photograph maintains the original meaning while changing the dimensions)

Events and packages of events can pass from the skeleton stream into the live stream which is then recorded as the on-demand stream.

Where a provider knows what the content will be in advance but doesn't know the timing in advance (eg for the pre-prepared events described above) it should be possible to make the content available for download and add the time to the event when this is known.

Context of engagement

As previously discussed LIMO interaction is intended to adapt to the context of use: live, time-shifted or on demand. For each, the user should receive an enjoyable experience and need not necessarily be aware of any formatting changes. For example, if content, chapters or entire packages are removed for the on demand version then the presentation should still flow seamlessly (see fig 17 later in this document).

Menus for navigation and programme description

The visibility and/or availability of programme content for the purposes of navigation (eg the contents of a chapter menu) should also be considered. In this case we suggest that chapters and content are made visible to the user, but remain inactive until the point of broadcast.

This facility enables a user to view information about the programme and its chapters and content in advance. Post broadcast, menu content can remain available, enabling to the user to navigate the programme timeline non-linearly, affording them control.

Relationship between events

The relationships between events can be categorised as follows:

- Linear
  - Sequential (eg one after another)
  - Cumulative (eg a combination of events that build on previous events)
- Non-linear

The fact that a user can join a live stream at any point or may watch a stream on demand raises the need for the event player to force linear events to be viewed in an intended order.

We suggest that for quizzes, where a score is cumulatively awarded, a user is *only* able to progress linearly. It is possible to allow users to join cumulative events late, however their ability to build a score should either be disabled (play for fun only) or they should be given an option to start from the beginning as a time-shifted or on demand version.

For other types of sequential content a user should be able to navigate freely, with prompts to join at suggested chapter points.

Building blocks

The basic elements of LIMO required to support most interactive television scenarios can be grouped together to simplify wireframe design as follows:

- Video – the main timeline driver
- Subtitles – displayed in the lower portion of the screen
- Chapters – give context to a sequence within a programme
- Overlays – lower thirds
- Info panels – eg maps, URLs
- Content – interactive content
- Quiz logic †

† Note that quizzes are in fact an application of several of the building blocks listed above

Conclusions on these requirements

The LIMO data format and engine must be (and have been) designed in such a way that attributes for the variations described above can be added ie the different contexts of use they imply can all be supported.

Sketching out the issues

Having considered the basic users requirements we looked at the different event and graphic types required to create a flexible interaction framework. We sketched out a number of alternatives before settling on the approach used in the demonstrator described later in this document.

The initial sketch on the next page highlighted the need for code fragments pieced together as separate events on a timeline. For example lower-third ‘straps’ (used for naming presenters on screen) can be visually rich with the use of HTML, images (PNG, JPG etc), web fonts, CSS and jQuery animations. Compound events were explored here as a way to nest events into a quiz format (see bottom right):

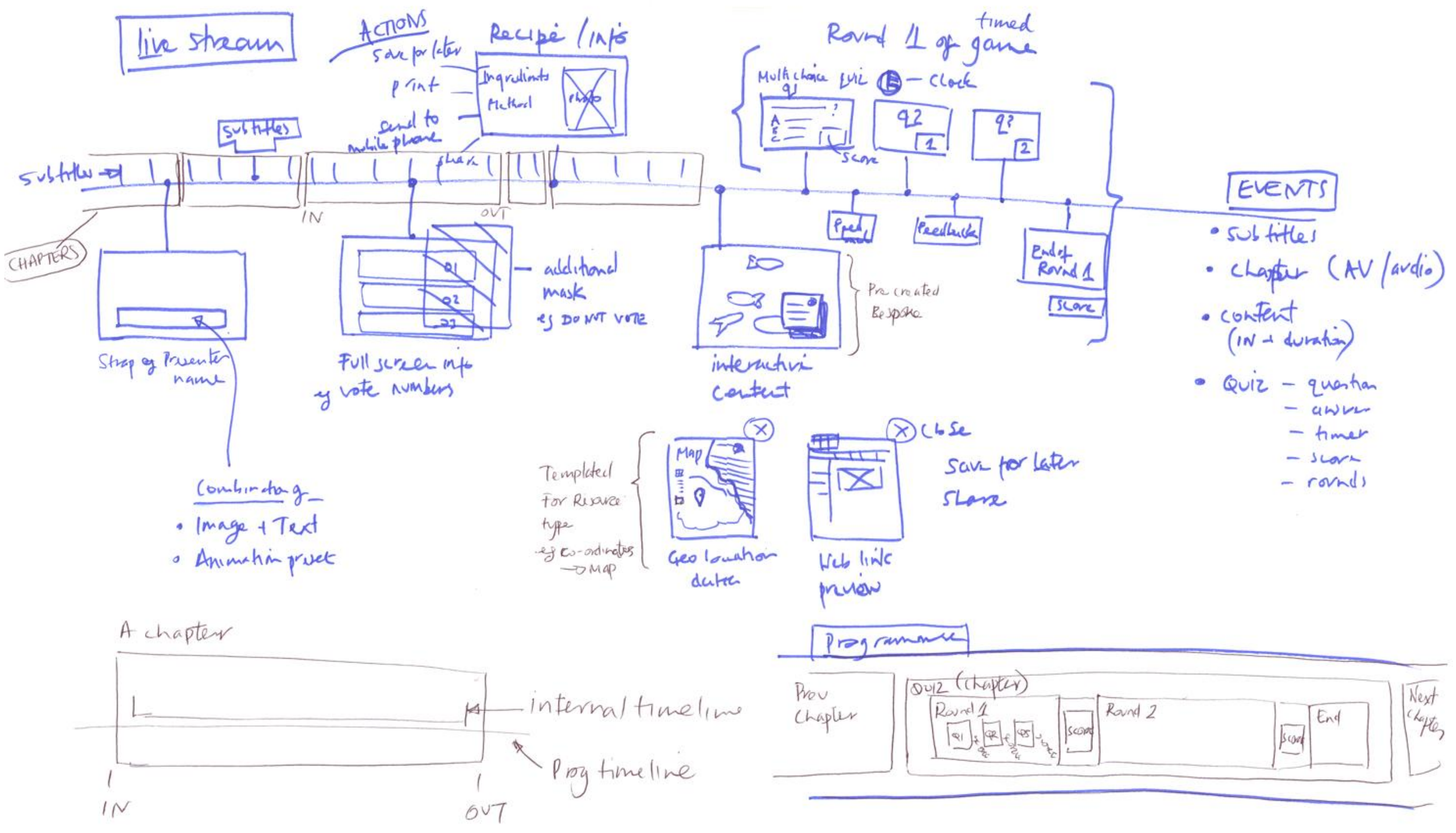


Fig 7 – Initial sketch of how events occur on timeline

The following sketch explores different ways to display additional content – DVD style menus, multi-screen, leading to simple content screens.

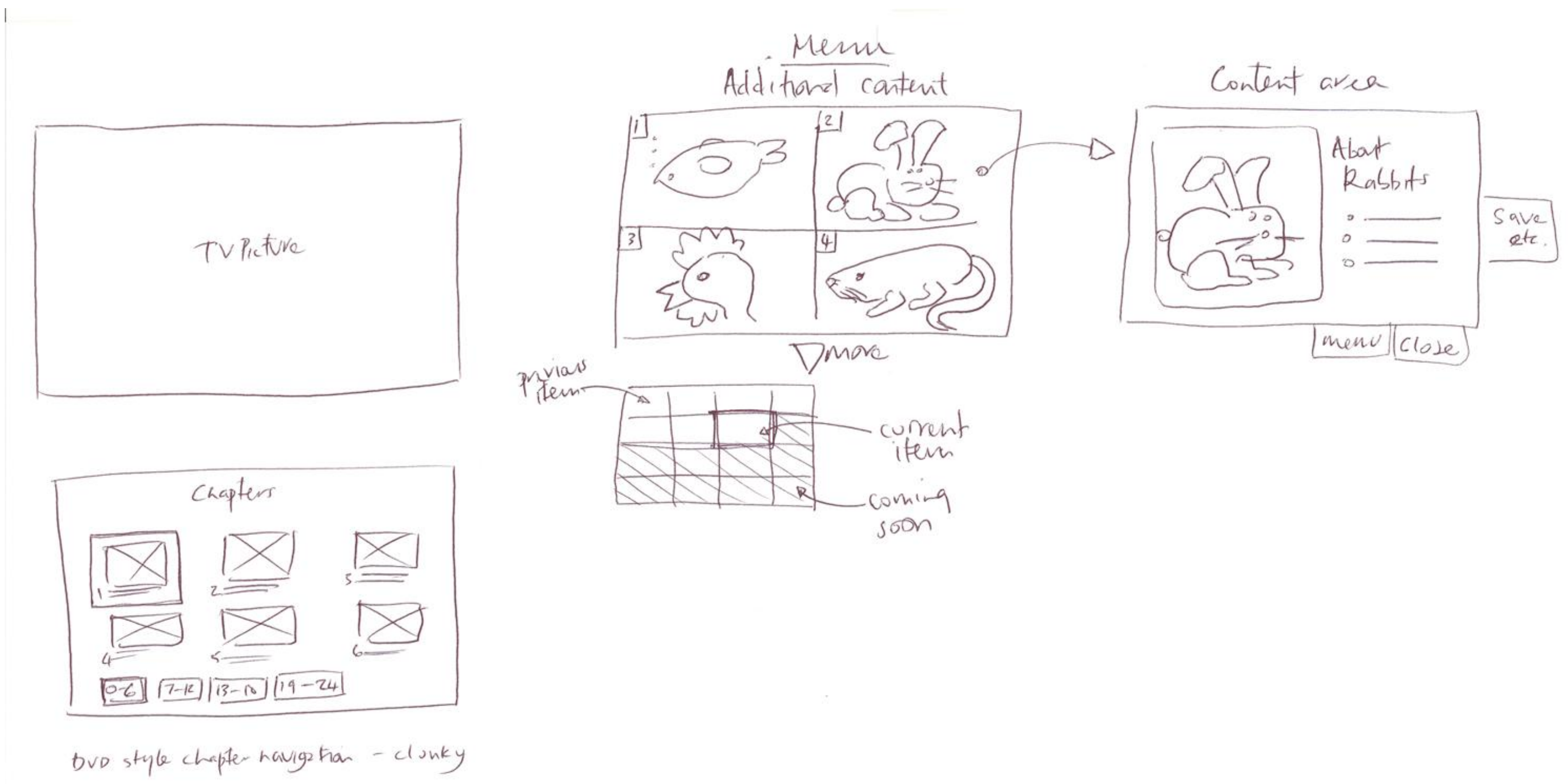


Fig 8 – Initial sketch showing modes of display

Finally this final sketch looks at how to navigate content by chapter, how to move between screen states, and how to show progress through a video or chapter.

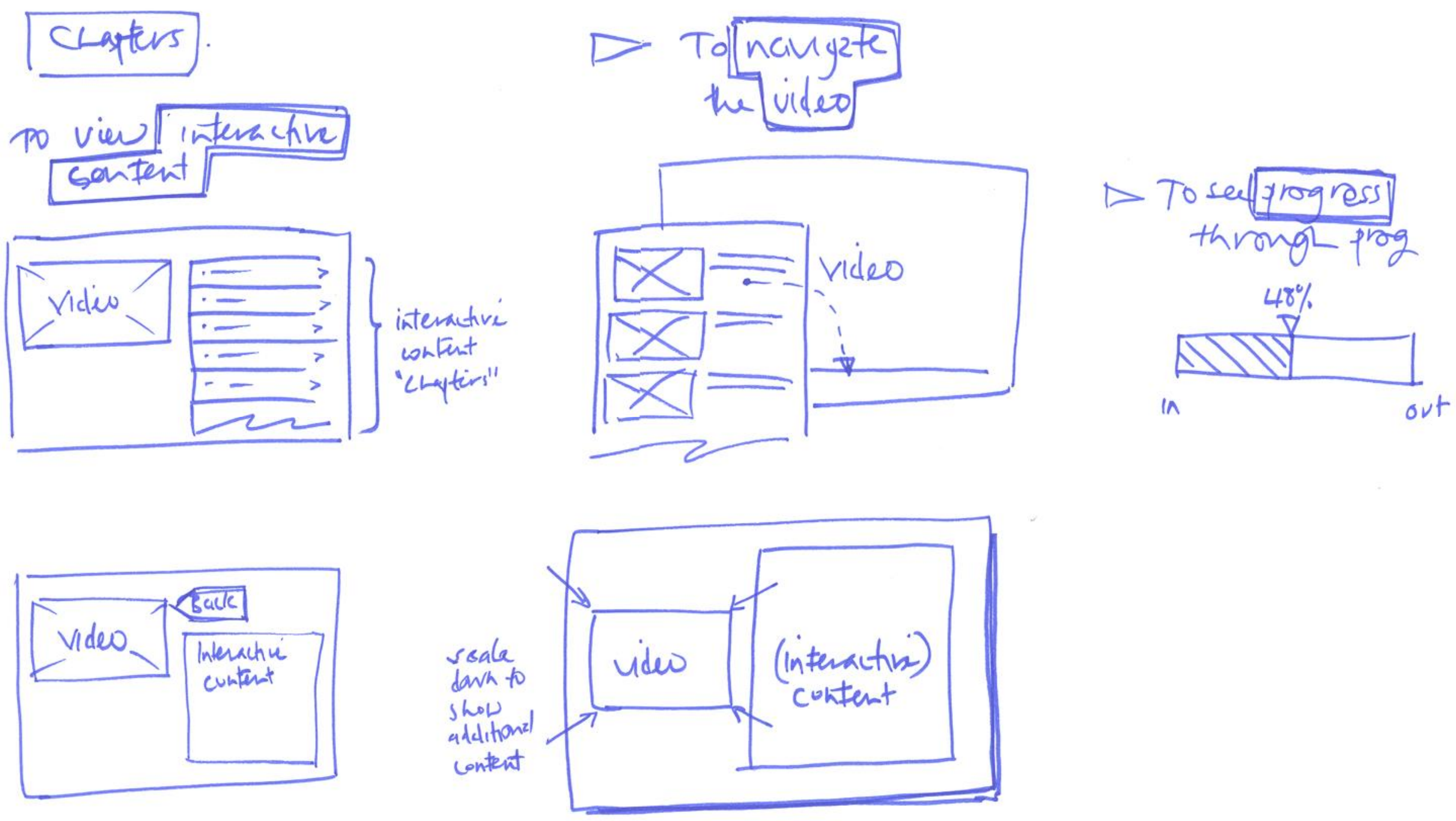

Chapters.
To view interactive content
video
interactive content "chapters"
To navigate the video
video
To see progress through prog
48%
in
out
back
video
Interactive content
scale down to show additional content
video
(interactive) content

Fig 9 – Initial sketch showing navigation options

Wireframe layouts

Combining: video / content / event timeline

Having sketched out various options on paper we turned to creating wireframe layouts on screen. The following shows a simple four state interface for the user to navigate content:

1. *Top left:* Video view, showing chapter and progress bar
2. *Top right:* Video view with pop alerting user to new content
3. *Bottom left:* Dual view of video and content with description
4. *Bottom right:* Content view with video in picture window

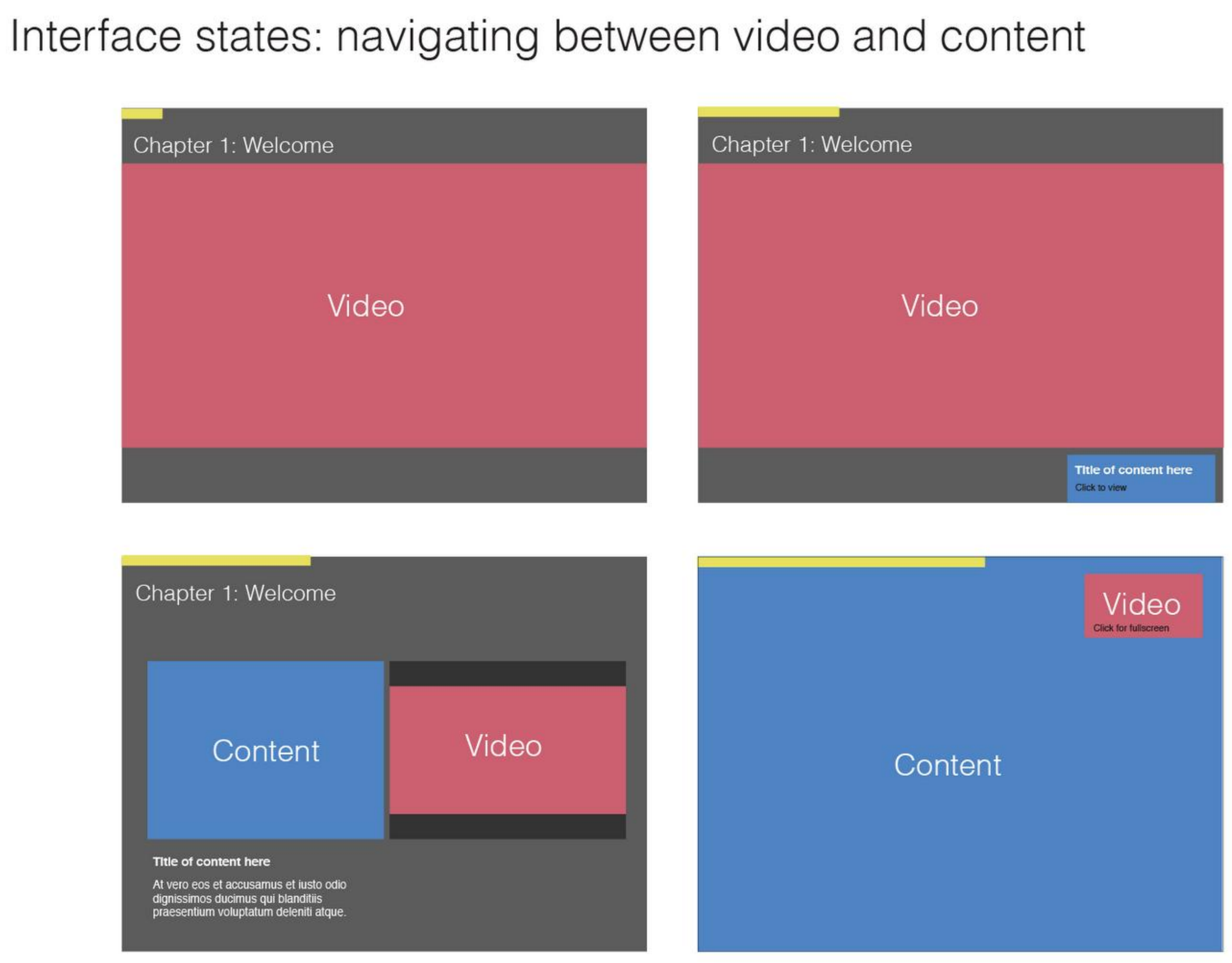

Interface states: navigating between video and content
Chapter 1: Welcome
Video
Chapter 1: Welcome
Video
Title of content here
Click to view
Chapter 1: Welcome
Content
Video
Title of content here
At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque.
Video
Click for fullscreen
Content

Fig 10 – Interface states

The design becomes more complex with the addition of an event timeline. This is a useful panel for producers to see where they have made interactive content available but we believe it’s also useful for consumers as a means of tracking where they are within a video

The diagram below shows three potential ways of displaying events:

- *Top left:* Displaying events in a vertical timeline at the left of the screen
- *Bottom left:* Displaying events in a horizontal timeline along the bottom of the screen
- *Bottom centre:* Adding a tab so users can toggle between content, video and timeline

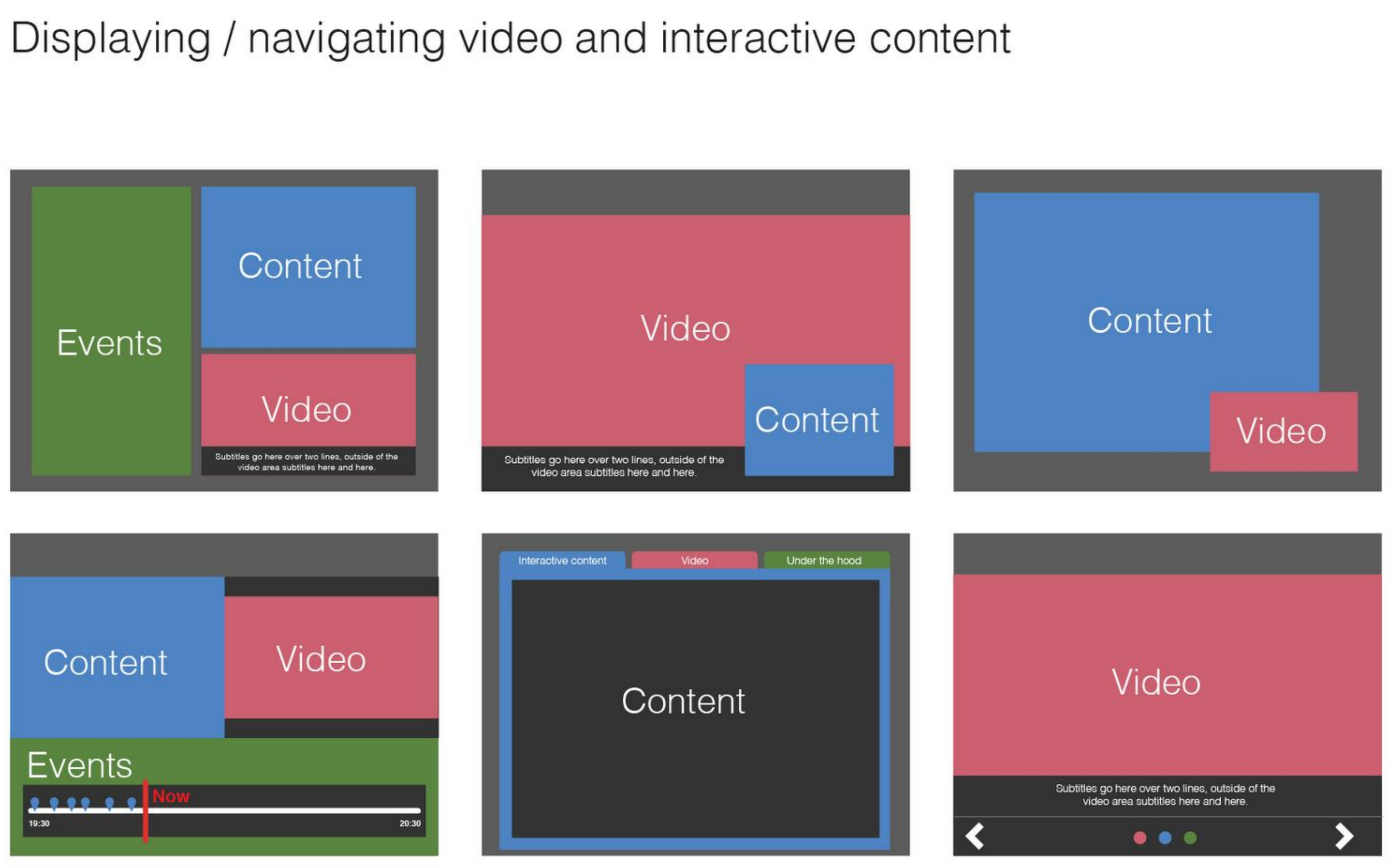

Displaying / navigating video and interactive content
Events
Content
Video
Subtitles go here over two lines, outside of the video area subtitles here and here.
Video
Content
Subtitles go here over two lines, outside of the video area subtitles here and here.
Content
Video
Content
Video
Events
Now
19:30
20:30
Interactive content
Video
Under the hood
Content
Video
Subtitles go here over two lines, outside of the video area subtitles here and here.

Fig 11 – Interface states (complex)

We concluded that horizontal timelines are the common paradigm for time-based media such as video so we should explore this approach in more detail (see next section).

Event data display

Horizontal timelines are the common paradigm for time-based media such as video. In the case of LIMO such timelines may need to contain several tracks or layers.

We decided to give each event type its own layer for easy identification. As highlighted earlier in the document the most common event types can be grouped as follows with the diagram showing those marked with an asterisk as an example:

- Video – the main timeline driver
- Subtitles – displayed in the lower portion of the screen *
- Chapters – give context to a sequence within a programme *
- Overlays – lower thirds
- Info panels – eg maps, URLs
- Content – interactive content *
- Quiz logic

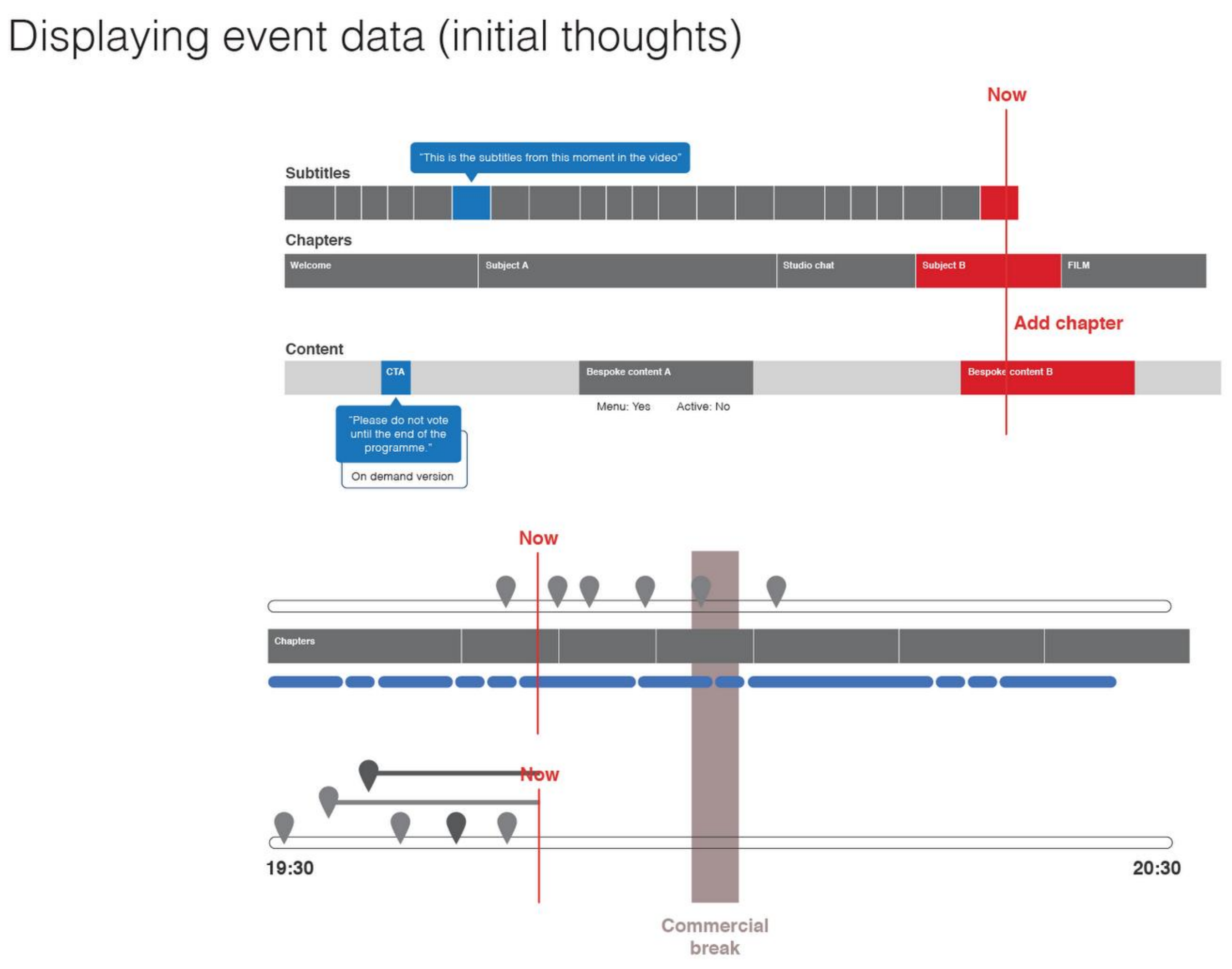
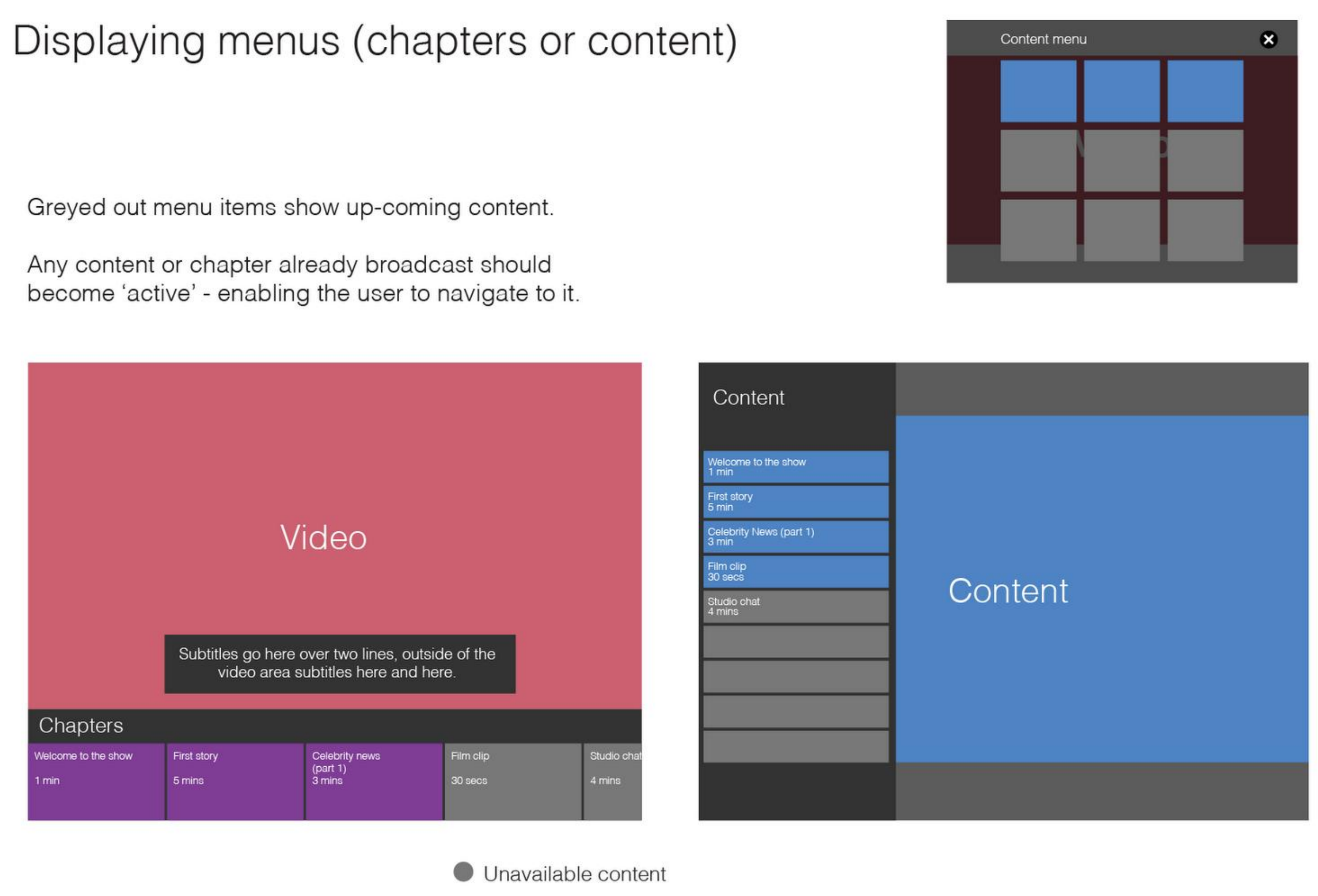


Fig 12 – Displaying event data (initial thoughts)

Chapter and content menus

LIMO’s ability to expose events opens up a variety of opportunities for user navigation and programme presentation. This gives the user the ability to control the experience, but also the content author a method to show the breadth of offering in advance, which should improve content discoverability.


Displaying menus (chapters or content)

Greyed out menu items show up-coming content.

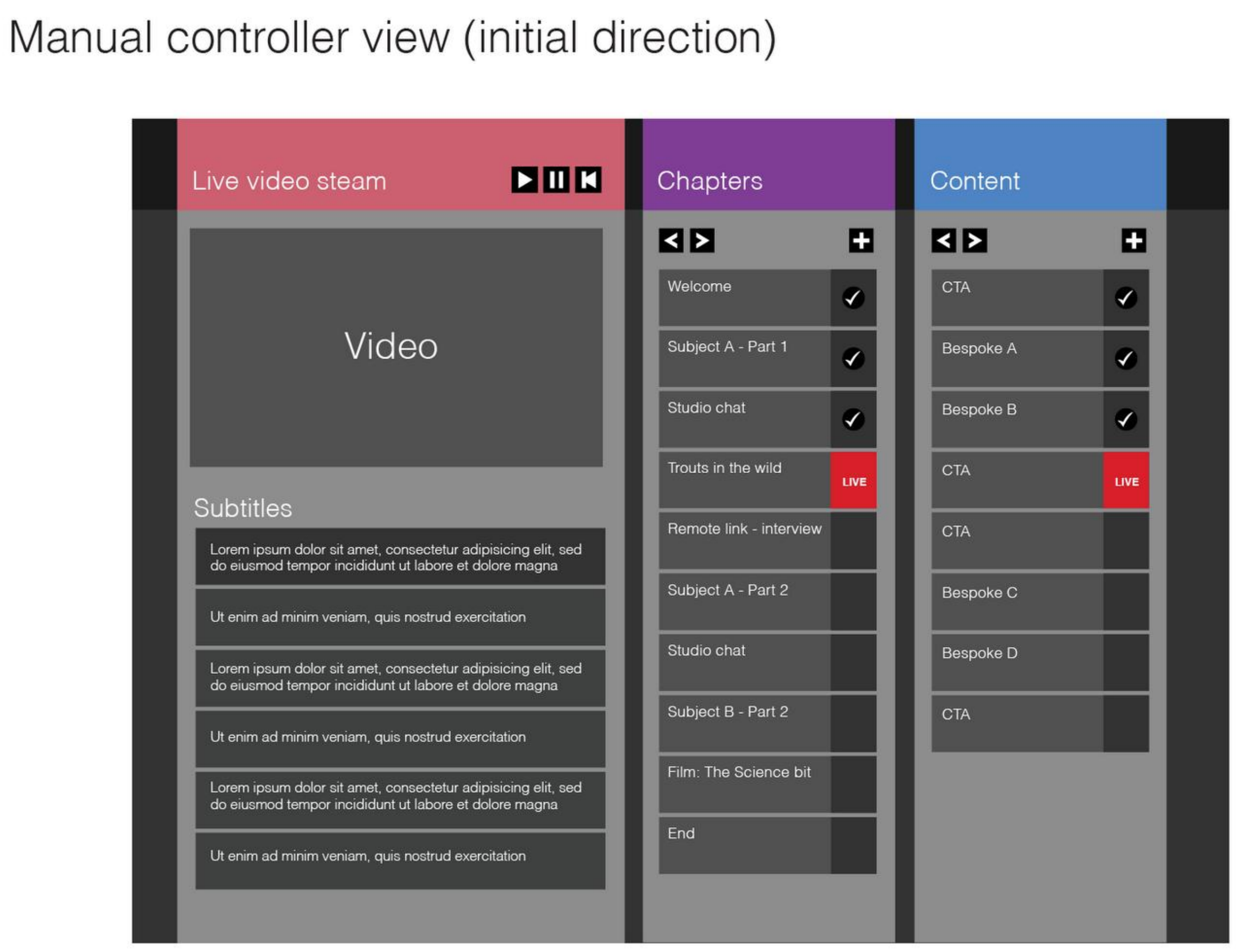
Any content or chapter already broadcast should become ‘active’ - enabling the user to navigate to it.

Content menu
Video
Subtitles go here over two lines, outside of the video area subtitles here and here.
Chapters
Welcome to the show 1 min
First story 5 mins
Celebrity news (part 1) 3 mins
Film clip 30 secs
Studio chat 4 mins
Content
Welcome to the show 1 min
First story 5 min
Celebrity News (part 1) 3 min
Film clip 30 secs
Studio chat 4 mins
Content
● Unavailable content

Fig 13 – Displaying menus

Controller interface

The simplest control interface would simply give the operator a series of lists per event type to activate manually during broadcast. This way the event timeline is built on-the-fly with the results being recorded for on demand and time-shifted users. In this way the events can be ordered to create running orders, each item has a status: used, live or unused.



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna

Ut enim ad minim veniam, quis nostrud exercitation

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna

Ut enim ad minim veniam, quis nostrud exercitation

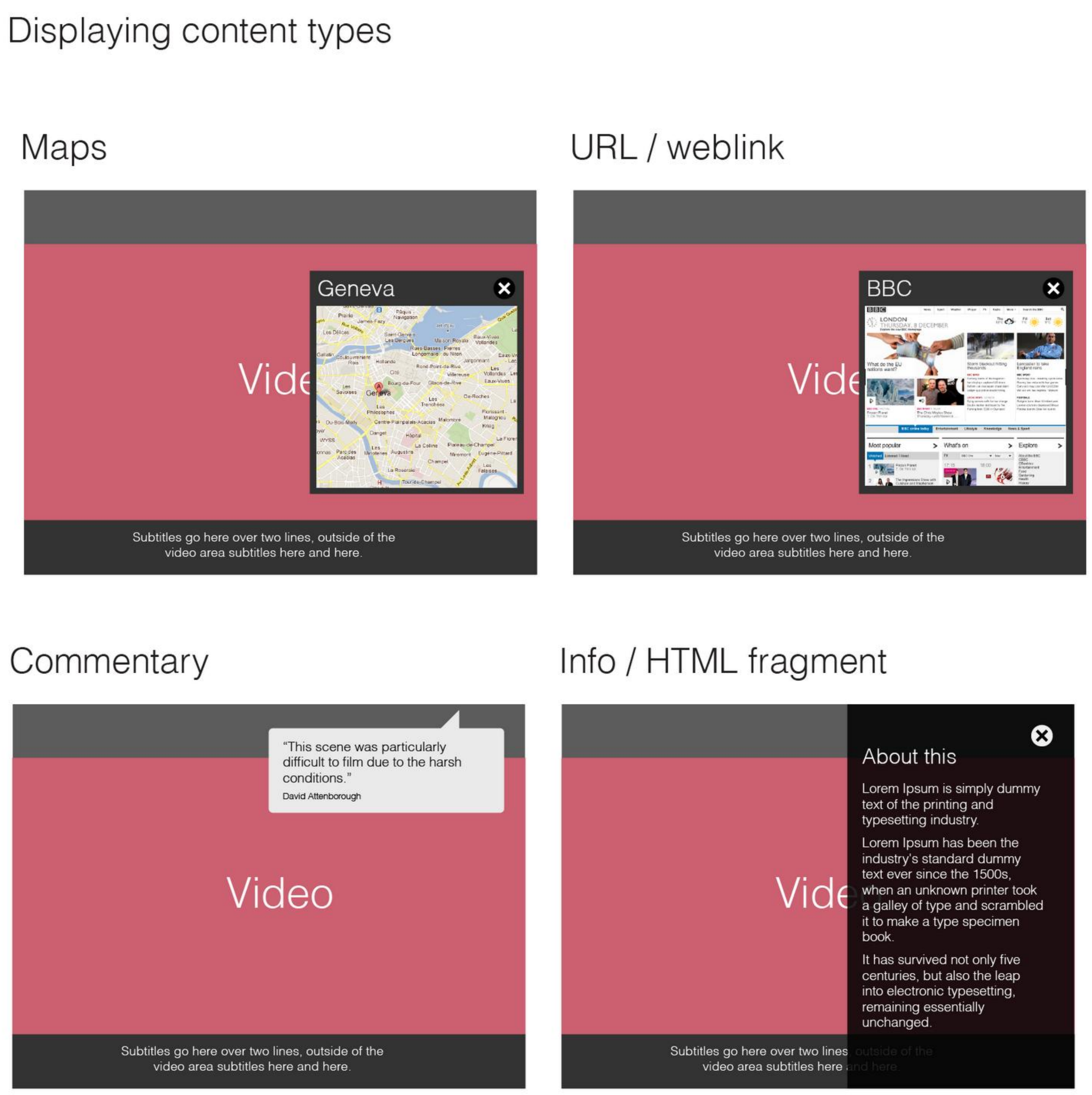
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna

Ut enim ad minim veniam, quis nostrud exercitation

Fig 14 – Controller view

Content module types

We considered different examples of generic information display for overlay content:



Subtitles go here over two lines, outside of the video area subtitles here and here.

"This scene was particularly difficult to film due to the harsh conditions."
David Attenborough

Lorem Ipsum is simply dummy text of the printing and typesetting industry.

Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged.

Fig 15 – Different content types

Event context

Some events will require the flexibility to adapt to play out context. The example below demonstrates a simple overlay graphic inviting the viewer to vote.

The overlay would behave as required for an initial broadcast, however if the user were to pause the video during the broadcast they would enter the time-shifted ‘near live’ context. In this instance there may still be opportunity for the user to vote as well as a prompt to rejoin the live stream which would be especially useful for live event TV.

Finally the on-demand version behaves like a traditional programme repeat with replaced message of ‘vote lines are now closed’. This demonstrates the potential for event based graphics to replace the expensive alternative of re-editing and reversioning.

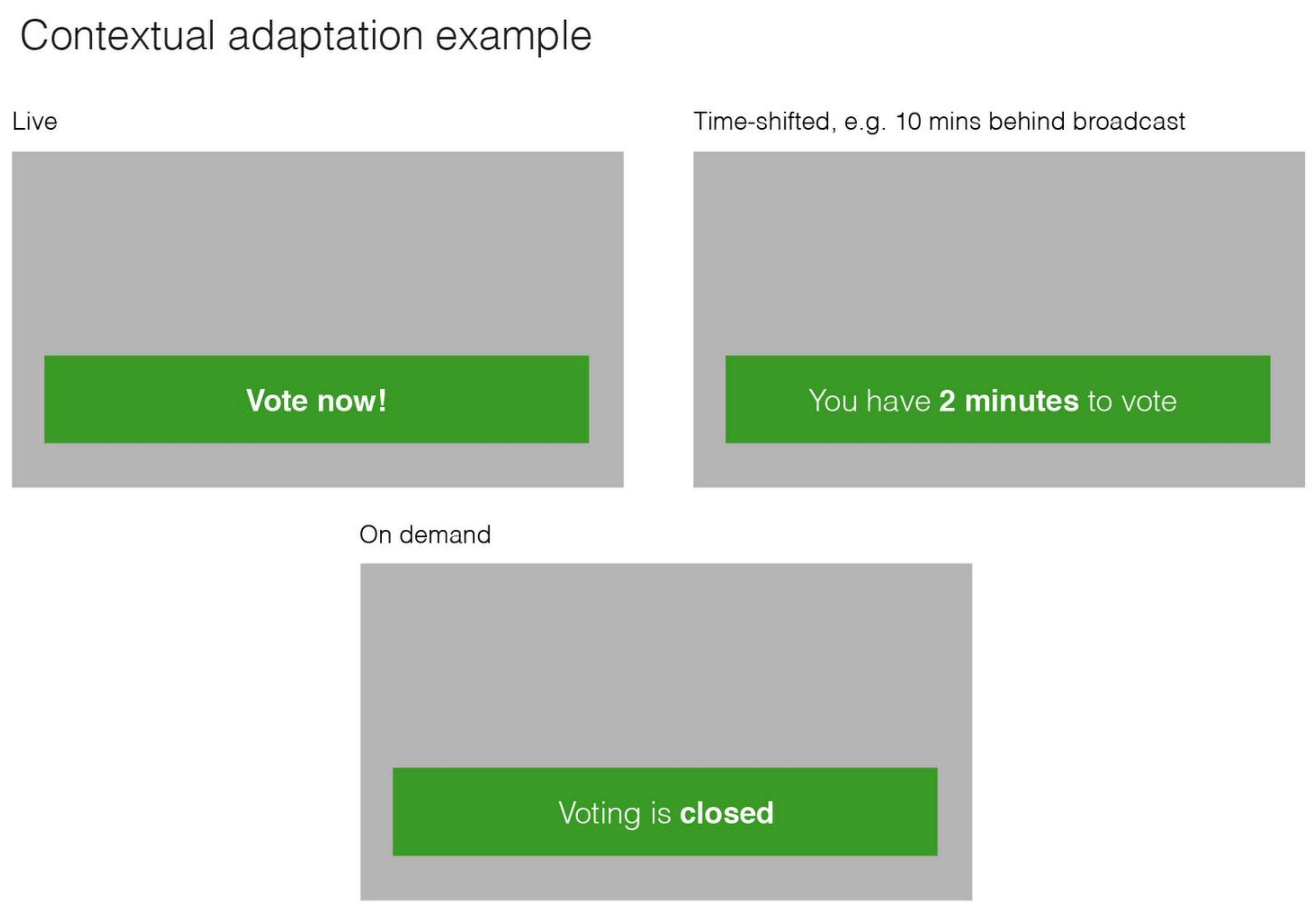

Contextual adaptation example
Live
Vote now!
Time-shifted, e.g. 10 mins behind broadcast
You have 2 minutes to vote
On demand
Voting is closed

Fig 16 – Event context

The following diagram illustrates the messaging required for users joining a live stream at different points of the broadcast.

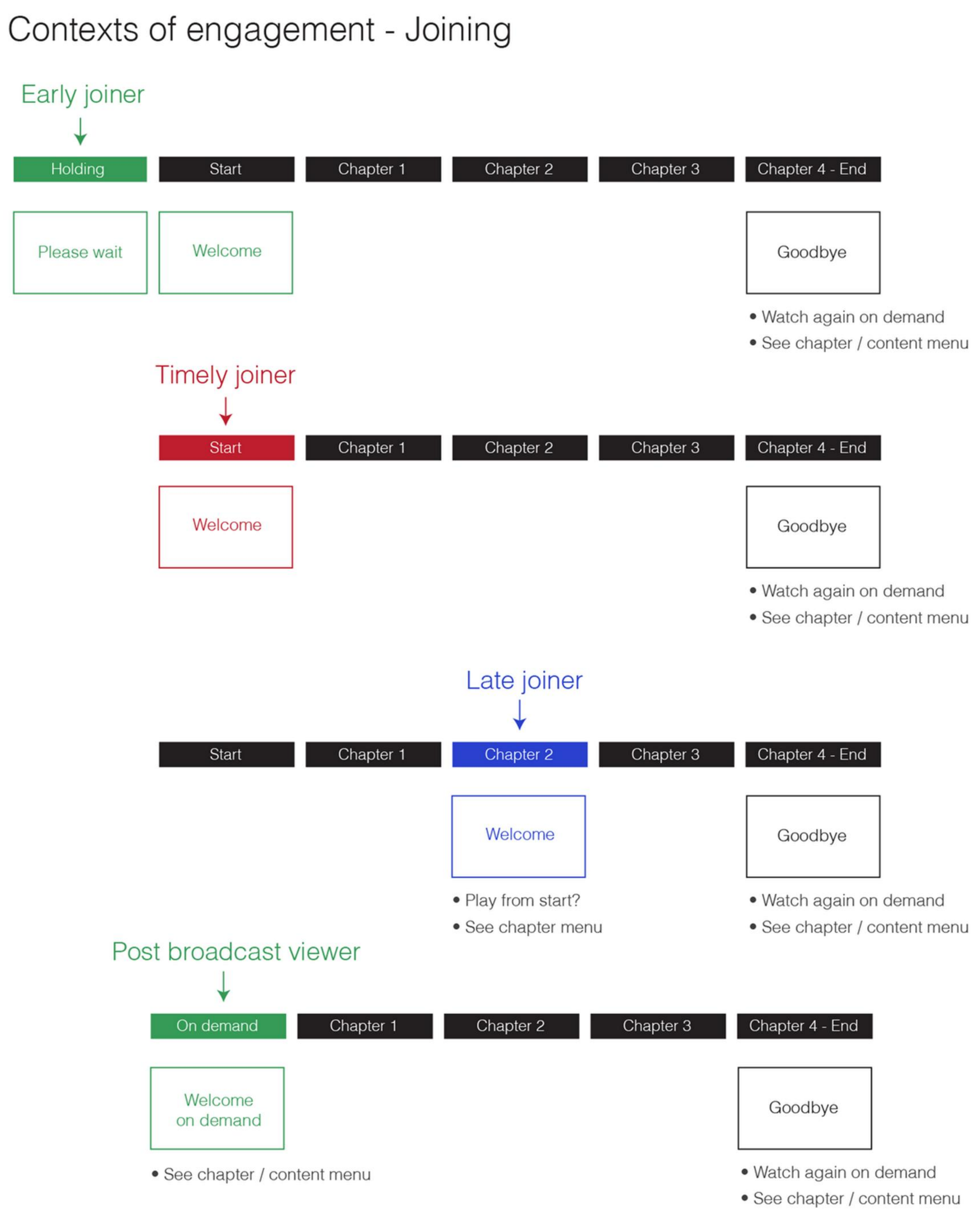


Fig 17 – Joining a stream

Control interface – the asset bin with skeleton timeline

Having considered user display we turned our attention to the control interface for displaying the asset bin with a 'skeleton' timeline. Prepared timelines or 'packages' can themselves be opened on a separate timeline for editing.

The timeline contains the following tracks:

- Packages
- Content
- Graphics
- Subtitles
- Chapters
- Video

Events are dragged into the skeleton timeline to assemble the play out. They can be altered at any time.

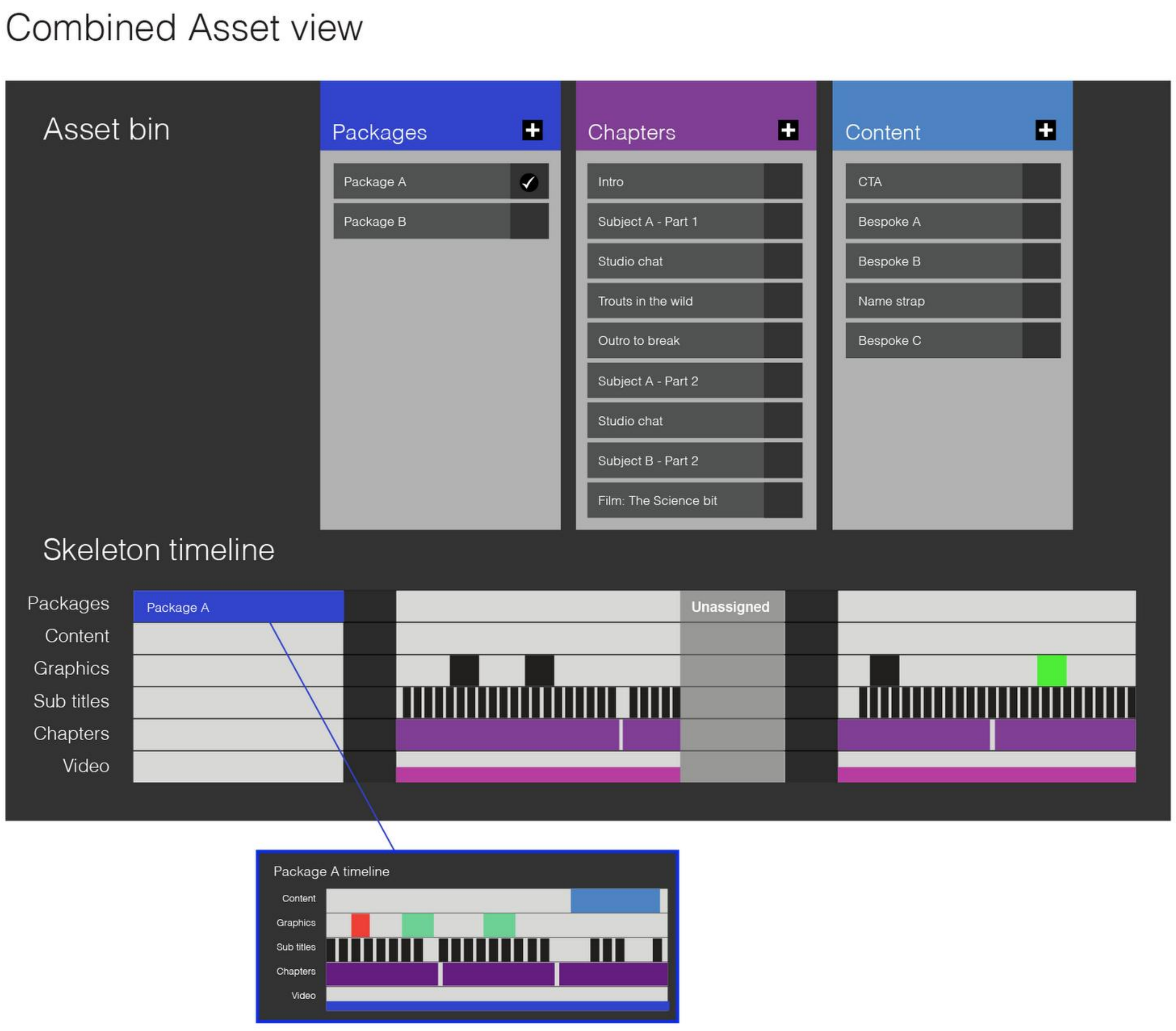


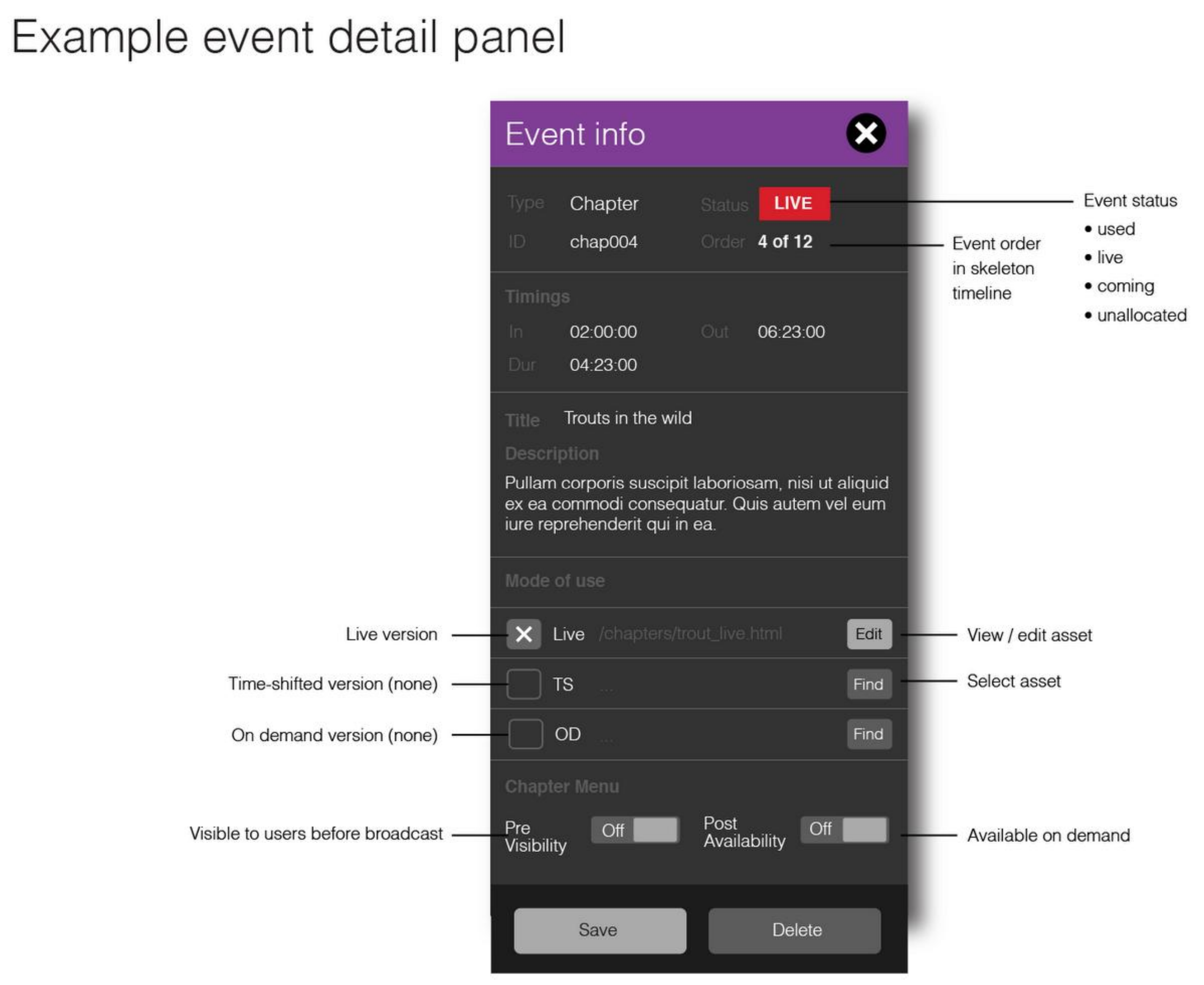
Fig 18 – Combined asset view. Note that Package A is a prepared piece of content embedded within the working skeleton timeline. Here it is exploded out of the main timeline to reveal it's internal content.

Event detail panel

This panel shows the data and functionalities required to give the author of a timeline the power to specify the context of use for each event.

For example, in the *Mode of Use* section the author is able to specify different content for the event for each context: live, time-shifted and on demand. It should be possible for one version to fulfil all contexts.

Additionally *Chapter Menu* allows the author to choose whether the chapter is visible and or available to the users in advance and after of the live
broadcast.



Type Chapter Status LIVE
ID chap004 Order 4 of 12

Timings
In 02:00:00 Out 06:23:00
Dur 04:23:00

Title Trouts in the wild
Description
Pullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur. Quis autem vel eum iure reprehenderit qui in ea.

Mode of use
[✓ Live] /chapters/trout_live.html
TS ...
OD ...

Chapter Menu
Pre Visibility Off Post Availability Off

Fig 19 – Event detail panel

The lifecycle of the timeline

Reading the diagram on the following page from the bottom up, you can see that the skeleton timeline contains a scheduled package to start the programme. A graphic is overlaid on-the-fly during play out (marked by the orange rectangle below it). Following this is a fixed event – the commercial break, treated as a separate package containing its own content.

The middle section combines play out from the skeleton timeline – in this instance a live show that requires flexibility with timings, so a prepared package can be edited to fit available time and manually added to the timeline as an ad hoc element. The final section contains a *live only* event that will not be available to other playback contexts.

Above the packaged timelines (which feed into the main skeleton timeline) we see the combined timeline potentially available to time-shifted viewers (including later joiners).

The on demand timeline is a condensed edit of the live experience where both the commercial breaks and an entire package has been removed. Finally at the top right of the diagram is an example of an on demand only graphics event.

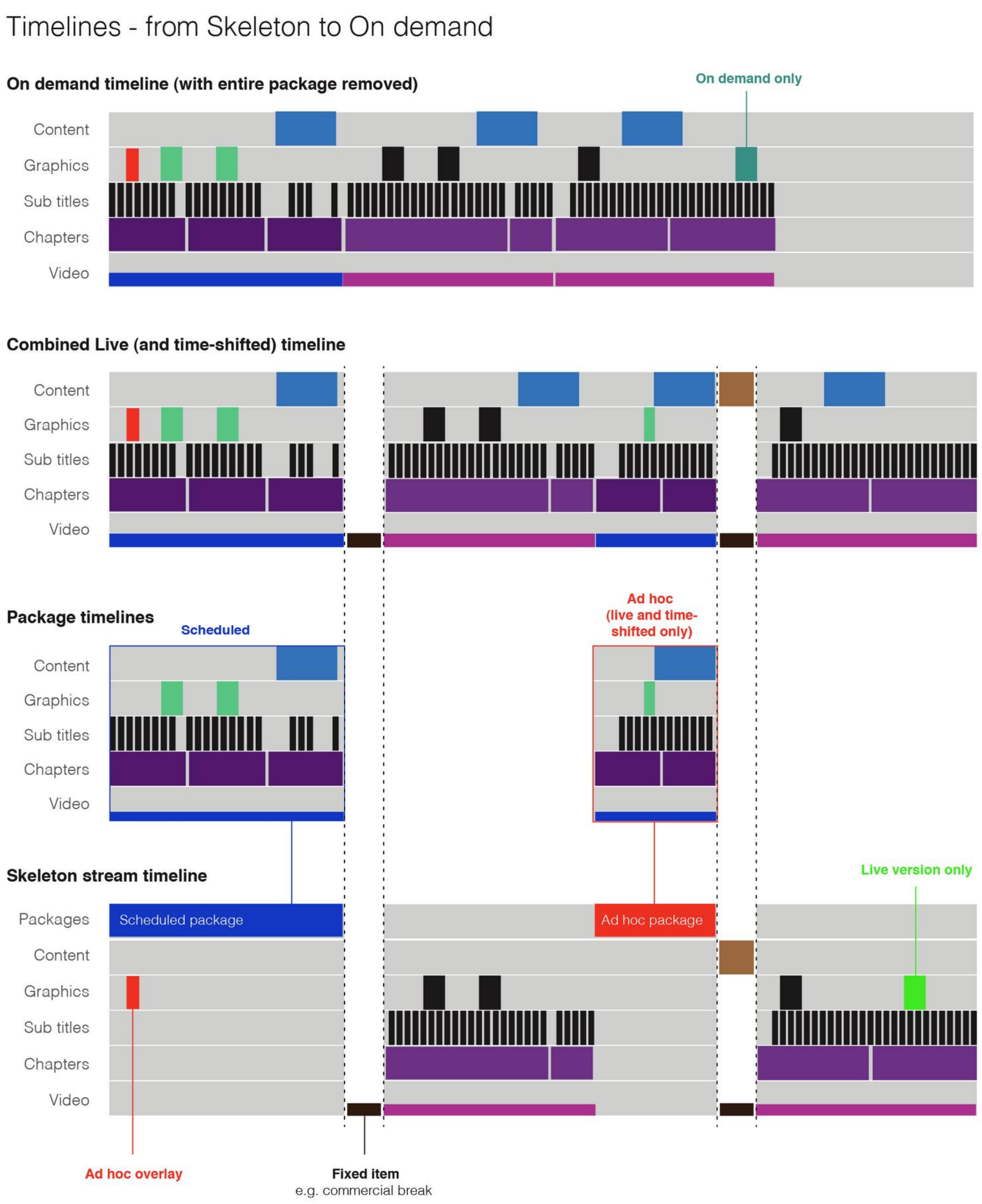
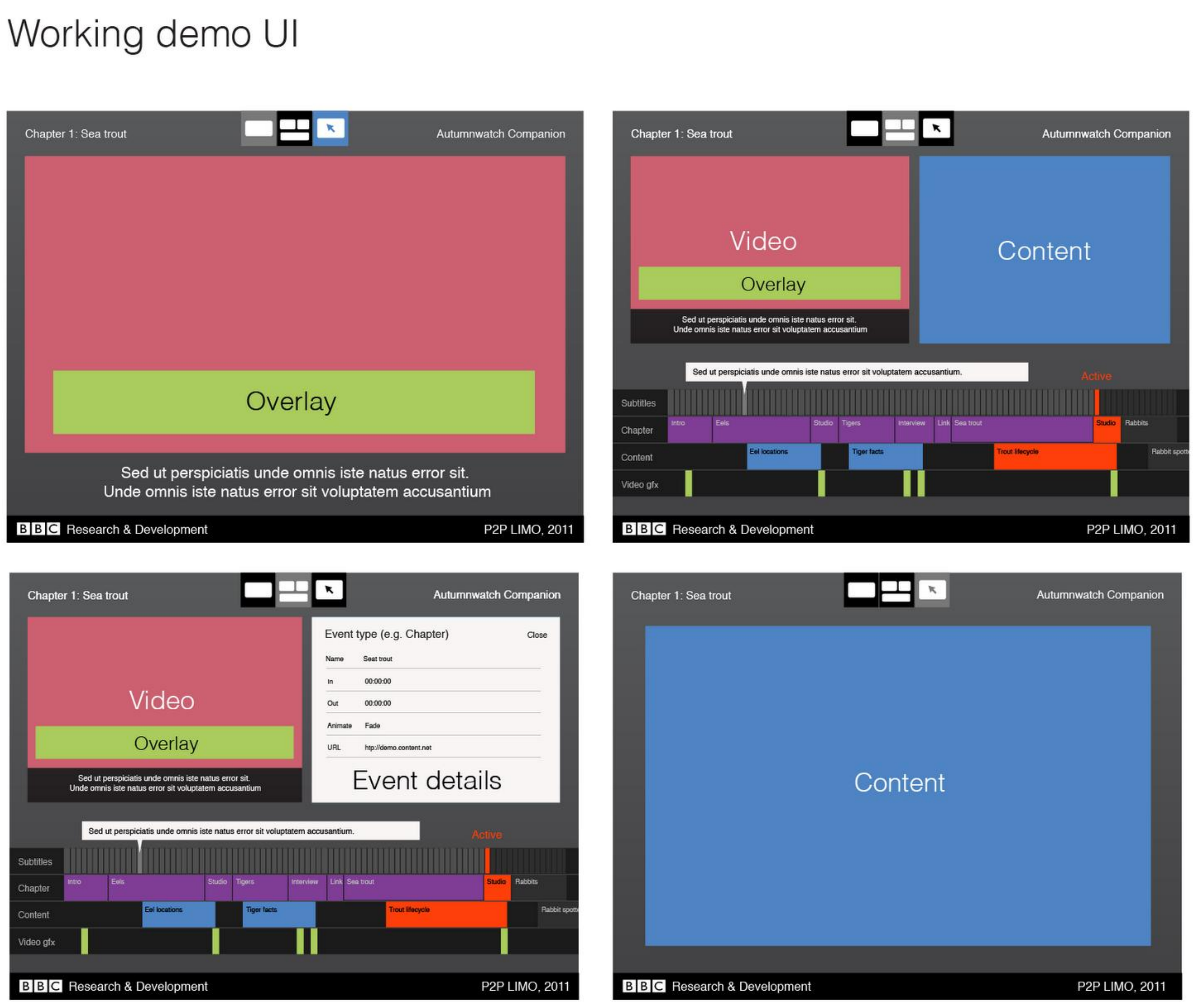


Fig 20 – Timelines

Consumer interface

This final wireframe shows a more refined version of UI from previous iterations and forms the basis of the demonstrator described later in the document. Key features include:

- The addition of buttons at the top of the screen to navigate between views
- Event details accessed by double clicking an event on the timeline



Sed ut perspiciatis unde omnis iste natus error sit.
Unde omnis iste natus error sit voluptatem accusantium

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium.

Fig 21 – User interface for demonstrator

Technical design considerations

Having identified that we want to synchronise out-of-band delivered LIMO events to a live media stream, we set about investigating how current technologies could be used or adapted to meet our goals. The requirements the system must satisfy are:

- The media stream must contain a time reference that is monotonically increasing, so that clients are able to determine the current playback time within the stream without ambiguity. For example, if the system generating the media stream restarts, then the timing reference should not reset back to zero
- The media play-out system must make this time reference available to other server-side components that need to synchronise to the media stream, for example, to generate LIMO events
- The browser client must be capable of playing live media streams, and must make the stream's time reference, the current playback time, available to JavaScript code in the browser
- The LIMO events must be sent such that they are received by clients ahead of the current playback time, so that they can be triggered at the right time in the client

HTML5 video

The HTML5 standard provides `<audio>` and `<video>` elements for media playback in the browser. These elements implement the HTMLMediaElement interface (http://www.w3.org/TR/html5/video.html, retrieved 12 Dec 2011). This interface provides attributes that allow JavaScript code to synchronise to the playing media stream. The HTML5 standard describes these attributes as follows:

currentTime:

> 'Media elements have a current playback position, which must initially (ie in the absence of media data) be zero seconds. The current playback position is a time on the media timeline.
>
> The currentTime attribute must, on getting, return the current playback position, expressed in seconds'

startOffsetTime:

> 'Some video files also have an explicit date and time corresponding to the zero time in the media timeline, known as the timeline offset
>
> Establishing the media timeline: If the media resource somehow specifies an explicit timeline whose origin is not negative, then the media timeline should be that timeline. (Whether the media resource can specify a timeline or not depends on the media resource's format.) If the media resource specifies an explicit start time and date, then that time and date should be considered the zero point in the media timeline; the timeline offset will be the time and date, exposed using the startOffsetTime attribute.'

initialTime:

> ‘initialTime returns the initial playback position, that is, time to which the media resource was automatically seeked when it was loaded. Returns zero if the initial playback position is still unknown.
>
> Media elements have an initial playback position, which must initially (ie in the absence of media data) be zero seconds. The initial playback position is updated when a media resource is loaded. The initial playback position is a time on the media timeline.
>
> The initialTime attribute must, on getting, return the initial playback position, expressed in seconds.’

Also of interest is the timeupdate event, which is generated as the playback position changes during playback:

> ‘The event thus is not to be fired faster than about 66Hz or slower than 4Hz (assuming the event handlers don't take longer than 250ms to run). User agents are encouraged to vary the frequency of the event based on the system load and the average cost of processing the event each time, so that the UI updates are not any more frequent than the user agent can comfortably handle while decoding the video.’

WebM video format

WebM (http://www.webmproject.org) is an open standard for video on the web. It is based on the Matroska container, and uses the VP8 video codec and Vorbis audio codec. Unlike other video formats, such as H.264/AVC, WebM has a royalty-free license.

We decided to investigate live streaming using WebM, rather than other formats such as Ogg, not only because of its royalty-free license position, but also because we found WebM to be well documented and therefore straightforward to understand and modify.

In the WebM format, audio and video data is contained within clusters. Each cluster has a timestamp, which is used to synchronise playback of the audio and video streams. WebM also defines a DateUTC element as part of the Segment Information section, which is part of the stream header. This field is documented as being the ‘date of the origin of timecode (value 0), ie production date’ (http://www.webmproject.org/code/specs/container/). Notice that this agrees with the definition of startOffsetTime in the HTML5 specification.

In order to provide the time reference we needed, the current playback time, we determined that we could set the DateUTC element to a known value, such as the wall-clock time of the start of the stream, and extract this value in the browser and make it available to JavaScript code in the form of the startOffsetTime attribute. Then, JavaScript code could add the startOffsetTime and currentTime values together to obtain the current playback time.

An alternative approach would have been to modify the cluster timestamps themselves, to use an origin value other than zero. We did not do this, however, to avoid potentially introducing problems in code that uses these timestamp values.

Current browser implementations

When we tested current browser versions (Mozilla Firefox 8.0 and Google Chrome 15.0.874.121) with a live WebM stream, we found that both the initialTime and startOffsetTime attributes of the HTML5 `<video>` element were not implemented, returning ‘undefined’ when accessed.

We also found an inconsistency between Firefox and Chrome in the currentTime attribute with live streams:

- In Firefox, currentTime returns the media stream position relative to the point at which the browser joined the stream. For example, if the browser joins the stream 60 seconds after the start of the stream, currentTime returns 0.
- In Google Chrome, currentTime returns the media stream position relative to the start time of the stream. For example, if the browser joins the stream 60 seconds after the start of the stream, currentTime returns 60.

This appears to be a difference in interpretation of the HTML5 specification with respect to what the specification calls ‘media timelines’ and streaming resources (http://www.w3.org/TR/html5/video.html, retrieved 12 Dec 2011):

> ‘In the absence of an explicit timeline, the zero time on the media timeline should correspond to the first frame of the media resource. For static audio and video files this is generally trivial. For streaming resources, if the user agent will be able to seek to an earlier point than the first frame originally provided by the server, then the zero time should correspond to the earliest seekable time of the media resource; otherwise, it should correspond to the first frame received from the server (the point in the media resource at which the user agent began receiving the stream).’

FFmpeg

As our first experiment with generating a live WebM stream, we used FFmpeg to encode video from a webcam into WebM format and output this stream into a ‘stream-m’ streaming server (http://code.google.com/p/stream-m).

WebM support in FFmpeg is provided as part of the libavformat library, but we found that the WebM (Matroska) encoder does not write a DateUTC value to the output stream. We therefore modified libavformat to write DateUTC as the current system time.

We encountered difficulties using stream-m, as it did not appear to always generate a valid output WebM stream, so we decided instead to investigate Flumotion as an alternative open-source streaming server.

GStreamer and Flumotion

GStreamer is an open source set of cross-platform media processing components that can be connected together into complex media processing pipelines. It is widely used in AV processing by clients such as the Totem and Amarok media players, video editors such as PiTiVi, by media servers such as Flumotion and in embedded systems.
Flumotion is a open source streaming and on-demand media server built upon the GStreamer framework which supports distributed AV processing and streaming WebM and Ogg encoded audio/video to web-connected clients.

Demonstrator configuration

Fig 22 shows the system we set up, using Flumotion to serve a live WebM stream from an input WebM video file. The LIMO Event Server listens for timing notifications from the WebM Muxer and plays out a pre-prepared set of events.

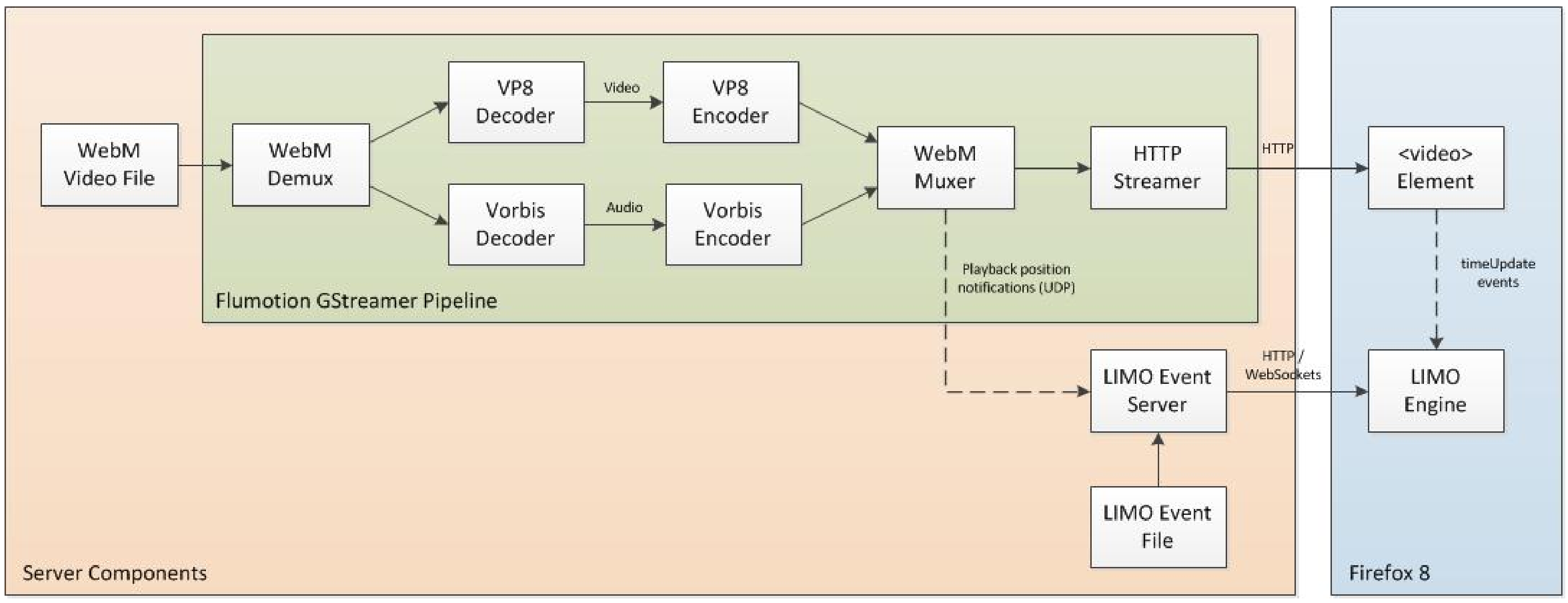


Fig 22 – System set up

In order to synchronise LIMO events to a live WebM stream we had to change the Flumotion Looper component to accept WebM input files instead of Ogg. This involved changing the Looper's GStreamer pipeline to use a Matroska demuxer and a VP8 video decoder. We also changed the GStreamer Matroska muxer component to output timing notifications to a UDP socket for use by the LIMO Event Server.

The LIMO Event Server is a node.js (http://nodejs.org) application that receives timing notifications from the Matroska muxer component in the GStreamer pipeline and schedules playback of LIMO events to client browsers. Browsers connect to the LIMO Event Server using the Socket.IO library (http://socket.io) to receive the events, which are pushed from the server.

Firefox browser

We chose to use Mozilla Firefox as our client development platform because it has built-in WebM support and is open source, with comprehensive information on-line for how to compile (https://developer.mozilla.org/en/Build_Documentation).

We modified Firefox 8.0 to implement the startOffsetTime attribute for `<audio>` and `<video>` elements. This involved modifying libnestegg to extract the DateUTC value from the Segment Information section of the WebM stream (see http://matroska.org/technical/specs/index.html and http://matroska.org/technical/diagram/index.html), and adding the startOffsetTime attribute to the HTMLMediaElement interface, implemented to return the DateUTC value. In doing so we deviated slightly from the HTML5 specification, returning the value as the number of milliseconds since the Unix epoch, as a floating-point number, instead of a JavaScript Date object. This number is readily converted to a Date object from JavaScript code.

Outcomes

The changes we made to the open source software we used (FFmpeg, GStreamer, Flumotion, and Firefox) are all available as patch files as part of the LIMO code release.

Visual presentation demonstator

Visualising LIMO internals

While previous demos focused on building a polished user experience using simple LIMO events and combining events to enable more complex applications such as quizzes (eg Tree of Life) in this instance, we believed it would be more beneficial to focus on a visual representation of the interactive package as a whole rather than something that would be applicable to end users.

Our prototype visualises the internals of the LIMO engine on a timeline, demonstrating what happens when a client connects to a live stream and displays events as they're received from the server and indicates their status of played, currently active or queued to play in the future.

The prototype displays several types of LIMO event: subtitles, chapters, video overlays and interactive content. We have re-purposed interactive content from the BBC R&D Autumnwatch second-screen trial. We also manually created chapter points and example video overlay content. As previously discussed, the interface shows how all this content could be visualised, but also displays a timeline of LIMO's internal events.

Building the demonstrator

The demonstrator interface is built using JavaScript, HTML5 and CSS3. CSS3 Transitions are used throughout to provide animation between the three views: full-page video, full-page interactive content and the timeline view.

Wherever possible, existing open source libraries and UI components were used for example modal dialogues and tooltips from 'Twitter Bootstrap' (http://twitter.github.com/bootstrap/, last retrieved 16 Dec 2011). The timeline component is extracted from Mozilla Popcorn Maker, a web application for authoring video meta-data (http://mozillapopcorn.org/popcorn-maker/, last retrieved 16 Dec 2011).

Joining a stream

When the visual interface is loaded within a browser the client must:

1. Extract the current time within the live media stream exposed by the browser
2. Retrieve any events that have occurred before the client was loaded
3. Establish a connection to the server to receive new events

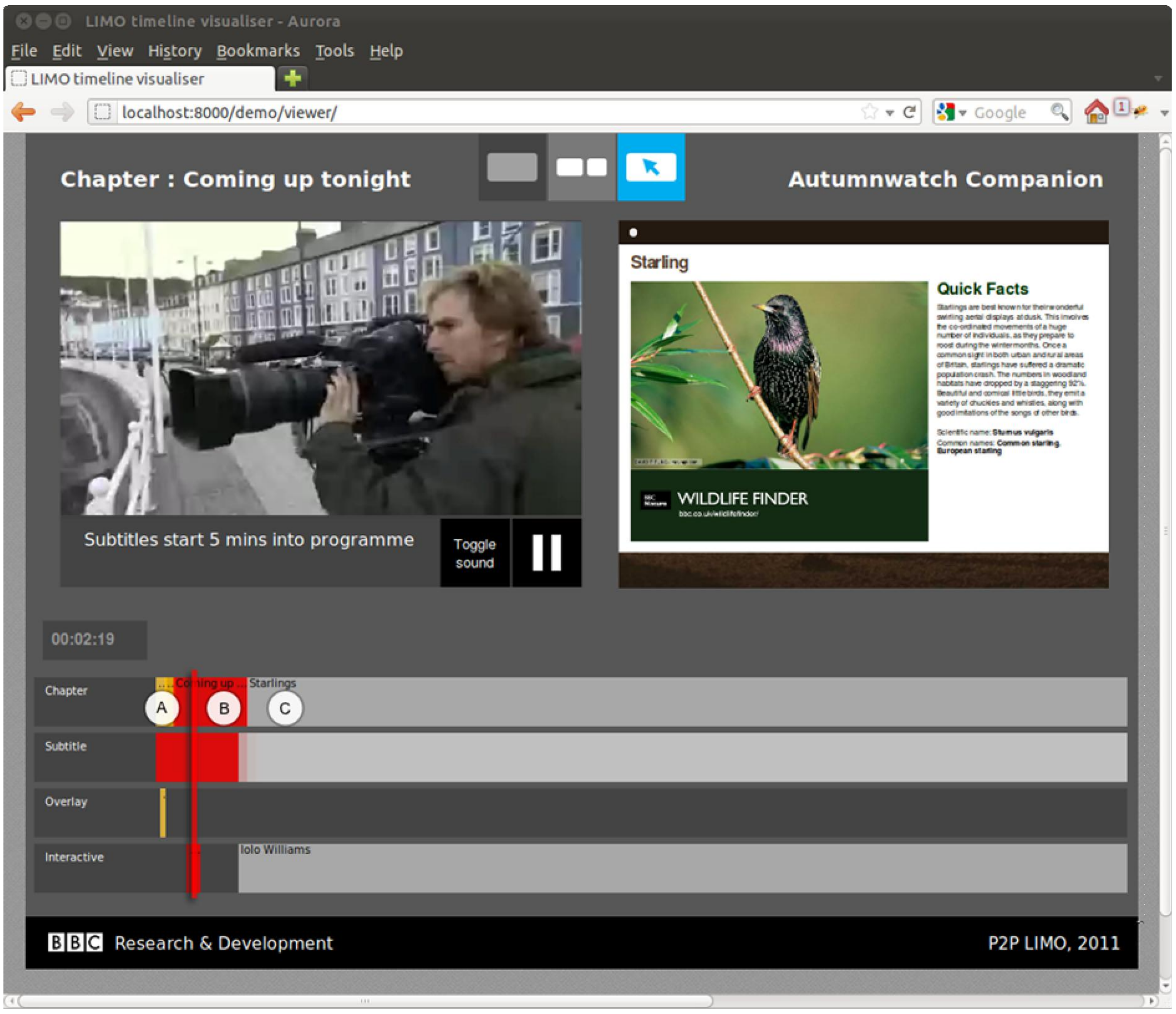

LIMO timeline visualiser - Aurora
File Edit View History Bookmarks Tools Help
localhost:8000/demo/viewer/
Chapter : Coming up tonight
Autumnwatch Companion
Starling
Quick Facts
WILDLIFE FINDER
Subtitles start 5 mins into programme
Toggle sound
00:02:19
Chapter
Subtitle
Overlay
Interactive
Coming up ...
Starlings
A
B
C
Iolo Williams
BBC Research & Development
P2P LIMO, 2011

Fig 23 – Screenshot of visualiser demonstrator where played events (A) are yellow, active events (B) are red and queued events (C) are grey.

For each single logical event, there are two actual events sent by the server, a ‘start’ event and an ‘end’ event:

A ‘start’ event includes timing information and content:

```
        {
            "id"         : "event-chapters-2",
            "event-type" : "start",
            "limo-type"  : "chapter",
            "time"       : 1310052667,
            "data"       :
                {
                    "title"       : "Coming up tonight",
                    "description" : "What's on tonight's programme"
                }
        }
```

An ‘end’ event includes just timing information:

```
{
    "id"         : "event-chapters-2",
    "event-type" : "end",
    "limo-type"  : "chapter",
    "time"       : 1310052736
}
```

When the timeline viewer is notified by the LIMO engine of a new start event, it cannot determine how long the event will be, so it assigns it a duration of the length of the timeline. When the corresponding end event arrives, the event’s representation on the timeline is adjusted to its actual duration.

Since a client can join the live video stream at any point, the LIMO engine must request a collection of historical events from the server. Primarily, this is because the client may join after a ‘start’ event has been sent but before the corresponding ‘end’ event.

There are other events that may have played by the time the client joins the stream, but that are still useful for the client to receive eg a client may display a list of all interactive content so far in the programme as a menu for the user to select from, or a quiz client may display the questions posed so far so the user can catch up with the game.

The choice of what is ‘interesting’ for the client is application-specific. In our demonstrator, the server decides which of these interesting events to send to the client when it joins the stream and initialises the LIMO engine. It sends all events as they’re needed for display on the timeline.

A client application aimed at end-users would probably only send the subset of events that were going to be used in the interface. In addition to requesting historical events, the client must also establish a stream connection to the server so that it receives new events from that point on. The LIMO engine API provides three endpoints to allow client access to this information:

limoEngine.setInitialLoadEventCallback()
On connection to the live stream, the LIMO engine makes an AJAX request to the server which responds with a collection of interesting historical LIMO events. The events passed to this callback are displayed on the timeline (see fig 23 label A).

limoEngine.setStreamEventCallback()
When a stream connection has been established, every event received from the server will invoke this callback. This indicates that a stream event has been received by the client and stored within the LIMO engine, ready for activation (see fig 23 label C).

limoEngine.registerEventHandler()
Fired whenever whenever a LIMO event becomes activate or inactive. This is used to change the state of events on the timeline when they’re activated (see fig 23 label B) and then deactivated (see fig 23 label A). This is also used to update the other user interface elements such as displaying the chapter title, triggering interactive content and the video overlay.

These API methods provided by the LIMO engine are essential enablers for the client interfaces described earlier in the document (see ‘Chapter and content menus’ section). For example, to show the user a menu of all interactive content in the programme, or to show a list of chapter points to navigate through the stream, the client must be able to access a list of historical events.

Running the code

The code is available as a Linux virtual machine that's compatible with VirtualBox. The project contains a README file that explains how to access the demo.

The virtual machine is available for download: http://limo.prototyping.bbc.co.uk.downloads

The source code is also available in the P2P-Next Subversion repository: https://ttuki.vtt.fi/svn/p2p-next/LIMO/trunk/LIMO. The README file contains instructions for how to install and run the demos.

The project includes several demos, described briefly below:

Download Events Demo

This demo shows subtitles and chapters against the BBC R&D TV video. The browser client downloads the entire list of LIMO events before video playback starts.

AJAX Events Demo

This demo shows subtitles and chapters against the R&D TV video, and demonstrates "chunked" download of LIMO events. During video playback, the browser client makes AJAX requests every 10 seconds to retrieve upcoming LIMO events.

Stream Events Demo

This demo shows subtitle events played against a live video stream. Events are delivered to the browser using Socket.IO, which in recent browsers (eg Firefox 8.0) uses Web Sockets as the transport.

WebM Live Stream with Timestamps Demo

This demo is shows the result of our investigations into providing a time reference suitable for synchronising real-time generated LIMO events. The demo shows various time values obtained from the playing video:
currentTime: The `currentTime` attribute of the `<video>` element (specified in HTML 5)
currentTimeAbsolute: The absolute time reference of the media stream (non-standard)
startOffsetTime: The offset to add to `currentTime` to obtain the absolute time reference of the media stream (specified in HTML 5, but not implemented in current browser versions)
startTime: The time of the browser joining the stream, relative to the start of the stream (non-standard)
dateUTC: The absolute timestamp of the start of the media stream, from the WebM stream header (non-standard)

Autumnwatch Live and On-Demand Timeline Viewer Demos

These demos are described fully in the 'Visual presentation demonstrator' section of this document.

Glossary

- Current stream time – the time a video stream has reached live (even though all users may not be at this point ie if they may have time-shifted playback)
- Current playback time – the point at which a user is watching at that moment (which is the same as current stream time when watching live but different when watching on demand or time shifted). Note that current playback time ≤ current stream time

Appendix 1 – LIMO data format

This section presents a detailed description of the LIMO data format, including the LIMO manifest and LIMO event types.

LIMO manifest

The LIMO manifest describes all the parts that are needed to compile and run a synchronised LIMO experience. The manifest specifies a list of HTML resources and event resources, which allows HTML presentations and events to be associated together in different combinations for presentation on different capability devices. The manifest is designed to expose as much as possible how components of the LIMO experience are coupled together. This is the basis for extensibility and mash-up. The manifest may not always be a static file. For instance, it may change as one of the event-resources changes from live to on-demand.

The `transport` type for each event resource signifies whether the resources are to be downloaded, retrieved via AJAX or streamed.

```
{
  "limo": {
    "html-resources": [
       {
         "screen-type": "normal",
         "link": "index.html",
         "using-resources": [ "events1" ]
       }
    ],
    "event-resources": [
      {
         "link": "subtitle_and_chapter_events.json",
         "id": "events1",
         "language": "en",
         "limo-types": [ "subtitle", "chapter" ],
         "transport": "download"
       }
    ],
    "manifest": {
      "link": "manifest.json"
    },
    "updated": 1302090165
  }
}
```

LIMO events

The basic element of timed metadata in LIMO is the “event”, which defines either a point in time or a period of time within the associated media content, together with the event’s metadata. LIMO defines a set of event types, based on common media types. These form a basis for common tools and components in the presentation layer. Presentation authors can expect event data to be of the specified format. This is not a fixed set of types. Custom types and vendor specific types may be easily specified just by giving a new type name.
The LIMO engine treats all event types in the same way, so new event types can be introduced without affecting the design of the LIMO engine. LIMO events are encoded in JSON format, which is easily consumed by modern web browsers, which have built-in JSON parsers.

audio

The `audio` event links a segment in the content item to related audio

```
{
  "start": 22.0,
  "end": 221.0,
  "limo-type": "audio",
  "data": {
    "link": "http://server/theaudio.mp3",
    "type": "audio/mp3"
  }
}
```

chapter

A `chapter` event contains the title and description of a chapter segment within the media content, to provide a visual indication of the current and available chapters, and to allow the user to navigate through the content.

```
{
  "start": 15.0,
  "end": 123.345,
  "limo-type": "chapter",
  "data": {
    "icon": {
      "type": "image/jpeg",
      "link": "http://server/chapter1.jpeg"
    },
    "description": "This chapter blah blah...",
    "title": "Chapter 1"
  }
}
```

comment

A `comment` event contains a comment from a user, which could come from a chat application or a social media site such as Twitter.

```
{
  "time": 123.0,
  "limo-type": "comment",
  "data": {
    "text": "This video is great!",
    "user": {
      "display_name": "johnd",
      "full name": "John Doe",
      "image_url": "http://example.com/users/123/profile_image.png",
      "user_profile_url": "http://example.com/users/123/"
    }
  }
}
```

geolocation

A `geolocation` event contains information on the location of a scene:

```
{
  "start": 0.0,
  "end": 10.0,
  "limo-type": "geolocation",
  "data": {
    "title" : "My House",
    "description": {
      "type": "text/html",
      "text": "This is where I live"
    },
    "image": {
      "type": "image/jpeg",
      "link": "http://server/image.jpeg"
    },
    "location": {
      "lat": 37.4219720,
      "lng": -122.0841430
    }
  }
}
```

image

An `image` event links a segment in the content item to a related image

```
{
  "start": 180.0,
  "end: 223.3,
  "limo-type": "image",
  "data": {
    "link": "http://server/theimage.jpg",
    "type": "image/jpeg"
  }
}
```

information

An `information` event contains further information about what is shown on screen.

```
{
  "start": 234.0,
  "end": 250.0,
  "limo-type": "information",
  "data": {
    "title": "Mammals",
    "type": "text/html",
    "content": "Mammals are members of a class of <a
href='http://example.com/link.html'>air-breathing</a> vertebrate animals characterised by
the possession of hair, three middle ear bones, and mammary glands functional in mothers
with young."
   }
}
```

poll

A `poll` event allows content providers to ask for feedback during a broadcast. The poll event must contain a URI to which the LIMO client should post the user’s response.

```
{
  "id": "poll1234",
  "start": 100.0,
  "end": 150.0,
  "limo-type": "poll",
  "data": {
    "question": "Who was your favourite performer?",
    "options": ["Cher Lloyd", "Diva Fever", "Aiden Grimshaw", "Wagner Carrilho"],
    "submit-uri": "http://server/limo-poll/"
  }
}
```

quiz

The quiz event allows content providers to build play-along quiz games. Note the example below shows an in-line question - correct answers are provided along with the question itself. Other formats could be supported.

```
{
  "start": 12,
  "end": 24,
  "limo-type": "quiz",
  "data": {
    "questionType": "multipleChoice",
    "questionNumber": "1",
    "question": "1. From where, what or who do the band Richmond Fontaine take their
name?",
    "choices": [
      "From a town in the Pacific Northwest of North America",
      "From a make of cigarette popular in British Columbia",
      "From an American expat they met in Mexico"
    ],
    "answer": 2,
    "feedback": [
      "Wrong! The band is named after an American expat who had helped bassist Dave Harding
when his car was stuck in the desert in Baja Mexico.",
      "Wrong! The band is named after an American expat who had helped bassist Dave Harding
when his car was stuck in the desert in Baja Mexico.",
      "Correct! The band is named after an American expat who had helped bassist Dave
Harding when his car was stuck in the desert in Baja Mexico."
    ]
  }
}
```

subtitle

A `subtitle` event contains timed text for closed captioning. Subtitles can be plain text or HTML and may contain multiple lines:

```
{
  "start": 124.4,
  "end": 155.0,
  "limo-type": "subtitle",
  "data": {
    "type" : "text/plain",
    "lines" : ["Line 1 goes here", "Line 2 goes here"]
  }
}
```

video

The `video` event links a segment in the content item to a related video

```
{
  "start": 35.0,
  "end": 65.0,
  "limo-type": "video",
  "data": {
    "link": "http://server/thevideo.mp4",
    "type": "video/mp4"
  }
}
```

weblink

A `weblink` event contains a link to relevant content on the web.

```
{
  "start": 123.0,
  "end": 321.0,
  "limo-type": "weblink",
  "data": {
     "title": "Steve Jobs (Wikipedia)",
     "type": "text/html",
     "link": "http://en.wikipedia.org/wiki/Steve_Jobs"
  }
}
```

LIMO event formats

There are two format representations for LIMO events: “download format” and “stream format”.

In download format, the start and end time of the event are contained within the same JSON object. This format is suitable for on-demand content where the timing of all events can be prepared in advance.

```
{
  "start": 124.4,
  "end": 155.0,
  "limo-type": "subtitle",
  "data": {
    "type" : "text/plain",
    "lines" : ["Line 1 goes here", "Line 2 goes here"]
  }
}
```

In stream format, the start and end of each LIMO event is encoded as a distinct JSON object. Each LIMO event must have a unique `id` value so that the start and end can be paired together.

Start event:

```
{
  "time": 124.4,
  "event-type": "start",
  "limo-type": "subtitle",
  "id": "subtitle-1",
  "data": {
    "type": "text/plain",
    "lines": ["Line 1 goes here", "Line 2 goes here"]
  }
}
```

End event:

```
{
  "time": 155.0,
  "event-type": "end",
  "id": "subtitle-1"
}
```

Appendix 2 - LIMO engine API

This appendix presents a detailed description of the LIMO engine API.

Synopsis

```
var limoEngine = new LIMO.Engine();

var limoManifest = new LIMO.Manifest();
limoManifest.load("http://example.com/limomanifest.json");

var eventResources = limoManifest.getEventResources();

if (eventResources) {
  $.each(eventResources, function(index, item) {
    limoEngine.load(item["link"]);
  });
}

limoEngine.registerEventHandler("subtitle", function(event, isActive) {
  if (isActive) {
    var subtitle = event.data["lines"].join(" ");
    $("div#subtitle").html(subtitle);
  }
  else {
    $("div#subtitle").html("");
  }
});

var videoElement = $("video#mainVideo")[0];
limoEngine.setMediaElement.(videoElement);
```

The main classes that the application uses are LIMO.Manifest, LIMO.Engine, and LIMO.Event.

LIMO.Manifest

The LIMO.Manifest class is responsible for loading a LIMO manifest document and providing access to its contents to clients.

constructor
Creates a new LIMO.Manifest object.

```
var limoManifest = new LIMO.Manifest();
```

load(manifestUrl)
Loads the LIMO Manifest from the supplied URL. This URL expected to return a JSON data structure.

```
var manifestUrl = "http://example.com/limo/manifest.json";
limoManifest.load(manifestUrl);
```

getEventResources()
Returns an array of the LIMO event resources from the manifest.

```
var eventResources = limoManifest.getEventResources();
```

Each event resource is an object, for example:

```
{
  "id": "sub1",
  "limo-types": ["subtitle", "chapter"],
  "transport": "download",
  "language": "en",
  "link": "http://server/somesubtitles.json"
}
```

LIMO.Engine

The LIMO.Engine class is responsible for triggering LIMO Events according to the current time of an associated LIMO Clock. The LIMO Clock provides timing notifications to the LIMO Engine from the playback position of an audio or video media stream, which may either be displayed either within the same web page as the LIMO content or on a remote device.

constructor

Creates a new LIMO.Engine object.

```
var limoEngine = new LIMO.Engine();
```

load(eventsUrl)

Adds events from the LIMO event file at the supplied URL. This URL expected to return a JSON data structure containing LIMO events in download format. Clients can call this method any number of times to load multiple LIMO event files.

```
var eventsUrl = "http://example.com/limo/events.json";
limoEngine.load(eventsUrl);
```

setAjaxUrl(ajaxUrl)

Sets the URL for the LIMO engine to request events via AJAX calls. During media playback, the LIMO engine will make requests to this URL, with start_time and end_time query parameters specifying the time range of interest. The response should be a JSON document with stream format LIMO events.

```
limoEngine.setAjaxUrl("http://example.com/limo/events");
```

setStreamUrl(streamUrl)

Sets the URL for a Socket.IO (see http://socket.io) connection to receive events.

```
limoEngine.setStreamUrl("http://example.com/limo/events");
```

registerEventHandler(limoType, handler)

Registers a handler function for LIMO Events of the specified LIMO type. Handler functions are invoked at the start and end times of each Event. The handler function receives two arguments: a LIMO.Event object that contains the event data, and a flag that indicates whether the Event is to be activated or deactivated. The handler function should take appropriate action depending on the type of event, such as displaying or removing a subtitle, updating a chapter selection, etc

```
limoEngine.registerEventHandler("subtitle", function(event, isActive) {
  if (isActive) {
    // Display the subtitle in the event.
  }
  else {
    // Remove the subtitle.
  }
});
```

update(time)
Updates the LIMO presentation to the correct state for the given time in the media playback (in seconds). Updating may involve activating or deactivating LIMO Events as appropriate.

```
var time = 12.34;
limoEngine.update(time);
```

setMediaElement(element)
Binds the LIMO engine to an HTML5 `<audio>` or `<video>` element by adding a listener for `timeupdate` events. The listener invokes the LIMO engine's `update` method.

LIMO.Event

LIMO.Event objects are created internally by the LIMO Engine, and are passed to client code via the handler functions registered with the LIMO Engine. LIMO.Event objects have public attributes listed below, which may be used from within handler functions. Handler functions may modify elements under the data attribute, if modification is allowed by the LIMO Type or appropriate to the presentation. Handlers should not modify the limoType, startTime, endTime or isActive values.

limoType
The LIMO Type identifier of the event.

startTime
The start time of the event, in seconds.

endTime
The end time of the event, in seconds.

isActive
A flag that indicates whether the event is active, ie the current media playback position lies within the range [startTime, endTime).

data
The type-specific data associated with the event. Refer to Appendix 1 for examples.

Appendix 3 – MSV (related work by NORUT)

The focus of this document so far is synchronisation between media components, *internal* to a web browser. In particular, the document discusses how various kinds of extra material may be synchronised with the progression of live video, thereby enhancing the basic video experience in profound ways.

In parallel with this work, NORUT has focused on synchronisation between media components on *different* devices. This work brings another dimension to the utility of LIMO, by taking LIMO from single screen to multiple screens. The idea is that media components on multiple screens should behave as if part of a single media experience. In order to make this possible, progress navigation must be shared between devices. For instance, if the video stream is paused at the TV, the iPad showing synchronised content in a browser must pause too, immediately.

NORUT has implemented a generic, lightweight and effective mechanism that allows a large number of devices (including web browsers) to participate in a single synchronised media presentation. The mechanism is based on the novel concept of Media State Vector (MSV). A MSV is a compact representation of ‘motion’. The synchronisation mechanism works on ‘motion’ exclusively (not content), allowing a single ‘motion’ to be shared and synchronised between many devices. Any media presentation that can be expressed as a deterministic function of ‘motion’, can be synchronised using this mechanism. In particular, these requirements are perfectly met by LIMO-style media presentations, where presentations are synthesized from multiple components, synchronised in real time, by the client itself, based on a single source of motion.

This work has many applications and implications. To name only a few, it means that a broadcaster such as the BBC may start to produce content *designed* for multiple screens in the same way LIMO suggests design of content for a single browser, by means of synchronising multiple independent media components. It also takes away the limitation that time-sensitive, web-based extra material, such as Formula One lap times or Twitter comments, is only available for live TV content. Using the MSV synchronisation mechanism, such time-sensitive material may be re-played on demand, to synchronise with time-shifted or on demand TV content.

The planned date for NORUT to publish its work on the MSV synchronisation mechanism is in the first half of 2012.

Appendix 4 – Smoothlink (related work by KTH)

During 2011, KTH conducted research and development in the related area of media migration. This has led to the development of Smoothlink: a web-based framework for media context migration.

If a user needs to switch devices while consuming content, then doing so is fairly easy for text as all a user has to do is transfer a URL (eg manually or with an email/ instant message) and easily pick up where he or she left off. However, no good solution exists for managing media in a similar way.

For example, if you are watching a video clip on a smartphone, you may want to continue watching it on your big screen TV once you get home. Unfortunately, there is currently no simple mechanism to migrate media context information related to what a user is currently listening or watching and at what point in time between two devices.

An orderly hand-off is a process that migrates media context between two devices, enabling the user to suspend her multimedia activity and resume it in another device. With users increasingly being surrounded by internet-enabled media devices, being able to migrate media context is something that is becoming increasingly important, both in the workplace and privately.

To devise a cross-platform online solution for media context migration becomes an even more difficult task as devices are heterogeneous in terms of screen resolution, internet access networks, operating systems, web browsers or transport mechanisms. On the other hand, current heterogeneous devices support one common feature – the ability to connect to the internet by means of web browsers that render HTML5.

KTH has designed and implemented a fully web-based framework called Smoothlink that enables the user to easily migrate media context between her personal internet-enabled devices, regardless of the networks they may be connected to – wired, wireless, or 3G.

Using Smoothlink, a user can discover devices and migrate media context with minimal intervention. Context can, for instance, be automatically migrated from the smartphone to the big screen TV if the user pauses the content on the smartphone after arriving at home. Experimental results show that the latency to migrate media context between devices is around 3 seconds, with 6 seconds in the worst case.

KTH has devised a new handshake protocol for media context migration between internet-enabled devices. An origin device can, on user request, migrate context to a target device. In addition to media context migration, the framework employs media playback adaption for each device accordingly.

KTH also specifies an interface that enables Smoothlink to communicate with any of the underlying transport protocols using a simple HTTP gateway. The interface employs two simple methods to facilitate the communication of the framework with the transport protocol, namely:

1. Retrieve information for the current media stream
2. Set initial information for a new media stream

Smoothlink design is modular so add-ons and new modules can be added to enhance existing functionality with, for example, new transport protocols, compatibility with other solutions, or different media adaptation algorithms.

As an example, KTH has integrated the swift transport protocol to retrieve online content.