



**Fachhochschule
Kaiserslautern**
University of Applied Sciences

**Informatik und
Mikrosystemtechnik**
Zweibrücken

Studiengang

Medieninformatik

Bachelorarbeit

Handling of Multimedia Files in the Invenio Software

vorgelegt von

Björn Oltmanns

31. August 2011

Betreuung: Prof. Dr. Jörg Hettel

Zweitkorrektur: Prof. Dr. Bernhard Schiefer



Abstract

'Handling of multimedia files in the Invenio Software' is motivated by the need for integration of multimedia files in the open-source, large-scale digital library software Invenio, developed and used at CERN, the European Organisation for Nuclear Research. In the last years, digital assets like pictures, presentations podcasts and videos became abundant in these systems and digital libraries have grown out of their classic role of only storing bibliographical metadata. The thesis focuses on digital video as a type of multimedia and covers the complete workflow of handling video material in the Invenio software: from the ingestion of digital video material to its processing on to the storage and preservation and finally the streaming and presentation of videos to the user. The potential technologies to realise a video submission workflow are discussed in-depth and evaluated towards system integration with Invenio. The focus is set on open and free technologies, which can be redistributed with the Invenio software. The state-of-the-art of these open technologies is discussed. Based on the results of the analysis, a conceptual solution as well as a technical solution is presented. Completely new modules for the processing of videos in Invenio are introduced and diverse enhancements and changes to the existing Software for better integration are portrayed. These solutions are implemented as portable base code, applicable for every Institution using Invenio for their digital libraries, as well as prototypical code, which serves as a demonstration on how to implement a complex individual video submission workflow with Invenios newly added video capabilities.



Table of Contents

1	INTRODUCTION.....	1
1.1	MOTIVATION.....	1
1.2	ENVIRONMENT.....	1
1.3	ASSIGNMENT	1
1.4	THESIS AND INTERNSHIP.....	1
2	DESCRIPTION OF THE EXISTING SYSTEM.....	3
2.1	INVENIO OVERVIEW.....	3
2.1.1	MARC21 AND MARCXML.....	4
2.1.2	INVENIO MODULES.....	6
2.2	THE MEDIA ARCHIVE WORKFLOW.....	9
2.3	WHY RE-CONCEPTION?	11
3	SYSTEM ENHANCEMENT REQUIREMENTS.....	13
4	TECHNOLOGY OVERVIEW	14
4.1	VIDEOS ON THE WEB	14
4.1.1	INTRODUCTION TO FLASH VIDEO	14
4.1.2	INTRODUCTION TO HTML5 VIDEO	14
4.1.3	VIDEO FORMATS AND CODECS	16
4.1.4	HTML5 VIDEO MARKET PENETRATION	19
4.1.5	CONCLUSIONS ON VIDEO FORMATS.....	22
4.2	VIDEO QUALITY, RESOLUTIONS AND BITRATES.....	23
4.3	VIDEO ENCODING TOOLS	24
4.3.1	FFMPEG, LIBAVCODEC	24
4.3.2	MENCODER.....	26
4.3.3	HANDBRAKE	26
4.3.4	MEDIAINFO	27
4.4	UPLOAD TECHNOLOGIES	28
4.4.1	UPLOADS IN INVENIO	28
4.4.2	HTTP AND HTML BASED UPLOAD	29
4.4.3	FLASH AND OTHER PLUGIN BASED UPLOADS.....	30
4.5	STREAMING TECHNOLOGIES	31
4.5.1	PROGRESSIVE STREAMING	31

4.5.2 HTTP LIVE STREAMING.....	32
4.6 DATE EXCHANGE AND CONFIGURATION.....	34
4.6.1 INI vs. XML vs. JSON	34
4.6.2 MARCXML AND PBCORE FOR VIDEO METADATA.....	37
5 SYSTEM AND WORKFLOW ANALYSIS.....	41
5.1 SYSTEM STRUCTURE.....	41
5.2 USE CASES	44
6 DESIGN AND IMPLEMENTATION.....	46
6.1 OBJECT ORIENTED VS. PROCEDURAL	46
6.2 FUNCTIONAL STRUCTURE OF BIBENCODE.....	47
6.3 WRAPPING FFMPEG AND MEDIAINFO IN PYTHON	48
6.4 COMMAND LINE INTERFACE AND OPERATION MODES	50
6.4.1 ENCODING MODE.....	50
6.4.2 EXTRACTION MODE.....	50
6.4.3 METADATA MODE.....	51
6.4.4 DAEMON MODE.....	51
6.4.5 BATCH MODE	51
6.5 INTEGRATION AND INTERACTION WITH OTHER INVENIO MODULES.....	52
6.5.1 BASIC VIDEO SUBMISSION WORKFLOW	52
6.5.2 BIBSCHED AND BIBTASK INTEGRATION	53
6.5.3 BIBDOC INTEGRATION.....	55
6.5.4 WEBSUBMIT INTEGRATION.....	56
6.6 METADATA AND MARC INTEGRATION	64
6.7 JSON CONFIGURATION AND PROFILES.....	65
6.8 HANDLING OF VIDEO ASPECT RATIO	72
6.9 VIDEO PRESENTATION AND PLAYER	73
6.9.1 CONCEPTUAL AND VISUAL DESIGN OF THE VIDEO PLAYER	74
6.9.2 TECHNICAL ASPECTS OF THE VIDEO PRESENTATION.....	79
6.10 MULTI-NODE ENCODING WITH BIBENCODE	84
6.10.1 MODIFICATIONS TO BIBSCHED	85
6.11 QUALITY ASSURANCE, TESTS AND SECURITY	88
6.11.1 TESTING DIFFICULTIES	88
6.11.2 CODE QUALITY AND GUIDELINES.....	89
6.11.3 DOCUMENTATION	90

6.11.4 SECURITY	90
7 EVALUATION	92
7.1 TESTING THE WORKFLOW WITH BIBENCODE AND MULTINODE	92
7.1.1 NODE CONFIGURATION.....	92
7.1.2 STRESS TESTING	96
7.1.3 DETECTED ISSUES.....	97
7.2 FULFILMENT OF REQUIREMENTS	98
7.3 OPEN ISSUES	99
7.3.1 SUPPORT FOR HTTP LIVE STREAMING.....	99
7.3.2 IMPROVED VIDEO UPLOADS	100
7.3.3 SCALABILITY OF MULTINODE	101
8 CONCLUSIONS.....	103
9 BIBLIOGRAPHY.....	104
10 TABLE OF FIGURES.....	109
11 TABLE OF TABLES.....	113
12 APPENDICES	114
APPENDIX A - INVENIO CODE EXTENSION OVERVIEW.....	114
APPENDIX B - EARLY ANALYSIS AND DESIGN DOCUMENTS	115
APPENDIX C – USE CASE DESCRIPTIONS	118
APPENDIX D – FLOW CHART OF THE BATCH PROCESSING ENGINE.....	121

1 Introduction

1.1 Motivation

In the recent years, multimedia assets like images, audio podcasts and videos have become abundant in the world of digital libraries. While these libraries were designed to only preserve the bibliographical metadata of print documents in the beginning, they need to be able to handle a huge variety of different records today. This includes the processing of many different types of digital files and media. Video streams have become an import aspect of the Internet we know. Various video platforms such as YouTube or Vimeo offer millions of video streams and serve millions of customers each day. While the integration of video streams in digital library has already taken place, the market lacks open solutions that are portable and applicable for institutions of any size.

1.2 Environment

The *CERN Document Server (CDS)* is a large scale, digital and web driven library at *CERN*, the European Organization for Nuclear Research. CDS serves a wide variety of documents like scientific papers, preprints, articles, journals and experiment related documents. In the recent times, it is more and more used for multimedia content such as pictures, presentations, talks, posters, audio podcasts and videos. *CDS* is based on the open-source software *Invenio*, initiated and developed at CERN by the *IT-UDS-CDS* section. *Invenio* is used and supported by numerous external developers and institutions worldwide such as *Deutsches Elektronen Synchrotron (DESY)*, *École Polytechnique Fédérale de Lausanne (EPFL)*, *Fermi National Accelerator Laboratory (Fermilab)* and *Stanford Linear Accelerator Center (SLAC)*.

1.3 Assignment

The aim is to develop an extension to the *Invenio* Software that is capable of handling multimedia content in the form of digital videos, under the constraints and requirements of digital libraries, while being flexible and easily adaptable for the needs of different institutions running *Invenio*, but providing a basic, common an reusable workflow to submit, process and publish digital video content.

1.4 Thesis and Internship

This document is a combined bachelor thesis and internship project report of the authors six month long stay from March to August 2011 at CERN. The internship part focuses on the development of the modules for video encoding, frame extraction and batch processing and the wrappers for the *ffmpeg* and *Mediainfo* libraries. The thesis part focuses on the integration of the new modules into the existing *Invenio* infrastructure by providing a *WebSubmit* submission workflow, video record presentation and playback with *BibFormat* and a distributed video

processing concept based on modifications to BibSched. Both parts share a comprehensive system and technology analysis.

Common Chapters:

- 1 - Introduction
- 2 - Description of the existing System
- 3 - System Enhancement Requirements
- 4 - Technology Overview
- 5 - System and Workflow Analysis
- 6.11 - Quality Assurance, Tests and Security

Internship Chapters:

- 6.1 - Object Oriented vs. Procedural
- 6.2 - Functional Structure of BibEncode
- 6.3 - Wrapping FFmpeg and Mediainfo in Python
- 6.4 - Command Line Interface and Operation Modes
- 6.6 - Metadata and MARC
- 6.7 - JSON Configuration and Profiles
- 6.8 - Handling of Video Aspect Ratios

Thesis Chapters:

- 6.5 - Integration and Interaction with other Invenio Modules
- 6.9 - Video Presentation and Player
- 6.10 - Multinode Encoding with BibEncode
- 7 - Evaluation
- 8 - Open Issues

2 Description of the existing system

The following chapter grants an overview on how Invenio basically works as a digital library and the modules that are potentially involved in the processing of digital video are portrayed. Because a form of video processing and archival already exists at CERN in the form of the Media Archive, this environment is described and its interactions with CDS are analysed.

2.1 Invenio Overview

Invenio is a free and open software suite for running large-scale digital libraries and document repositories on the web. It covers all aspects of digital library management: from ingestion through submission or harvesting; to conversion, classification and indexing; to formatting and publishing. It is both fit for small libraries as well as large-scale libraries with millions of records. Invenio uses the *MARC21* metadata standard as its underlying bibliographic format. [1]

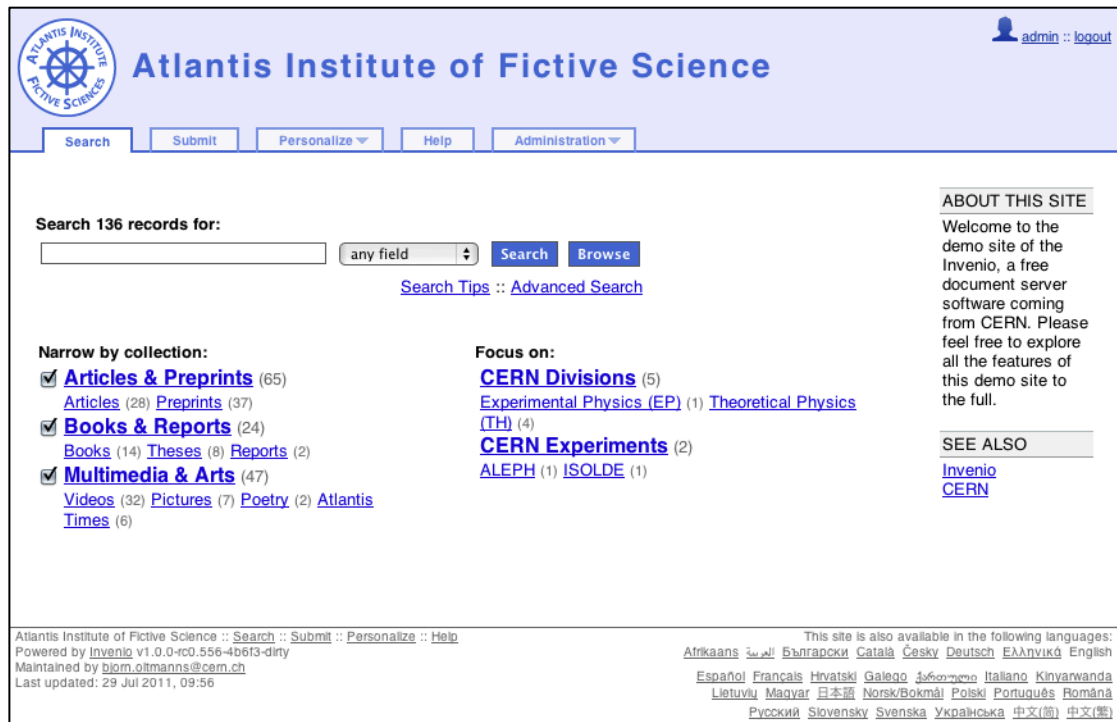


Figure 2-1 Screenshot of the web interface of a demo installation of the Invenio software.

Invenio is written in *Python*, a modern and versatile programming language often used for web development. Because *Python* as an interpreted language can cause performance issues, some of the core modules of *Invenio* are written in *Common Lisp*. Invenio currently requires an *Apache* webserver running on a *GNU/Linux* system. Other unixoid systems might work, but are not officially supported. For the communication between *Python* and *Apache*, *mod_wsgi* is used. Invenio uses a *MySQL* database as its data and metadata storage infrastructure. [2]

The software is subdivided into a number of modules, responsible for different aspect of digital library management. The modules interact with each other, the database and the surrounding layers, as well as the user and administrator through the web tier. While being structured through the module folders and individual code files, these modules do in most cases not operate on their own. A deeper look into the code reveals that Invenio is rather monolithic than modular.

The user accesses the digital library through a web interface where she can browse collections of records and search for specific information. Privileged user can submit new records through a submission interface. Administrators can create new types of records and their associated submission workflows. The user has various possibilities to customise her experience such as creating personal collections and setting up periodical reports on new documents. The administration interface allows the complete customisation of the appearance and behaviour of the library as well as the manual curating of records.

Administrators can utilize a series of command line tools to access several modules and functionalities directly.

2.1.1 MARC21 and MARCXML

The internal representation of records in *Invenio* is based on the *MARC21* standard. *MARC* stands for **MA**chine-**R**eadable-**C**ataloging. It is a standard format for the storage and exchange of bibliographic records and related information in a machine-readable form.

```

1  001__ 1350780
2  003__ SzGeCERN
3  020__ $$a9781430230908$$uprint version, paperback
4  041__ $$aeng
5  080__ $$a004.916.HTML
6  100__ $$aPfeiffer, Silvia
7  245__ $$aThe definitive guide to HTML 5 video
8  260__ $$aNew York, NY $$bApress$$c2010
9  300__ $$a321 p
10 6531_ $$9CERN$$aHTML
11 6531_ $$9CERN$$aweb design
12 6531_ $$9CERN$$aweb page design
13 65017 $$2SzGeCERN$$aComputing and Computers
14 690C_ $$aBOOK
15 916__ $$d201120$$sh$$w201119
16 964__ $$a0001
17 960__ $$a21
18 963__ $$aPUBLIC
19 980__ $$aBOOK

```

Figure 2-2: A record representation in MARC21. A record consists of fields (numbers in red colour) and subfields (alphas and numbers in green colour, preceded by '\$\$') with their corresponding content. Additionally, there can be two indicators per tag (alphas and numbers in blue colour). The mapping between the content and the fields is either part of the standard defined by the Library of Congress or an internal convention defined by the institution.

According to Pepe, A., Baron, T., Simko, T., Vesely, M., Gracco M., Le Meur, J-Y. and Robinson, N. [3] *MARC21* was chosen for several reasons:

- Well-established in the library world and used since the 1960s
- Easily transformable into a standardized XML representation: *MARCXML*
- Flexible enough to guarantee a long-term reliability
- Extendable and adaptable to any metadata structure, at least at CERN

Instead of storing the records in MARC21 notation, Invenio uses the MARCXML format that maps MARC21 to a XML structure. Converting from MARC21 to MARCXML and vice versa is fully information preserving because MARCXML does not add any new features to the MARC standard.

Records are stored in the bibliographical metadata database based on MySQL in three different structures. One DB table stores the record in serialised and ZIP compressed MARCXML as well as in a serialised, dictionary based Record Structure (RecStruct). RecStruct is just another representation of MARC that can easily be processed in Python. Both MARCXML and the Dictionary are stored as a BLOB. As a third structure there is for every MARC field in a given range, for example all fields beginning with '05', a separate table inside the metadata DB. The record is exploded into these tables to feed the search algorithm with fields that have not yet been indexed for fast searches.

```

1 <collection xmlns="http://www.loc.gov/MARC21/slim">
2   <record>
3     <controlfield tag="001">1350780</controlfield>
4   <.../>
5     <datafield tag="245" ind1=" " ind2=" ">
6       <subfield code="a">The definitive guide to HTML 5 video</subfield>
7     </datafield>
8   <.../>
9     <datafield tag="260" ind1=" " ind2=" ">
10      <subfield code="a">New York, NY</subfield>
11      <subfield code="b">Apress</subfield>
12      <subfield code="c">2010</subfield>
13    </datafield>
14    <datafield tag="653" ind1="1" ind2=" ">
15      <subfield code="9">CERN</subfield>
16      <subfield code="a">HTML</subfield>
17    </datafield>
18  <.../>
19    <datafield tag="650" ind1="1" ind2="7">
20      <subfield code="2">SzGeCERN</subfield>
21      <subfield code="a">Computing and Computers</subfield>
22    </datafield>
23    <datafield tag="100" ind1=" " ind2=" ">
24      <subfield code="a">Pfeiffer, Silvia</subfield>
25    </datafield>
26  <.../>
27 </record>
28 </collection>

```

Figure 2-3 The same record as in Figure 2-2, formatted in MARC XML. Some parts are omitted. The MARC tags are red, the indicators blue and the subfields are green

Although Invenio additionally supports the output of other metadata standards, such as Dublin Core, metadata will only be stored in MARCXML. All of the other metadata output formats are generated by a conversion from MARCXML.

2.1.2 Invenio Modules

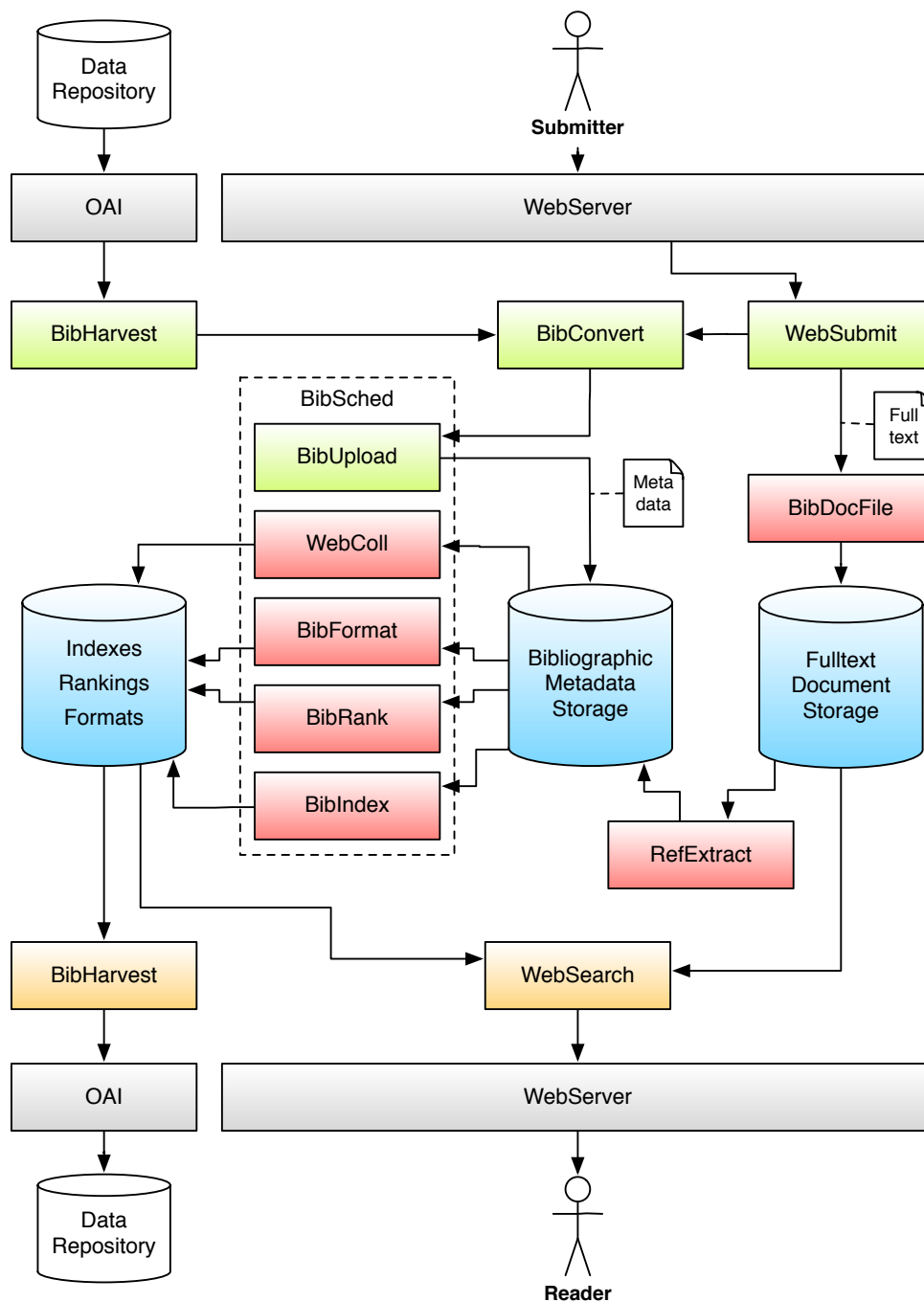


Figure 2-4 A Simplified Invenio module overview. Some modules are omitted as they are not relevant for multimedia related records. Green modules are responsible for the ingestion of metadata and for the creation of records. Red modules are for the processing of metadata and fulltext information. Orange modules are for accessing metadata and records. Data storage pools are shown in blue colour.

2.1.2.1 BibDocFile

Besides metadata representing a specific document, records might also contain the document itself or even different versions of the document. The MARC standard is only able to describe the media and reference the digital place of storage

for example through an URL. Coming from the cataloguing of physical media, MARC has no real concepts for handling digital media.

The BibDocFile module of Invenio handles the storage of digital assets such as fulltext files and images. Every record is associated to a number of BibDocs (Bibliographic Documents). A BibDoc encapsulates a digital asset in its various forms. A fulltext document can for example be available as a Microsoft Word, Open Office or Adobe PDF document. The content of these files is the same, but their format is different. The BibDoc is designed to house all available formats. The formats themselves are stored as BibDocFiles (Bibliographic Document File). Every BibDocFile must be different in Format to the other BibDocFiles residing in the same BibDoc. Every file has a Version, so that multiple versions of the same file and Format can be preserved if the file is revised.

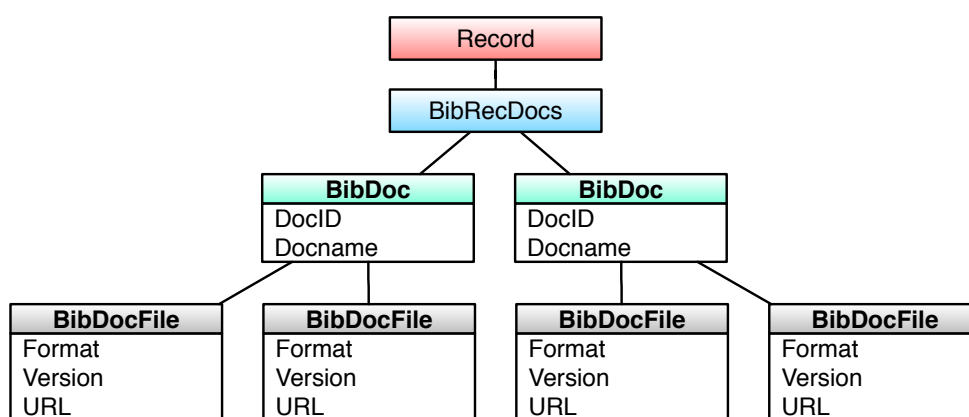


Figure 2-5 Basis structure of the BibDoc hierarchy.

BibDocFiles are stored on the file system. They rely on a specific folder structure where every BibDoc has its own folder with a name defined by the DocID. Inside the BibDoc folder, the BibDocFiles reside. The Docname of the BibDoc and its Format and Version defines the filename of a BibDocFile. The Docname is in most cases the name of the file that was uploaded to Invenio via the WebSubmit module. The Format is made up of the file extension and a customisable Subformat. The Version is a continuous Integer. The complete name of the BibDocFile is built as following:

docname.extension;subformat;version
 e.g. *example.pdf;printing;1*
 e.g. *example.doc;draft;1*

2.1.2.2 BibSched

BibSched stands for **B**ibliographic Task **S**cheduler. It is an infrastructure for running intensive tasks and services in the background, decoupled from the web-interface. Many modules of Invenio are BibSched enabled. The modules BibIndex, BibUpload and BibRank can for example be run as a BibSched Task.

Tasks might either be run as single execution tasks or as periodic tasks. They can be scheduled for immediate execution or for a specific time and date.

BibSched runs BibTasks. To run as a BibTask, a module must implement a specific API to communicate with BibSched, given by the BibTask module. The API contains functions for initiate tasks, check and validate task parameters, run and halt tasks. BibTasks can either be launched through their Command Line Interface (CLI) or through the *Low Level Submission API* of the BibTask module.

BibSched will neither guarantee that tasks are executed in a specific order nor that they are immediately executed or at the exact specified time. The set time will only function as the earliest possible execution time. Tasks might be postponed if the scheduler is running other tasks, which enjoy a higher priority. As BibUpload tasks create and alter records, these tasks enjoy for example the highest priority in BibSched

The task scheduler offers a semi graphical command line interface, as seen in figure 2-6.

ID	PROC [PRI]	USER	RUNTIME	SLEEP	STATUS	PROGRESS
4	webcoll	admin	2011-08-02 09:58:05	20m	CONTINUING	Part 1/2: done 281/831
723152	bibupLoad:FFT	bibdocfile	2011-08-02 09:58:37		WAITING	
1	bibindex	admin	2011-08-02 10:00:04	5m	WAITING	
2	bibreformat	admin	2011-08-02 10:00:44	5m	WAITING	
3	bibrank	admin	2011-08-02 12:51:04	4h	WAITING	
9	webcoll	admin	2011-08-02 20:00:00	5h	WAITING	Postponed 1 time(s)
6	oarepositoryupdater [-1]	admin	2011-08-02 21:44:08	1d	WAITING	
7	oaiharvest [-10]	admin	2011-08-02 21:46:04	24h	WAITING	
12	dbdump		2011-08-02 22:52:50	20h	WAITING	
13	inveniogc		2011-08-02 23:03:57	20h	WAITING	
11	bibindex:Council [-1]	admin	2011-08-02 23:27:05	2d	WAITING	
10	bibcircd [5]		2011-08-03 03:00:18	1d	WAITING	
8	atnappreminder	admin	2011-08-05 12:56:07	14d	WAITING	
5	bibrank	admin	2011-08-09 00:17:22	7d	WAITING	

Automatic Mode [A Manual] [1/2/3 Display] [P Purge] [L/L Log] [O Opts] [Q Quit] 2011-08-02 10

Figure 2-6 A picture of the BibSched (Bibliographic Task Scheduler) in the automatic scheduling mode with multiple waiting tasks and a running task.

2.1.2.3 WebSubmit

The WebSubmit module is responsible for the ingestion of user submitted records. Submission Workflows are created with WebSubmit to cover the creation of different types of records like journal entries or images. The workflows created with WebSubmit basically present a form-based interface to the user where she can insert metadata and upload fulltext or media files.

2.1.2.4 BibConvert and BibUpload

Form based metadata coming from a WebSubmit Workflow is fed to the BibConvert module to generate a MARCXML representation. The MARCXML is then fed to BibUpload, which inserts the record and its metadata into the different database tables.

2.1.2.5 BibFormat

The BibFormat module is responsible for the creation of record representations in HTML or different data exchange formats. Templates can be defined that determine how MARC fields are to be rendered as HTML code. By using BibFormat, the administrator can create individual representations for each type of record in the system or share the same representation among multiple types.

2.2 The Media Archive Workflow

Since its beginnings in 1954, CERN has build up an archive of thousands of audio and video recordings from conferences, talks and professional productions. These recordings were not available for the general public until the Media Archive was established in 2006. The aims of the Media Archive were the digitalisation of physical recordings and their storage on CERN server infrastructure. The system should in addition store new digital productions created by the Audio-Visual section. The project aimed to utilise existing CERN infrastructure and minimum manpower to maintain low costs.

Audio, photo and video archives are now stored on a Distributed File System (DFS). The original, high-quality and large-size files coming from digitalisation or professional production equipment are called Master. They are the most important files to preserve. The Master files are not to be consumed by users, but needed for conservation and future reprocessing and productions.

Slaves are created from the Master files. Slaves are lower quality versions meant to be consumed by users over the Internet. For every Master, various Slaves in different formats, qualities and sizes exist to fit the consumer needs. The transcoding from Master to Slave is done by Worker Nodes. These are servers running a set of conversion software, mainly video and audio encoders to transcode from one format to another.

For the preservation of the materials metadata and for making the videos, photos and audio available to the users, CDS is utilized.

It was decided by the Media Archive to store the video Masters in two different formats: A DVCPRO HD version in a QuickTime MOV container that delivers the best quality for preservation but is difficult to process in a Windows environment according to Kwiatek [4] and lower quality but said to be easier to process MPEG2 version. Both versions are created by the AV section staff in professional production software, which means that they are doing at least one additional video conversion on their local machines, instead of using the described infrastructure.

When a user submits a new video record to CDS, the submission does not include any video upload. It only contains the metadata to create the basic record with fields like title, description and additional metadata for the encoding server. The additional metadata includes the width and height of the video in pixels, the aspect ratio as a fraction and the framerate in frames per seconds.

The user then needs to copy the Master to an exact path created during the submission and communicated to the user through CDS. The path follows a strict naming schema, in which the videos reside in specific folder and the name follows a specific pattern:

`\\cern.ch\dfs\Services\MediaArchive\Video\Masters\Movies\YEAR\CERN-MOVIE-YEAR-NUMBER.mpg`

After the user uploaded the file to DFS, a script on the Media Archive server automatically contacts CDS through an exclusive API to receive the metadata inserted in the first step. If the metadata can not be received, e.g. due to not yet finished processing on CDS, the Media Archive server tries again periodically until a timeout is reached and the conversion will go into an error state.

If the metadata is successfully received, the Media Archive server initiates the creation of the Slaves based on predefined profiles.

When the Slaves were created successfully, the Media Archive server communicates the URL of the newly created files to CDS through the same API used for receiving the metadata.

The video is made available for streaming on CDS through an embedded Flash or Windows Media Player. An Adobe Flash Streaming Server and a Windows Media Services server are running on top of the Media Archive. They use real streaming protocols to serve the videos from DFS. The video files are also accessible through the HTTP protocol for progressive streaming (downloading and playing simultaneously).

When the user revises the Master by uploading a new version under the same name to DFS, the server side scripts detect this circumstance and reinitiate the creation of Slaves.

Slaves currently consist of 4 different transcoded video formats and 3 different sets of thumbnail images extracted from the video:

- A low quality WMV version with 512x288 pixels and a bitrate of 480 kbit/s for Windows Media Streams.
- A medium quality WMV version with 640x360 pixels and a bitrate of 753 kbit/s for Windows Media Streams.
- A FLV version encoded with the Sorenson Spark codec with the same resolution and bitrate as the medium WMV version for Flash Video Streams.
- A MP4 file encoded with H.264 at 640x360 pixels and a bitrate of 600 kbit/s.
- 10 JPEG thumbnails, one every 10% of the video duration with at size of 640x360 pixels.
- 10 JPEG thumbnails, one every 10% with a size of 80x45 pixels.
- 1 JPEG thumbnail at 5% of the video duration with a size of 180x101 pixels.

The Slaves are generated in the named formats regardless of Master video quality, bitrate or resolution.

A big issue is the lack of transparency of the conversion process for both the administrator managing the system and the regular staff uploading material to it. The conversion queue that executes the transcoding of Slaves is opaque. For every Master, there exists only a XML file that stores the latest status of the process per Slave: INSYNC for finished conversions, BUSY for running conversion, ERROR for failed conversion and NEW in case the Master was changed and the processing needs to be reinitiated.

The time to encode a video cannot easily be predicted. It depends on various factors like the used codecs and encoders, the desired quality and especially the load and general performance of the servers running the encoding. It is important for all users of the system to be able to access at least an estimate or a status of the process. The conversion might take a long time, but if no hints are given, should people worry after several hours or after a day if their videos are not appearing on CDS?

In general, the Media Archive solution relies heavily on proprietary software by Microsoft:

- Servers for the Media Archive workflow are running Microsoft Windows Server 2003 or later.
- The programs on the server-side to initiate and control the encoding of Slaves are written in Microsoft Visual Basic.
- The Distributed File System is a Microsoft product, delivered with Server 2003 and later.
- To create a Windows Media Video (WMV) version of the Master, the Windows Media Encoder 9 is used. It is freeware but proprietary.
- Windows Media Services is used for streaming WMV to clients.

2.3 Why re-conception?

We have seen in that the Media Archive and the CERN Document Server are two completely individual entities. They interface with each other to publish the archived video material on CDS, because the Document Server is the central place for any kind of document and media published at CERN. There is no integration for the handling of videos in CDS and the underlying Invenio software at all.

The Media Archive has been specifically developed for CERN and relies on proprietary software by Microsoft. Even if all the code related to Media Archive would be published as open-source software, the proprietary infrastructure would block its adoption and is completely contrary to the philosophy behind Invenio.

On the other hand, there are many inconveniences for the users due to the nature of the Media Archive and its loose coupling with CDS. Unnecessary steps are taken when the videos produced by the AV section are first converted into a format that the Media Archive can handle. Users submitting a new video must first go to CDS and create a new record through a WebSubmit workflow. Instead of simultaneously uploading their video through the same interface, they receive a specif-

ic path on a file server where they should upload the video. After the upload, the video encoding is completely opaque for the user. Uploading the file does not yield any feedback. If the encoding takes up a larger timespan, the user might wonder if the process is still on going or if it might have failed.

Even at CERN, the Media Archive is only accessible for trained personnel of the AV section. Other sections or the experiments are not able to add video records to CDS on their own. There have been several cases where the CDS was contacted regarding video submissions. The customers were redirected to the AV section to handle the request. With CDS as the prototype for other Invenio installations, there have also been requests by other Institutions on video handling in Invenio, as videos are obviously available on CDS. These requests were answered with an unsatisfactory reference to the non-distributable Media Archive solution.

The project described in this Thesis is not about replacing the media archive, but to explore deep media integration in Invenio and CDS, independent of the Media Archive. The goal is to allow non-AV section users at CERN the submission of videos to CDS and to enable video support in Invenio based libraries of external institutions.

3 System Enhancement Requirements

The requirements for the project were initially based on the capabilities of the Media Archive and the ability to reproduce its results, while removing its inconveniences with a solution completely integrated into the Invenio software:

- Upload of a video file through a web interface.
- Adding metadata information to the video to create a record.
- Preserving and storing the original (*Master*) video for archiving and later use.
- Extracting metadata information from the video file/container. For example the duration, size etc.
- Appending the extracted metadata to the record for search features and later use.
- Transforming the original video into other video formats. (Creation of *Slaves*)
- Using open-source and licence free encoders and libraries for the transformation.
- Storing the *Slaves* for later streaming.
- Extracting thumbnails (still images) from the video to create previews.
- Using open-source and licence free libraries to extract the thumbnails.
- Storing the thumbnails.
- Generating a video representation by combining the user inserted and the extracted metadata together with the *Slaves* as sources for video streams.
- Streaming the video to the user.
- Classic download of videos.

These requirements were never strictly specified from the beginning of the project, but were put together continuously by the analysis of the Media Archive solution as well as numerous discussions and incremental analyses during the first month of the project.

During the analysis and implementation phase, other potential requirements of the system have arisen:

- Systematic scheduling of video transformations.
- Decoupling of web-server and video encoding, e.g. running the video encoding on a different machine than the webserver.
- YouTube like video experience: A video player suitable for any web browser; Video recommendations; Commenting system.

To fulfil these requirements, an extensive analysis of potential technologies is presented in the following chapter.

4 Technology Overview

4.1 Videos on the Web

When we think about how video was presented on the web before the year 2002, we will come to Windows Media and Real Media for embedded videos, playable in the browser, but only if the user had installed the right proprietary video player on her Windows PC. But then came Flash and changed the way videos were presented on the web. A combination of well appreciated authoring tools and a fast growing market penetration on the client side made Flash and especially Flash Video a huge success in the online media landscape.

4.1.1 Introduction to Flash Video

Adobe Flash is ubiquitous today on the Web: it is used for animations, rich Internet applications, games, audio and video streams. The Flash Player is available as a browser plugin for all common browsers like Internet Explorer, Mozilla Firefox, Opera and Safari on Windows, Mac OS X and Linux. Google even integrated Flash directly into its Chrome browser, removing the need of an external plugin.

Macromedia, a company later bought by Adobe, released the first version of the Flash Player that supported videos in the year 2002. This 6th version included the Sorenson Spark H.263 video codec, as well as their own proprietary live streaming protocol RTMP [5]. Flash was already supporting MP3 audio since version 3 in 1998, now used as the audio stream for the videos images. One year later in version 7, support for HTTP progressive streaming was added. Flash Player 8 added support for the VP6 video codec by On2. In 2007, support for H.264 video and AAC audio was added in version 9.

Flash supports video playback in full screen mode, taking up the whole screen estate and removing all operation system related UI elements.

While Flash and all related technologies were created as proprietary formats, Adobe lifted all licensing restrictions in 2008 through the Open Screen Project and made the RTMP, SWF and FLV specification available for free use [6].

But even with a market penetration of 99% in the year 2011, according to Adobe [7], the problem of the proprietary player remains, as no open-source Flash Players has a significant market penetration.

4.1.2 Introduction to HTML5 video

With the development of the new HTML5 standard, the W3C wanted to specify and establish open standards for videos on the web. The HTML5 video tag specifies a way of embedding video files into a website and establishes an API for manipulating the appearance and behaviour of the embedded player and video. The HTML5 working group considered defining at least one standard video and audio codec that all browsers must support to be HTML5 compliant. In an early draft,

the OGG Theora video codec and the OGG Vorbis audio codec were considered as such in the HTML5 standard. But the recommendation for OGG was later removed due to protest of companies including Nokia, Apple and Microsoft. For them, OGG was neither without patent and licensing issues, nor delivering superior video and audio quality [8]. The passage in the W3C draft was altered to be in no way specific and to only make a general statement about the nature of a codec for an open web:

“It would be helpful for interoperability if all browsers could support the same codecs. However, there are no known codecs that satisfy all the current players: we need a codec that is known to not require per-unit or per-distributor licensing, that is compatible with the open source development model, that is of sufficient quality as to be usable, and that is not an additional submarine patent risk for large companies. This is an on-going issue and this section will be updated once more information is available.” [9]

This passage was later completely removed because no consent was reached and the issues have not been resolved.

With no codec specified as the standard for HTML5 video, it was up to the browser developers to support whatever codec they preferred. Microsoft and Apple, being against the first W3C draft, decided to only support H.264 in Internet Explorer and Safari. While Mozilla and Opera, as supporter of the first W3C draft, decided to go for Theora.

Google on the other hand was supporting both H.264 and Theora from early Chrome versions. Regarding Theora, Google was also fearing “submarine patents” and was not fully satisfied with its quality. H.264 was neither issue free for Google, because licence fees were to be paid for decoders built into Chrome, which was not suitable for its open-source version Chromium [10]. Google later bought On2, the developers of the VP3 codec (on which Theora is based) and the more modern VP8 video codec. They used the MKV container format for VP8 and combined it with OGG Vorbis to create the WebM format. Mozilla Firefox, Google Chrome and Opera currently support WebM.

Browser	Cur. Ver.	Theora	WebM	H.264
Internet Explorer	9.0	Plugin	Plugin	Yes, 9.0
Mozilla Firefox	5.0	Yes, 3.5	Yes, 4.0	No
Google Chrome	12.0	Yes, 3.0	Yes, 6.0	Yes, 3.0 (to be removed [11])
Safari	5.0	Plugin	Plugin	Yes, 3.1
Opera	11.11	Yes, 10.50	Yes, 10.6	No

Table 4-1: Browser support for video codecs Theora, H.264 and WebM in their latest versions (20/06/2011)

The HTML5 video tag offers a rich JavaScript API to create ones own controls and to manipulate the video.

A drawback is, that HTML5 video can currently not be played in full-screen mode in every browser, like Flash video can. There is no API for such a feature defined in the standard, but browser vendors may implement it on their own:

“User agents may allow users to view the video content in manners more suitable to the user (e.g. full-screen or in an independent resizable window).” [12]

The WebKit engine on which Apple Safari and Google Chrome are built, offers a proprietary JavaScript function [13] to enlarge any element to full-screen. By using standard JavaScript techniques to control the position, layer and size of an element, it is at least possible to enlarge the video up to the size of the HTML content area of the browser.

While the Flash Player supports proprietary RTMP streaming besides playback during progressive download (progressive streaming), the HTML5 video tag is in principle protocol agnostic. This means that any protocol the browser supports could be used within the video tag. Unfortunately, only Apple Safari supports any advanced streaming protocol [14].

4.1.3 Video formats and codecs

4.1.3.1 OGG Theora

Theora is an open-source and royalty free video codec developed by the Xiph.Org Foundation. It is based in the VP3 codec of On2. On2 has granted an irrevocable, royalty free licence over its VP3 patents. The source code is published under a BSD style licence that allows the use in proprietary software. It is considered to be a multi purpose codec that delivers good visual quality for low bandwidths for both small and large (e.g. HD) video sizes. Theora is a block-based, lossy codec similar to MPEG4 and others. It assumes a psychovisually-aware encoder. The codec is designed to be used in the OGG container with an .ogv extension and is therefore in general accompanied by OGG Vorbis encoded audio, but other codecs are possible according to the specification [15]. [16]

Theora is natively supported by Mozilla Firefox [17], Google Chrome and Chromium [10] as well as in Opera [18]. Microsoft Internet Explorer and Apple Safari do not offer native support for Theora, but are able to play it through plugins developed by the Xiph.Org Foundation [19] [20].

The OGG container offers support for metadata. The container can store so called *VorbisComments* a key value pair like “ARTIST=Elvis”, “TITLE=Blue Suede Shoes” that is associated to a stream within the container. *VorbisComments* were initially meant for simple comments on OGG Vorbis audio only, but have later been extended to Theora and other codecs in OGG. The keys are not standardised. Only recommendations exist. Several drafts have been published for the standardisation of metadata within the OGG container, using more sophisticated means of storing the information, but these have yet to be consolidated and implemented. [21]

4.1.3.2 Google WebM

WebM is an open-source and royalty free codec like Theora, developed by Google after its acquisition of On2 Technologies. Instead of VP3 as in Theora, WebM is based on the more modern VP8 codec.

Just like Theora, VP8 is a lossy, block based video codec. Very simplified speaking, the codec compresses video material by removing unchanging image areas within a series of images and later reconstructing based on complete keyframes and the changing parts. It uses algorithms to specify parts of the image (blocks) by referring to other blocks of previous images. These blocks can even move spatially. [22]

Google created a simplified version of the Matroska (MKV) format for the container of WebM video.

Contradictory to Google's strong efforts in metadata and data mining, the stripped down version does not include MKV Tags. These Tags are nestable elements for systematic metadata storage in MKV. [23]

The specification recommends to only use VP8 video and OGG Vorbis audio within the WebM container, instead of being totally open for numerous codecs that are supported by MKV. This should not be seen as a mindless constraint, but rather as a real simplification in handling video files on the web. With the WebM container used, the codecs required to play the video are defined by the file extension (.webm) if the creator followed the recommendation. Without the need of additional mime-types given, or downloading the video header, the client can be quite sure about the playability the video. With containers like MP4 that openly support numerous codecs, the playability is uncertain. VP8 and Vorbis were probably not set in stone and described as "should" for the introduction of future codecs to WebM. [24]

Table 7-1 shows that WebM is currently supported by Google Chrome and Chromium since version 6.0, Mozilla Firefox [25] since version 4.0 and the Opera browser [26] since version 10.6. Both Internet Explorer [27] and Safari [28] play WebM video after installing third party plugins, similar to OGG Theora.

4.1.3.3 H.264/MPEG-4 AVC

While OGG Theora can be seen as the open-source codec of choice and WebM as the codec for an open-web, the H.264 codec was not developed for a single purpose, but rather as a completely universal codec collection for various use-cases and usage scenarios. It is used for Blue-Ray discs, DVB broadcasting, within Adobe Flash and Microsoft Silverlight on the Web, videoconferencing as well as mobile applications. The H.264 standard [29] defines a set of "profiles" that relate to usage scenarios, ranging from low-quality for mobile and videoconferencing to standard-quality for digital television and internet broadcasting to high-quality profiles for Blue-Ray discs and high-definition broadcast to ultra high and near lossless profiles for professional applications. Every profile relates to a set of algorithms that are used to achieve the projected quality, bitrate and performance.

Many technologies H.264 uses underlie software patents. The company MPEG LA, responsible for the patent pool surrounding H.264, has stated in 2010 [30] that all H.264 videos free to the end-user on the Internet are free of royalties. The publication explicitly states, that all other products and services continue to be royalty bearing. This includes all encoding technologies involved in the creation of these videos, which means that one could be charged in a country supporting software patterns for using and/or distributing these tools and technologies.

It is therefore not recommended to include H.264 directly in distributed code (e.g. in Invenio), even if available as open-source, but rather creating an interface to easily communicate with external codec libraries. It is then up to the institution or user to enable H.264 encoding through the installation of these libraries.

H.264 has replaced the older Sorenson H.263 codec as the codec of choice for Flash Video. The Flash Player now supports H.264 video embedded into a MP4 container together with MP3 or AAC audio. It is thereby possible to supply both an embedded flash player and a HTML5 video player in H.264 enabled browser with only one file.

Apple Safari currently supports H.264 for HTML5 video since version 3.0, Microsoft Internet Explorer since version 9.0. Google added H.264 support in early versions of Chrome, but stated that it will drop support at some point in the future in favour of the WebM codec. Because of the automatic updating scheme Google applies to Chrome installations, Chrome should no longer be counted on for H.264 support, even if it is still there. For both Safari and Internet Explorer a MP4 container must be used for H.264 in combination with either AAC or MP3 audio.

The MP4 container offers metadata capabilities as far as this could be determined from libraries and tools being able to handle MP4 metadata such as *iTunes* and *AtomicParsley*. Unfortunately, the MP4 specification is not freely available to the author and the topic cannot be further investigated.

4.1.3.4 Comparison of H.264, WebM and Theora

The video quality of Theora was criticised by Microsoft and Apple not being on par with other modern video codecs such as H.264.

Various tests and comparisons showed that the Theora codec is far superior compared to the old Sorenson Spark H.263 codec of Flash at the same bitrate for small video sizes (420x270). Maxwell [31] and Loli-Query [32] have shown that the differences become minor for larger resolutions.

WebM is regarded as being slightly inferior to H.264 in quality in pure subjective tests. Comparisons done with YouTube like Settings by QuavLive [33] show, that the image quality of VP8 is significantly lower at low bitrates and low resolutions compared to H.264. When comparing 480x270 pixel video at 256 kb/s, VP8 shows a tremendous amount of artefacts and blur, both in uniform areas and detailed areas. H.264 shows the same amount of blur on uniform areas, but is a lot sharper in the detailed areas. With rising resolutions and bitrates, the differences

become less visible. At 1920x1080 and 3600 kb/s both codecs offer a mediocre quality, due to the low bitrate. The H.264 image still looks sharper on still images, but the differences are more difficult to see in moving images.

Vatolin D., Kulikov D., Parshin A., Arsaev M. and Voronov A. have shown in their detailed report [34], that objective measurements using Signal To Noise Ratios (STNR) support the results of the subjective tests. Vatolin et al. compared several H.264 encoders with the WebM and Xvid encoders. It has to be said, that the differences between all the encoders become smaller with rising bitrates, not only between WebM and H.264 in general. On the average, the x264 encoder scored best of all the H.264 encoders. Regarding encoding speed, the WebM encoder performs between 30 and 50 frames/s slower than the x264 encoder on a “normal” preset. The difference was only between 5 and 10 frames/s for a “high quality” preset. The tests were done on a latest generation quad-core Intel Sandy Bridge machine, which supports hardware based H.264 encoding.

Unfortunately, there are no studies available comparing all the three video codecs in one study under the exact same conditions. However, during the course of the project described in this thesis, various videos were encoded using the latest versions available of all the three codecs, for testing and evaluating the video integration in Invenio, but not the video codecs. All encoders were using the same settings for bitrates, frame dimensions and frames per second on the same original video file. No objective analysis was conducted on the video material like evaluating the STNR, but the subjective impression complies with the general trend in the studies described in this chapter. The subjective impression of the author is, that the Theora codec delivers the worst quality at low bitrates (854x480 pixels at 1000 kbit/s) as well as at medium bitrates (1280x720 pixels at 2500 kbit/s). The WebM codec delivers a substantially better quality than Theora in both bandwidth scenarios. The best quality is without a doubt achieved with the H.264 codec, both for low bitrates and small frame sizes as well as high bitrates and HD frame sizes. The author needs to mention that none of the codecs is tweaked in any way regarding optimal codec specific parameters. The H.264 encoder is used with the ‘baseline’ preset.

4.1.4 HTML5 video market penetration

It is imperative to analyse the penetration of HTML5 video supporting browsers in the target market before deploying HTML5 video. It is senseless to deploy videos encoded with certain codecs that might deliver better quality than others, but there is no support in the market for these formats.

At CERN, the target audience mostly consists of scientific institutions and users in the high-energy physics field. As the project does not aim to provide a CERN specific solution for videos in Invenio, but a solution for other institutions using the software, the focus should not be too narrow on HEP, because only some of these institutions operate in this field. Therefore both CERN specific statistics and global statistics from StatCounter are analysed regarding HTML5 video support. StatCounter generates statistics from tracking codes deployed on several million pages worldwide.

The free log analyser AWSTATS was used to generate browser statistics from Apache log data of the CDS production server (<http://cdsweb.cern.ch>) collected in May 2011. One has to keep in mind that different collections might target different audiences (for example scientific paper vs. video collections). The browser shares might therefore differ from collection to collection. Unfortunately, with the current URL structure in CDS and the limitations of Apache logs and AWSTATS, there is no way to gain browser statistics on video collections only or specifically on video records.

Browser	Share	Hits
Firefox	38,13%	562666
IE	31,60%	466427
Chrome	11,49%	169575
Safari	9,10%	134296
Opera	1,72%	25448
Other	7,96%	117422
Total	100,00%	1475834

IE	Share	Hits
v9.0	1,61%	23695
v8.0	14,19%	209401
v7.0	11,01%	162443
v6.0	4,24%	62550
v5.5	0,49%	7248
Other	0,07%	1090
Total	31,60%	466427

Chrome	Share	Hits
v13.0	0,05%	736
v12.0	0,21%	3129
v11.0	6,45%	95135
v10.0	3,66%	54081
v9.0	0,63%	9367
v8.0	0,15%	2169
v7.0	0,09%	1343
v6.0	0,10%	1442
v5.0	0,09%	1295
v4.0	0,04%	661
v3.0	0,01%	159
Other	0,00%	58
Total	11,49%	169575

Firefox	Share	Hits
v4.0	12,58%	185730
v3.6	20,29%	299511
v3.5	1,53%	22520
v3.0	2,63%	38756
Other	1,09%	16149
Total	38,13%	562666

Safari	Share	Hits
v6.0	0,05%	696
v5.0	7,10%	108430
v4.0	0,60%	10271
v3.0	0,70%	12287
Other	0,70%	2612
Total	9,10%	134296

Opera	Share	Hits
v9.8	1,66%	24562
Other	0,06%	886
Total	1,72%	25448

Table 4-2 Browser statistics by browser family based on Apache logs from CDS of May 2011, analysed with AWSTATS. The particular tables for one browser family show percentages based on the total hits in the top left table.

AWSTATS outputs a tabular HTML overview, which shows the number of hits and a calculated percentage by exact browser version (e.g. Firefox 3.6.1) in the given period. The raw hit-count was then used for evaluation.

All subversions of one browser version are summarised in a spreadsheet. For example Firefox version 3.6.1, 3.6.2, 3.6.3 and so on are summarized under version 3.6. All summarised versions of one family were then summarised to a total value for the whole browser family. The percentages were recalculated due to inaccuracies in AWSTATS.

Table 7-2 shows the statistics for all pages of CDS. Firefox shows the highest usage rate with 38,13%, followed by Internet Explorer with 31,60%. Chrome and Safari are positioned around 10% with 11,49% for Chrome and 9,10% for Safari.

Within the 117442 hits of “other” browsers, 68733 or 4.6% are by unknown browsers. These unknowns are most often crawlers, bots or other non-human clients. The unknown browsers are therefore ignored, and removed from the total number of hits for further analysis. 16906 hits or 1.1% are caused by the old non-Firefox Mozilla browser, which does not support HTML5. There are 6011 hits or 0.4% by Safari on iPhone, supporting HTML5 video with H.264, 5096 hits or 0.3% were Android browsers, which might support HTML5 video with WebM or H.264 as well as Flash video. Due to various Android versions in use, it is very difficult to predict which codecs are really available [35]. The rest of “others” is seen as HTML5 incompatible in the further analysis; assuming that they do either not support HTML5 video at all or having only minor statistically impact on the final results.

As shown in Table 4-1, different browsers support different codecs for HTML5 video. For each browser, the lowest version number that supports a given codec is indicated in Table 4-1. Based on the data of Table 4-2, for each browser the sum of hits of all versions supporting a specific codec is calculated. The percentage is calculated against the total number of hits in May minus “unknown browsers” as explained above. Safari includes hits from the mobile version.

Browser	Theora	WebM	H.264	Any
Firefox	36,09%	13,20%	0,00%	36,09%
Chrome	12,05%	11,90%	(12,05%)	12,05%
Safari	0,00%	0,00%	9,79%	9,36%
IE	0,00%	0,00%	1,68%	1,68%
Opera	0,00%	0,00%	0,00%	0,00%
Total	48,13%	25,10%	11,47%	59,18%

Table 4-3 HTML5 video compatibility based on the CDS statistics of May 2011. The table shows the overall compatibility including all the current HTML5 video codecs: H.264, Theora and WebM. Google Chrome is not counted towards the total percentage of H.264 because the support is going to be cut in future versions. Unknown browsers have been removed from the total number of hits to calculate the percentages.

The general adoption of HTML5 video compatible browsers on CDS almost reaches 60%; this includes all of the current HTML5 video codecs: H.264, Theora and WebM. For H.264, there is an adoption of only 11,47%. Chrome is not counted because the support is going to be dropped in the near future and the automatic updating policy of Google keeps most of the installations updated. Table 4-2 shows that there are a great number of users still using Mozilla Firefox 3.6, which does not support WebM. This reduces the number of total users supporting WebM video to 25,10%, as shown in Table 4-3. But versions 3.6 and 3.5 of Firefox support Theora, thereby raising the number of users with Theora support above those of WebM to 48,13%.

For the global statistics, browser shares by browser version were downloaded from the StatCounter [36] website for May 2011. The statistics come as comma-separated values: “Browser Version, Percentage”. The data is processed in the same way the AWSTATS data is, but summing up the given percentages of the browser versions directly, because no hit counts are given. The Safari statistics include mobile versions. Chrome includes Chromium installations.

Browser	Theora	WebM	H.264	Any
Firefox	28,18%	14,23%	0,0%	28,18%
Chrome	19,35	19,03	(19,35%)	19,35%
Safari	0,0%	0,0%	4,98%	4,98%
IE	0,0%	0,0%	4,57%	4,57%
Opera	1,67%	1,63%	0,0%	1,67%
Total	49,20%	34,89%	9,55%	58,75%

Table 4-4 HTML5 video compatibility based on StatCounter statistics of May 2011. The table shows the overall compatibility including all the current HTML5 video codecs H.264, Theora and WebM. Google Chrome is not counted for the total percentage of H.264 because the support is going to be cut in future versions. Safari includes all mobile versions with support for H.264.

With 58,76%, the global adoption of HTML5 video supporting browsers is almost the same as on CDS with 59,18%. The same is true for Theora with 49,20% globally and 48,13% on CDS and H.264 with 9,55% globally and 11,47% at CERN. Only the numbers for WebM differ greatly, with a global adoption of 34,89% and a local adoption of 25,10%. This difference is due to the fact, that the Firefox adoption at CERN is around 10% greater than on a global scale. The situation is vice versa for Chrome.

4.1.5 Conclusions on video formats

We have seen that both WebM and H.264 deliver superior quality compared to Theora. While both WebM and Theora are supported by almost half of the browser market through Opera, Chrome and Firefox at the same time, neither Internet Explorer nor Safari support any of them. If one would only look at the general browser adoption, there would be little to no reason to deliver Theora video to the users, as any of the latest browser supporting Theora also supports WebM. But as shown in chapter 7.1.3, there is a great amount of Firefox versions still in use on CDS, which do not support WebM. The usage of old versions is often not an active decision of the user, but rather tied to institution policies preventing updates on workstations. The presented numbers are therefore not expected to decrease in the near future.

We have seen that H.264 delivers superior image quality compared to WebM, but bears strong licensing and patenting issues. For HTML5 video, it is only supported by Internet Explorer 9.0 and Apple Safari, with market shares around 5% altogether in the worldwide market, and even above 10% on CDS. But for Flash Video, H.264 is a very interesting solution. Through the Flash Player, H.264 can reach more than 90% of the market [7].

In principal, developers should not rely on third party plugins being installed in the users browser. One big exception might be Flash, which is almost ubiquitous. Plugins enabling non-native video codecs in browsers are therefore not treated in any video codec consideration.

As we aim for an open web, the primary support should go to HTML5 video, mainly through WebM. H.264 should serve HTML5 video in proprietary browsers as well as delivering a fallback mechanism with Flash for older browser versions that do not support HTML5 video and WebM. JavaScript could load an embedded Flash player if it fails to detect a HTML5 and WebM compatible browser. The Flash Player would be configured to use the H.264 file to play the video.

4.2 Video quality, resolutions and bitrates

For digital broadcasting and video streaming, many different resolutions exist and they differ largely by the broadcasting standards in use.

'*Standard Definition TV*' has a resolution of 640x480 pixels in the NTSC standard and a resolution of '768 × 576' pixels in PAL. Many video platforms on the web are based in the United States and have therefore adopted NTSC like resolutions for their video streams.

In the recent times, broadcasters and Internet streaming company improve their service towards *High Definition* (HD) video. Digital TV channels offer HD video today as well as online video portals like YouTube. Unfortunately, what HD really means is not defined by any international standard. The general idea of HD is to deliver high resolution, detail rich and sharp images to the clients. High Definition is de facto referred to high frame resolutions at or above 1280x720 pixels in the broadcasting industry. This resolution is often referred to as 'half full HD', while 'full HD' is referred to a resolution of 1920x1080 pixels.

A higher resolution of a video does not necessary improve the perceived image quality of the video. The quality is basically a function of the video codec used and the selected frame size and bitrate. A Blue-Ray disc for example offers a video bitrate of 40 Mbit/s at a resolution of 1920x1080 pixels and is often encoded with H.264. Is generally considered as the highest consumer video quality currently available. On the web, these bitrates would be impossible to stream for most of the Internet users. In Germany for example, the average advertised broadband bandwidth is 17 Mbit/s according to an OECD study conducted in September 2010 [37]. Even if the available user bandwidth would be completely used, this would create enormous requirements for the storage and streaming infrastructure of online video portals. The real bitrates used by YouTube for example are not officially available, but McFarland, P. has calculated approximate bitrates for H.264 encoded videos on YouTube [38].

The current standard resolutions for video streams on the CERN Document Server, transcoded and streamed by the Media Archive is 640x480 pixels at a bitrate of 600 kbit/s encoded with H.264. Many of the videos available in the Media Archive lack sharpness and contain visible artefacts due to their low bitrate. It is the

absolute bottom line that should be made available today, mostly for streaming to mobile devices with limited screen estate and available network bandwidth.

Resolution (pixel)	Video Bitrate (kbit/s)	Audio Bitrate (kbit/s)
1920x1080	3500	128
1280x720	2000	128
854x480	1000	128
640x360	500	96

Table 4-5 Average bitrates for different video resolutions of H.264 video with AAC audio on YouTube calculated by McFarland, P. [38].

No clear recommendation for the quality of videos on the web can be given. Future technologies will for sure allow better qualities, both at low bitrates with new codecs as well as through increasing broadband bandwidths and extended capacities on the server side. The quality made available will therefore depend on the technical constraints of the Institution offering video streams and the target audience. The numbers shown in Table 4-5 can be seen as the bare minimum bitrates for the different video resolutions with the best video codecs currently available. Bear in mind that HD quality will be substantially worth when compared to a Blue-Ray video. Even for a DVD like resolution of 854x480 pixels, a 1 Mbit/s streams has an up to 8 times lower bitrate than a DVD video usually has.

4.3 Video Encoding Tools

To generate digital video content of any type, video codecs are needed. Modern lossy codecs are required to generate video at reasonable quality and bitrates. As different clients such as browsers support the playback of some, but not all modern codecs, not only one codec is required to generate streamable output, but a set of codecs. In most cases, the different codecs exists as C libraries or command line tools. All of them have different input and output mechanisms and do not share a common API in most of the cases. To generate videos encoded with different codecs in a unified and consistent way, people have developed comprehensive encoding libraries and tools to make a common API available for different codecs.

4.3.1 FFmpeg, Libavcodec

FFmpeg is a set of free and open-source, cross-platform tools for media recording, streaming and conversion under a GPL license. It is built upon the *Libavcodec* library, which allows the decoding and encoding of many different media formats and includes an API to make use of external codec libraries.

Regarding video encoding it includes among others:

- *ffmpeg* - A command line tool to convert multimedia files between formats
- *ffprobe* - A command line tool to extract metadata information.
- *ffserver* - A streaming server that is able to stream video and audio over HTTP, RTP and RTSP

- *libavcodec* – A library containing de- and encoders for audio and video codecs.
- *libavformat* – A library for handling multimedia container formats.

All the included tools and libraries are written in C. An analysis done on the code base shows that it is quite complex and makes use of high performance coding techniques in many cases. The API is therefore very low level. There is no official wrapper for any other language. The code documentation is very weak, with only 6.3 lines of comments per 100 lines of code in total. The user documentation is quite comprehensive, covering the command line tools mentioned before and including some usage examples. Unfortunately, there are many cases in which options are not clearly described and useful examples are missing.

Several wrappers for Python exist like *PyMedia*, *pyffmpeg* or *wffmpeg*, but none of them is in any way feature complete or up-to-date.

The community around FFmpeg is very active. Major new versions are released every several month. Bug fixes and improvements are almost committed every day to the GIT repository. The project aims to keep the old API usable over new releases and regularly releases backports of new features to old versions. The libraries provided by FFmpeg are used in many different projects including various Linux distributions such as Ubuntu and Debian and the Google Chrome browser.

Using external libraries, *ffmpeg* allows the encoding of H.264 video with MP3 or AAC audio as well as Theora and WebM video with OGG Vorbis audio, while the built-in libraries allow to decode almost every known codec [39].

```

1  [STREAM]
2  index=0
3  codec_name=h264
4  codec_long_name=H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10
5  codec_type=video
6  codec_time_base=1/5994
7  codec_tag_string=avc1
8  codec_tag=0x31637661
9  width=1920
10 height=1056
11 has_b_frames=1
12 pix_fmt=yuv420p
13 r_frame_rate=2997/125
14 avg_frame_rate=2997/125
15 time_base=1/2997
16 start_time=0:00:00.000000
17 duration=0:02:28.565232
18 nb_frames=3562
19 TAG:creation_time=2011-03-07 14:52:53
20 TAG:language=eng
21 [/STREAM]
```

Figure 4-1 *ffprobe* example output for one video stream within a video container.

The *ffprobe* tool can extract metadata from all the video containers and video codecs *libavcodec* is able to read. The output is available in an INI like, easy to parse format.

4.3.2 MEncoder

MEncoder is part of the open-source, cross-platform media player MPlayer. Just as FFmpeg, it uses *libavcodec* as its core library and is written in C. The encoder therefore offers a similar feature set and command line interface, using the same or similar parameters and options. In addition player as well as the encoder can play and transcode physical video media like DVDs and Blue-Ray discs.

MEncoder additionally supports configuration files for default values as well as profiles. Both are INI like files that store key-value pairs grouped together in sections. But the parameters for encoding are almost all stored in one single key value pair related to *libavcodec*, greatly decreasing the ease of use of these profiles. While profiles will definitely be important for the configuration of video encoding in Invenio workflows, the usage of MEncoder profiles would make the configuration inconvenient, would take away power from Invenio and require the administrator to study the MEncoder manual to set it up correctly.

Regarding documentation, MEncoder offers a much more complete manual than FFmpeg does. Most options are explained more clearly and hints are given how the video output is influenced. Because the parameter usage is similar to *ffmpeg*, the MEncoder manual can partly be seen as an improved FFmpeg manual.

The activity within the MPlayer community is relatively high, but less active compared to FFmpeg. Major versions are released around every half a year. Many MPlayer developers also contribute to the FFmpeg project.

Regarding the video submission and encoding needs for Invenio, MEncoder offers no real advantages compared to FFmpeg. The way profiles are handled would be of no great use and only documentation then speaks for MEncoder. The player part and handling of physical media are neither needed nor usable in Invenio.

4.3.3 Handbrake

Handbrake is an open-source video encoder GUI and CLI tool. Like MEncoder and FFmpeg it is built around Libavcodec.

While Handbrake can ingest almost every video format through Libavcodec, the output formats are limited by its philosophy: supporting only state-of-the-art codecs to achieve best quality while maintaining simplicity in usage. It focuses on H.264 video encoding through Libx264 with MP3 or AAC audio in a MP4 or MKV container. Besides, Handbrake supports MPEG-4 and Theora video and Vorbis audio. Like MEncoder, it can read and transcode DVDs.

Due to its reduced and simplified functionality, the command line usage differs greatly from the one of MEncoder or FFmpeg. Parameters are differently named, and the number of options is limited to the most basic options like codec, bitrate, framerate and resolution. For H.264, advanced Libx264 parameters can be passed to the encoder and some device specific presets (e.g. iPod and Apple TV) are provided.

The documentation of Handbrake is comprehensive and usage oriented and CLI parameters are explained adequately.

Major releases of Handbrake happened less often compared to FFmpeg and MEncoder in the past. The timeframe between the last two versions was about a year. The repository is quite active; commits are done regularly every several days by various committers.

At CERN, Handbrake is currently used by the Media Archive to transcode videos of the Audio-Visual section to H.264 for the current Flash video player used in the video collection of CDS.

Besides the more simple usages compared to FFmpeg and MEncoder, Handbrake lacks many advanced features and does not support WebM video. The future of the Theora support is questionable due to the projects dedication to H.264.

4.3.4 Mediainfo

Mediainfo is free and open cross-platform software to retrieve metadata from multimedia files. It supports both technical metadata such as bitrates, framerates, resolutions and used codecs, as well as intellectual metadata such as title, author and year. Mediainfo offers both a GUI and a CLI.

For the use case of Invenio, only the CLI is relevant. From the CLI, Mediainfo can export media metadata to plain text or a structured XML output, which makes it ideal for importing the metadata into code, as Python offers sophisticated XML manipulation tools out of the box.

Compared to ffprobe, the information gathered by Mediainfo is more complete and better structured. It supports both human readable and raw output. A bitrate can for example be exported as "1000 kbit/s" or "1000000".

```

1  <track type="Video">
2    <ID>1</ID>
3    <Format>VP8</Format>
4    <Codec_ID>V_VP8</Codec_ID>
5    <Duration>37s 0ms</Duration>
6    <Bit_rate>1 835 Kbps</Bit_rate>
7    <Width>852 pixels</Width>
8    <Height>480 pixels</Height>
9    <Display_aspect_ratio>16:9</Display_aspect_ratio>
10   <Frame_rate>30.000 fps</Frame_rate>
11   <Compression_mode>Lossy</Compression_mode>
12   <Bits_Pixel_Frame_>0.150</Bits_Pixel_Frame_>

```

```

13  <Stream_size>8.09 MiB (92%)</Stream_size>
14  <Language>English</Language>
15  </track>

```

Figure 4-2 Mediainfo example output for one track within a video container.

Tests show that Mediainfo cannot be trusted in any case. While *ffprobe* for example does not output any video display aspect ratio if not explicitly declared in the video stream, Mediainfo seems to simply divide width and height and thereby outputs the storage aspect ratio as the display aspect ratio. When handling anamorphic video, where the real proportions differ from the stored ones, the information from Mediainfo might lead to wrong calculations for automated encoding in the portrayed system and thereby distorting the results.

Mediainfo would be ideal to supplement *ffprobe*. The combined information would increase the reliability greatly, because Mediainfo does not rely on Libavcodec or Libavformat compared to others as discussed before.

4.4 Upload Technologies

When handling potentially huge files like video Masters on the web, it is advantageous to have options to ease the possible problems regarding their upload and movement in distributed networking structures.

4.4.1 Uploads in Invenio

There is a general problem with the upload of large files in the current way the *WebSubmit* module works. The requests handler that directly captures the data received by *mod_wsgi* through *Apache* creates a temporary file on disk where the data is written during the upload. An open file handler is then passed to *WebSubmit*, keeping the temporary file alive. *WebSubmit* will then copy the temporary file to the directory of the currently running submission and close the file handler afterwards to remove the temporary file from disk. This means that the file is moved in every case and is not directly written to the correct folder on disk.

If the system must give the user an immediate feedback after the upload, it won't be possible because it has to move large files before they are available in a permanent file on disk.

For a local file system on a moderate workstation, it might take up to 10 seconds to move a 1GB file (100MB/s sequential reading/writing). For a synchronous feedback, it would be almost too long. It would be possible to have asynchronous feedback with a „currently preparing preview“ message and afterwards presenting additional options like aspect correction, based on the preview. But if we think of possible master video sizes of up to 50GB as they occur in professional video productions such as the CERN AV Section, it might take more than 8 Minutes to move the file to a non-temporary place. This even makes the asynchronous feedback impossible.

The whole problem raises the need for an adaption of the request handler as well as WebSubmit to write uploaded files in the first place to the appropriate directory, to avoid temporary files and moving.

4.4.2 HTTP and HTML based upload

Form based uploads based on HTML and regular HTTP requests have been on the Internet for many years. Until the introduction of HTML5, the HTML4 implementation had strong limitations in the handling of file uploads on the client side.

The file input element “`<input type=“file”/>`” in HTML4 is only able to select one file at a time. As the upload is triggered on the submission of the form, the status or progress of the file upload is only visible in the progress bar of the browser. In general, there is no API in HTML4 and the associated JavaScript standard to access local files or to retrieve the status of file uploads. JavaScript based asynchronous uploads are possible in general, but their progress is opaque. As no JavaScript API exists, regular form-based HTML4 Uploads are not able to send a file using byte-range like requests to start the upload at a specific position within the file. This way, it is not possible on the client side to resume the upload in case of failure at some point.

HTML5 lifts many of the limitation of HTML4. The new file selection dialog allows the selection of multiple files at a time. A new JavaScript API is introduced to handle files and uploads on the client side. The File API [40] finally allows to read local files selected in a file input dialog that can for example be triggered by a “`<input type=“file”/>`” element. The File API provides readonly information on the name, size and Mime type of a selected file. Additionally, a Blob object can be constructed from a File object, which represents a slice of the original file, determined by byte ranges. The Blob enables the chunking of larger files, possibly to create resumable uploads for large files. In addition to the File API, the HTML5 spec includes a new *Progress Events* [41]. A *Progress Event* is triggered when the status of a file upload or download changes. It is thereby possible to implement an in-site progress bar that gives the user feedback on the upload status.

Standard HTML upload through forms, as well as through AJAX, is done over the HTTP protocol. As HTTP is based on TCP, there are checks within the protocol stack to detect corrupted or lost packets and send them again. But the protocol stack does not protect the upload from connection timeouts or a completely dropped connection. In this case, the upload will fail. For a small file, the risk might be negligible, but for large video uploads of several gigabytes it becomes more problematic. To handle this problem, a way of resuming failed video uploads would be well appreciated.

While the client is able to chunk with HTML5, the request handler on the server side must be able to handle the chunked files and reassemble them. In case of a failure, it must know that there are chunks missing and enable the resume feature. The client must be able to recognize failure and store information about the last sent chunk. It then might try to resume the upload by sending the failed and following chunks again. For these problems, custom AJAX based solutions must

be developed because the HTML5 standard only offers the chunking API and the server-side implementation is completely independent.

When HTML5 is considered for advanced features, it is important to bear in mind that HTML5 is still a draft and not all of the features are supported in every browser. It might therefore still be better to rely on well-established solutions, such as Adobe Flash plugins, for handling large uploads.

4.4.3 Flash and other Plugin based uploads

Before the HTML5 spec was developed and browser vendors started to adopt the new features, developers had to look for other solutions to enable sophisticated and user-friendly file uploads.

The Adobe Flash Platform and its ActionScript API have capabilities for accessing local files. Files can be uploaded and the progress of the upload can be accessed similar to the new *Progress Events* in HTML5. As it is possible to interface to a Flash object using JavaScript, it is possible to have a completely invisible Flash object in the background. This background object handles only the technical aspects of the upload. The elements to initiate the upload and messages coming from Flash can be displayed within normal HTML and CSS constructs.

Flash also allows for chunking of input files on the client side. A mechanism similar to HTML5 to chunk uploads can be implemented with Flash objects within HTML4 [42].

There are various open-source scripts available that use Flash objects for file uploads with fall back to HTML4 if Flash is not available, as well as scripts that rely on HTML5 and fall back to Flash and in the end to HTML4. Two of these scripts are *Uploadify* and *Plupload*.

Uploadify [43] is already in use in the Invenio software for asynchronous picture uploads. Multiple pictures can be uploaded and viewed in the *DEMOPIC* submission workflow included in the Invenio demo installation. *Uploadify* is based on a Flash object that handles files and uploads. For its JavaScript API, the script functions as a *jQuery* plugin, meaning that *jQuery* is needed to run *Uploadify*, which comes fortunately with Invenio. *Uploadify* can be set-up on top of a regular form, replacing the file input element of HTML if JavaScript and Flash are available. The script can be used to upload multiple files and it offers a visual upload queue. *Uploadify* has a rich API to customise the behaviour of the uploader through JavaScript. For every action taken by the user and possible response by the server, there is an Event where custom code can be appended. The upload is done asynchronously and additional GET parameters can be send together with the file to a server-side URL. The script does unfortunately not support chunking.

Uploadify is free and open and comes with the JavaScript and Flash sources. To total size of the deployed scripts is about 50kb excluding *jQuery*.

Plupload [44] is a multi file uploader that offers support for various standard uploading techniques and browser plugin based uploads. Depending on browser capabilities and installed plugins, it will either use HTML4, HTML5, Adobe Flash, Microsoft Silverlight, Google Gears or the BrowserPlus plugin. Plupload comes with a complete App-like UI that can be integrated in a website or a custom UI can be developed using the rich API. Plupload is built modular; only the necessary plugin modules need to be deployed. The JavaScript API is constant over the different plugins. Similar to Uploadify, Plupload relies on JQuery. Not every feature of the API is available with every plugin. Chunking is for example only possible with browser plugins, but not with HTML5. In general, the API supports similar Events compared to *Uploadify*. Additional features depending on the available plugins are chunking and client-side image resizing. The HTML5 features of *Plupload* seem to be in an early development stage and are not supported by any browser, because they have not yet been adopted or vendors have locked the File API down due to security concerns.

The script is available free and the source is open, including the JavaScript, Flash and Silverlight sources. The pure JavaScript solution is about 50kb in size. The Flash object needed for the Flash version is about 20kb in size and the Silverlight object is about 45kb large. The total size of a feature complete deployment would be about 115kb.

Comparing *Uploadify* and *Plupload*, it comes mostly down to the comparison of their Flash features. In these regards, only the chunking support is a relevant advantage of *Plupload* over *Uploadify* for video uploads. The other plugins available in *Plupload* have an unpredictable market penetration and should therefore not be relied upon. In general, both plugins have similar dependencies and offer a similar API. *Uploadify* has a smaller footprint but does not offer a rich out-of-the box UI like *Plupload* does. The trump of *Uploadify* is, that it has already been integrated into Invenio for the picture submission.

4.5 Streaming Technologies

In principal, the HTML5 video tag is protocol agnostic. One could use any protocol the browser supports for streaming a video. Unfortunately, none of the current Browsers with significant market share supports any real streaming protocol like the open standard *Real Time Streaming Protocol (RTSP)* or Adobes proprietary *Real Time Messaging Protocol (RTMP)*. The later is obviously well supported by the Adobe Flash Player and open-source implementations for the streaming server such as *Red5* [45] exist. For pure Browser based solutions with HTML5 video, only the HTTP protocol is left for video streams. While it is not a streaming protocol in any sense, technologies have adapted to the constraints of the protocol to deliver video over the web.

4.5.1 Progressive Streaming

The most simple and standard way to deliver video to the user is by just downloading it. The client requests the video file with an HTTP requests and the server answers with a HTTP response containing the binary file stream. The client con-

tinuously writes the binary stream to an appropriate file in its local file system until the download finishes. After the download finishes, the client can load the video file and start the playback. For a small video file sizes this might be acceptable, but for long and potentially big files the user would need to wait for the playback for an unacceptable long time.

To elevate this problem, the playback is started by the browser while the video is still downloading. It works as long as the download rate is bigger than the playback rate. The browser might wait for a buffer built up before the playback is really started. The process is called *Progressive Streaming*.

With HTTP 1.0 requests that either serve the header of the file or the header plus the content starting from the beginning of the file, the video player could not jump to a later position in the video without having downloaded the video up to this position.

Fortunately, the HTTP 1.1 standard defines *Byte Range* Requests. After the file header and the beginning of the file content including the video container information were received, the client can send a GET request that specifies to not start the content of the file to be send from the beginning, but to send only a range of bytes defined by the client. The browser has to approximate the byte range from the duration and total size of the video file to jump to a specific position within the video.

The Apache 2 Webserver required for running Invenio supports the HTTP 1.1 protocol and Byte Range Requests. As described in the Invenio overview, full-text files and other files attached to records are served through the BibDocFile infrastructure. File downloads are not directly handled by the Apache Server but handled by an in-house Python implementation. This is necessary because authorization checks are implemented in Python, and the internal filesystem representation is opaque for the users. When a file is downloaded, a full Python process will be devoted. To enable Byte Range Requests based on this infrastructure, Invenio is compatible to *mod_xsendfile* for Apache. If *mod_xsendfile* is enabled, Python will still handle the authorization etc., but Apache will finally handle the streaming of the file to the client. There is no drawback associated with the usage of *mod_xsendfile*, as it removes the burden from Python partially, it will improve the overall performance of the Invenio installation when many BibDocFile download requests are handled simultaneously.

4.5.2 HTTP Live Streaming

Progressive Streaming allows the client to start the video playback while downloading the video file and enables seeking within a video. But it still does not allow the stream to adapt in quality in bandwidth constrained situations or does allow to stream content that is currently generated on the server side, such as live streams. As live streams are not relevant for the future video processing workflow in Invenio, this part is not discussed.

The basic idea of *HTTP Live Streaming* for variable bandwidth scenarios, also called *Adaptive Streaming*, is to encode a video in different bitrates and resolutions. Thereby creating low quality versions for low bandwidth scenarios like mobile phones on mobile networks and high quality versions for high bandwidth scenarios such as consuming the video at home with a broadband Internet connection. The slaves of different quality are then each segmented into a series of chunks. Each file just contains several seconds of the total video duration. The available qualities are associated to a bandwidth range they should be used in. A manifest file contains the definition of the available qualities and the path to another manifest file containing a list of video segments belonging to the specific quality. The URL of the first manifest file is set as the video source in an HTML5 video element. When the video is to be played, the browser needs to determine the available bandwidth and select the appropriate list of video segments. From this list it then determines the video segment to play. If the browser detects a change in the available bandwidth during playback, for example if the download rate is not sufficient for continuous playback, it selects a list with a lower bitrate and continues with the last segment played.

```

1 #EXTM3U
2 #EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=350000
3 high/video.m3u8
4 #EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=200000
5 medium/video.m3u8
6 #EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=100000
7 low/video.m3u8

```

Figure 4-3 Example HTTP Live Streaming M3U manifest file containing different bandwidth dependent references to other manifests.

```

1 #EXTM3U
2 #EXT-X-TARGETDURATION:10
3 #EXT-X-MEDIA-SEQUENCE:0
4 #EXTINF:10,video_part_0.ts
5 #EXTINF:10,video_part_1.ts
6 #EXTINF:10,video_part_2.ts
7 #EXT-X-ENDLIST

```

Figure 4-4 Example for one of the referenced M3U manifests of Figure 4-3 containing the path to the video segments and their duration.

HTTP Live Streaming is currently not a standard but an Internet Draft submitted by Apple to the IETF [46]. For the manifest files, the M3U playlist standard is extended. An example for such manifests can be seen in Figure 4-3 and 4-4.

On the server side, the video encoding of different qualities can be achieved with *ffmpeg*. The segmentation is possible with a little command line tool written by Chase Douglas that is built upon the *libav* library of *ffmpeg*. Douglas explains in his Blog [47] how to setup a encoding and segmentation workflow for live streams.

Currently, the Apple Safari browser for the desktop and on mobile devices such as the iPhone and iPad support HTTP Live Streaming. Other browsers currently offer no support for the protocol.

4.6 Date Exchange and Configuration

Invenios format of choice for storing and ingesting metadata is *MARCXML*. Different built-in converter for other metadata standards such as *Dublin Core* and *BibText* are available. Regarding video metadata, it is necessary to discuss if and how the *MARC* standard is suitable.

On the other hand, the internal configuration of Invenio is not done in XML like configuration files, but in flat INI-like text files or even hardcoded within Python code files. Already now, this way of storing configuration information is no longer suitable. In some cases, more complex, nestable and in general more semantic solutions are needed. Parsers for Python-like dictionaries within INI configurations have been introduced to Invenio to serve these needs. As the requirements for the project of this thesis have shown that there are needs to decouple the webserver and encoding server infrastructure, ways to exchange data between these instances need to be defined.

This chapter focuses on data representation standards for metadata, configuration and communication.

4.6.1 INI vs. XML vs. JSON

The INI file format is a simple way to permanently store configuration. It defines a one level structure of key-value pairs that can be grouped in sections. In addition, there can be comments in INI files by using a semicolon ';' to indicate the beginning of a comment. In principal, all keys and values in INI files are strings. The INI format is no real standard but a de-factor standard used within many software packages. As there is no definition of Integers, Booleans or Floats given by a standard, a program parsing INI has to handle potential types by its own means.

INI parsing is relatively trivial as long as the INI file is valid: The parser reads line by line. If a section is reached, all the following lines will be associated to it and stored as key-value pairs (mostly as Dictionaries or Maps) belonging to the section, until the next section is reached. Per line, the key-value pair is determined by splitting the String at the first occurrence of the equals '=' character. If there is a semicolon ';' within a string, the information behind the semicolon is discarded.

```

1 ;Encoder configuration file
2 [WEBM_480P]
3 audicodec=libvorbis
4 videocodec=libvpx
5 audiobitrate=128k
6 videobitrate=2000k
7 width=854
8 height=480
9
10 [H264_720P]
11 audicodec=libfaac
12 videocodec=libx264
13 audiobitrate=128k
14 videobitrate=3500k

```



```

15 width=1280
16 height=720

```

Figure 4-5 Example for an INI file storing configuration profiles for a video encoder.

When a multilevel structure is needed, INI can barely be used as it only supports the notion of sections. Formats such as XML and JSON are needed in this case.

```

1  <!-- Encoder configuration file -->
2  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
3  <profiles>
4    <profile>
5      <name>WEBM_480P</name>
6      <audiocodec>libvorbis</audiocodec>
7      <videocodec>libvpx</videocodec>
8      <audiobitrate>128k</audiobitrate>
9      <videobitrate>2000k</videobitrate>
10     <width>854</width>
11     <height>480</height>
12   </profile>
13   <profile>
14     <name>H264_720P</name>
15     <audiocodec>libfaac</audiocodec>
16     <videocodec>libx264</videocodec>
17     <audiobitrate>128k</audiobitrate>
18     <videobitrate>3500k</videobitrate>
19     <width>1280</width>
20     <height>720</height>
21   </profile>
22 </profiles>

```

Figure 4-6 Example for an XML file storing configuration profiles for a video encoder.

XML is almost ubiquitous in the modern software world. As virtually any data structure and information can be represented in XML, the mapping from INI to XML is trivial. But we see that the complexity to retrieve the information programmatically instantly increases. If the Schema is simple and fixed, one might be able to write a dirty parser with some regular expressions matching the XML Tags shown in Figure 4-6. But a DOM aware parser is more complex. For the simple example, where Tags are used, the semantic overhead already doubles the number of characters compared to INI. The overhead can be reduced dramatically if Attributes are used instead of Tags.

```

1  <!-- Encoder configuration file -->
2  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
3  <profiles>
4    <profile name="H264_720P" audiocodec="libfaac" videocodec="libx264"
5      audiobitrate="128k" videobitrate="3500k" width="1280" height="720"/>
6  </profiles>

```

Figure 4-7 Example for a simplified configuration file using Attributes instead of Tags.

JSON stands for *JavaScript Object Notation*. As the name suggest, it is basically derived from the way JavaScript stores associative arrays as objects. JSON is defined by RFC4627 [48] as an Internet standard by the IETF. In difference to Schema-less XML and INI, JSON defines Types. JSON defines *Objects*, *Arrays*,

Numbers, Strings, Boolean and Null. Objects are Maps/Associative Arrays that store key-value pairs. The key has to be a String; the value can be any of the Types.

The structure of the INI file can easily be converted to a JSON object. The overhead in this case is very limited. In the example shown in Figure 4-8, the Python function *eval()* could be used to transform the JSON into a Python dictionary for internal usage, because the JSON code is valid Python code in this case. This shows how near the JSON representation is to Python. The function would fail for Boolean and Null values, as their notation is different in Python. A real parser should of course not use *eval()* as this would be an unacceptable security problem.

```

1  {
2    "MP4_480P": {
3      "videocodec": "libx264",
4      "audiocodec": "libfaac",
5      "videobitrate": "1000k",
6      "audiobitrate": "128k",
7      "width": 854,
8      "height": 480,
9    },
10   "H264_720P": {
11     "videocodec": "libx264",
12     "audiocodec": "libfaac",
13     "videobitrate": "3500k",
14     "audiobitrate": "128k",
15     "width": 1280,
16     "height": 720
17   }
18 }
```

Figure 4-8 Example for a configuration file in JSON notation.

Python is able to handle all the three formats out of the box since Python 2.6.

The JSON parser is available in the *json* module. For Python 2.4 there is an external library called *simplejson*, which is basically the *json* module available in 2.6. For XML parsing and processing there is for example the *minidom* module available since version 2.0. Since Version 2.3 there is the *ConfigParser* module to handle INI files.

Accessing INI data with the *ConfigParser* is extremely simple in Python. An object is created that parses a given file. The configuration options are accessed with the *get()* function with the name of the section and the name of the key as shown in Figure 4-9.

```

1  from ConfigParser import SafeConfigParser
2  parser = SafeConfigParser()
3  parser.read('config.ini')
4  print parser.get('WEBM_480P', 'videocodec')
```

Figure 4-9 Parsing INI in Python with *ConfigParser*.

The *json* module of Python transforms a JSON object directly into a Python dictionary. All dictionary functionalities are therefore directly available on JSON data. An example is shown in Figure 4-10.

```
1 import json
2 config = json.load(open('config.json'))
3 print config['WEBM_480P']['videocodec']
```

Figure 4-10 Parsing JSON in Python with the *json* module

With the *minidom* module, one can access XML data similar to the way JavaScript handles the DOM. Accessing a value in our simple configuration does unfortunately already require to use loops and conditions as shown in Figure 4-11. No simple access of values is possible.

```
1 from xml.dom.minidom import parse
2 dom = parse("config.xml")
3 for node in dom.getElementsByTagName('profile'):
4     if node.attributes['name'] == 'WEBM_480P':
5         print node.getElementsByTagName('videocodec')[0].firstChild.data
```

Figure 4-11 Parsing XML with the *minidom* module.

To handle the example profile configuration effectively in Python, a parser on top of *minidom* could be written to transform from XML to some kind of Config Object or to a dictionary representation just like JSON does.

Seeing the limitations of INI and the unnecessary complexity of XML for the needs of configuration and data exchange in an encoding infrastructure, JSON is strongly advocated for its simplicity in creation, parsing and accessing in Python. The only drawback that comes with JSON is that the standard has no support for comments, as it was developed for data-exchange scenarios with as little overhead as possible. The *json* module does therefore not understand any comment type at all. This problem can be bypassed by implementing a custom pre-parser that removes comments and feeds the parser with a cleaned file.

4.6.2 MARCXML and PBCORE for Video Metadata

The core part of the Invenio software is the ingestion, preservation and representation of document asset metadata. Chapter 5.1.1 explained that metadata in Invenio is stored in the MARC21 format or its XML representation MARCXML. The problem is that video assets or digital video files can carry intellectual or user-settable metadata as well as technical or implicit metadata. While the intellectual part is well represented in the MARC standard by the Library of Congress (LOC), the technical part is not, and the serial way of representing metadata in MARC is not fully suitable for digital video media.

The intellectual metadata often includes the title of the video, the year in which the video was produced or filmed, the names of the producers or authors as well as copyright information and other comments or descriptive texts. User-settable video metadata matches standard MARC fields defined by the Library of Con-

gress in most cases. Fields for the title, year, author, description and copyright existed from the beginning of the standard for print media bibliographies. These fields can naturally be reused for video records: The title can for example be stored in field '245\$a' [49] and the author or producer in the '100\$a' field [50], the year can be stored in the '260\$c' field [51] and a video description or summary can be stored in the '520\$a' field [52]. Between the intellectual metadata fields normally no interdependencies exist. In the case of multiple authors or producers, the author field (e.g. '100') can appear just multiple times. If multiple versions or formats of the same video belong to one record, the intellectual metadata is valid for any version.

Technical metadata contains implicit and unchangeable information about the technical nature of the video, its container and the streams inside the container. The video container, often determinable by the videos extension, encapsulates different streams that can per stream either contain moving images, audio tracks, subtitles or other types of data. Both the container and the streams carry technical metadata. The container might contain information about its own nature, the total duration of the video, the total bitrate of the video or the number of streams included. The streams for example carry information about the nature of the stream (moving images, audio track etc.), the bitrate of the stream, the start time and the codec or encoding type of the stream. In case of audio streams, typical information is the number of audio channels, the sampling rate and the bit-depth. Typical information for moving images is the dimension of the video frames, the aspect ratio of the video or the number of frames per second. The MARC standard defined by the Library of Congress for bibliographical record has no concept to store the technical metadata of digital media such as video in regards of containers and streams. The LOC standard only defines fields and sub-fields for analog video recordings that are mostly stored on tapes or digital video stored on discs. The classification is limited to the physical format of the video and has no notion of the encoding of the video, its digital dimensions or other digital related information such as bitrates or sampling rates [53].

In principal, the MARC format has no notion of which field should store which metadata. It is the bibliographical standard by LOC based on MARC that contains the field definitions. This means that new field in the form of custom fields on top of the LOC standard could be created to store the technical metadata. An important question for the technical metadata is how the units of measurement should be stored in the custom fields. A bitrate for example can be expressed as bits per seconds, bytes per second or kilobytes per seconds and so on. Should the unit be stored together with the value in one field, like '128 kbit/s' or '128000 bit/s'? Or should only the value be stored and the unit be part of the convention of the metadata contained in the custom field? The first way would require the parsing and evaluation algorithms to be aware of different notations and the second way would need the convention to be more strict and the parser aware of the convention.

Another problem in the case of video metadata is how to define the connection between the records intellectual metadata, the medias instantiation and the streams within the media inside a MARC record. As fields can be repeated in

principal, there could be multiple fields 'A' that refer to the different video formats associated to one record. As the video formats have multiple streams each, multiple fields 'B' for the streams could be added to the record. But how can the fields 'B' now be associated to the fields 'A'? The A fields would need to contain a custom identifier in a specific subfield. The 'B' field then would need to refer to their corresponding 'A' fields by containing the custom identifier in a specific subfield again. In Invenio, this kind of association is often done by referring to the BibDoc ID of an asset in the subfield '\$8'. This kind of notation is not natural to the serial MARC format and depends on arbitrary external references, in this case the BibDoc ID.

Public broadcasting institutions in the United States faced the same problems related to digital media metadata. In a cooperative project, they created the PBCore metadata standard for audiovisual media [54]. PBCore is used within and in between these institutions to manage media metadata in data exchange and archival. The standard describes a XML format with a broad set of tags and attributes to handle various types of digital media. It is both suitable for classic metadata fields such as title, author or description and technical metadata fields such as bitrates and pixel dimensions. PBCore defines a notion of Documents, Intellectual Content, Instantiations and Tracks combining the bibliographical metadata with the metadata of video containers and their audio and video streams. The PBCore Document describes a media asset similar to a MARC record. The Document can contain Intellectual Content in the form of descriptive metadata as well as concrete information on how the media is instantiated (Instantiation) as a media file or multiple files in different formats [55]. The Instantiation can then contain multiple Tracks that represent the audio, video or data streams within the files container. All information is represented in a XML tree. The *pbcoreDescriptionDocument* is the root element of the tree. Within this element, the different intellectual content elements such as *pbcoreTitle* and *pbcoreDescription* as well as the *pbcoreInstantiation* reside. The *pbcoreInstantiation* is the concrete representation of the digital media asset. Within the Instantiation element, there can be multiple *InstantiationEssenceTrack* elements that represent the media streams.

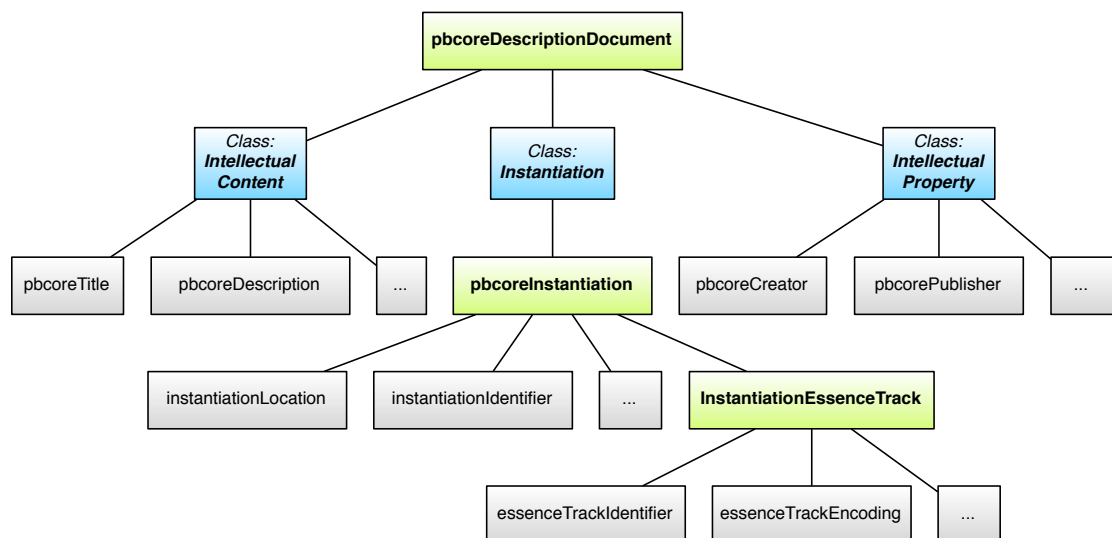


Figure 4-12 Basic structure of the Document, Instantiation and EssenceTrack relation in PBCore. For the leaf elements only some examples are presented.

By using an original XML schema, PBCore also solves the problem of storing the unit of measure in which the values are stored, by offering a *UnitsOfMeasure* attribute on various elements such as the *InstantiationDataRate* element where different units could potentially be applied.

Because the PBCore standard defines both the structure of PBCore records as well as the available fields, it is by its nature limited to what is available in the standard. This makes PBCore less flexible compared to MARC, as the later is in principal just a way to structure metadata in a machine readable way and the fields and subfields could be mapped to any metadata based on the individual requirements of the institution. Most of the institutions might stick to the field definitions by the Library of Congress to ease information interchange between digital libraries. Only if the LOC standard fails to provide the necessary fields for an individual type of record, the institution defines custom fields.

As MARCXML and PBCore are both based on standard XML, one can in principal use *XSLT* to translate from one format to the other automatically. For this purpose a *XSLT* template that defines an adequate mapping between PBCore and MARCXML or vice versa has to be defined and passed together with the source metadata format to a *XSLT* parser. Such a parser is available in the *BibFormat* module of Invenio.

5 System and Workflow Analysis

5.1 System Structure

Different approaches are thinkable to fulfil the requirements for a new and open video processing system in the Invenio software. In general, a complete system to handle the ingestion, processing, storing and playback of video content would consist of four subsystems, each handling one of the different steps:

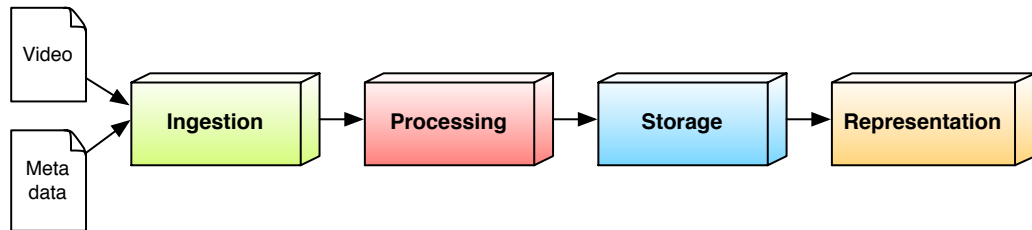


Figure 5-1 General concept for a chain of systems handling videos.

- **Ingestion** - The first subsystem is responsible for the ingestion of the video and its metadata and the creation of a video record in Invenio. This subsystem is basically available in the Invenio software with its WebSubmit framework to create submission workflows and the BibConvert and BibUpload modules to transform the data coming from the workflow into a new record. Based on the available modules, a new workflow and record type would be introduced.
- **Processing** - A second subsystem handles the transcoding of the video material provided by the first system. This processing systems ingests a Master video and creates several Slaves for various usage scenarios. The second subsystem would be a complete new development as Invenio currently lacks any modules or features in these regards.
- **Storage** - The third subsystem stored the Master video and the Slaves created by the previous subsystem. It also handles the access and authorisation of the stored data. The system could either be a specialised system to store and serve videos in a highly efficient way, such as a media streaming server, or be based upon the available file storage and access infrastructure of Invenio and its BibDocFile module.
- **Presentation** - A fourth system prepares the video records visual presentation and enables the playback or download of the different available video formats for the user. This system would be based on the BibFormat framework to generate a record representation, combining metadata from MARC and files provided by BibDocFile.

In principal, the browser and its video playback plugins or a dedicated video player could be seen as a fifth system.

All but the second system could be built upon existing Invenio modules and infrastructure. The focus would then be set on the development of the processing

system. An interesting aspect in this case is the interaction between the first and second and third system for ingestion, processing and storage.

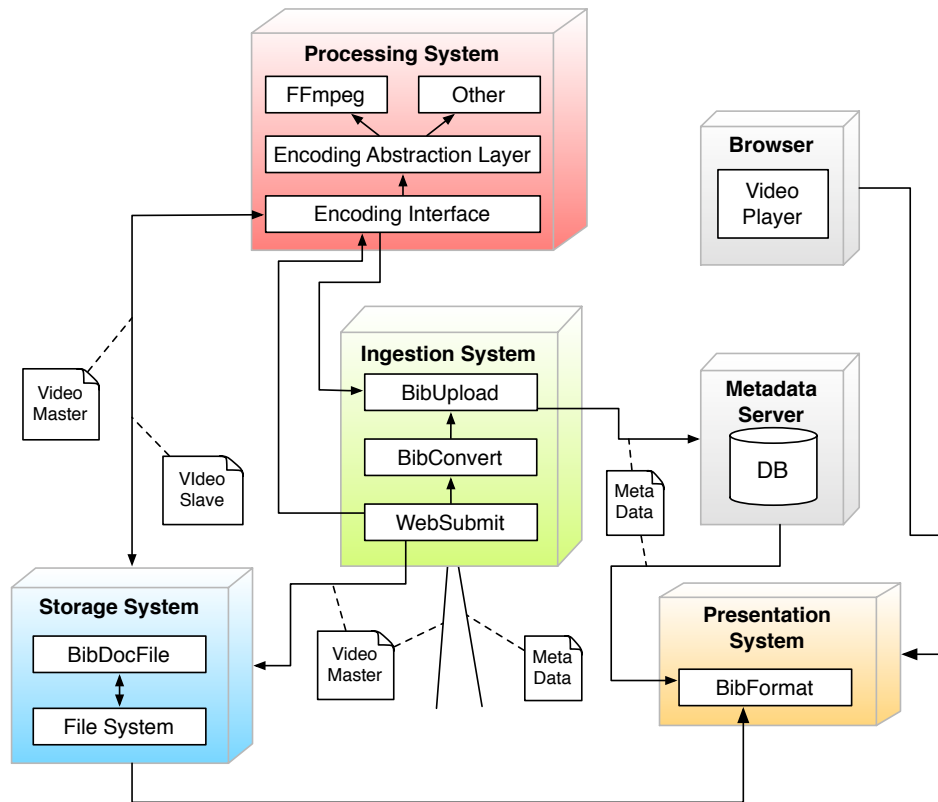


Figure 5-2 First concept of an encoding infrastructure tied to Invenio based on the 'four systems' approach.

Figure 5-2 shows the first concept for a video processing workflow. The submission interface represents the *Ingestion System*, which ingests the records *Metadata* and *Video Master* through a WebSubmit Form. The system would create a record and insert it into the metadata database. At the same time, the Ingestion System would store the Video Master in the *Storage System*.

The *Processing System* is a black box for the Ingestion System. The two systems communicate through an *Encoding Interface*. The Ingestion System upon receiving the Video Master calls the Encoding Interface through which it communicates the *Processing Parameters* and the location of the Video Master to the Processing System. Internally, the Processing System communicates with *Video Encoders* or *Encoding Libraries* to transform the Video Master to different *Video Slaves*. To store the transcoded files, the Storage System is used again. During the processing, the Processing System calls back to the Invenio Server to update the *Encoding Status*. At the end of the processing, metadata that was not available during the submission time, like the final path of the processed files, is send back to the Ingestion System to append it to the record.

The *Presentation System* is not called by any of the other systems, but upon a browser accessing the video record. The system accesses the stored metadata

and Video Slaves to generate a user centred representation in form of a website with an embedded video player.

With the modular system infrastructure, slightly different approaches to the communication and information flow are thinkable.

Figure 5-3 shows a modified version of the workflow presented in Figure 5-2. The unchanged parts are omitted in the visual representation. The Video Master file is directly uploaded to the Storage System, while the metadata is still submitted the Ingestion System. The WebSubmit submission will call the Processing System to look for the Video Master in its respective location and start the encoding process according to transmitted parameters. This workflow resembles the one of the Media Archive at CERN, but the conversion is not automatically triggered on the encoding server. The basic idea would not be to have the separate input as an inconvenience for the user, but make for example the Storage System of a professional video production available for the Ingestion and Processing System. By this means removing the inconvenience to upload the Video Master.

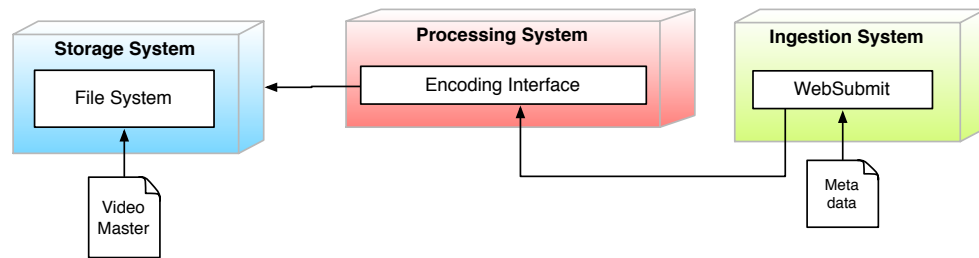


Figure 5-3 An alternate concept for the encoding infrastructure, where the video master and the metadata are handled separately.

Figure 5-4 shows a third approach where the process chain is not initiated by uploading metadata. Instead, a daemon regularly watches a 'hot' folder on the Storage System and automatically starts conversions according the predefined parameters. When the encoding finishes, the Processing System calls BibUpload to create a new record based on the predefined parameters and metadata extracted from the video. This approach resembles the automatic way the processing is initialised on Media Archive, but in this case a record is required in the first place before the processing can be triggered. The concept for this workflow is to start a lengthy video encoding while the user has time to prepare comprehensive metadata and update (and unlock) the record later.

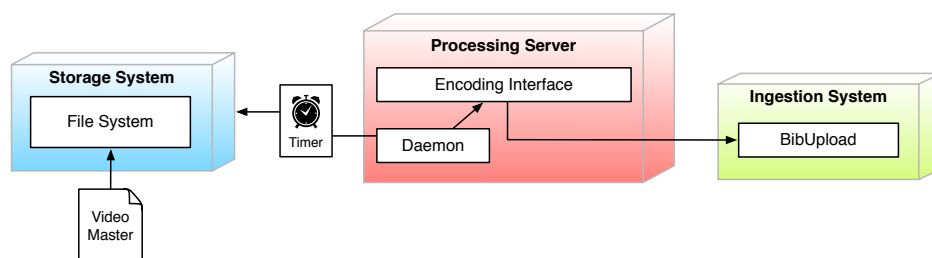


Figure 5-4 An alternate concept where the video master is uploaded to a file server and the processing is automatically initiated.

5.2 Use Cases

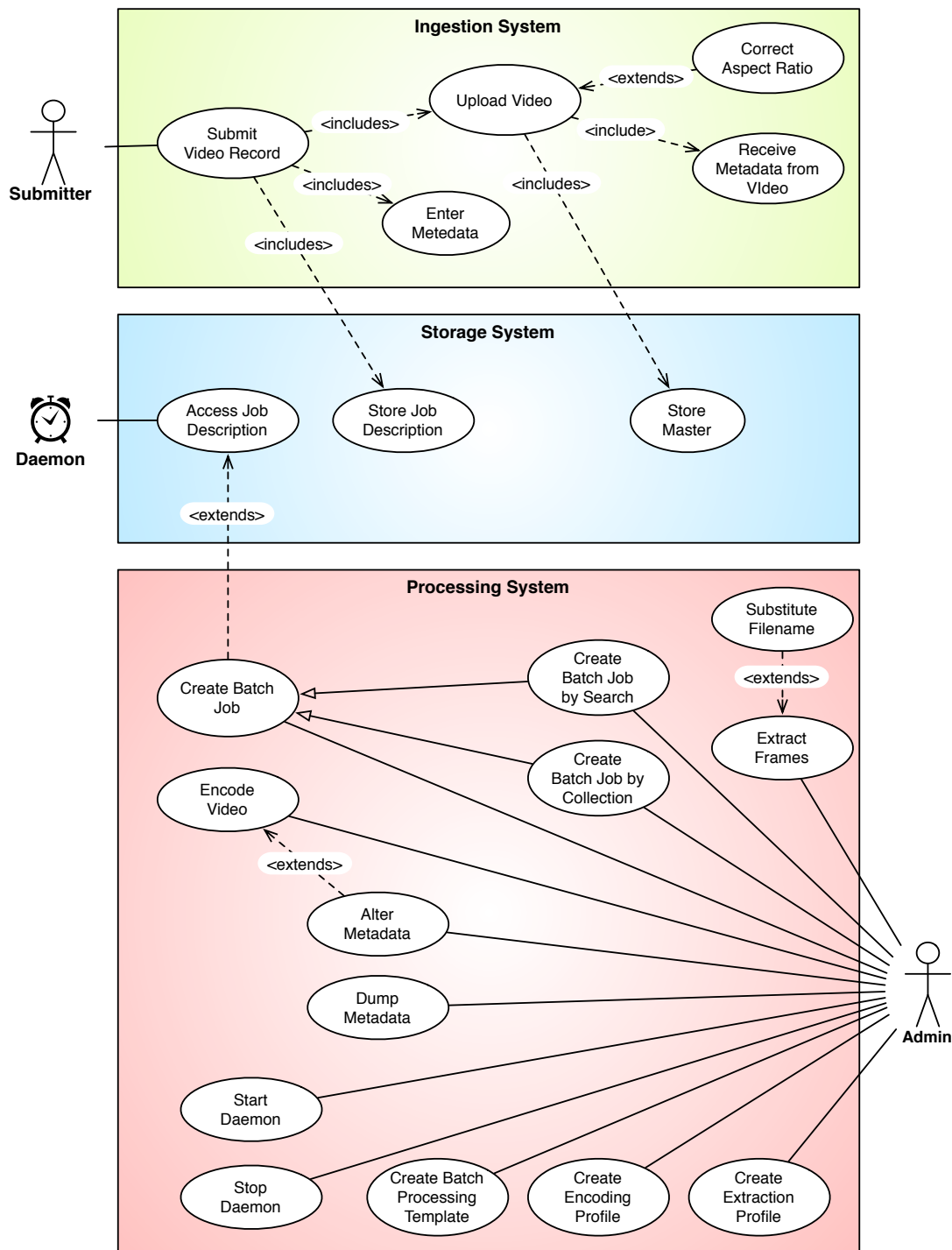


Figure 5-5 Use-Case Diagram for the Video Workflow. Showing two human actors in the form of an end-user who submits a new video record through the Submission System and an administrator who manages the Processing System. In addition, a non-human actor is shown in the form of a Daemon that automatically launches processing jobs based on the results of the human actors.

The Use Cases of the previously described system focus on the user of the Ingestion System and the Processing System. The Ingestion System is the front-end for the end-user. The *Submitter* Actor uses it. A direct interaction with the Processing System is only performed by the *Administrator* Actor. The Use Cases of

the system are shown in Figure 5-5. Only Use Cases with direct user or daemon interaction are considered. Back-end processing and logic are not shown. Appendix C of this document contains the descriptions of the most important Use-Case.

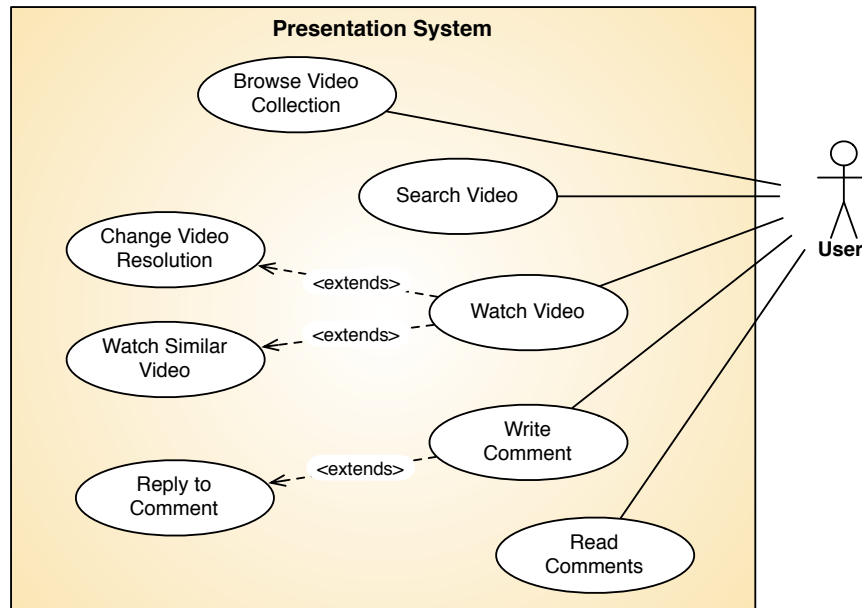


Figure 5-6 Use-Case Diagram for video consumers. The human actor only consumes video material from the Presentation System and never gets in touch with any of the processing related systems.

The Use-Cases for the end-user or video consumer focus on the Presentation System. Figure 5-6 shows the Use-Case Diagram for such a user. The user can browse collections of videos and watch individual videos. The video quality and resolution can be adapted to the needs of the user (e.g. low quality content for mobile devices and high quality content for broadband desktop computer access). The user receives recommendations on similar video content. In addition, the user is able to read and write comments to discuss the video.

6 Design and Implementation

This chapter focuses on the solutions developed based on the requirements and technologies analysis described in the previous chapters.

6.1 Object Oriented vs. Procedural

Currently almost none of the Invenio modules are developed object oriented. The excessive use of objects and class hierarchies in Python is reducing the performance of the code. This was at least true in 2004 for old Python versions, as tests have shown by Tibor Simko [56]. According to his work, object function calls perform at least 10% worse than non-object functions. Subclassing decreases the performance even further by 1% per level. Simko states, that simple object orientation is okay, as long as no complex class structures are developed and objects are not used in performance critical tasks. This philosophy limits object orientation to structuring the code in a one level class structure where objects are neither passed around as data container, nor used for doing their data processing independently. This limited way of doing OO can be seen in many modules of Invenio such as BibDoc and BibSched, where the classes only structure a set of functions that could easily live outside the class structure, as there are already numerous functions outside the classes in these modules, working on the same data.

Most modules in Invenio rely on Python dictionaries to structure data. Dictionaries store key-value pairs where both can be any atomic type, structure or object. Most of the time, they store keys as integers or strings and values as atomic types, lists, tuples or other dictionaries. When passing around these dictionaries through the procedural structure of Invenio, every function relies on conventions on the structure of these dictionaries that are not clearly defined in most cases, because no schema for these dictionaries can be defined in Python. This is a huge drawback compared to OO programming, where the object encapsulate the data and its structure is centrally defined in the objects classes. On the other hand, when the convention is at least documented in the code, the usage of dictionaries reduces overhead in creating data structures greatly, as no class structure, constructors or getter and setter methods are needed.

The module developed and further designated as *BibEncode* adopts the notion of submodules to structure code instead of following an object oriented approach or implementing a misleading classes that fulfil no purpose but structuring some functions. Submodules reside in their own Python code files each. The module hierarchy of Invenio is flat in reality. To access a module within Invenio to notation '*invenio.module_submodule*' is used. Within each of the submodules, the code structure is completely procedural.

6.2 Functional structure of BibEncode

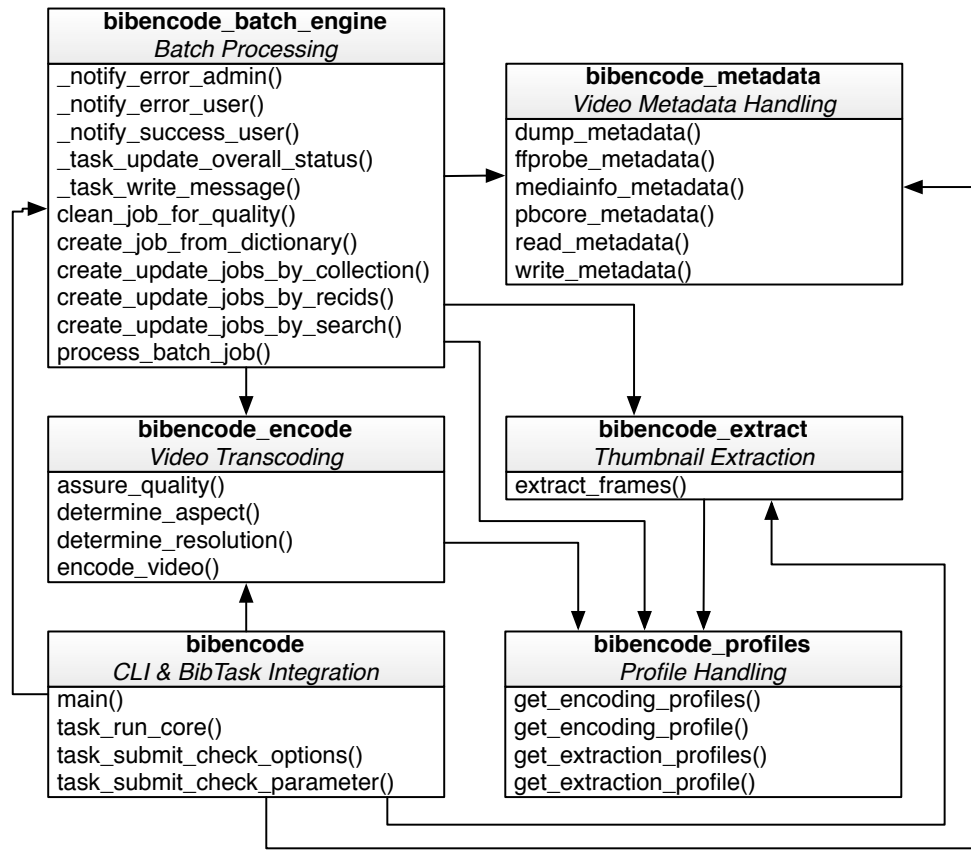


Figure 6-1 A structural overview of the BibEncode module with its most important submodules and functions. Be aware that this is not a UML class diagram, as BibEncode is developed in a procedural style.

The BibEncode module is divided into several submodules that handle the different aspects of video processing. Some of the submodules work completely stand-alone; others integrate into available Invenio infrastructure to do their tasks.

bibencode – Defines the command line interface of BibEncode and interfaces with the BibTask API to systematically schedule BibEncode tasks in BibSched.

bibencode_encode – Enables the transcoding of video files by wrapping around the *ffmpeg* video encoder. In addition, it handles the detection and calculation of video aspect ratios and resolutions for optimal transcoding results.

bibencode_extract – Responsible for the creation of thumbnails from video files to generate previews.

bibencode_metadata – A module to import video metadata into Python code. It wraps both the *ffprobe* and *Mediainfo* tools and parses their outputs. The combined metadata can be used to export metadata XML files according to the *PBcore* standard. The module can also write new metadata to video files by utilising *ffmpeg*.

bibencode_profiles – Allows the ingestion and usage of profiles with predefined configurations in the other modules of BibEncode. Profiles are stored in JSON notation.

bibencode_batch_engine – Implements the processing of complete video submission and reprocessing workflows. Ingests JSON based batch job descriptions.

bibencode_daemon – Defines a daemon that watches ‘*hot-folders*’ where job descriptions are placed and launches these jobs. Decouples the Processing and Submission System.

bibencode_utils – A Library of utility functions related to video encoding and required by all of the other submodules. Handles the parsing of JSON files used in BibEncode.

bibencode_config – Central storage for BibEncode internal configuration variables.

bibencode_tests – Contains Unittests for the BibEncode submodules.

bibencode_websubmit – Defines the prototype for an AJAX enabled WebSubmit interface for video submissions.

6.3 Wrapping FFmpeg and Mediainfo in Python

The technology analysis described in chapter 4.3 showed that the *FFmpeg* video encoding library and its set of tools such as *ffmpeg* and *ffprobe* are most suitable for the needs of video processing in Invenio. In addition the *Mediainfo* software was selected to supplement missing or unreliable metadata capabilities of *ffprobe*. As explained, several wrappers for FFmpeg in Python exist, but none of them implements the necessary feature set. Wrapping the C-code of *Libavcodec* with CTypes or similar solutions is possible in principal, but the low-level and highly performance optimized functional structure of FFmpeg would make this effort very complex and time consuming and would take the focus away from the Invenio related problems of this project.

There is a set of utilities in Invenio called *shellUtils*, which offer a simple API to execute external binaries or scripts. *Shellutils* were first designed to use the built-in *run()* function of the *os* module, but they have recently been refactored to use the *subprocess* module. *Subprocess* is designed to run external programs as background processes and receive their results asynchronously. But *shellutils* makes only light use of its capabilities and is not able to run all processes in background or to receive results or output of the process asynchronously during its execution.

To wrap the *FFmpeg* and *Mediainfo* binaries and their command line APIs, direct use of *subprocess* is made. The *Popen* class is used to execute the binaries. A *Popen* object is instantiated by passing a shell command to the *Popen* constructor. The command consists of a list of arguments, beginning with the path to the

binary, followed by the parameters that would normally be used on the command line terminal. Additionally, the targets for the message output and error output of the process can be defined. These can either be open file handles to write directly to the file system or pipes to memory. Once a *Popen* object is instantiated, the binary is immediately execute. Code execution in Python can now either continue normally, without a need for the process to finish or terminate, or halted until the process finishes with the *wait()* function of the *Popen* object. During the execution, the pipes can be read with the *communicate()* function. The latest output of the process is retrieved and the pipes are flushed afterwards. To receive the current status of the process, the *poll()* function is used. It returns either a *None* type if the process is still running or the operation system return code after the process has finished. Return codes are positive or negative integers that range from 0 for a successful execution, to 1 for an error during the execution and to -15 if the process was terminated by the system, with several other codes in between.

The commands for *ffmpeg* differ greatly between the transcoding of videos and the extraction of video frames. The kind of messages received from *ffmpeg* are also different in both cases and the parsing of the encoding status is far more complex when transcoding a video compared to just generating several frames. For these reasons, *bibencode_encode* and *bibencode_extract* wrap *ffmpeg* individually, adapted to their needs. *Mediainfo* and *ffprobe* are wrapped in *bibencode_utils*. The parsing of the their outputs is done in *bibencode_metadata*.

A difficulty is the parsing of *ffmpeg* output while running video encodings. *FFmpeg* itself has neither a human nor a really machine friendly status presentation when it is running on the command line.

```

1  Metadata:
2  major_brand      : qt
3  minor_version    : 537199360
4  compatible_brands: qt
5  creation_time    : 2011-03-07 14:52:53
6  encoder          : Lavf53.5.0
7  Stream #0.0(eng): Video: libx264, yuv420p, 1920x1056, q=2-31, 2 kb/s, 2997 tbn,
   23.98 tbc
8  Metadata:
9  creation_time    : 2011-03-07 14:52:53
10 Stream #0.1(eng): Audio: libfaac, 44100 Hz, stereo, s16, 0 kb/s
11 Metadata:
12 creation_time    : 2011-03-07 14:52:53
13 Stream mapping:
14 Stream #0.0 -> #0.0
15 Stream #0.1 -> #0.1
16 Press [q] to stop, [?] for help
17 frame= 42 fps= 0 q=69.0 size= 1kB time=00:00:00.-4 bitrate=-244.2kbits/s
18 Mframe= 50 fps= 39 q=69.0 size= 2kB time=00:00:00.29 bitrate= 53.2kbits/s
19 Mframe= 58 fps= 32 q=69.0 size= 3kB time=00:00:00.62 bitrate= 33.4kbits/s
20 Mframe= 66 fps= 28 q=69.0 size= 3kB time=00:00:00.95 bitrate= 27.3kbits/s
21 Mframe= 74 fps= 26 q=69.0 size= 4kB time=00:00:01.29 bitrate= 24.4kbits/s

```

Figure 6-2 Output of *ffmpeg* to the command line during the encoding of a video.

To generate meaningful and transparent feedback for the administrator, the *ffmpeg* output needs to be parsed and transformed into a human readable representation. The last lines in Figure 6-2 show the current progress of the video encoding as ‘time’ parameters. This parameter shows the current position within the video in a timecode notation (*hh:mm:ss.ss*). Unfortunately, the total duration of the video is not presented in the output. To generate a progress bar or a similar indicator, metadata that includes the total duration of the video is first parsed from *ffprobe*. The video encoding is then started and the output of the sub process is piped to a log file. The last line of the log file is read periodically. The current time is parsed from the timecode and divided by the total duration. For the task status column in the visual manager of BibSched, the calculated percentage is transformed into a text based progress bar shown in Figure 6-3.

```
[4/7][##### ][1/1] 057% > ...s/g0/675/CERN-MOVIE-2010-0
```

Figure 6-3 Encoding status representation in the visual task manager of BibSched. The first bracket shows the overall task status. The 4th of 7 subtasks is in process. The second bracket shows a progress bar for the current subtask. The third bracket shows the sub process status of subtasks (e.g. for multi pass encoding). The right side shows the file that is currently processed.

6.4 Command Line Interface and Operation modes

When it comes to the usage of BibEncode through the command line interface or the low-level task submission API, the module offers several distinct modes in which it is able to run.

The use-cases of the CLI are mainly maintenance and manual intervention if the automatic submission processing fails or very specific and singular needs of clients are to be fulfilled.

6.4.1 Encoding Mode

The Encoding mode focuses on the *bibencode_encode* submodule to do single video transcoding using either direct parameters to the CLI or predefined profiles. Basically, the CLI parameters are mapped to those of the *encode_video()* function. Input and output files are defined and either the name of a profile is given, or parameters such as the video resolution, audio and video codecs and bitrates are given directly by the administrator. There is no need to provide all the parameters; the mandatory parameters are only the input file path and the output files path. The settings for the encoding are in this case auto-detected by *ffmpeg* according to the inputs videos metadata and the output files future video container, based on the output file extension.

6.4.2 Extraction Mode

In the extraction mode, command line parameters are mapped to the *extract_frames()* function of the *bibencode_extract* submodule. Just like in the Encode mode, the administrator can either use predefined profiles or direct parameters. Both single and multiple frames can be extracted from a video to a given destination. Multiple frames can either be extracted by providing a number of

frames to be extracted or by giving a list of positions as either timecodes or percentages of the total duration as integer values. If multiple frames are to be extracted, BibEncode will automatically add running numbers at the end of the output filename. The formatting and the position of the numbers can be individualised by using substitution tags within the output filename. Besides the numbers, the timecode at which the frames was taken, the dimensions of the frame and the input filename can be substituted into the output filename. The tags are basically Python string substitution markers like `'%(timecode)s'` or `'%(number)02d'`. By doing so, the output name can be completely constructed based on the input with all necessary information included for displaying the frames as thumbnails in video previews etc.

6.4.3 Metadata Mode

The metadata mode of the BibEncode CLI operates in different sub-modes to handle video metadata. In dump mode, metadata can be dumped from a video into a text file in JSON format, using either *ffprobe* or *Mediainfo* output. It is also possible to create a PBcore XML document. The PBcore document combines the most reliable information from both the *ffprobe* and the *Mediainfo* library. Besides dumping data, the metadata mode can insert metadata into a video container. Due to the limitations given by FFmpeg, only a limited set of metadata key-value pairs can be inserted and a copy of the video needs to be created in order to do so. Input metadata is either given as a serialised JSON dictionary with key-value pairs on the CLI or by an input file containing a JSON dictionary.

6.4.4 Daemon Mode

In Daemon mode, BibEncode is watching a given folder and launches new BibEncode tasks according to job description files found in these folders. Job descriptions are files containing a JSON dictionary. The description can either be of a single job type or a batch processing type. Single jobs just store CLI parameters for specific operation modes of BibEncode like *Encode* or *Extract*. This allows for a remote execution of BibEncode Tasks if access to the job directory is given. A batch job is executed by the Batch Engine submodule of BibEncode. Batch jobs normally work on records and produce output in BibDocs. Launching Batch Tasks through the daemon allows the decoupling of the submission and processing infrastructure where the submission system defines what is to be done and the processing system decides how and when it is done. Jobs that are started or done are moved to a different directory by the Daemon. Multiple Daemons should never work on the same directory. The Daemon can be launched as a periodic task through the `'-s'` parameter available to every BibTask. The parameter is followed by a value the task should be rescheduled after in seconds or minutes etc. like `'10s'` or `'10m'`.

6.4.5 Batch Mode

The Batch mode corresponds to the Batch Processing Engine of BibEncode. The Batch Engine can either ingest a Batch Description for a specific record or create a new description based on a Batch Template. If a complete Batch Description is

ingested, a new Batch Job is instantly submitted as a BibTask. When new Batch Jobs are to be created from a template, the user of the command line interface can specify a record or a list of records where the template should be applied. It is also possible to apply it to a collection of records or to every record that matches a given search pattern. Instead of launching the Tasks directly, Job Descriptions are generated and placed inside a folder the Daemon can evaluate. If the Batch Job creation is used on Templates that use the “Update From Master” concept, whole collections of video records can be updated with new video formats defined in the templates. If the master videos were appended to the records in the first place, it is simple for the administrator to make new video formats available by creating a new template and feeding it to Batch Mode.

6.5 Integration and Interaction with other Invenio Modules

While most of the submodules of BibEncode can be used stand-alone, the *bibencode_batch_engine* and *bibencode_websubmit* modules tightly integrate into the Invenio infrastructure. In addition, several new features, elements and functions were added to existing modules of Invenio to enable full video capabilities.

6.5.1 Basic Video Submission Workflow

The video submission workflow designed for Invenio focuses on the Batch Processing module of BibEncode. The *bibencode_batch_engine* module implements a systematic approach to create all the necessary video and image formats for rich video records. The module handles the evaluation of the input material, the sequential execution of configurable processing steps, the appending of the output to the record in a controllable way and the necessary alternation of the records metadata after the different processing steps. The flow chart for the BibEncode Batch module can be found in Appendix D. The module does not work alone, but integrates tightly into the existing Invenio infrastructure:

- A *WebSubmit* based submission Interface ingests the video metadata and video master file through a form. The submission form is a custom creation for video records and uploads and defined in *bibencode_websubmit*. Video uploads are stored in a temporary directory.
- BibConvert converts the metadata from the form to a *MARCXML* representation. For this purpose, a custom BibConvert conversion template was created for the project.
- The *MARCXML* file is ingested by *BibUpload* and inserted as a new record into the bibliographic metadata database.
- The custom WebSubmit video submission functions create a batch processing job for *BibEncode Batch*, based on a predefined template, that includes the necessary steps to create all video slaves and preview thumbnails for the video record. The batch job description is stored in a hot folder that is watched by the *BibEncode Daemon*.
- The *BibEncode Daemon* ingests the batch description and launches a new *BibTask* for *BibEncode Batch* to process the job.
- *BibEncode Batch* follows the instructions of the batch job and creates different new versions and formats of the video stored in the temporary directory.

It utilises the *Encode* and *Extract* module, which wrap *ffmpeg*, to create them. The slaves as well as the master are then stored in *BibDocs*, attached to the record that was created by *WebSubmit* and *BibUpload*. In the end, Batch will create a new *MARCXML* file with additional metadata that was extracted during the processing. The file is send to *BibUpload* to append the data to the record.

The whole process is shown in Figure 6-4.

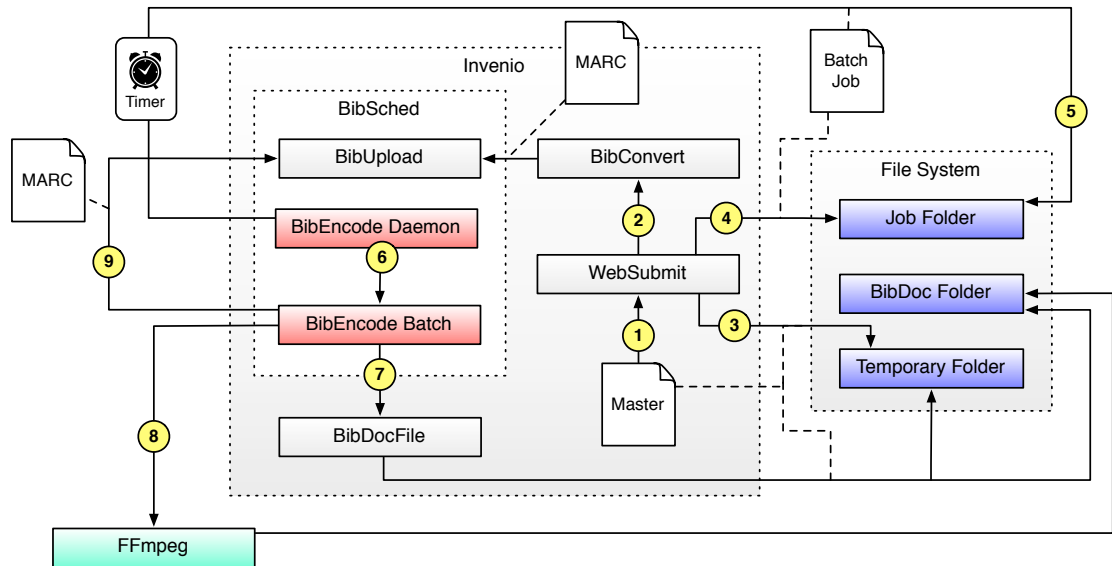


Figure 6-4 Basic workflow between the background modules during a video submission. (1) A video master is submitted to WebSubmit through the web interface. (2) WebSubmit calls BibConvert and BibUpload to inject a new video record into the database. (3) WebSubmit stores the master in a temporary file. (4) WebSubmit creates a batch job description file describing the necessary steps to generate all the required video formats for the record. The description is stored in a predefined job folder. (5) The BibEncode daemon is scheduled periodically in BibSched and analyses the job folder for new job descriptions. (6) If a new description is found, the daemon launches a BibEncode batch task with the path to the job as a parameter. The BibEncode batch tasks reads the job description and (7) calls BibDocFile to copy the master from the temporary directory to its final position in a new BibDoc folder. (8) BibEncode batch calls FFmpeg to transcode the slave versions directly into the previously created BibDoc folder. (9) After the transcoding is finished, the newly created files and their metadata are appended to the record through BibUpload.

6.5.2 BibSched and BibTask Integration

Transcoding video from one format to another is a time and resource-consuming task. Depending on multiple factors, the encoding of a single transcoding can often not be done in real-time, but will take up a multiple time of the duration of the video. The time the encoding takes is always a function of the number of video frames to encode, the video frame dimension, the performance of the codec with given output parameters and the performance of the machine running the encoding. It is apparent that such a heavy task cannot be done on the fly during the submission of a video. It also makes sense to not run these encoding processes somewhere in the background, where they are opaque to monitoring and maintenance and the status of the lengthy operation stays unknown until it either succeeds or fails. In case of error and failure, the process should be able to

be reinitialised after intervention. It should also be possible to schedule the encoding at times when resources are available and the CPU time consuming encoding does not interfere with other processes that have an direct impact on the performance of the system and most importantly the perceived performance by the user.

For resource intensive tasks, Invenio comes with the BibSched and BibTask modules, which allow the systematic scheduling and monitoring of processes. BibEncode integrates into BibSched as a BibTask. For a Python script to run as a BibTask, the script needs to implement a specific API. The BibEncode base module *bibencode.py* implements the BibTask API in the form of three functions. The *main()* function initialises the BibTask by passing the functions *task_run_core()*,

task_submit_check_options() and *task_submit_elaborate_specific_parameter()* to the BibTask factory function *task_init()*. A BibTask can both be launched by its command line interface and a low-level submission function in the BibTask module. In both cases, the parameters for the BibTask are passed as concatenated key-value pairs like on a CLI interface. In this way, the parameters can be evaluated by the same functions for both the function call and CLI submission.

The *task_submit_elaborate_specific_parameter()* function is responsible for validating individual parameters passed to the command line interface of BibEncode. The function for example checks if the frame height is given as an integer value to the CLI. In some cases parameters are sanitized before they are passed to further functions or converted into a different type and stored in an internal parameter dictionary. If the validation of the parameter fails, an error is raised and a problem description is presented to the user on the command line interface or sent to the admin via email if the task was submitted via function call.

The *task_submit_check_options()* function is if the validation of every individual parameter was successful. The function checks if interdependencies between parameters are met. For example in the extraction mode of BibEncode, the user can either define the frames to be extracted by using the *'numberof'* or the *'positions'* parameter but not both or none. The handling of error is done in the same way as in the previous function for individual parameters.

If the interdependencies are met, the task is submitted to the BibSched queue. The task stays in the BibSched queue in the *'WAITING'* status until its scheduled execution time is reached and the task is finally launched. Upon launch, the *task_run_core()* function is called to start the processing. This function calls, depending on the operation mode and given parameters, the functions responsible for the real processing of the video material in the BibEncode submodules such as *encode_video()* in *bibencode_encode.py*. The processing functions such as *encode_video()* or *extract_frame()* can in principal be run as stand alone functions without being called by *task_run_core()*. The said functions implement an API to send update, status and error messages to other functions during their execution. For the case of running inside a BibTasks, *task_run_core()* passed the functions *write_message()* and *task_update_status()* to the processing function. The messages sent are then written into the BibSched log and the status column in Bib-

Scheds visual task monitor. The status of the BibEncode processes thereby become transparent for the administrator and is recorded in the Invenio logging infrastructure.

6.5.3 BibDoc Integration

We have seen that within the Media Archive workflow, video masters, slaves and generated thumbnails reside on a DFS, the distributed file system of Microsoft, in a folder structure with subfolders for the years of creation and a strict file naming schema with running numbers.

As we have seen in chapter 5.1.2, BibDocs do follow their own folder and naming structure. Every BibDoc resides in its own folder named after the BibDocs ID. Within the BibDocs folder, the BibDocFiles appended to the BibDoc reside. Their *Docname*, *Extension*, *Subformat* and *Version* define BibDocFile filenames.

When the BibDoc API is used to append new Files to a record, these files will be copied to the BibDocs directory and renamed according to the set *Docname* and *Format*.

The videos master and all slaves should go into the same BibDoc, because they can clearly be seen as different formats of the same video.

Moving files in some file systems can be cheap tasks, if only metadata is altered with no physical impact. If the file is physically moved on one disk, between disks on the same machine or even between machines through a network, an immediate performance impact is seen.

At CERN, MOV format master files of up to 50GB are produced by the AV section and copied to the DFS. Slaves are usually ten times and more smaller than the master, but the need for a number of different slave formats and resolutions can add up to several hundred megabytes. The goal is to get rid of unnecessary file movements as the type of filesystem and its capabilities the Invenio installation is residing on cannot be predicted.

Currently, when uploaded through a WebSubmit submission workflow, a master would first be created as a temporary file, and then moved to the submissions directory. If added to a record as a BibDocFile, the BibDocFile module moves it again to the BibDocs directory. In a later chapter, we will see that we can get rid of the first move, but the second move of the master to be BibDocs directory has to happen in any case, if we store the video related files in BibDocs, just like every other file stored currently in Invenio, excluding the CERN specific Media Archive.

In principle, once a BibDoc has been created and its folder is available in the file system, files can be manually moved into this folder. The path to the BibDocs folder can be retrieved with its *get_base_dir()* function. If one is about to copy a file to this folder manually or programmatically, but not through the BibDoc API, the new filename has to fit the naming scheme given by the BibDoc. To stream-

line the assembly of correct BibDoc filenames, two new functions are added to BibDocFile. One is *compose_file()*, which composes a full path including the filename for a new file by receiving the BibDocs directory, the Docname, the Extension and additionally the Subformat and the Version. Another function *compose_format()* generates a correct BibDoc Format by passing a files extension and wanted Subformat. With these new tools one is able to easily add files to BibDocs with move operations of the OS or stream generated files in the very moment of generation directly into a BibDoc without the need to copy them again. Afterwards, the file is accessible through the BibDoc API, but not added to the MARC of the record, as this is only done when the API is used for adding new files. Fortunately, there is a function called *cli_fix_marc()* in the command line tool of BibDocFile module to repair the MARC for a given BibDoc, which adds missing metadata.

For adding the master file, we can use the regular BibDoc API to append it to a record. If the internal metadata of the master is to alter before it is added, we can pipe the master through FFmpeg to alter the metadata and write the updated file directly into the BibDocs directory, saving one step of copying. For the creation of slaves, we can directly pipe the encoders output into the BibDocs directory. For every slave, it would otherwise be necessary to write the file to a temporary directory, let BibDoc copy it, and remove the temporary file afterwards. As generated video thumbnails are generally very small in their files size (several kilobytes or up to several hundred kilobytes depending on the images format and size), the regular BibDoc API can be used for them without significant drawbacks.

6.5.4 WebSubmit Integration

As explained in chapter 5.1.4, WebSubmit is one of the oldest modules in Invenio and misses many state-of-the-art features that would today be mandatory requirements for systems handling user submitted content.

On the other hand, it is possible to alter the server side conditions of a submission in any thinkable way, as WebSubmit allows the execution of Python code within custom elements and functions. This also makes it possible to send back custom HTML and JavaScript to the client to bypass the limitations of standard workflows regarding asynchronous interaction, file uploads and form validation, and to allow a custom styling of the submission.

As an overhauled version of WebSubmit is currently in development, any current workflow needs to be revised in the future to integrate in the new WebSubmit environment, which offers many of the currently missing features and completely alters the ways submission workflows are created.

The proposed workflow of this thesis should be seen as a prototype for a future video submission workflow, as it integrates in the current version of WebSubmit and uses a great amount of both custom Python and JavaScript code, it would not be easily portable.

6.5.4.1 Video submission form

Instead of using regular WebSubmit elements to create the submission form and its fields, the prototype uses a single WebSubmit response element. This element includes Python code to import and call a function in the BibEncode module. The called function generates custom HTML and JavaScript code and sends it back to the response element. The response element then renders the custom code into the form.

The function will generate a valid HTML form that includes fields for the title of the video, the year the video was created, the authors or producers of the video and a description. In addition, there is a file field to select a video file from the clients file system to be uploaded. Without any JavaScript additions, or deactivated JavaScript in the Browser, the webpage would behave like a normal WebSubmit generated form. With activated JavaScript, the form is heavily altered by the JS code appended to the form.

The client side code makes use of JQuery, an all-around JS library that alleviates many tasks web-developers face when creating interactive websites.

For the validation of the form, a JQuery plugin named *Validation* is used. *Validation* enables the live validation of forms and offers a set of predefined and customisable rules to evaluate field values and report problems to the user. *Validation* hooks directly into the form and validates on submission, generating labels with error messages if problems are detected. Figure 6-6 shows some example error messages.

Demo Video Submission

Title:
Inauguration du tramway au CERN

Year:
2011

Author(s), one per line:
CERN Video Productions

Description:
Le samedi 30 avril, le tram Meyrin-CERN a été inauguré sous un ciel radieux. Les Meyrinois-es ont pu profiter d'une belle fête populaire sur la place de Meyrin-Village.

Upload Video Finish Submission

Submission number(1): 1311680994_28532

Figure 6-5 Simple HTML form styled with CSS for video submissions with YouTube like fields and no advanced options. Two buttons are provided to upload a file and to submit the form.

Title:
Please enter a title.

Year:
Please enter a year.

Figure 6-6 Detail of the form shown in 6-5, when the form is submitted, but mandatory fields are missing

The JQuery plugin *Uploadify* is used for asynchronous file uploads. *Uploadify* is partly Flash based to allow multi-file uploads and the dynamic evaluation of the upload status on older browsers, that do not support these features in HTML5 or HTML5 at all. It has a HTML fallback if Flash is not available. *Uploadify* is chosen over *Plupload*, because it has already been integrated into Invenio for the photo

upload workflow and its server side infrastructure for asynchronous file uploads with *Uploadify* can partially be reused for the video uploads. *Uploadify* removes the regular file input field and inserts a button like Flash object into the form to handle the upload. Once the button is pressed, a standard OS file selection dialog is opened. After a file is selected and confirmed, the upload starts automatically, sending the file to a specific URL where a sever side script handles the asynchronous upload. The status of the upload including the total size of the file and the percentage uploaded are displayed at the end of the form, shown in Figure 6-3.

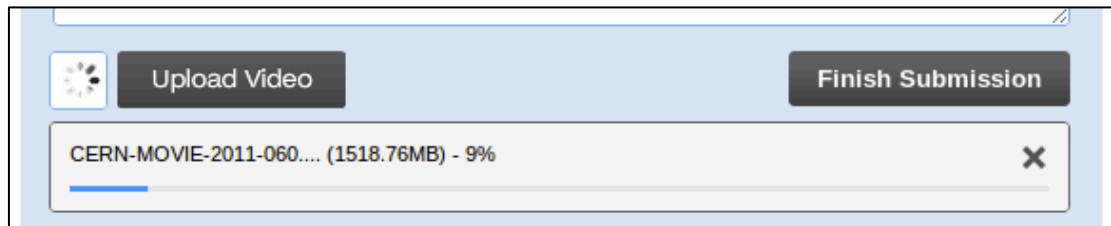


Figure 6-7 A detail of the form shown in figure 6-5, after a video from the local file system was selected and the upload was started.

When the upload is finished, the server script answers back with a HTTP response in JSON that either reports an error to the client or sends back information about the uploaded video.

If an error is reported, the client displays an error message informing the user that the file was not readable by the server, either because it was corrupted or not a video at all, or the server does not support the type of the video.

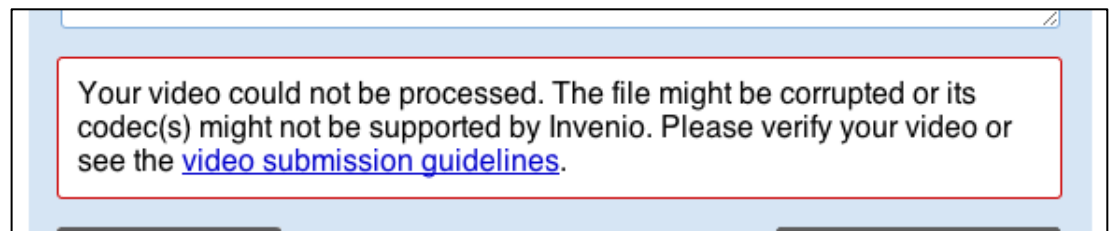


Figure 6-8 Detail of the form after a corrupted, non-video file or video of unsupported type has been uploaded.

If the video upload is successful and the video file is readable, the server will generate a set of preview images and store them on disk. The server will also try to determine the correct aspect ratio of the video and additional metadata information to prefill the title, year, author and description field if the user has not yet filled them. The URL of the images and the aspect ratio and metadata are send back to the client in the JSON response. The client will then request the images from the server and generate an animated slideshow through a custom script. Every 3 seconds, the slideshow displays another image of the video with a blending effect between the images. A sample image is shown in Figure 6-9.

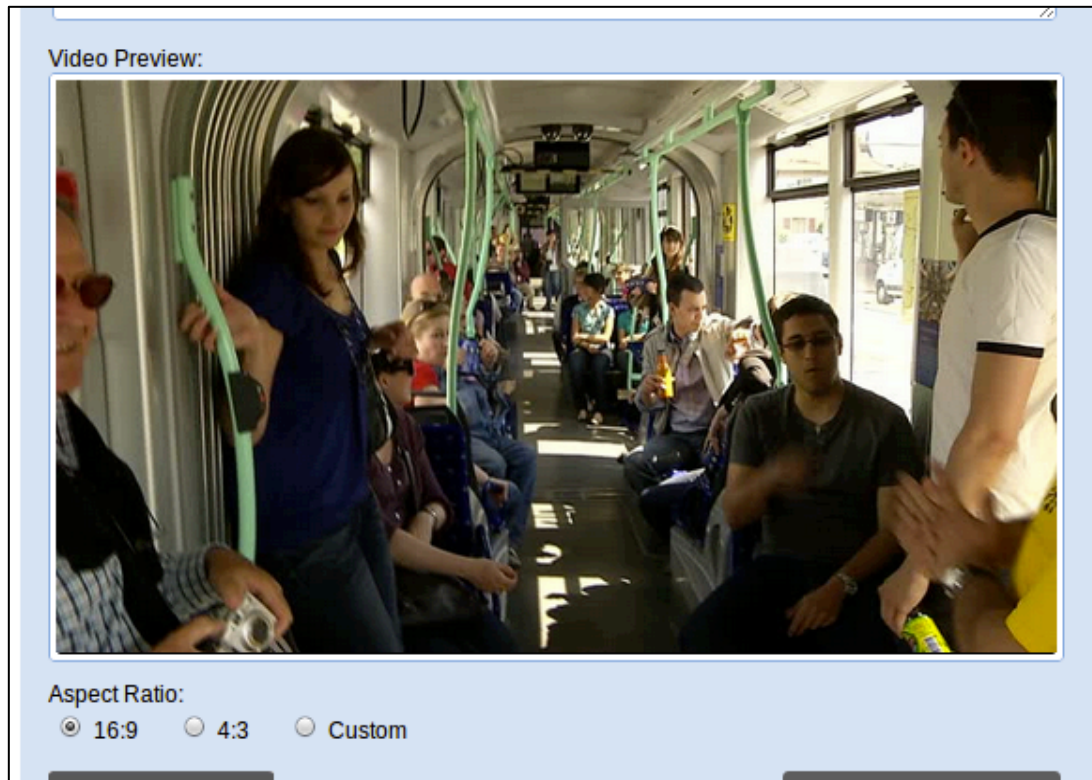


Figure 6-9 Detail of the form shown in figure 6-5 after the video upload was successful. The page shows a slideshow of images generated from the uploaded video. Additionally, the user can alter the aspect ratio of the uploaded video.

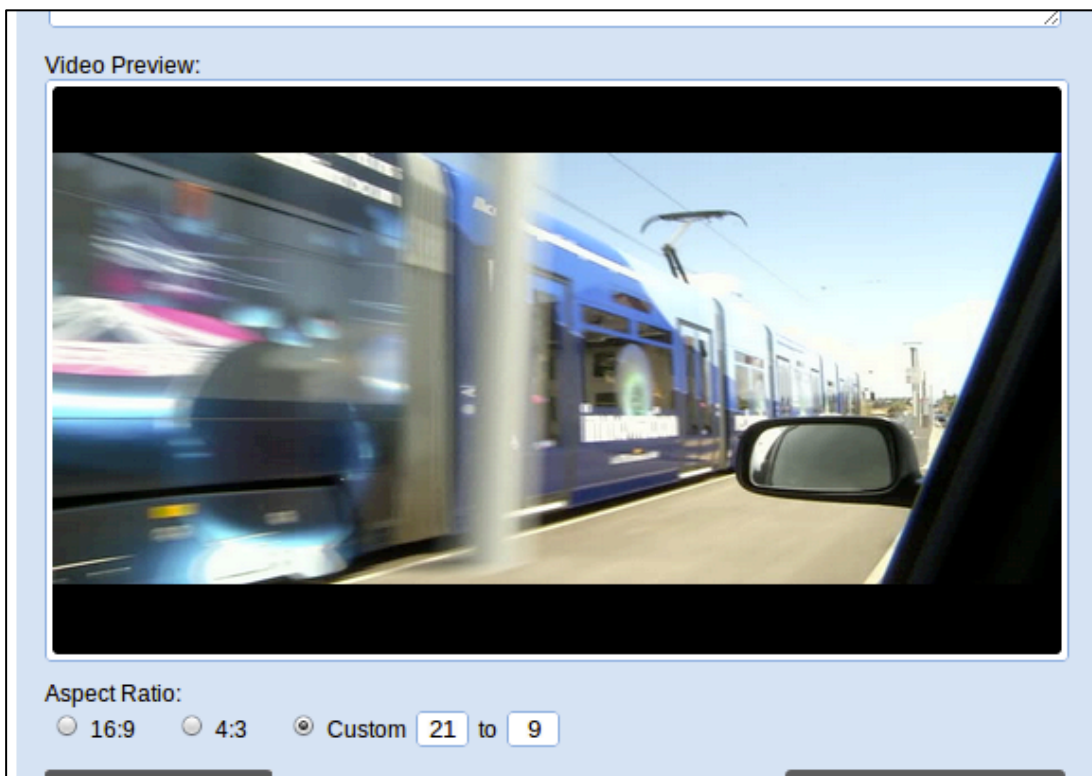


Figure 6-10 The preview images shown in figure 6-5 will dynamically adapt in their appearance if the aspect ratio is changed. The user can also enter a custom ratio.

Within the preview box, the images are first displayed in the aspect ratio that was determined by the server. This information might not be correct because it was either not set in the video metadata at all or incorrect values are stored. The user is able to modify the aspect ratio of the later to be generated video slaves through the aspect ratio fields provided below the video preview. The most common aspect ratios 16:9 and 4:3 are provided as radio buttons (Figure 6-9). But the user is also able to set a custom aspect ratio as shown in Figure 6-10. When the aspect ratio is changed, a JS event is triggered to adapt the images to the new ratio. This is done on the client side, stretching the images with CSS properties.

When the form is finally submitted, the user is taken to a standard WebSubmit confirmation page and the server side processing of the record and the uploaded video is initiated.

6.5.4.2 Asynchronous video upload handler

A new function *upload_video()* is added to the *websubmit_webinterface* module that handles the asynchronous video uploads of the previously described submission form prototype. The URL handler of WebSubmit is updated to call this function and pass the HTTP request to it when the address */submit/upload_video* is called. In reality, this function is the last in a chain of functions handling the request.

As described in chapter 7.4.1, HTTP requests travel through the *webinterface_handler* module that pre-evaluates form data and the uploaded files are stored as a *NamedTemporaryFile* Python object in a local temporary folder in case of multipart forms that include binary file uploads. A *NamedTemporaryFile* only lives in the file system as long as the file handler in Python is kept open. During that time, other code and programs can access the temporary file just like a normal file. The temporary file does not carry the name of the original file, but a uniquely generated one with a 'tmp' prefix. The original name of the file is stored in the modified request that is later passed to other functions. With Python 2.5 it is possible to keep the file alive after the file handler is closed, by setting the *delete* attribute to *false*. Currently, the file handler is not closed until all the functions associated to the URL for handling the request are evaluated.

The *webinterface_handler* module is modified to evaluate the URL before the temporary file is created. If the URL equals */submit/upload_video*, the temporary file will not be stored in the local temporary directory */opt/invenio/var/tmp* but instead in the shared temporary directory */opt/Invenio/var/tmp-shared*, making the file available for other Invenio instances in a multi instance scenario with shared file systems.

When *upload_video()* is finally called, the open file handler of the *NamedTemporaryFile* object is passed in the pre-evaluated HTTP request. The *upload_video()* function first evaluates the GET parameters of the URL that will include the identifier of the current user session, the WebSubmit identifier of the running submission and other information needed to authenticate the user and select the

correct folder within WebSubmits folder structure to store files generated by the function. The parameters are verified and washed to prevent malicious injections. The function will then check if another video has previously been uploaded in the same WebSubmit session. Because a wrong video was uploaded first by the user for example. If so, *upload_video()* removes the old upload from the filesystem and immediately freeing disk space, instead of waiting for the filesystem garbage collector of Invenio. Especially when one thinks about the very large video files produced by the Audio-Visual section, it is very unwanted to have tens to hundreds of gigabytes of dead files in the shared file system.

The function will also check if the file is a readable video, by calling *ffprobe*. As *ffprobe* might be able to read some image and audio formats if uploaded, these formats are excluded by checking the file extension against a list of unsupported formats. If the file is not detected to be a video, the function removes the uploaded file from disk by closing the temporary file handler and returning an HTTP error to the client.

The temporary file handlers *delete* attribute is then set to *false* to keep the file alive after it is closed. The handler is then closed and the file is renamed with a 'video_upload' prefix and unique identifier of the submission. The identifier was checked against in a previous step, when old uploads were deleted. For Python 2.4, which is still supported by Invenio, the *delete* attribute does not exist. To keep the file alive, it is renamed and a dummy is created with the old name, then the file handler is closed. This should be no problem in general, as *upload_video()* is the only function at that point accessing the open file handler.

Instead of copying the video to the WebSubmit storage folder, the path to the renamed file and its original filename are stored in a *filename* and *filepath* file inside the storage folder, preventing unnecessary copy operations of potentially large files, and allowing an immediate feedback for the user after the upload.

The *extract_frames()* function of the *BibEncode* module is then called to generate several preview images of the video, with their URLs to be send back to the user, to verify the upload and adapt the aspect ratio with a visual feedback. The frames are stored inside the submission folder. In the *websubmit_webinterface()* module, there is already a function *getuploadedfile()* to make generated files during a submission accessible for the user in a secure way, through a specific URL.

Additionally, *BibEncode* tries to determine the display aspect ratio of the video and send it back to the user to have it verified. If the aspect ratio is not stored inside the video container, the function will just divide the width of the video in pixels through the height of the video in pixels, thereby sending the storage aspect ratio to the user, which might not be identical with the real aspect ratio to display the video in case of an anamorphic storage.

The URLs of the extracted frames and the aspect ratio are copied into a Python dictionary structure, which is then dumped into a JSON representation using the building JSON module of Python. The JSON representation is then send back as an HTTP response to the users clients.

At this point, any other video metadata could be extracted, evaluated and send back to the client. The function can easily be extended in the future in these regards, as all the necessary functions to access the metadata are available in the BibEncode module. The response dictionary just needs to be extended with new keys, being filled from the metadata dictionary that BibEncode creates.

6.5.4.3 Server side validation and batch job creation

After the user submitted the submission form, WebSubmit offers the possibility to call server-side functions to initiate the record creation and process all the dependencies. Multiple functions can be called sequentially. These functions have to live in the WebSubmit module custom functions folder, within Python files of the same name as the function, and the function implementing a specific API.

The first function being called after the submission is *Video_Validation()*. It is a custom function to validate the video submission on the server-side, before any other action is taken. As explained before, the current WebSubmit module has no built-in way of validating forms except for a primitive validation on the client side, which is only able to check if fields are filled at all. The *Video_Validation()* function is therefore tied to the form prototype and will hopefully no longer be needed in future versions of WebSubmit, which offer integrated support for real form validation both on the client and the server side.

When the form is submitted, all field values are stored in the file system by WebSubmit. Every field value is written to a corresponding file with the fields name as its filename inside the submission directory. As these files are the basis for any future processing, their stored values are evaluated in the *Video_Validation()* function. The function checks for mandatory values being present and restrictions in value type and minimum or maximum value length being fulfilled. However, it does not wash the field values for HTML or JavaScript to protect from Cross-Site-Scripting (XSS) or SQL statements to protect from SQL injections. The SQL injection protection is done at the level of the database module and the BibUpload module, where a behaviour similar to *Prepared Statements* in JAVA is implemented. At the level of HTML and JavaScript, the BibFormat module, based on information stored in the MARCXML, generates the HTML output of a record. BibFormat is able to wash the MARC fields and to remove or escape HTML and JavaScript code.

If the validation finishes with a negative result, a redirect is triggered and the user is brought back to the submission form to correct the input. No other WebSubmit functions are called in this case.

If the validation is successful, a series of standard WebSubmit functions is called to create the record and inject the metadata from the form, as well as informing the user that the video has been successfully received by an onscreen message and an email send.

As a last step, the custom *Video_Processing()* function is called. The *Video_Processing()* function is generalised to apply it to user created WebSubmit workflows for video submissions. When configuring a WebSubmit workflow, the

admin can set parameters to be passed to custom functions. These parameters define the batch-processing template to be used, the file storing the video or containing the path to the video, the file storing the aspect and the file storing the title of the submission. *Video_Processing()* loads a batch template, which stores the batch description in JSON format as a Python dictionary. The parameters or default values are injected into the batch dictionary and then dumped back into a JSON file inside the BibEncode Daemon's directory, where the Daemon and the BibEncode Batch module later ingest it.

6.6 Metadata and MARC integration

Metadata preservation and representation is the core business of Invenio and the digital libraries using it.

For video metadata, one has to discriminate between user-settable metadata and implicit or technical metadata. The differences are explained in chapter 4.6.2.

The user-settable metadata is often inserted during the creation of a digital video by its authors using video production software. In some cases, this type of metadata might match or at least be a subset of the metadata needed to create a record for the video in Invenio and make it available and searchable. We have seen in chapter 4.6.2 that MARC fields for this type of metadata exists in the standard defined by the Library of Congress. Because the user normally fills these fields during a record submission, the advocated video submission prototype retrieved the corresponding metadata from a video file after the upload and presents it to the user in the submissions fields, if no information was yet manually inserted. The metadata is put into the corresponding fields in the MARC of the record through the BibConvert and BibUpload process chain upon submission.

Technical metadata plays a key role in automatic video processing. When video records are to be reprocessed, for example when new video formats for the web arise, the reprocessing system needs to access the technical metadata to determine suitable output parameters like the correct aspect ratio. If a variety of output formats with different target resolutions and bitrates is to be created, the system needs to determine if the once uploaded video is able to sustain the quality of the new formats.

As explained in chapter 4.6.2, the MARC bibliography standard by the Library of Congress lacks any field definition for storing technical metadata of digital video. At most, it defines fields for storing metadata of physical video material, such as analog film or videotapes [53]. Luckily, there is the PBcore metadata standard that was designed especially for digital video material in broadcasting associated institutions. The PBcore concept of *Documents*, *Instantiations* and *EssenceTracks* perfectly matches records related to digital videos in containers with different streams.

When video metadata is to be extracted from videos and appended to records, BibEncode first creates a PBcore representation of the metadata. To do so, BibEncode uses both *ffprobe* and *Mediainfo* to retrieve the video metadata. The

two data structures are then combined using a user configurable micro-language that matches metadata keys from both libraries to PBcore tags. The PBcore XML file can then be appended to a record as a BibDocFile. A subset of the PBcore metadata can actually be translated into MARCXML. For this purpose, a mapping between PBcore tags and MARC fields and subfields is defined. It makes use of the \$950 and \$951 fields in MARC, as these fields are in the custom field range of the Invenio standard fields and currently not user otherwise. Field \$950 stores information about the container format of the video in its subfields. Field \$951 exists multiple times per \$950 field and stores information about each of the streams inside the video container in its subfields. The video bitrate is for example stored in the PBcore tag *instantiationDataRate*. The current unit of measure is bit per second. Only the raw information without a unit is stored in the MARC field \$950e. The unit of measure is lost and further processing retrieving the metadata from MARC relies on the convention to use the most atomic unit. To do the conversion between PBcore XML and MARC XML, the XSL capabilities of BibFormat are used. The *format()* function of the *bibformat_xslt_engine* module ingests a XSL template and an input XML file. A template covering all PBcore Instantiation and EssenceTrack tags is developed. It can further be extended to include more tags. The XML input comes from the *pbcore_metadata()* function of the *bibencode_metadata* module.

6.7 JSON configuration and profiles

For the configuration of BibEncode and for the creation of job descriptions and profiles, JSON is chosen over other data exchange and configuration file standards such as XML and INI.

The first implementation of the encoding profiles was done in a simple XML markup. But even this was not straightforward to parse and to feed the information to the functions responsible for the video transcoding. Based on the Python modules for parsing XML such as *minidom*, a custom parser for the profiles was build that translated from XML to a Python dictionary structure to pass the profile data around. When profiles or job descriptions are altered or dynamically created and to be written back to the file system, a XML constructing function would be needed to translate from Python structures back to XML.

Creating the profiles in JSON and ingesting them into Python code is much more natural. So is the opposite direction when going from Python structures back to JSON. As explained in Chapter 4.6.1, the JSON structure almost perfectly matches the structure of Python dictionaries and the built-in parser translates instantly from JSON to a dictionary in Python.

JSON in BibEncode is used for the creation of batch processing templates and job descriptions, encoding profiles and extraction profiles as well as for the metadata mapper from *ffprobe* and *MediaInfo* metadata to the *PBcore* multimedia metadata XML standard.

A drawback of the JSON standard is that it offers no support for comments like XML or INI does. When providing predefined profiles and templates, comments

would greatly improve the ability of administrators and developers to understand these files when custom workflows are created. When using the standard JSON parser in Python, any type of comment notation within JSON would cause the parser to raise an exception. A wrapper around the parser is therefore developed, allowing C or JavaScript like comments in JSON. The comments are removed before the String containing JSON is passed to the parser. The wrapper keeps the overall structure of the file intact, including the line numbering by inserting blank lines for comment lines. If an error while parsing occurs, the developer can quickly find the problem by the given line number in the parser's exception. As Invenio currently does not fully support Unicode and parsed JSON comes as *Unicode* Strings, the wrapper encodes from *Unicode* to *utf-8*.

6.7.1.1 Encoding and Extraction Profiles

The description of encoding and extraction profiles matches a subset of the parameters available in the *encode_video()* and *extract_frames()* functions of BibEncode.

```

1  // Extraction profiles for BibEncode
2  {
3  // Extracts one representative frame at 5% of the video's duration
4      "POSTERS": {
5          "width": 640,
6          "height": 360,
7          "positions": [5, "00:00:10.00", 20],
8          "extension": "jpg"
9      },
10 // Extracts 10 small thumbnail frames evenly over the video's duration
11     "SMALLTHUMBS": {
12         "size": "160x120",
13         "numberof": 10,
14         "extension": "png"
15     }
16 }
```

Figure 6-11 Example for a file containing extraction profiles in BibEncode with two profiles 'POSTERS' and 'SMALLTHUMBS' and C-style comments for additional explanations.

Figure 6-11 shows an example for a file that contains two profiles for the extraction mode of BibEncode. Within the profiles, all transcoding related parameters of the frame extraction function are configurable. The overall structure is a dictionary that contains keys for every profile. The profiles themselves are then dictionaries again. For the dimensions of the extracted thumbnail image, aspect respecting boundary dimensions can be set by using the 'width' and/or 'height' parameters as shown in line 5 and 6. If an exact dimension should be met, the 'size' parameter can be used as shown in line 12. If a number of frames evenly distributed over the duration of the video is to be extracted, the 'numberof' parameter could be used. If frames at exact positions are to be extracted, the 'positions' parameter must be used. The positions are either given as a percentage of the total duration in integer values like 5 and 20 for a frame at 5% and 20% of the videos duration, or given as exact timecodes like "00:00:10.00" for a frame at exactly 10 seconds from the beginning of the video. The two notations can be mixed within one list of positions. If no image format is given through a file extension in the

output files name, it will be taken from the profiles extension parameter shown in line 8 and 14. Any image format that *ffmpeg* supports is possible, for example *jpeg*, *png* or *bmp*. As there is currently no clearly defined way to influence the image quality of the extracted frames in *ffmpeg*, no quality parameter is present in the *extract_frames()* function or in the extraction profiles.

```

1  // Encoding profiles for BibEncode
2  {
3  // MP4 with H.264 video and AAC audio in HD at 720 lines
4  "MP4_720P": {
5      "videocodec": "libx264",
6      "audiocodec": "libfaac",
7      "videobitrate": "2500k",
8      "audiobitrate": "128k",
9      "width": 1280,
10     "height": 720,
11     "passes": 1,
12     "extension": "mp4",
13     // For the x264 encoder, a H.264 preset must be given
14     "special": "-vpre baseline -ac 2",
15     // Additional metadata to be injected into the video
16     "metadata": {
17         "copyright": "CERN"
18     }
19 }
20 }
```

Figure 6-12 Example of a file containing an encoding profile for BibEncode with one profile 'MP4_720P' defined and C-style comments for additional explanations.

Figure 6-12 shows an example file containing one encoding profile for transcoding to H.264 video with AAC audio. Just like the extraction profiles file in Figure 6-3, the overall structure here is a dictionary with a key for each profile and the profiles being dictionaries themselves again. The 'videocodec' and 'audiocodec' parameters shown in line 5 and 6 correspond to the 'vcodec' and 'acodec' parameters of *ffmpeg*. The codecs specified have to be the codec libraries names, just like they would be used in *ffmpeg* if explicitly defined, and not auto detected for a given video container. The 'audiobitrate' and 'videobitrate' parameters in line 7 and 8 can either be defined as integer like 128000, which stands for 128.000 bit/s, or as Strings like "128k" with a 'k' for kilo at the end. Just like for single frames, the video dimensions can either be defined by the boundary values 'width' and 'height' in shown in line 9 and 10, which keep the aspect ratio of the original video intact, or by the 'size' parameter where dimensions are strictly applied. As BibEncode supports multipass encoding up to two passes with *ffmpeg*, the number of passes to run in line 11 can either be set to 1 or to 2. If output destinations are defined without file extensions, the 'extension' parameter defines the video container format to be used for the encoding. As BibEncode and its profiles should be usable without a deep knowledge of the *ffmpeg* command line parameters, no direct mapping of profile keys to *ffmpeg* parameters was applied. Most of the previously described profile parameters should be sufficient for web-oriented video encoding and most codecs. If more sophisticated parameters for *ffmpeg* are needed, these can be directly passed to the *ffmpeg* CLI by using the 'special' parameter shown in line 14 of Figure 6-12. If the same metadata is to be

injected into the transcoded video for every conversion using the profile, a metadata dictionary can be provided like in line 16.

6.7.1.2 Batch Descriptions

```

1  // Batch Job Template Example for a fresh submission
2  {
3      "isbatch": true,
4      "input": "/opt/invenio/var/tmp/movie02.mp4",
5      "recid": 124,
6      "jobs": [
7          {
8              "mode": "encode",
9              "profile": "WEBM_480P",
10             },
11             {
12                 "mode": "extract",
13                 "profile": "POSTER",
14             }
15         ]
16     }

```

Figure 6-13 A compact batch description with mandatory parameters only.

Batch jobs in BibEncode are created to work directly on records and create all the necessary output for a video submission, while tightly integrating into Invenio's record and file storage infrastructure. The complete back-end workflow to create the necessary output is defined within a batch job description file. The batch job description is a dictionary where the first level of key-value pairs relates to general parameters for all jobs, options directly relating to the record and the handling of the video master. Within the first level, there is the 'jobs' key, which contains a list of jobs to be executed sequentially. Some of the parameters within the job description are mandatory to execute the job others are optional.

As the system is designed to only contain one video in different formats per Invenio record, the batch processing can only contain one input file and record ID. The parameters 'input' and 'recid' are therefore the most important and mandatory parameters. Besides, there must be jobs in the 'jobs' dictionary or nothing would be done during the processing. Figure 6-13 shows a very simple batch description with two jobs: the first for encoding the input to *WebM* and the second one to extract a thumbnail.

The configurability of batch jobs goes far beyond these basic options if needed. As we have seen in chapter 6.2.3 BibEncode integrates tightly into the BibDoc structure to store the records video files and thumbnails. With a minimal batch configuration, the BibDoc specific such as *Docname*, *Format* and *Subformat* are auto discovered based on the filename and other information. All BibDoc related information manually configurable in the batch job description for both the master and the individual slaves. Full control over the BibDoc parameters allows the creation of very sophisticated workflows that include multiple video formats in multiple resolutions for advanced video players embedded into the HTML record representation. Figure 6-6 shows a more advanced configuration with many optional parameters.

The order of parameters inside a dictionary has no influence on the execution of an encoding or extraction job based on that dictionary, as all values are accessed by their keys, while the order of the jobs inside the 'jobs' list of a batch job is strictly followed by the batch processing engine of BibEncode.

Within the list of jobs shown in Figure 6-14, lines 82 to 121, it is possible to use profiles as well as the keys normally set inside a profile directly, as well as a mixed scenario with both a profile given and profile keys. Allowing a maximum of flexibility. The keys inside the job description will override the keys given by the profile in this case. As the batch description might have been created based on a profile that is applied to every submission, jobs might exist with parameters that are not compatible with the input file. Example: A low bitrate video with a DVD like resolution is uploaded, but the batch template includes a job that transcodes to a high definition, high bitrate format. In this case, it would make no sense to execute the job, as the videos quality is not sufficient to deliver HD video streams. The *'assure_quality'* parameter could be set in the general part of the batch description. The batch process will not execute jobs whose quality requirements are not met by the input video. In case that no jobs would be executed because the quality is too low for all of them, jobs with the *'fallback'* parameter set to *true* would be executed to have at least some output. Jobs with the *'enforce'* parameter are executed regardless of the quality assurance.

A special type of Batch Job Description is shown in Figure 6-15: When the *'update_from_master'* parameter is set to *'true'*, the BibEncode Batch Engine tries to find a master video inside the records BibDocs, based on hints given in the Job Description. At least one BibDocFile Comment, BibDocFile Description or BibDocFile Subformat to find the master must be provided. If more than one is provided, all of them have to match in order to determine the video master BibDocFile. To be able to use this feature, it is important to append the master to the records during submission setting the *'add_master'* parameter to *'true'* and setting at least one of the potential hints to a unique identifier.

```

1  // Batch Job Template Example for a fresh submission
2  {
3      // MUST
4      // Defines that this JSON document describes a batch job
5      "isbatch": true,
6
7      // MUST
8      // The input file that is to be processed. Only one input for the whole
9      // batch is possible
10     "input": "/opt/invenio/var/tmp/movie02.mp4",
11
12     // MUST
13     // The recid that will the batch process is going to work on
14     "recid": 124,
15
16     // OPTIONAL
17     // Alters the collection to the specified after the batch job is completed
18     // Makes use of MARCXML and BibUpload
19     "collection": "VIDEO",
20
21     // OPTIONAL
22     // Defines if the master video should be appended to the record
23     "add_master": true,
24
25     // OPTIONAL
26     // Defines the BibDoc name for the master and all slaves
27     // Neces. For the websubmit workflow because we use temporary filenames
28     "bibdoc_master_docname": "DEMOVIDEO",
29
30     // OPTIONAL
31     // Defines BibDoc extension for the master, necessary for the demo
32     // websubmit workflow because we use temporary filenames without extension
33     "bibdoc_master_extension": "mp4",
34
35     // OPTIONAL
36     // Defines the BibDoc comment, description and subformat for the master
37     // Highly recommended to set this for reprocessing existing records
38     "bibdoc_master_comment": "MASTER",
39     "bibdoc_master_description": "MASTER",
40     "bibdoc_master_subformat": "master",
41
42     // OPTIONAL
43     // Removes jobs from the batch that are not suitable for the input video
44     // e.g. high resolution profiles are not suitable for low res material
45     "assure_quality": true,
46
47     // OPTIONAL
48     // Aspect ratio support. If BibEncode cannot detect the aspect, this
49     // value is used as a fallback value for the encoder
50     "aspect": "16:9",
51
52     // OPTIONAL
53     // Sends notifications to the user/admin in case of failure/success
54     "notify_user": "bjorn.oltmanns@cern.ch",
55     "notify_admin": "bjorn.oltmanns@cern.ch",
56
57     // OPTIONAL
58     // The original name of the file uploaded and the title of the submission
59     // Used in the notification email
60     "submission_filename": "movie02.mp4",
61     "submission_title": "A demo video",
62
63

```

```

64 // OPTIONAL
65 // Deletes the input file after the batch process was successful
66 "delete_input": true,
67
68 // OPTIONAL
69 // Only deletes if pattern is in the input name
70 "delete_input_pattern": "video_upload_",
71
72 // OPTIONAL
73 // Creates a BibUpload containing extracted Metadata
74 "add_master_metadata": true,
75
76 // OPTIONAL
77 // Creates a pbcore metadata bibdoc for the master
78 "add_master_pbcore": true,
79
80 // MUST
81 // Dictionary of the jobs to process
82 "jobs": [
83   {
84     // MUST
85     // Mode in which BibEncode should run
86     "mode": "encode",
87
88     // OPTIONAL
89     // Profile to use, here for encoding
90     "profile": "WEBM_480P",
91
92     // OPTIONAL
93     // Bibdoc subformat, comment, description for this specific job
94     // Highly recommended! In case of multiple slaves with the same
95     // extension previously created files would be overwritten!
96     "bibdoc_subformat": "480p",
97
98     // OPTIONAL
99     // Bibdoc comment and description
100    "bibdoc_comment": "WEBM_480P",
101    "bibdoc_description": "WEBM_480P"
102
103    // OPTIONAL
104    // Sets the job as fallback if no job survives the cleaning.
105    // "fallback": true,
106
107    // OPTIONAL
108    // Sets the job as enforced, regardless of quality
109    // "enforce": true
110  },
111  {
112    "mode": "extract",
113    "profile": "POSTER",
114
115    // OPTIONAL
116    // Bibdoc docname for the extracted frame.
117    "bibdoc_docname": "${bibdoc_master_docname}_POSTER",
118    "bibdoc_comment": "POSTER",
119    "bibdoc_description": "POSTER"
120  }
121 ]
122 }

```

Figure 6-14 A complete batch job description in JSON with extensive comments and optional parameters. The first part contains general options for all sub jobs and the video master. The second part contains the sub job of the batch process.

```

1  // Batch Job Example for adding new formats to an existing record
2  {
3      "isbatch": true,
4
5      // MUST
6      // Defines that the new formats are to be created based on a master
7      // video that is already appended to the record's BibDocs
8      "update_from_master": true,
9
10     // MUST (ONE OR MORE)
11     // BibEncode Batch will try to match these to find the master.
12     // If none of them is provide, it will raise an exception
13     "bibdoc_master_comment": "MASTER",
14     "bibdoc_master_description": "MASTER",
15     "bibdoc_master_subformat": "master",
16
17     "recid": 121,
18     "assure_quality": true,
19     "jobs": [
20         {
21             "mode": "encode",
22             "profile": "THEORA_480P",
23             "bibdoc_subformat": "480p",
24             "bibdoc_comment": "THEORA_480P",
25             "bibdoc_description": "THEORA_480P"
26         }
27     ]
28 }

```

Figure 6-15 A batch description for updating a record with new video formats transcoded from a master video that is already appended to the record.

6.8 Handling of video aspect ratio

An important part of the automatic processing of videos that was not foreseen during the analysis phase of the project was the handling of different aspect ratios. Only later when prototypes of the system were tested with various video material, some of the material was distorted during the processing. The issue was caused by faulty assumptions on the aspect ratio of the videos.

When talking about video aspect ratios, one has to discriminate between three different types of aspect ratios to refer to: the Storage Aspect Ratio (SAR), the Display Aspect Ratio (DAR) and the Pixel Aspect Ratio (PAR).

The SAR is implicitly given by the quotient of the width and height in pixels of the frames stored in a video stream. When *ffmpeg* is used to transcode from one video format to another, the SAR is set by the width and height inside the value of the 'size' parameter (e.g. '-s 480x360'). The stored dimensions of the frames might not be the ones that should be presented to a user watching the video. The video might have a stored frame size of '480x360' pixels, but it should be displayed to the user as '640x360' pixels and stretched to a wider dimension. The discrepancy between the stored frame size and the real video dimensions is known as Anamorphic Video. It allows a widescreen video to be stored in smaller dimension in bandwidth or total storage space constrained situations, because the smaller frame size requires less bits per frame to be stored. It is for example

applied on DVDs where the total amount of space is limited by the physical media or in the CERN Audio-Visual section, where the bandwidth of DVCPRO HD recordings is limited by the magnetic tapes in use.

The best way to store videos for the web is in a square pixel format, where the aspect ratio of a single pixel is 1.0 and the DAR and SAR are identical. The video can be fed to any browser or player with no need to rely on aspect ratio detection. With the uploaded video having possible pixel aspect ratios unequal 1.0 and imperfect detection algorithms, the user is able to select the appropriate aspect ratio during a submission, supported by the best auto-detection the algorithms can offer. Afterwards, the automatic processing is able to set the correct dimensions for transcoded videos in square pixels. When new video formats arise in the future, and collections of videos are to be transcoded to these formats automatically, access to information such as the aspect ratio is needed again to achieve optimal results. Chapter 6.6 explained that the aspect ratio is stored together with other technical metadata in the MARCXML of the video record.

After it was first only possible to set a hard value for the dimensions of transcoded videos and extracted frames, basically all functions of BibEncode that accepted frame dimensions now also accept soft boundary dimensions and aspect ratios. These functions then call the *determine_resolution_preserving_aspect()* function of the *bibencode_encode* module to determine the correct dimensions. This function will try to retrieve the correct aspect ratio from the video metadata and falls back on information given by the user. If both fail, it calculates the aspect ratio on its own based on the original frame dimensions. It then takes the boundary dimensions at fits the new frame size into these, preserving the aspect.

6.9 Video presentation and player

As we have seen before, the submission frontend and the background and processing modules are put in place to fulfil the jobs of the Ingestion, Processing and Storage Systems. The Presentation System is completely independent from these Systems and is never called during the ingestion and processing phase. But the processing system needs be configured to produce all the content for the presentation of video records in the right way. The idea is to have well defined content creation according to both the needs of preservation and a user centric video presentation.

In general, the BibFormat module does the representation of records and their metadata in Invenio. BibFormat templates contain a mixture of HTML code and BibFormat Element declarations. BibFormat Elements are Python code files that return string values or HTML code fragments. The output of the Elements can in principal come from everywhere, but is most often received and processed from the stored MARC metadata and BibDocFiles appended to the record. Within the Elements, one can make use of any available Python or Invenio module, giving the Representation System unlimited powers. As no real templating system is used, the developer has to make sure that the code remains maintainable.

The task to create the video representation was as following:

- Create a player interface based on the Use-Cases discussed in chapter 5.2.
- Create processing templates able to feed the necessary data to the player.
- Implement the video player using BibFormat Templates and Elements.

6.9.1 Conceptual and Visual Design of the Video Player

A wireframe mock-up of the proposed video record representation is shown in Figure 6-16. The design is inspired by video platforms such as YouTube and Vimeo. The idea is to show that Invenio, with the additions developed during the course of this project, is ready to be used for video platform like user experiences.

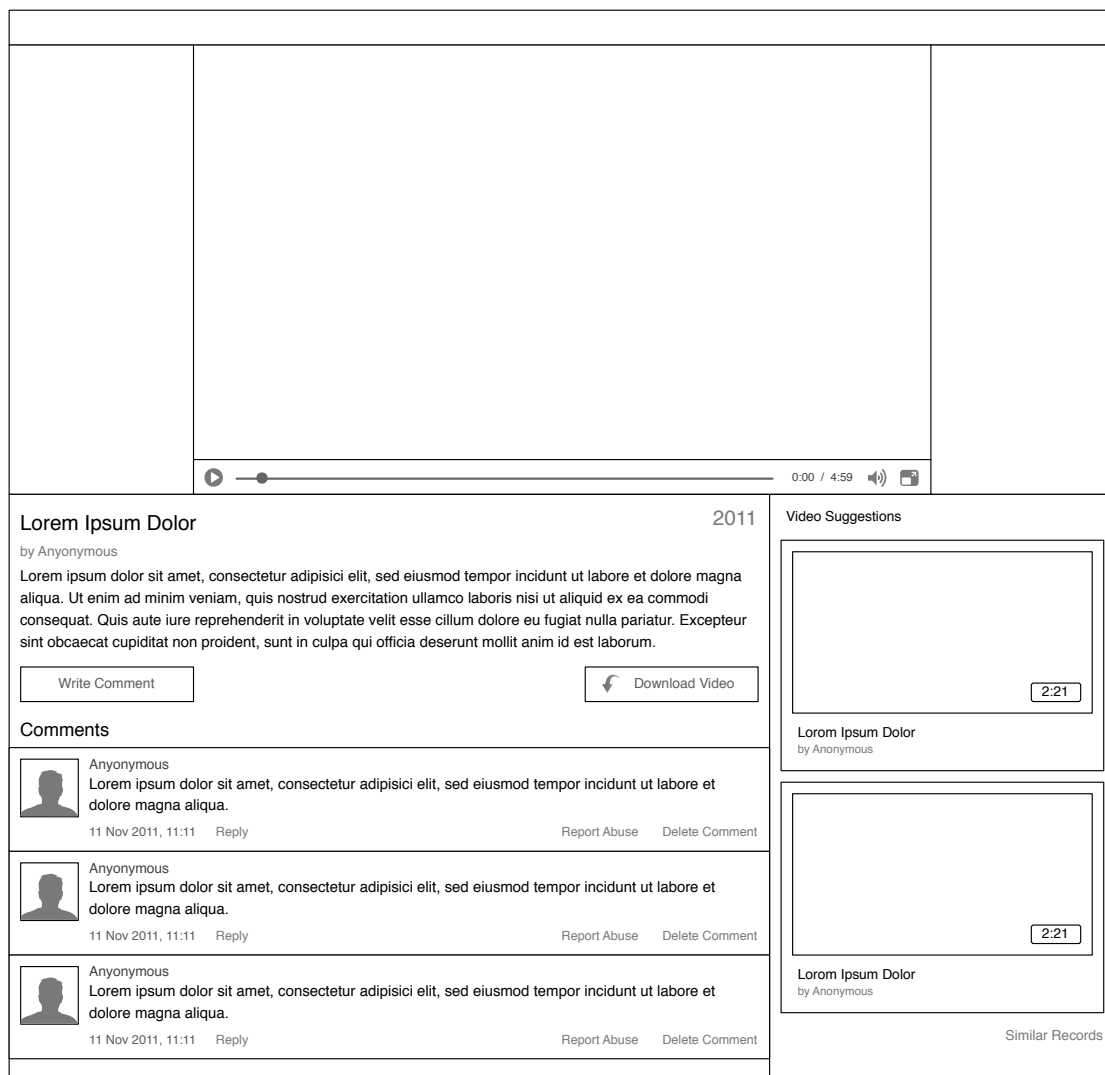


Figure 6-16 Wireframe for a YouTube like video player in Invenio.

In the uppermost part of the mock-up shown in Figure 6.16, the video is played within a media player like element with standard video controls like a seek bar, play-pause buttons, volume controls, duration display and fullscreen button. Below the player, the design splits into two columns. The left column hosts the intellectual metadata of the video record such as the title, the author, the year of production and the abstract or description. Below this content, there are buttons

to download the video file and to write a comment on the video. Below the buttons, the column continues with the comments that have been posted on the video. The right column shows preview elements of recommended or similar videos to the user. The preview element contains a thumbnail of the video with its duration, the title and author of the video.

The screenshot displays the CERN Document Server interface for a video record titled "The MoEDAL Experiment: The Monopole and Exotics Detector At the LHC". The interface includes a navigation bar with links like "Search", "Submit", "Help", and "Your CDS". Below the navigation bar, there are tabs for "Information", "Discussion", and "Files". The main content area features a large video player with a play button and a progress bar. To the right of the video player, there is a sidebar with "See also:" recommendations, including "Energie pour le LHC et le CERN", "Générique LHC first physics", "Clip CERN Director General Rolf Heuer looks forward to first physics at the LHC", "LHC News", "Interview CERN Director General Rolf Heuer looks forward to first physics at the LHC", "LHC First Physics : CERN Press Conference, March the 30th 2010", and "Le Tunnel du LHC". Below the video player, there is a "Download Movie" section with options for "Mp4:", "Flash:", and "Windows Media:", each with "High" and "Medium" quality choices. A "Download high-res version" link is also present. At the bottom, there are links for "Add to personal basket", "Export as BibTeX, MARC, MARCXML, DC, EndNote, NLM, RefWorks", and "Similar records". The footer contains information about the CERN Document Server, its version, and the languages it is available in.

Figure 6-17 Presentation of a video record in CDS. The video sources and thumbnail images are produced and hosted on the Media Archive server.

Comparing the wireframe to the video representation currently found on CDS for the videos of the Media Archive (Figure 6 -17), the new layout is reduced drastically in the number of elements and information shown. The layout structures the information and makes it much easier for the user to scan the content of the video record. It removes a lot of the visual noise and clutter present in the CDS representation. The download element, which takes up an unnecessary huge

amount of space on CDS, is built into a single button that shows an in-page popup menu when pressed. The visual representation of the popup is shown in Figure 6-20. The video suggestion is not meant to be scrollable, compared to CDS. Frame-like in-page scrolling is never flawless and should be avoided, especially if the page itself is scrollable. As enough vertical space is available, extending the pages length is not a problem.

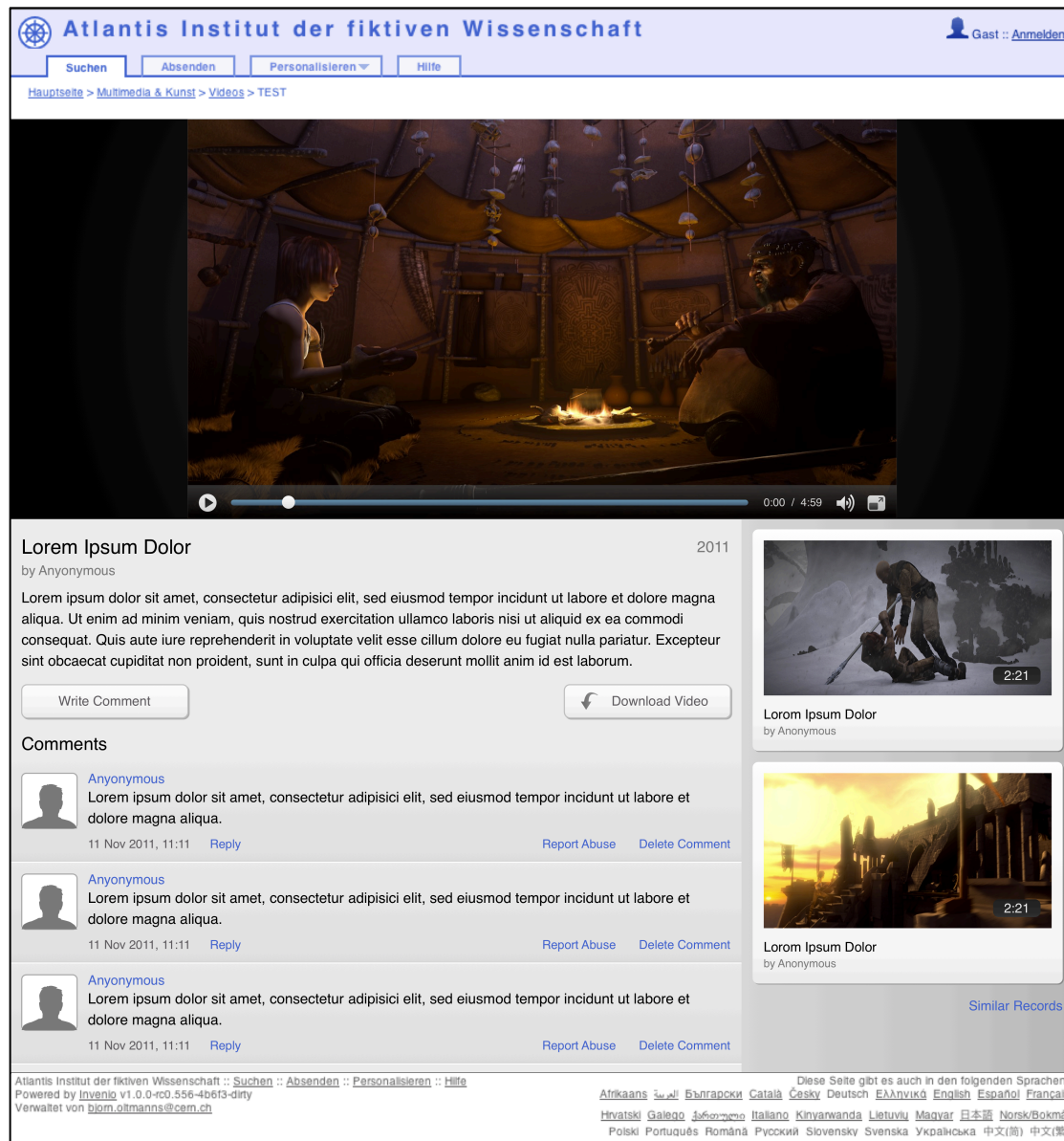


Figure 6-18 Visual design mock-up for a video You-Tube like video player in Invenio.


A visual design for the video record is created based on the wireframe. Figure 6-18 shows the visual design. Except minor changes, the layout of the visual design remains the same, compared to the wireframe. In addition, the standard Invenio header and footer are added. The border normally shown around a record representation with tabs on top to switch between description, discussion, files and other options related to the record is completely removed. The tabs can be seen in the upper part of Figure 6-17 for the CDS videos. The discussion tabs content is partly included in the record representations comments section. It can be

reached by selecting one of the option links within a comment box or the ‘show all’ link below. The file tab normally shows the BibDocFiles associated to a record. In the case of a video record, it is not necessary to make this tab available to the video consumer, as the appropriate files are automatically presented in the video player and via the download button. The colour scheme of the visual design is kept in grey with a strong contrast in form of a black bar that surrounds the actual video player. The black bar fills the complete horizontal space of the page and lets the video stand out against the grey and white background, making it easier for the user to focus on the video. It is easily visible in the screenshot of CDS in Figure 6-17 that the elements surrounding the video player create a distractive moment.

Especially the video preview filmstrip presented above all the other elements begs for attention, and its function is not clearly defined for the user. It might even be confused with the recommendation frame, which has a similar appearance, but positioned vertically. The function of the filmstrip is questionable at all, because the video record is already opened when the strip is presented. The proposed design for the Invenio video player does not carry such an element. The left and right column make subtle use of gradient to visually separate the individual comments and create an affordance as clickable elements for the comment and download button. The fonts colours are adapted in contrast to their importance for the user. Less important text elements are presented in grey shades.


By its aesthetically simple nature, the visual layout and design is easily portable into a HTML and CSS representation. Using the visual capabilities of CSS3, like rounded borders, gradients and shadows, the design can be completely recreated without any image files, except for the icons of the video player. The design will gracefully fall back to CSS2 if the browser does not support certain CSS3 features.

Figure 6-19 shows the actual appearance of a video record with real content coming from the metadata and comment database and videos and images, created during the processing and stored in BibDocFiles. The real appearance is not far from the visual design. One can see that the appearance of the video player is slightly different to the visual design. The comment section shows a comment reply with a nested markup.


Atlantis Institute of Fictive Science
admin :: [logout](#)

[Search](#)
[Submit](#)
[Personalize ▾](#)
[Help](#)
[Administration ▾](#)

[Home](#) > [Multimedia & Arts](#) > [Videos](#) > Group Session Demo Video




Group Session Demo Video

by [Bjorn Oltmanns](#) 2011

This is a Demo Video for the Group Session. Here are some keywords for the ranking: Bjorn, Group, Session, Park Joy

[Download Video](#)

3 Comments




[admin](#)

Pulvinar porttitor, dictumst proin augue ultricies, ut magna enim pellentesque ac, vut, facilisis in urna turpis. Proin egestas. Mattis in integer, odio porttitor odio? Porttitor, integer tincidunt, aenean montes enim nec egestas! Dis, habitasse tincidunt phasellus. Porttitor adipiscing scelerisque, a purus sed? Duis purus tempor facilisis magna dignissim odio mattis.

Non a cras dolor urna, urna ut ac mid, velit, mid massa dis scelerisque dapibus integer quis sed rhoncus, aenean amet vut dis augue augue! Sagittis pid turpis augue, pulvinar lundium enim lundium. Augue et pellentesque mauris in tempor! Pulvinar adipiscing? Sed tempor. Sit, aliquam, elit sit vel etiam.


13 Aug 2011, 17:27 [Reply](#) [Report abuse](#)



[admin](#)

Non a cras dolor urna, urna ut ac mid, velit, mid massa dis scelerisque dapibus integer quis sed rhoncus, aenean amet vut dis augue augue! Sagittis pid turpis augue, pulvinar lundium enim lundium. Augue et pellentesque mauris in tempor! Pulvinar adipiscing? Sed tempor. Sit, aliquam, elit sit vel etiam.

13 Aug 2011, 17:28 [Reply](#) [Report abuse](#)



[admin](#)

admin wrote on 13 Aug 2011, 17:28:


Non a cras dolor urna, urna ut ac mid, velit, mid massa dis scelerisque dapibus integer quis sed rhoncus, aenean amet vut dis augue augue! Sagittis pid turpis augue, pulvinar lundium enim lundium. Augue et pellentesque mauris in tempor! Pulvinar adipiscing? Sed tempor. Sit, aliquam, elit sit vel etiam.

Pulvinar porttitor, dictumst proin augue ultricies, ut magna enim pellentesque ac, vut, facilisis in urna turpis. Proin egestas. Mattis in integer, odio porttitor odio? Porttitor, integer tincidunt, aenean montes enim nec egestas!

13 Aug 2011, 17:28 [Reply](#) [Report abuse](#)

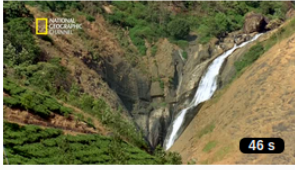
[View all 3 comments](#)

[Write a comment](#)




Group Meeting Sample Video
by Bjorn Oltmanns

1 m



Another Group Session Demo Video
by Bjorn Oltmanns

46 s



Group Session Demo Video
by Bjorn Oltmanns

10 s

Atlantis Institute of Fictive Science :: [Search](#) :: [Submit](#) :: [Personalize](#) :: [Help](#)
 Powered by [Invenio](#) v1.0.0-rc0.6604981d0
 Maintained by bjorn.oltmanns@cem.ch

This site is also available in the following languages:
 Afrikaans العربية Azərbaycanca Català Česky Deutsch Ελληνικά English Español
 Français Hrvatski Galego Հայերեն Italiano Kiswahili Lietuvių Magyar 日本語
 Norsk/Bokmål Polski Português Română Русский Slovenščina Svenska Українська 中文(簡) 中文(繁)

Figure 6-19 Actual appearance of the video player after its implementation.

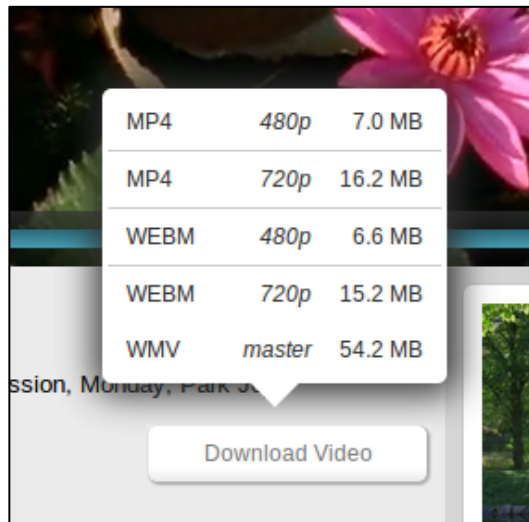


Figure 6-20 Actual appearance of the download and popup element after its implementation in HTML, CSS and JS.

6.9.2 Technical Aspects of the Video Presentation

To generate the representation seen in Figure 6-19, several new BibFormat elements and a new BibFormat template are introduced. For the actual video player, the HTML5 video library *MediaElement* is integrated. The comments are based on the WebComment module of Invenio. The video suggestions rely on the ranking algorithms provided by the search module of Invenio.

A new BibFormat template named *Video_Platform_HTML_detailed.bft* hosts the basic HTML skeleton for the video record. It includes several new BibFormat Elements and instantiates the *MediaElement* video player.

- *bfe_video_platform_downloads.py* – Creates the download button and the popup.
- *bfe_video_platform_sources.py* – Creates the video sources and the necessary technical metadata for the video player.
- *bfe_video_platform_suggestions.py* – Creates a list of video suggestions with thumbnails. Makes use of Invenios search and ranking algorithms.

These files reside inside the '*lib/elements*' folders of the BibFormat module

A download and installation script for the *MediaElement* player is added to the Makefiles of Invenio. The installation is done via '*make install-mediaelement*' from the command line in the Invenio source directory. The default Apache host configuration files are altered to include a path for the *MediaElement* files.

The JavaScript and CSS sources for the video record presentation are hosted in the '*www*' folder of the BibEncode module. They are required for all of the new BibFormat elements and injected into the HTML through the new BibFormat template.

6.9.2.1 Video Downloads

bfe_video_platform_downloads.py retrieves the list of BibDocFiles attached to the record. It searches for files having a video extension such as *mp4*, *webm*, *ogv*, *avi* or *mov*. If one of these signatures is found, the BibDocFile URL together with the video extension, Subformat and file size are added to a new list. The list elements are then encapsulated into HTML and returned to the BibFormat template. The HTML code is portable and its appearance is customisable through CSS. To generate the appearance and behaviour of the popup and the button for the demo installation, *video_platform_record.css* and *video_platform_record.js* from the BibEncode module are required.

The user is able to click the link to the video file on the records webpage, but as the modern browsers are able to play some of the video formats, it might just open a new tab or window to play the video behind the link. This is due to the fact that BibDocFiles are always delivered with the files original Mimetype and the HTTP header field '*Content-Disposition*' set to '*inline*'.

There are two possible solutions for this problem:

- The Mimetype of the video file could be set to '*application/octet-stream*' for the purpose of downloading. As the browser cannot know how to handle a file with this Mimetype, it will either directly start a download or ask the user for assistance on some Operation Systems.
- The '*Content-Distribution*' header field is set to '*attachment*'. This will force the browser to start a download, rather than trying to open or play the file itself.

In both cases, a signal would be needed for the webserver to switch from the regular content delivery to the download. As the first solution might cause inconveniences for the user, the second one is preferred.

The file streaming functions of the BibDocFile module and the request handlers of WebSubmit are altered to serve downloads as '*attachment*'. A new GET parameter '*download*' is introduced for BibDocFile URLs. The download URL in the *bfe_video_platform_downloads.py* Element is constructed with the '*download*' parameter set to 1. In absence of the parameter or any other case, the BibDocFile is delivered normally.

6.9.2.2 HTML5 Video Sources

To make the videos attached to the record available for streaming, the HTML5 video player needs to be supplied with video sources. Normally, these sources are directly inserted into the HTML5 video tag as '*<source/>*' elements. The video player follows the order of the sources and plays the first source it is able to play with its available codecs. If one wants to supply the player with different qualities or resolutions of a video, this technique is not sufficient. Instead, one either needs to create different static pages with different sources for different versions of the video, or needs to dynamically exchange the video sources via JavaScript

and reload the player. But then all the sources and their attributes need to be available for the script.

Similar to the download element, *bfe_video_platform_sources.py* generates a HTML select element that contains all available video formats for streaming sorted by their quality or resolution as HTML option elements. If for example four video files are available, two MP4 and two WebM versions, each with a resolution of 1280x720 or 854x480px pixels, two option elements are created. One is created for 720p and the other for 480p videos. The option elements encapsulate the information of the video files in HTML5 data attributes. Attributes starting with 'data-' are considered as injected information from the backend system and perfectly valid in HTML5. This allows developers to pass data in a convenient way from the backend to a JavaScript based frontend. The data attributes store information about the URL of the different video formats, the Mimetype of these formats, the resolution of the video and the URL of a thumbnail image. An example of the described option and select elements is shown in Figure.

The JavaScript functions provided in *video_platform_record.js* will hook into the select element, built video sources from the data attributes and dynamically exchange the video sources when the user selects a different option. In the record representation for the demo site, the select element is even moved into the control elements of the video player. This allows the user to easily learn the functionality based on potential experience with YouTube.

```

1  <select id="mejs-resolution">
2    <option data-src-webm="/record/175/files/video.webm?subformat=480p"
3      data-type-webm='video/webm; codecs="vp8, vorbis"'
4      data-src-mp4="/record/175/files/video.mp4?subformat=480p"
5      data-type-mp4='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'
6      data-poster="/record/175/files/poster.jpeg"
7      data-video-width="854px"
8      data-video-height="480px">
9      480p
10  </option>
11  <option data-src webm="/record/175/files/video.webm?subformat=720p"
12    data-type-webm='video/webm; codecs="vp8, vorbis"'
13    data-src-mp4="/record/175/files/video.mp4?subformat=720p"
14    data-type-mp4='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'
15    data-poster="/record/175/files/poster.jpeg"
16    data-video-width="1280px"
17    data-video-height="720px">
18    720p
19  </option>
20 </select>

```

Figure 6-21 Option and Select Elements created by the BibFormat video record template to feed the HTML5 video player with video sources.

6.9.2.3 Video Suggestions

The *bfe_video_platform_suggestions.py* Element uses the search engine and ranking modules of Invenio to create a list of video suggestions or recommendations, based on the metadata of the video that is currently watched. Based on a given collection, the search engine is called to receive the list of record IDs within this

collection. For the demo site, it is the ‘Videos’ collection. The list of IDs is then passed to the ranking function together with the record ID of the record currently visited. The functions ranking method is set to *word similarity*. The result set of the ranking method contains two tuples: one contains a list of record IDs with similar content and the other one contains a ranking grade for each of the record IDs from 0 up to 100. The video suggestion BibFormat Element can be called with a threshold for the ranking grade and a maximum number of suggestions to show. In addition, the results can be shuffled. Based on these parameters, a sample set from the ranking results is taken and the record URL, title, authors, duration and thumbnail URL for each of the records are retrieved. The data is inserted into a HTML skeleton and returned to the BibFormat template.

6.9.2.4 Video Record Comments

To display the comments attached to the video record, the already existing *bfe_comments.py* Element is used. This Element normally calls a function of the WebComment module to display comments as well as reviews attached to a record. The element and the corresponding function in WebComment are updated to allow the display of comments only. At this point, the problem still was that the WebComment templates were infected with HTML ‘*style*’ attribute and had no CSS classes at all. This prevented the customisation and styling of the comment output to achieve the design shown in Figure 6-10. To resolve this problem, the WebComment templates are completely refactored. All ‘*style*’ attributes are removed and meaningful CSS classes are applied to every relevant element. The overall HTML structure of the template is optimised. The style definitions are moved to the Invenio CSS file and tweaked to achieve the same look as before in the Discussion tab of standard records. The refactoring now allows the complete customisation of the comments element and the desired look for the video record is achieved. The custom CSS of *video_platform_record.js* simply overrides the default style. This technique can now be applied to any future record type, where comments are directly needed inside the record representation.

6.9.2.5 Video Player

Silvia Pfeiffer [57] has shown that the appearance of the default HTML5 video player differs greatly between different browsers. If a consistent look across browsers is desired, one needs to implement a custom player interface with HTML, CSS and JavaScript. The default controls can then be deactivated. As described in chapter 4.1.2, the video element offers a rich JS API to handle all necessary commands.

There are fortunately JavaScript libraries available to handle this task and enable easy styling of the HTML5 video player. Besides, these libraries offer additional features such as fullscreen video playback and fallback mechanisms to Adobe Flash and other browser plugins.

VideoJS [58] is a HTML5 video player that allows the easy styling of the video controls through CSS. The player is able to fall back to Adobe Flash if HTML5 video is not available, but then requires an external Flash Video Player. This means that the video API and the visual appearance of the player will not remain con-

sistent after the fallback. The player additionally offers a fullscreen feature. *VideoJS* is released as open-source code under the LGPL license.

Projekktor [59] is a HTML5 video player with Adobe Flash fallback support. It aims for a consistent API and appearance in case the fallback is used. The player can be easily styled with custom HTML templates and CSS styles. In addition to the default player feature it offers fullscreen and advertisement support as well as playlist. *Projekktor* is open-source and licensed under GPLv3.

Similar to *Projekktor*, **MediaElement** [60] is a video player that enables a consistent JavaScript API and visual appearance across HTML5, Adobe Flash and Microsoft Silverlight video. HTML5 video can thereby fall back to Flash or Silverlight if H.264 or WMV files are provided. Out of the box, it comes with fullscreen, video loop and caption features. *MediaElement* offers a Plugin API to extend the players functionality. The code is open-source and licences under GPLv2.

Fallback capabilities and a consistent appearance and API are the most important characteristics for choosing a video player. Both *Projekktor* and *MediaElement* fulfil these criteria. The two players offer slightly different features out of the box. The additional support for Silverlight in *MediaElement* is welcome, because WMV files are still produced at CERN. Video captions have been a long time discussion for videos on CDS and are therefore welcome as well. *MediaElement* seems to be more open for plugin like extensions and features like advertisement overlays can be easily realised. *MediaElement* is therefore chosen over *Projekktor*.

Two custom plugins for *MediaElement* are developed to fulfil the Use-Case of video quality or resolution selection. The plugins reside in *video_platform_record.js* of the BibEncode module. The first plugin moves the resolution select box described in chapter 6.5.2.2 inside the video controls and makes it behave like one of them. For example by not hiding the control bar while the select element is used. The other plugin is a modified version of the fullscreen feature coming with *MediaElement*. It handles the correct calculation of the player dimensions if the video source is changed during fullscreen playback.

The appearance of the player is not altered as its out-of-the-box appearance fits the proposed visual design very well. The minor differences can be seen if Figure 6-18 and 6-19 are directly compared.

To instantiate the *MediaElement* player, a regular HTML5 video element with a unique ID is created inside the BibFormat template. The element is then called from JS code with the *MediaElement* constructor. The constructor relies on the existence of video sources inside the video element, meaning that the Sources Element described in chapter 6.5.2.2 needs to be present. For a very simple Use-Case, where only one set of video formats in one quality is available, a single BibFormat element could be created that directly includes the video sources and instantiates the *MediaElement* player. All the code to do so is available in the current iteration. A BibFormat element that just creates video sources for one resolution is still available from an earlier iteration in the *bfe_video_sources.py* file.

Invenio is configured with *mod_xsendfile* to enable progressive streaming with Byte Range Requests for seeking within the HTML5 video content.

6.10 Multi-Node Encoding With BibEncode

One important consideration for running video encoding within the Invenio software is the CPU hunger of the video encoders. When running conversions, encoders normally eat up all the CPU time they can get, often utilizing multiple CPU cores through multiple threads. This will in fact slow down all the other processes running on the same machine. If video encodings and the Webserver are running on the same machine, the web tier will see a severe performance impact. This is unacceptable. One aspect of the project is therefore the decoupling of web-tier and encoding, meaning basically that the video encoding has to run on a different machine than the webserver. As introduced in chapter 5.2, this is the current way of transcoding videos at CERN. The CDS server holds the record with the videos metadata and does no video processing at all, while the Media Archive does the heavy transcoding jobs on dedicated encoding server infrastructure.

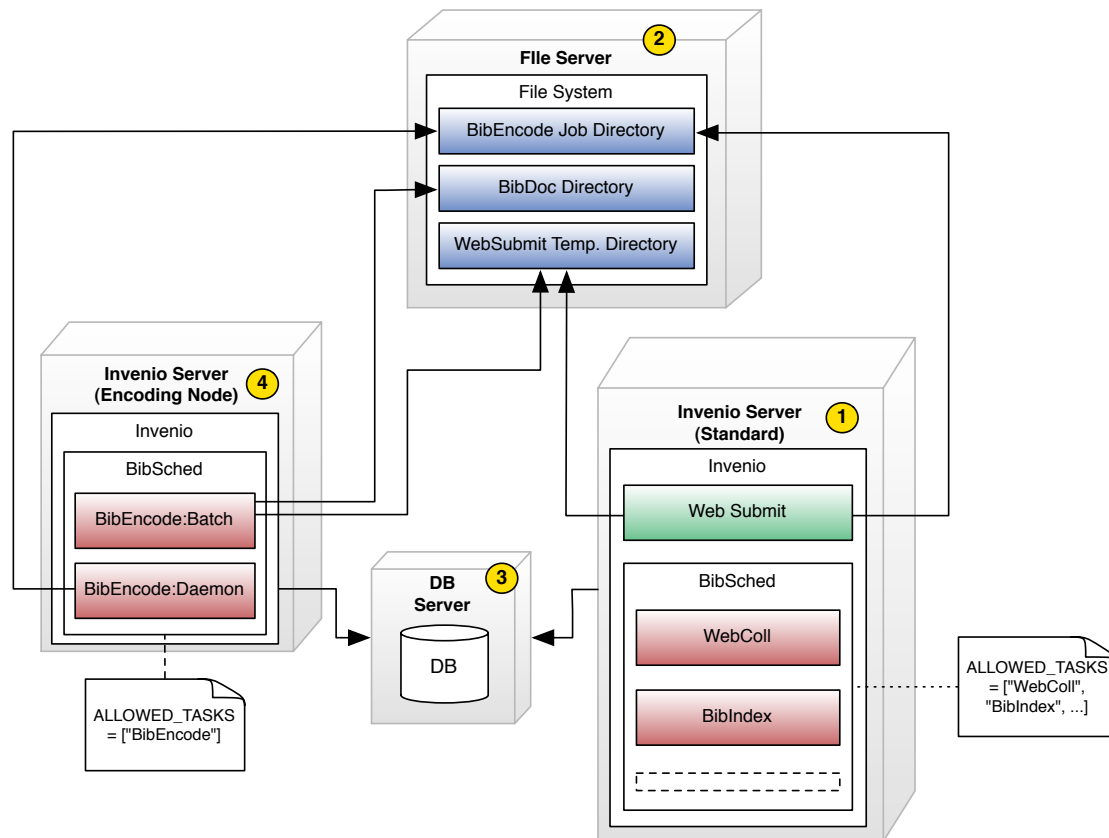


Figure 6-22: Multiple Invenio instances working together. (1) The main server, serving HTTP requests through Apache such as record submissions in *WebSubmit* and running *BibSched* for bibliographic tasks in the background. (2) A files server, serving the *BibDocs* and other files that are commonly used by all the instances. (3) A database server, containing the shared database between all hosts. (4) The encoding server, not running Apache, only running *BibSched* set to only execute *BibEncode* tasks. Both (1) and (4) are mounting some shared folders of (2) into their Invenio directory. Both share the same database (3), which means that all the instances of *BibSched* (1 + 4) are working on the same task queue.

6.10.1 Modifications to BibSched

Currently, there is no known case where BibSched is running multiple times on the same Invenio installation. For Inspire [61] load balancing, Invenio is running on multiple webservers, sharing file system and database, but only one instance is running BibSched.

One concept is to have BibSched running on multiple machines, but accessing the same database that holds the task queue. Every BibSched instance watches the queue and tries to start tasks that are marked as *WAITING*. The type of tasks BibSched instance can start might be limited by an instance specific configuration. The configuration could for example limit the executable tasks to BibEncode ones, meaning that the specific instance of BibSched can only encode this type of tasks, making it to a kind of encoding-only node. This kind of concept makes the different BibSched instances kind of independent. Every instance just processes what it is able to process. There is no master instance that controls the execution centrally or is scheduling tasks for a specific instance.

The other concept of multinode processing is to actively schedule tasks for a specific node or host. A master instance would be used to schedule tasks for a specific instance. Every instance can only execute the tasks that are actively assigned to it. All the instances could still use the same database, but it holds additional information about the instance that should execute the tasks, for example the host-name of the instance to identify it. The BibSched database table that holds the task queue already has a host column in the current version of Invenio, but this column is never used and its usage is not specified.

A combination of both concepts is also thinkable, where nodes normally follow the best effort principle of the first concept, but tasks can also be scheduled for a specific node. The author follows the first concept, because the development of a central scheduling infrastructure would exceed the available time of the project.

To determine the machine running a specific task in a Multinode scenario, a *host column* is introduced to the BibSched visual task manager. Because the currently needed space in characters per line was already exceeding the 80 characters of a classic terminal in some cases, the limitation was completely lifted. Every modern terminal should be able to adapt to the screen resolution. Even with a low resolution of 1024px wide, a terminal window should be able to display 140 characters per line in a very readable 12pt monospaced font. As the *host column* stays empty until the task is actually running, it was placed before the *progress column*, with a similar behaviour, as the seconds last column. This prevents gaps in the lines and improves readability ¹.

¹ The progress column was later optimised by another developer based on the recommendation of the author to include information on the tasks parameters before its execution. The order of the columns should therefore be re-evaluated in the future.

Rows that are not executable on the local instance of BibSched or that are running on other Invenio instances are greyed out using terminal colours, just like the other coloured elements in the visual task manager.

A set of modifications is done to the BibSched scheduler code to make it both aware of multiple Invenio instances and to protect the instances from interfering in negative ways:

Before greater changes were applied, some inconveniences in the current BibSched code that were first refactored. The naming of several functions and variables was misleading or contradictory, such as a variable *'rows'* for tasks that are active (e.g. running) and a function *'handle_row'* that handles tasks, but not only active tasks. The handling of the task status was optimised and the related code was refactored. Repetitive code was removed.

The most important point is to prevent multiple instances of BibSched scheduling tasks at the same time, resulting the task to be executed or altered on two different machines at the same time. The main loop of BibSched retrieves the task queue periodically from the database and analyses the tasks statuses. It then decides if a task is due to be executed based on priority, scheduled time and task status. If so, it sets the task to the status *SCHEDULED* via SQL query, then creates a local process for the task. When the task process starts, the task itself sets its status to *RUNNING* via SQL query. BibSched will halt its loop until the task is set to *RUNNING*. If the task does not reach the running state, the BibSched queue stops and needs manual intervention. This is a critical part in the Multinode scenario and will be investigated later. BibSched might also decide to put a task to sleep if a very high priority task is to be executed immediately. It will then set the tasks status to *ABOUT TO SLEEP*. Tasks might implement an API that polls for this status and halts the process until it is set to *CONTINUING* by BibSched. If a task is altered in any way, BibSched performs the alteration and leaves to current loop instantly to retrieve the latest version of the task queue again. This process is repeated until no changes were applied in one loop. BibSched then puts itself to sleep for 5 seconds before the next main loop is executed.

To prevent multiple BibSched instances from interfering during the described process, there are in principal two possible ways: Using a transaction safe storage engine like InnoDB for the *schTASK* table and modifying the behaviour of all modules accessing the table towards transaction or Introducing MySQL table locking to the main loop of BibSched.

As *schTASK* currently uses the MySAM storage engine that is not transaction safe and neither the BibSched nor the BibTask module were developed in regards of transactions, table locking is easier and more economic to integrate.

In MySQL, a database table can be locked for all but the one connection that establishes the lock, excluding other connections from writing or even reading on the locked table. While the lock is established, queries by other connections are postponed and executed after the lock has been lifted.

With the current DB module of Invenio, the lock means that any code relying on the result set of an SQL query to a locked table will be halted in execution until the lock is lifted or a timeout is reached. This means that the lock should only be established for the shortest possible time, as the BibSched daemon and visual manager as well as BibTasks access the *schTASK* table.

When the main loop starts, a lock to the *schTASK* table is now introduced to BibSched with the SQL statement *"LOCK TABLES schTASK WRITE"*, granting only read and write access to the respective instance. After any change done by BibSched to the table, it is immediately unlocked with *"UNLOCK TABLES"*, giving relief to the pending requests by other instances or modules and allowing newly launched BibTasks to set their status from *SCHEDULED* to *RUNNING*. Afterwards, the main loops starts again, as described before. The lock is also lifted when no changes were done and BibSched goes into sleep for 5 seconds before going into the main loop again.

This behaviour allows multiple BibSched nodes to work on the task queue without interfering. It leads to a serial execution of one main loop of one instance after the other, intermixed with queries from BibTasks during the reliefs.

As nodes should be configurable to only handle specific types of BibTasks, such as BibEncode ones, the behaviour of BibSched needs to be altered in a way that it only executes or even cares about tasks that are supported with its configuration. In the untouched version of BibSched, there is a hardcoded list of valid BibTasks in the Python code file *bibtask_config.py*. Before a BibTask is launched, the tasks name is checked against this list. The purpose of this list is not completely clear, as it resides in a Python file, it is normally not meant to be altered, except by developers, implementing new BibTasks. If one wants to deploy the same sources on all machines of one Invenio installation, the current construct is not suitable for the configuration of allowed and non-allowed tasks on multiple machines.

The first step would be to introduce a new user configurable option in *"invenio-local.conf"*, the file holding the main configuration of Invenio, not overwritten by new source deployments. The option could then be set on each machine in each configuration file individually. For the time being, this is implemented as the new *CFG_BIBSCHEDED_ALLOWED_TASKS* configuration option. On the other hand, the configuration of multiple nodes would be facilitated if the entire configuration were done in one place. As all the nodes need to share some parts of their file systems anyway, a new folder *"etc-shared"* could be introduced for shared configurations. Within this folder a file like *"Invenio-multinode.conf"* could hold the configuration for all the nodes. Either as INI like file with a section for each node and a default section for all nodes or a JSON like file.

6.11 Quality Assurance, Tests and Security

6.11.1 Testing Difficulties

Three different types of automatic tests are currently applied in the Invenio software:

- **Unit Tests** that cover the isolated functionality of individual functions or classes. They verify the intended functionality of said structures. Unit Tests are performed as white-box tests.
- **Regression Tests** that cover the complex interaction within or between modules as well as the interaction with the database and file system. They verify that modules remain functional after big changes within themselves or other modules have been performed. The tests are performed as white-box tests.
- **Web Tests** that cover the behaviour of the web interface of Invenio. They verify that the interface stays functional and behaviour has not changed after changes in the back-end. These tests are always performed as black-box tests.

In BibEncode, there are several utility functions that simply transform input values to output values. These functions can be tested with Unit Tests. For example the function `seconds_to_timecode()` that transforms an integer of given seconds into a video timecode as a string. For BibEncode, these tests reside in the `bibencode_tests.py` submodule.

For functions that actually rely on data ingested from video files, it becomes more difficult to write Unit Tests and at the same time one has to think about the usefulness of such tests. One could at least check if the received data matches predefined data in the test suite. But the check does most likely not check the functionality of the own code, but the behaviour of the external library. Take for example the functions that parse the metadata from *ffprobe* and *MediaInfo*. If changes in the code of these tools alter the output format of the metadata, the parsing functions will fail, raising an exception or generating no output at all. One can easily test against this with a reference file and a unit test. If the changes only change the output slightly, but the format and structure remains, the parsers can still operate but the output set will be different. The test suite can still test with a reference file and a stored reference metadata structure. But the differences might effectively have no influence on the functionality of the system, if metadata output is altered but never used by any further processing. To really test if changes influence the system, the test suite would either need to know every video metadata key that is used or the internal usage must be decoupled completely from the tools metadata representation. The metadata must then be translated into a clearly defined internal structure that supports a fixed set of metadata and the test suite could then test against this structure. Such a structure could be a *Video Class* and its attributes could be lent from the *PBcore* standard.

Automatic testing becomes even more difficult if the functions are wrapping an encoding process. One could in principal compare the metadata of transcoded

reference files to reference values stored in the test suite. This could be done with Regression Tests. But unless Invenio restricts the possible video input and output to specific video containers, specific audio and video codecs and specific parameters for those, the necessary sample set would be very difficult to determine and potentially very large. But a basic level of support could be guaranteed and tested against. These tests would actually not test the BibEncode code, but the capabilities or availability of external libraries. As a simple approach to this problem, BibEncode implements some basic checks to verify the availability of a baseline set of encoding libraries linked to *FFmpeg*. The command line tools of *FFmpeg* report the codecs that were made available during compile time. A parser is written to evaluate the report against a preconfigured list in the BibEncode configuration. The test is executed during the Invenio installation process, on execution of the BibEncode CLI and on execution of the Unit Tests.

BibEncode only includes a single page submission prototype as its web interface. The interface can easily be tested manually, so no Web Tests are developed.

The complete video processing workflow described in chapter 5 and 6.2, including the Multinode support described in chapter 6.6, is tested in a virtual environment with a demo installation of Invenio. The system test is described in chapter 6.8.

6.11.2 Code Quality and Guidelines

The Invenio Coding Style follows the recommendations of PEP8, an official Style Guide for Python Code [62]. PEP8 defines how the code layout should be, for example regarding indentations and line length. It also describes how imports should be defined and how expressions and statements should be built. It goes on with recommendations on the style of comments and in-code documentation and introduces naming conventions for variables and functions as well as giving suggestions on general programming issues. While defining all these rules, the PEP8 manifest clearly states that consistency within a project is more important than consistency with an external style guide, such as the manifest itself. On the other hand, the rules might make the readability and understandability of the code worse, because not every situation of application can be foreseen. In this case the developer should make her own decision and not bend the code for the sake of consistency with the style guide.

The recommendations of PEP8 were followed in the project of this thesis as far as possible. *Pylint* is a tool to verify the consistency with most of the PEP8 standard and to beautify the code. It also checks for general Python errors like uninitialized variables. In addition, there is the *kwalitee.py* script in the Invenio sources that utilises *Pylint* as well as other checking methods to verify the quality of the code. *Kwalitee* is tuned for the requirements of Invenio and its coding guidelines. The output and readability of *Pylint* and *Kalitee* is different, and the author finds the first more appealing for daily usage. Both of them are used to ensure the code quality of the BibEncode module.

6.11.3 Documentation

At CERN, student projects are regularly taken over by other students in the following months, and the fluctuations within teams is relatively high in general. Additionally, the Invenio software is open-source and used by many different Institutions and many different developers contribute code. It is therefore crucial to provide a complete code, module and usage documentation.

The tools described in the previous chapter are performing checks on the existence of code documentation in the form of *Docstrings* in general, but can for sure make no assumption on the quality of them, nor can they check for adequate code comments. By manual checking, author made sure that every module, function and every important piece of code is extensively documented.

The *Docstring* of every module contains a description of its functionality and purpose. Every function in addition has a description for every parameter and the parameter types documented. This is crucial because of the dynamic typing system of Python. The style complies with the Invenio standards.

The usage of the command line interface is documented in the *bibencode.py* sub-module and can be accessed on the command line with the `--help` parameter or is presented automatically if no parameter at all is given.

The extensive customisability of the BibEncode module through its various JSON configurations and profiles is directly documented within these files. The predefined profiles and job templates make use of all configuration options and explain usage and effects of any of them through comments. Bear in mind that the JSON standard has no notion of comments and a non-compliant pre-parser for C-style or JavaScript style comments is introduced in BibEncode.

6.11.4 Security

The main security concern of the BibEncode module is the usage of *ffmpeg*, *ffprobe* and *Mediainfo*. An attacker could in principal forge a malicious multimedia file that exploits security holes in one of the said libraries. If the video submission system is open for public usage, the attacker could upload the forged file and its content will be evaluated by at least one of the tools during the submission and the later processing of the video record. An attacker could thereby cause a denial of service attack by crashing the application or executing arbitrary code with the privileges of the Apache user.

Security vulnerabilities like this existed in the past and are documented for FFmpeg in the Ubuntu Security Bulletin [63]. Just like any other software that has reached a certain complexity and usage, it is likely that new vulnerabilities will be discovered in the future. *Zero Day Exploits*, not yet known to the developers bear the biggest risk. This kind of security vulnerability is not limited to the video submissions in Invenio, but to all kind of submissions that ingest binary data and use external tools or libraries to evaluate it. It can be applied to the tools that handle PDF files or images too.

The administrators of an Invenio installation need to take care that the tools and libraries that potentially expose security vulnerabilities are updated to their latest version on a regular basis. On the other hand, it is not always clear if the functions within Invenio that wrap around these tools keep their full functionality if the tools are updated. They might even break. As described in chapter 4.3.1, FFmpeg at least has the philosophy to keep the APIs constant and features and fixes are backported to older versions that are then re-released.

7 Evaluation

7.1 Testing the Workflow with BibEncode and Multinode

7.1.1 Node Configuration

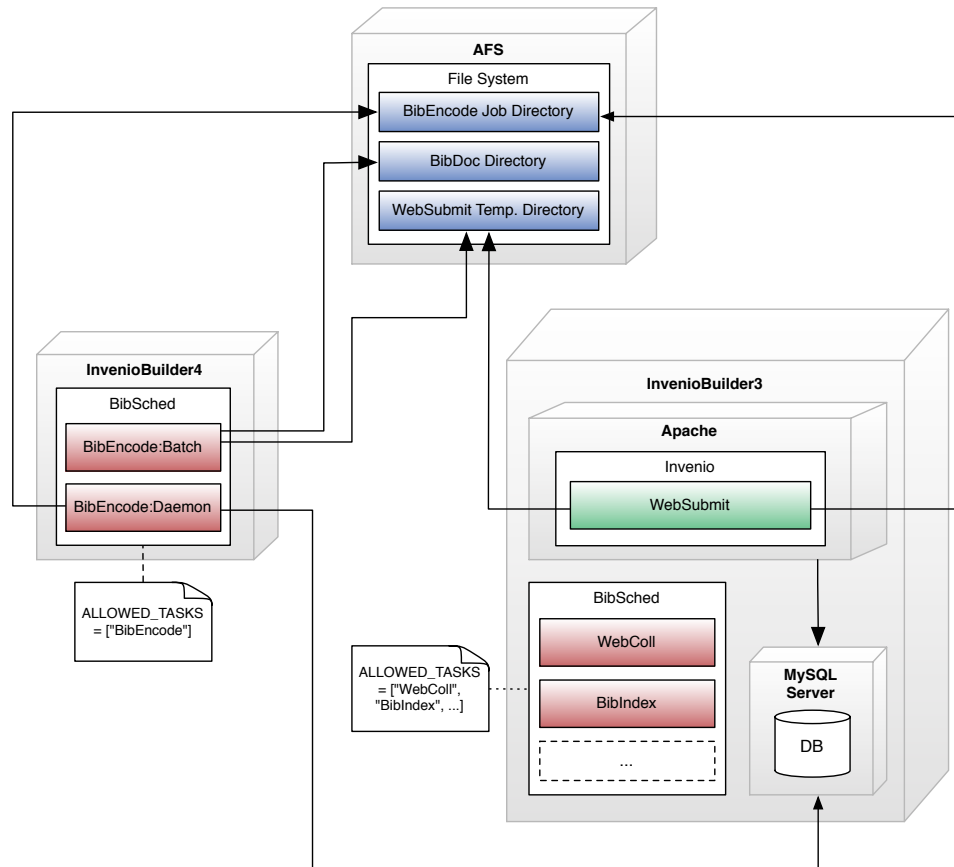


Figure 7-1 The virtual machine configuration for testing BibEncode and Multinode. InvenioBuilder3 hosts an Apache webserver and a MySQL database server. This machine runs the web tier of the Invenio installation. InvenioBuilder4 does not run a webserver and shares the database with InvenioBuilder3. Both machines run BibSched but Builder4 only runs BibEncode tasks and Builder3 does only run non-BibEncode tasks. Both machines share the same filesystem on AFS.

For testing the concept of multi-node encoding, two virtual machines running Scientific Linux CERN 5 (SLC5) are configured. SLC5 is a Linux distribution based on Red Hat Enterprise Linux (RHEL) and built by CERN to integrate nicely into their different services such as distributed file systems and security and authorisation services. The VMs are created through the CERN Virtual Machine service and are therefore running on CERN computing infrastructure, not on an office workstation.

Both VMs virtualise the same hardware:

An Intel Xeon E5410 CPU at 2.33GHz with 1 virtual core and 6144KB of cache as well as 1024MB of virtual RAM and 40GB of virtual disk space.

The virtual machines are called *InvenioBuilder3* and *InvenioBuilder4*. *InvenioBuilder3* functions as the master node, running the web tier of Invenio with an Apache v2.2.3 webserver, shown as (1) in Figure 6-24. *InvenioBuilder4* functions as an encoding only node, shown as (4) in Figure 6-24.

InvenioBuilder4 does not run an Apache server and is therefore not reachable through HTTP for the user.

Both machines share the same MySQL v5.0.77 database residing on *InvenioBuilder3*. The *IPtables* firewall of *InvenioBuilder3* is configured to allow access through the MySQL port with the following command:

```
1 iptables -A RH-Firewall-1-INPUT -p tcp -m tcp --dport 3306 -j ACCEPT
```

In addition, access to the database was granted in the MySQL configuration for *InvenioBuilder4s* hostname. From the MySQL command shell, the following command is executed:

```
1 grant all privileges on *.* to invenio@inveniobuilder3 identified by 'password'
```

Both machines are deployed with the latest BibEncode project code. The code and demo installation are installed according to the latest Invenio installation documentation with all the additions described in this project documentation. In addition, the Invenio configuration file is altered to hold the same configuration for both nodes. The configuration file is not shared but just identical:

```
1  ## invenio-local.conf:
2  [Invenio]
3  CFG_DATABASE_HOST = inveniobuilder3
4  CFG_DATABASE_PORT = 3306
5  CFG_DATABASE_NAME = invenio
6  CFG_DATABASE_USER = invenio
7  CFG_DATABASE_PASS = password
8  CFG_SITE_URL = http://inveniobuilder3.cern.ch
9  CFG_SITE_SECURE_URL = https://inveniobuilder3.cern.ch
10 CFG_SITE_ADMIN_EMAIL = bjorn.oltmanns@cern.ch
11 CFG_SITE_SUPPORT_EMAIL = bjorn.oltmanns@cern.ch
12 CFG_BIBSCHED_ALLOWED_TASKS = {
13   'inveniobuilder3' : ["bibindex", "bibupload", "bibreformat",
14    "webcoll", "bibtaskex", "bibrank",
15    "oaiharvest", "oairepositoryupdater", "inveniogc",
16    "webstatadmin", "bibclassify", "bibexport",
17    "dbdump", "batchuploader", "bibauthorid",
18    "bibtasklet"],
19   'inveniobuilder4' : ['bibencode'],
20 }
21 CFG_PATH_FFMPEG = /usr/local/bin/ffmpeg
22 CFG_PATH_FFPROBE = /usr/local/bin/ffprobe
23 CFG_PATH_MEDIAINFO = /usr/local/bin/mediainfo
24 CFG_BIBSCHED_MAX_NUMBER_CONCURRENT_TASKS = 10
```

Figure 7-2 *Invenio-local.conf* configuration file for the virtual machines.

As both machine use the same database, the DB configuration points to the same hosts and uses the same user (Figure 7-2, lines 3 to 7). Both machines are config-

ured to use the URL '<http://inveniobuilder3.cern.ch>'. While only one machine runs a webserver, the other machine sends emails after successful video processing that include the URL to the completed video record. These URLs should obviously point to the machine accessible through the web (line 8 and 9). Lines 12 to 20 of Figure 7-2 show the task configuration per BibSched host. *InvenioBilder4* is allowed to run every possible task except BibEncode tasks. *InvenioBuilder3* is only allowed to run BibEncode tasks. There were some problems with the path detection of the external tools *ffmpeg*, *ffprobe* and *Mediainfo* on SLC5. The paths are set manually in lines 21 to 23. The last line of Figure 7-2 sets the number of concurrent tasks in BibSched to 10. This means that every node can execute a maximum of 10 parallel tasks if these are non-interfering. While SLC5 blends nicely into CERN services such as AFS (Andrew File System, a distributed file system) and the CERN user authentication, it uses many old builds of packages and libraries and relies on a huge quantity of backports. This renders the default system unusable for state-of-the-art video encoding. All deprecated packages that relate to video encoding are therefore removed and the latest versions are installed from source afterwards.

The deprecated packages "lame.x86_64", "libtheora.x86_64", "libogg.i386", "libogg.x86_64", "libvorbis.i386", "libvorbis.x86_64" and "vorbis-tools.x86_64" are removed using the package manager "yum" under SLC5 with the command `$yum remove 'package name'`.

To encode all of today's HTML5 video codecs, the following libraries are necessary:

- **FFMPEG** – Version 0.8 – An open-source tools and libraries for decoding and encoding many different multimedia formats.
- **Libx264** – From git master branch on June 29th 2011– An open-source library for H.264 video, which is not redistributable due to patent and licensing issues regarding the H.264 codec.
- **Libfaac** – Version 1.28 – open-source library for Fraunhofer Advanced Audio Coding (AAC) decoding and encoding, not redistributable due to patent and licensing issues regarding AAC.
- **Liblame** – Version 3.98.4 – An open-source library for MP3 audio codec, not redistributable due to patent and licensing issues regarding MP3
- **LibOpenJPEG** – Version 1.4 – An open-source library for JPEG image decoding and encoding
- **LibTheora** – Version 1.1.1 – open-source library for Theora video codec
- **LibOGG** – Version 1.1.4 - open-source library for the OGG multimedia container
- **LibVorbis** – Version 1.2.3 - open-source library for decoding and encoding the Vorbis audio codec
- **LibVPX** – From git master branch on June the 29th 2011 - open-source library for the VP8, now called WebM, video codec
- **Mediainfo** – Version 0.7.45 - open-source CLI tool and library to retrieve metadata information

The installation from sources is straightforward. All libraries are built with a standard `./configure; make; make install` command chain.

For the shared file server, shown as (1) in Figure 7-1, the AFS at CERN is used. A 100GB partition is created inside the project space of CDS. Within the partition, multiple folders are created to host the shared files:

/tmp-shared – hosts video uploads in their temporary form

/tmp-shared/bibencode/jobs – hosts new job files for the bibencode daemon

/tmp-shared/bibencode/jobs/done – hosts job files that have been processed

/data/files – hosts BibDocs, files associated to records

/data/submit – hosts files created during the WebSubmit submissions

Symbolic links are then created in the Invenio installation directory of both *InvenioBuilder3* and *InvenioBuilder4*, pointing to the newly created folders on AFS. The AFS itself is mounted as a folder in the root directory of the machines.

On Linux, similar commands are used for this purpose:

```
1 $ln -s /afs/cds/bibencode/tmp-shared /opt/invenio/tmp-shared
2 $ln -s /afs/cds/bibencode/data /opt/invenio/data
```

After the configuration, the webserver on IB3 and the BibSched instances of both installations are restarted. The Invenio installations are reinitialised with the *inveniocfg* command line tool. The WebColl, BibReFormat, BibIndex and BibRank module are added to the BibSched queue with a periodic execution time of one minute. The Multinode system is fully operational afterwards. Figure 7-3 and 7-4 show the visual BibSched queues of IB3 and IB4 after the installation and the ingestion of some test records.

ID	PROC [PRI]	USER	RUNTIME	SLEEP	STATUS	HOST	PROGRESS
41	bibencode:batch-37-2011	bibencode:daem	2011-08-17 17:05:30		RUNNING	inveniobuilder4	[4/7][#####] [1/1] 003%
42	bibencode:batch-37-2011	bibencode:daem	2011-08-17 17:05:30		RUNNING	inveniobuilder4	[2/5][#####] [1/1] 071%
8	webcoll	admin	2011-08-17 17:08:20 1m		RUNNING	inveniobuilder3	Part 2/2: done 6/25
38	bibindex	admin	2011-08-17 17:09:38 1m		WAITING		Will execute: './bibindex'
7	bibreformat	admin	2011-08-17 17:09:47 1m		WAITING		Will execute: './bibreformat'
37	bibencode:daemon	admin	2011-08-17 17:09:48 1m		WAITING		Will execute: '/opt/invenio/
9	bibrank	admin	2011-08-17 17:09:52 1m		WAITING		Will execute: './bibrank' '-

Figure 7-3 BibSched running on InvenioBuilder3 with multiple running and waiting tasks. The cyan coloured tasks are executed on InvenioBuilder3 or are not allowed on InvenioBuilder4.

ID	PROC [PRI]	USER	RUNTIME	SLEEP	STATUS	HOST	PROGRESS
41	bibencode:batch-37-2011	bibencode:daem	2011-08-17 17:05:30		RUNNING	inveniobuilder4	[4/7][###] [1/1] 020%
42	bibencode:batch-37-2011	bibencode:daem	2011-08-17 17:05:30		RUNNING	inveniobuilder4	[2/5][#####] [1/1] 086%
8	webcoll	admin	2011-08-17 17:08:20 1m		RUNNING	inveniobuilder3	Part 2/2: done 10/25
38	bibindex	admin	2011-08-17 17:09:38 1m		WAITING		Will execute: './bibindex'
7	bibreformat	admin	2011-08-17 17:09:47 1m		WAITING		Will execute: './bibreformat'
37	bibencode:daemon	admin	2011-08-17 17:09:48 1m		WAITING		Will execute: '/opt/invenio/
9	bibrank	admin	2011-08-17 17:09:52 1m		WAITING		Will execute: './bibrank' '-

Figure 7-4 BibSched running on InvenioBuilder4 with multiple tasks. The cyan coloured tasks are not allowed on this machine or are running on InvenioBuilder3.

7.1.2 Stress Testing

To detect potential problems with the changes introduced to BibSched and to verify the stability of two Invenio instances working together, a stress test for the Multinode system is conducted.

The idea is to submit a continuous stream of both normal and video records. This will create numerous BibUpload tasks in the BibSched queue to insert the metadata into the shared database. At the same time, fulltext files are ingested and stored in the shared file system. The indexing, ranking and record formatting modules evaluate newly added metadata periodically. This part of the stress test is running on the InvenioBuilder3 BibSched instance. The processing of the videos is done on InvenioBuilder4 at the same time. Video metadata injections from this machine are pushed to InvenioBuilder3 after the processing.

The Firefox plugin *iMacros* [64] is used to record the submission of a thesis, poetry, journal and an article record on the demo site. All fields of the demo submissions are filled with placeholder data (e.g. “Lorem ipsum dolor ...”). A one page PDF is used as fulltext file upload. *iMacros* is then able to replay the recorded steps, thereby submitting new records based on the recorded form data. The plugin can unfortunately not record actions taken in Flash content. The video submission can therefore only be recorded up to the point where the video file has to be selected. The plugin is then set in looping mode in which the recorded macro is continuously played, thereby creating a continuous stream of record submissions. While the macro is running, 10 video submissions are created manually through the video submission interface created for BibEncode. This maxes out the total number of parallel encoding tasks on InvenioBuilder4. A 4-minute, 10MB big, Standard Definition (480p) WebM video was used for the video uploads. The submission uses a batch-processing template with 4 encoding jobs and 3 extraction jobs. Two 720p and two 480p encoding jobs in MP4/H.264 and WebM and one single thumbnail extraction and two 10 thumbnail extractions. The quality assurance should filter the 720p jobs out and only execute the 480p jobs. The record submission macro is stopped when the batch processing of all submitted videos finishes.

Both visual task managers are running during the test. This means that two task managers and two BibSched daemons are accessing the schTASK table periodically. In addition, every task accesses the table during its initialisation. Only BibSched daemons lock the table. All the other processes can only access the table during the unlock reliefs.

The processing of the 10 video submissions in parallel on InvenioBuilder4 takes about 25 minutes. During that time, about 120 normal submissions are ingested and processed on InvenioBuilder3. About 200 BibTasks are executed in total. 20 video slaves are created and 210 thumbnails extracted.

The test was repeated multiple times, delivering comparable results each time. No problems in scheduling or running tasks were detected. All created records were functional and available in their respective collections.

The stress test was executed at the very end of the project. Previous test of earlier iterations on the virtual machines detected some issues that presented in the next chapter.

7.1.3 Detected Issues

As both of the virtual machines run SLC5 installations, there is only Python 2.4 available. This caused trouble as some parts of BibEncode were unwittingly relying on modules and functions available only in later Versions of Python:

- The *NamendTemporaryFile* object that was used in the asynchronous video upload handler has no *'delete'* attribute in 2.4. The issue is discussed in chapter 6.2.4.2.
- The *'SEEK_END'* constant is not present in the *os* module in 2.4. It was used in the ffmpeg log parser of the encoding module of BibEncode to read from the end of a log file. The constant call was simply replaced by its integer value for older Python versions.
- The *json* module is not available in Python 2.4. This issue was discussed in chapter 4.6.2. The installation of the *simplejson* backport is mandatory in this case. The BibEncode code is altered to load the *simplejson* module instead of *json* if the Python version is less than 2.5.

Another issue was that the *fixmarc()* function in the command line module of BibDocFile was not using the shared temporary directories. The function creates a BibUpload that relies on a MARCXML file to add BibDocFile URLs to the metadata of a record. Because the BibUploads are not executed on the machine that creates the video slaves, the MARCXML file needs to be in a shared directory. The function is altered to use the shared temporary directory.

A persistent problem on the virtual machines is that some tasks crash before their code reaches the point where the task status is set to *RUNNING*. This causes the BibSched automatic task execution to halt until manual intervention. The bad thing is that no error or exception appears in the BibTask logs. This means that the task crashes even before the error logging is initialised. When no task is running but the periodic tasks, the problem appears in intervals between 12 and 24 hours. In most cases, the Linux system logs of the virtual machines showed that the AFS authentication token ran out just before the task that failed was executed. But the relevant folders for the task execution are not shared on AFS. The problem never occurred on the development workstation of the author, where the same code was deployed and one instance of BibSched was running for days. AFS was mounted on the workstation, but not within any Invenio folder. The author has the impression that the problem is not directly related to the changes to the BibSched code, especially the table locking, or the parallel running instances. Similar problems have been observed on the CDS production machines with unmodified code. The AFS token expiration seems to be more than a coincidence. The author was unfortunately not able to do any further investigations.

7.2 Fulfilment of Requirements

Upload of a video file through a web interface & Adding metadata information to the video to create a record – A custom WebSubmit workflow was implemented to handle the upload of videos and their metadata.

Preserving and storing the original (Master) video for archiving and later use – The processing of the uploaded video material can be configured to include the storage of the video master (JSON profiles). The stored master can later be used to reprocess the video if new formats are required.

Extracting metadata information from the video file/container. For example the duration, size etc. – The metadata module of BibEncode can extract any metadata available through ffprobe and Mediainfo. This metadata can either be user in code or dumped to a JSON or PBcore representation for data exchange.

Appending the extracted metadata to the record for example for search features and later use – Metadata extracted as PBcore can be converted to MARCXML using the provided XSL stylesheet. The MARCXML can then be appended to a video record. This method is used in the reusable and adaptable video processing workflow provided with the demo installation.

Transforming the original video into other video formats. (Creation of Slaves) – The encoding module of BibEncode can transcode from any to any format available in ffmpeg and its external codec libraries with a consisted API. For the systematic creation of multiple output formats at one time, the batch module can be used.

Using open-source and licence free encoders and libraries for the transformation – Both ffmpeg and Mediainfo are available as open-source, distributable and license-free code. Some of the codec libraries for ffmpeg must be licensed for commercial use as they rely on patented technologies. The use of these libraries depends on the needs of the institution running Invenio. A complete video workflow can be created using only open and free formats such as OGG Vorbis and WebM.

Storing the Slaves for later streaming – BibEncode integrates into BibDoc to store video slaves. The slaves are available for any client through the HTTP interface of BibDocFile.

Extracting thumbnails (still images) from the video for representation purposes – The extraction module of BibEncode handles the creation of video thumbnails. The extraction is highly configurable and integrates into the video processing workflow to create previews.

Using open-source and licence free libraries to extract the thumbnails – ffmpeg is used for the creation of thumbnails. The same remarks given for the video encoding are valid here.

Storing these thumbnails – Just like the video slaves, thumbnails are stored and served through the BibDocFile API.

Generating a video representation using the user inserted and the extracted metadata together with the Slaves as sources for video streams – Video records are presented in a complete web-interface that resembles popular video platforms such as YouTube or Vimeo in appearance and features. The video page contains the user-inserted metadata as title, description and authors displayed to the users. The extracted metadata is displayed to the user in form of the video duration and used for the technical instantiation of the video sources and player.

Streaming the video the user – Videos are streamed through a feature rich HTML5 video player, embedded into the video record page. Progressive streaming with Byte Range Requests is used to allow seeking within videos. The user can chose between streams of different qualities and resolution.

Classic download of videos – Besides streaming videos, the video files can be downloaded to be played locally by the user. Modifications to the BibDocFile API were introduced to allow this.

Systematic scheduling of video transformations – All video transformations are executed as BibTasks in BibSched.

Decoupling of web-server and video encoding, e.g. running the video encoding on a different machine than the webserver – The concept of multiple Invenio instances was introduced and a working prototype for at least small-scale installations is tested.

YouTube like video experience: A video player suitable for any web browser; Video recommendations; Commenting system – The introduced video player offers support for Adobe Flash and Microsoft Silverlight to allow video playback in browsers without HTML5 support. BibEncode is able to create the necessary video formats through ffmpeg. Video recommendations are generated based on the search and ranking engines available in Invenio and made available to the user through the video record presentation. The commenting system of Invenio is refactored to insert comments directly into the record representation for YouTube like video pages.

All requirements set in the beginning of the project of the project as well as those that evolved during its execution are fulfilled. New requirements will surely arise if the system goes into production. For this case, the general and flexible infrastructure of BibEncode will allow its extendibility and maintainability.

7.3 Open Issues

7.3.1 Support for HTTP Live Streaming

Chapter 4.5 explained the different approaches for streaming videos to a web client. The most standard and convenient way currently is progressive streaming

with the support for Byte Range requests. To enable this way of streaming, only the installation of *mod_xsendfile* for Apache is required and the configuration option *CFG_BIBDOCFILE_USE_XSENDFILE* in the Invenio configuration needs to be set to true. The current implementation in Invenio then handles the video requests just the right way.

If browser vendors other than Apple start supporting HTTP Live Streaming, it might be interesting to have such a functionality in Invenio. It would allow the automatic bandwidth detection and appropriate video format selection by the client. The video player implemented in this project only has a manual selection option for different video resolutions and qualities. The BibEncode Batch Processing engine is already able to create various video formats in different qualities and resolutions for various bandwidths. To allow HTTP Live Streaming, a *Video Segmenter* could be integrated into the Batch Engine. A new type of BibEncode mode (e.g. 'segment') could be introduced that uses the Segmenter to split the video and create appropriate BibDocs for the different sets of segments. The segmenter described in chapter 4.5.2 can also create the necessary manifest files.

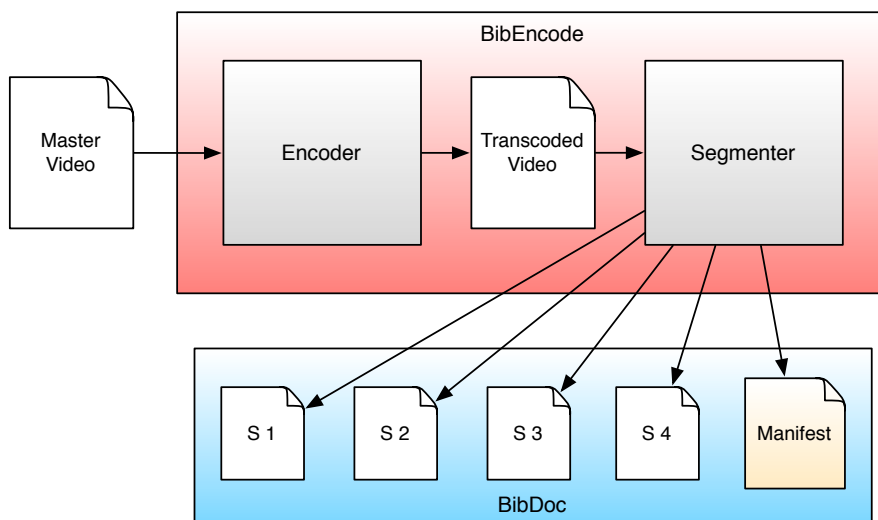


Figure 7-5 A concept of a video segmenter in BibEncode for HTTP Live Streaming

7.3.2 Improved Video Uploads

Chapter 4.4 showed that potentially large files such as videos need special treatment regarding their upload and server side storage. The concept of resuming uploads by implementing a chunking mechanism was presented. Due to the time constraints of project, it was not possible to implement such functionality. Instead, an asynchronous uploading system was implemented based on *Uploadify* and the already existing image uploading capabilities. If institutions are using the video submission only internally, over fast and reliable local networks, there are no big issues with the current system. If video submissions are open for uploading files over the Internet, the results are yet to be investigated. YouTube is for example using an experimental Java based uploader [65] for huge video files up to 20 Gigabytes that allows the resuming of uploads. Timeouts or disconnects might indeed make it a frustrating experience to upload a large video. It would

also make it possible for users to just stop the submission and continue the upload at a later time.

A future extension or project might want to focus on improving video uploads or uploads in general in Invenio. It would consist of a client part that handles the chunking and communicates with the server through an AJAX API. The client part could for example be based on the *Plupload* script presented in chapter 4.4.3. Currently *Plupload* only has chunking capabilities, but no API for resuming. The script could be extended to handle AJAX responses that indicate resuming capabilities of the server and specify the chunk to start the upload from again. On the server part, a new asynchronous upload handler would take care of associating and reassembling chunks as well as sending the messages to the client to resume uploads.

7.3.3 Scalability of Multinode

The way Multinode and distributed processing of video records is implemented can be seen as an ad hoc solution for the needs of irregular video submissions.

If multiple Invenio nodes are available, the processing would only be parallelised on a per batch job basis. The individual processing of batch jobs is not parallelised, because current versions of BibSched lack any notion of task interdependencies and hierarchies. If this would be available, each batch job could be split into smaller tasks. Dependent tasks would then wait until the tasks they depend upon have been successfully executed. It would also be necessary to visualise the dependencies inside the visual task manager of BibSched. It would be very difficult for the administrator to analyse problems otherwise.

On the other hand, the table locking mechanism introduced to allow multiple instances of BibSched on the same database forces a serialised, cyclic evaluation of the task queue node by node. For example: A Node 1 locks the table, evaluates the task queue, launches a task and unlocks the table; while this is happening, Node 2 tries to do the same and sends a lock command to the database; this command is pending until Node 1 unlocks; the code execution of Node 2 is halted during that time; the example starts from the beginning, but Node 1 and 2 change their roles. Figure 7-2 shows a graphical version of the example.

The results of chapter 6.8 show no significant performance impact for two nodes. If a larger number of nodes is deployed, task executions might be delayed noticeably. The pending queries might even go into timeout in an extreme case.

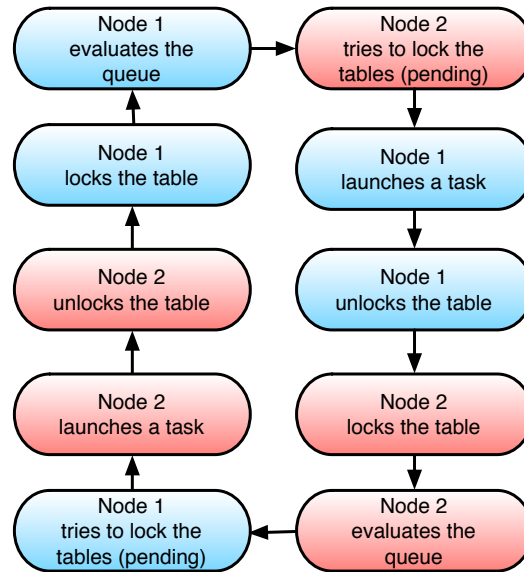


Figure 7-6 Serial, cyclic evaluation of the task queue with multiple BibSched instances.

Another problem is the strong dependency on BibUpload tasks. Many tasks rely on the availability of record metadata. As only BibUploads should insert the metadata, BibUpload tasks are executed with absolute priority compared to other types of tasks. The code of BibSched contains parts that are specially tweaked for an immediate execution of BibUploads. BibSched might even halt other tasks to execute pending BibUploads. This principle might be undermined with multiple BibSched nodes where some of them handle BibUploads and others don't. It might even be undermined if tasks are only executed in parallel on one machine. On current production systems, BibTasks are never executed in parallel. Some types of tasks should even not run in parallel, as they might work on the same data and could thereby cause inconsistencies. But parallel execution is in principle available for BibSched and is used for testing BibEncode. Parallel tasks were at least no problem for the demo installation environment.

All of the points mentioned above might cause unpredictable problems in a production environment. Potential problems could partially be resolved by introducing task dependencies as described before. The whole task execution infrastructure might need to be rethought and redesigned to allow the save execution of parallel tasks and the scalability of Invenio over multiple instances.

8 Conclusions

Solutions for the processing and presentation of videos already existed for the CERN Document server in the form of the Media Archive. But these solutions are highly specialised and not applicable for other institutions using Invenio. The goal of the project was the design and implementation of an open, portable and highly customisable video processing solution for Invenio that can be used by any institution.

The related technologies to create such a processing system were discussed in depth and the state-of-the-art in video technologies for the web was presented.

Potential solutions based on described technologies were analysed and a conceptual solution that is able to adapt to many Use-Cases was presented. The concept was then implemented as a comprehensive enhancement of the Invenio software. A fully functional video processing workflow was made available and Invenio administrators can take the provided examples and tweak them to their needs for very quick results. But they can also create completely custom video submission workflows based on the newly added video capabilities.

Regarding video formats, presentation and streaming, the focus was set on HTML5 video. The analysis and application of the new technology showed very positive results. HTML5 video already reached 60% of the market in the beginning of the project. The author is confident that the technology will reach a market penetration similar to Adobe Flash in the near future. Future additions to the video support in Invenio like subtitles or improved streaming should therefore be developed as HTML5 based solutions.

Future work should be devoted to further improvements in the handling of large uploads and the scalability of the processing system. Not only the video workflow but also Invenio in general would profit from such improvements.

9 Bibliography

- 1 Invenio Developers Community. About Invenio. [Internet]. 2011 [cited 2011 June 20]. Available from: <http://invenio-software.org/>.
- 2 Invenio Developers Community. Invenio Installation. [Internet]. 2011 [cited 2011 June 20]. Available from: <http://invenio-software.org/browser/INSTALL>.
- 3 Pepe A, Baron T, Simko T, Vesely M, Gracco M, Le Meur JY, Robinson N. CERN Document Server Software: the integrated digital library. [Internet]. 2005 [cited 2011 August 1]. Available from: <http://cdsware.cern.ch/invenio/doc/elpub2005.pdf>.
- 4 Kwiatek M. Automatic processing of CERN video, audio and photo archives. Journal of Physics: Conference Series. 2008;119(8).
- 5 Macromedia, Inc. MACROMEDIA AND SORENSON MEDIA BRING VIDEO TO MACROMEDIA FLASH CONTENT AND APPLICATIONS. [Internet]. 2001 [cited 2011 June 20]. Available from: http://www.adobe.com/macromedia/proom/pr/2002/flash_mx_video.html.
- 6 Adobe Systems Inc. Frequently Asked Question. [Internet]. 2009 [cited 2011 July 6]. Available from: <http://www.openscreenproject.org/about/faq.html>.
- 7 Adobe Systems Inc. Flash Player Penetration. [Internet]. 2011 [cited 2011 June 20]. Available from: http://www.adobe.com/products/player_census/flashplayer/.
- 8 Wegner S. Web Architecture and Codec Considerations for Audio-Visual Services. [Internet]. 2007 [cited 2011 June 20]. Available from: <http://www.w3.org/2007/08/video/positions/Nokia.pdf>.
- 9 W3C. The HTML5 Video Element. [Internet]. 2009 [cited 2011 June 20]. Available from: <http://www.w3.org/TR/2009/WD-html5-20090423/video.html>.
- 10 Google Inc. Differences Between Google Chrome and Chromium. [Internet]. 2011 [cited 2011 June 20]. Available from: <http://code.google.com/p/chromium/wiki/ChromiumBrowserVsGoogleChrome>.
- 11 Jazajery M. HTML Video Codec Support In Chrome. [Internet]. 2011 [cited 2011 June 20]. Available from: <http://blog.chromium.org/2011/01/html-video-codec-support-in-chrome.html>.
- 12 W3C. HTML5 Working Draft. [Internet]. 2011 [cited 2011 July 9]. Available from: <http://www.w3.org/TR/html5/>.
- 13 Apple Inc. HTMLVideoElement. [Internet]. 2010 [cited 2011 July 9]. Available from: http://developer.apple.com/library/safari/#documentation/appleapplications/reference/WebKitDOMRef/HTMLVideoElement_idl/Classes/HTMLVideoElement/index.html.
- 14 Apple Inc. HTTP Live Streaming. [Internet]. 2010 [cited 2011 July 9]. Available from: <http://developer.apple.com/library/ios/#documentation/networkinginternet/conceptual/streamingmediaguide/Introduction/Introduction.html>.

- 15 Goncalves I, Pfeiffer S, Montgomery C. Ogg Media Types. [Internet]. 2008 [cited 2011 July 4]. Available from: <http://www.xiph.org/ogg/doc/rfc5334.txt>.
- 16 Xiph.Org Foundation. Theora Specification. [Internet]. 2011 [cited 2011 June 20]. Available from: <http://www.theora.org/doc/Theora.pdf>.
- 17 Mozilla Foundation. Firefox 3.5 Release Notes. [Internet]. 2009 [cited 2011 June 20]. Available from: <http://www.mozilla.com/en-US/firefox/3.5/releasesnotes/>.
- 18 Jägenstedt P. Re-Introducing. [Internet]. 2009 [cited 2011 June 20]. Available from: <http://my.opera.com/core/blog/2009/12/31/re-introducing-video>.
- 19 Xiph.Org Foundation. DDirectshow Filters for Ogg Vorbis, Speex, Theora, FLAC, WebM. [Internet]. 2011 [cited 2011 June 20]. Available from: <http://www.xiph.org/dshow/>.
- 20 Xiph.Org Foundation. QuickTime Components. [Internet]. 2009 [cited 2011 June 20]. Available from: <http://www.xiph.org/quicktime/>.
- 21 Xiph.Org. OGG Metadata. [Internet]. 2011 [cited 2011 July 4]. Available from: <http://wiki.xiph.org/Metadata>.
- 22 Bankoski J, Koleszar J, Quillio L, Salonen J, Wilkons P, Xu Y. VP8 Data Format and Decoding Guide. [Internet]. 2011 [cited 2011 July 4]. Available from: <http://tools.ietf.org/html/draft-bankoski-vp8-bitstream-04>.
- 23 Matroska.Org. Matroska Tag Specifications. [Internet]. 2011 [cited 2011 July 4]. Available from: <http://www.matroska.org/technical/specs/tagging/index.html>.
- 24 WebM Project. WebM Container Guidelines. [Internet]. 2011 [cited 2011 July 4]. Available from: <http://www.webmproject.org/code/specs/container/>.
- 25 Mozilla Foundation. Firefox 4 Release Notes. [Internet]. 2011 [cited 2011 July 5]. Available from: <http://www.mozilla.com/en-US/firefox/4.0/releasesnotes/>.
- 26 Willis C. Opera supports the WebM video format. [Internet]. 2010 [cited 2011 July 5]. Available from: <http://dev.opera.com/articles/view/opera-supports-webm-video/>.
- 27 WebM Project. WebM Media Foundation Components for Microsoft Internet Explorer 9. [Internet]. 2011 [cited 2011 July 5]. Available from: <http://www.webmproject.org/ie/>.
- 28 Thompson R. WebM QuickTime Component. [Internet]. 2010 [cited 2011 July 5]. Available from: <http://code.google.com/p/webm/downloads/detail?name=WebM%20Component%20Installer.pkg>.
- 29 International Telecommunication Union. Advanced video coding for generic audiovisual services. SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS [Internet]. 2010 March.
- 30 MPEG LA. MPEG LA's AVC License Will Not Charge Royalties for Internet Video that is Free to End Users through Life of License. [Internet]. 2010 [cited 2011 July 5]. Available from: <http://www.mpegla.com/Lists/MPEG%20LA%20News%20List/Attachments/231/n-10-08-26.pdf>.

- 31 Maxwell G. Youtube and OGG Theora comparison. [Internet]. 2009 [cited 2011 June 20]. Available from: <http://people.xiph.org/~greg/video/ytcompare/comparison.html>.
- 32 Loli-Query E. Theora vs. H.264. [Internet]. 2007 [cited 2011 June 20]. Available from: <http://www.osnews.com/story/19019/Theora-vs-h.264/>.
- 33 QuavLive. H.264 vs VP8: a video codec comparison. [Internet]. 2010 [cited 2011 July 4]. Available from: http://www.quavlive.com/video_codec_comparison.
- 34 Vatolin D, Kulikov D, Parshin A, Arsaev M, Voronov A. MPEG-4 AVC/H.264 Video Codecs Comparison. [Internet]. Moscow 2011 [cited 2011 July 4]. Available from: http://compression.ru/video/codec_comparison/h264_2011/mpeg-4_avc_h264_video_codecs_comparison.pdf.
- 35 Google Inc. Android Supported Media Formats. [Internet]. 2011 [cited 2011 July 5]. Available from: <http://developer.android.com/guide/appendix/media-formats.html>.
- 36 StatCounter. Browsers May 2011. [Internet]. 2011 [cited 2011 August 19]. Available from: <http://gs.statcounter.com/#browser-ww-monthly-201105-201105>.
- 37 OECD. Advertised broadband download speed statistics in September 2010. [Internet]. 2010 [cited 2011 August 3]. Available from: <http://www.oecd.org/dataoecd/10/53/39575086.xls>.
- 38 McFarland P. Approximate YouTube Bitrates. [Internet]. 2010 [cited 2011 August 3]. Available from: <http://adterrasperaspera.com/blog/2010/05/24/approximate-youtube-bitrates>.
- 39 FFmpeg. FFmpeg General Documentation. [Internet]. 2011 [cited 2011 August 3]. Available from: <http://ffmpeg.org/general.html>.
- 40 W3C. File API. [Internet]. 2010 [cited 2011 August 3]. Available from: <http://www.w3.org/TR/FileAPI/>.
- 41 W3C. Progress Events. [Internet]. 2011 [cited 2011 August 3]. Available from: <http://www.w3.org/TR/progress-events/>.
- 42 Adobe Systems Inc. ActionScript 3.0 FileStream Class. [Internet]. 2011 [cited 2011 August 4]. Available from: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/filesystem/FileStream.html.
- 43 Garcia R. Uploadify. [Internet]. 2011 [cited 2011 August 4]. Available from: <http://www.uploadify.com/documentation/>.
- 44 Moxiecode Systems. Plupload. [Internet]. 2011 [cited 2011 August 4]. Available from: <http://www.plupload.com/index.php>.
- 45 Red5 Collaboration. Red5 Documentation. [Internet]. 2009 [cited 2011 August 4]. Available from: <http://trac.red5.org/wiki/Documentation/UsersReferenceManual>.
- 46 Pantos R, May W. HTTP Live Streaming. [Internet]. 2011 [cited 2011 August 4]. Available from: <http://tools.ietf.org/html/draft-pantos-http-live-streaming-06>.

- 47 Douglas C. iPhone HTTP streaming. [Internet]. 2009 [cited 2011 August 4]. Available from: <http://www.ioncannon.net/programming/452/iphone-http-streaming-with-ffmpeg-and-an-open-source-segmenter/comment-page-1/>.
- 48 D. C. The application/json Media Type for JavaScript Object Notation (JSON). [Internet]. 2006 [cited 2011 August 4]. Available from: <http://tools.ietf.org/html/rfc4627>.
- 49 Library of Congress. Marc Field 245 - Title Statement (NR). [Internet]. 2008 [cited 2011 July 31]. Available from: <http://www.loc.gov/marc/bibliographic/bd245.html>.
- 50 Library of Congress. Marc Field 100 - Main Entry-Personal Name (NR). [Internet]. 2008 [cited 2011 July 31]. Available from: <http://www.loc.gov/marc/bibliographic/bd100.html>.
- 51 Library of Congress. Marc Field 260 - Publication, Distribution, etc. (Imprint) (R). [Internet]. 2008 [cited 2011 July 31]. Available from: <http://www.loc.gov/marc/bibliographic/bd260.html>.
- 52 Library of Congress. Marc Field 520 - Summary, Etc. (R). [Internet]. 2008 [cited 2011 July 31]. Available from: <http://www.loc.gov/marc/bibliographic/bd520.html>.
- 53 Library of Congress. Marc Fields for Videorecording. [Internet]. 2008 [cited 2011 August 1]. Available from: <http://www.loc.gov/marc/bibliographic/bd007v.html>.
- 54 Corporation of Public Broadcasting. About PBCore. [Internet]. 2011 [cited 2011 August 1]. Available from: <http://pbcore.org/about/>.
- 55 Corporation for Public Broadcasting. PBCore Schema 2.0. [Internet]. 2011 [cited 2011 August 1]. Available from: <http://pbcore.org/schema/>.
- 56 Simko T. Python Performance. [Internet]. 2008 [cited 2011 July 20]. Available from: https://twiki.cern.ch/twiki/bin/view/CDS/PythonPerformance#4_Don_t_use_OO_paradigm_and_adva.
- 57 Pfeiffer S. The Definitive Guide to HTML5 Video. 1st ed. Vol I. Apress; 2010.
- 58 Zencoder, Inc. HTML5 Video Player. [Internet]. 2010 [cited 2011 August 16]. Available from: <http://videojs.com/>.
- 59 Kluger S. Projekktor Zwei. [Internet]. 2011 [cited 2011 August 16]. Available from: <http://www.projekktor.com/>.
- 60 Dyer J. MediaElement.js. [Internet]. [cited 2011 August 16]. Available from: <http://mediaelementjs.com/>.
- 61 Project Inspire. INSPIRE Project Information. [Internet]. 2011 [cited 2011 August 16]. Available from: <http://www.projecthepinpire.net/>.
- 62 Rossum Gv. Style Guide for Python Code. [Internet]. 2001 [cited 2011 August 9]. Available from: <http://www.python.org/dev/peps/pep-0008/>.
- 63 Canonical. FFmpeg vulnerabilities. [Internet]. 2010 [cited 2011 August 9]. Available from: <http://www.ubuntu.com/usn/usn-931-1/>.

- 64 iOpus. iMacros Firefox plugin. [Internet]. 2011 [cited 2011 August 20]. Available from:
<http://www.iopus.com/imacros/firefox/>.
- 65 Google Inc. Advanced (Java) Uploader for resumable uploads. [Internet]. 2011 [cited 2011 August 9]. Available from:
<http://www.google.com/support/youtube/bin/answer.py?answer=185316>.

10 Table of Figures

Figure 2-1 Screenshot of the web interface of a demo installation of the Invenio software.....	3
Figure 2-2: A record representation in MARC21. A record consists of fields (numbers in red color) and subfields (alphas and numbers in green colour, preceded by '\$\$') with their corresponding content. Additionally, there can be two indicators per tag (alphas and numbers in blue colour). The mapping between the content and the fields is either part of the standard defined by the Library of Congress or an internal convention defined by the institution.....	4
Figure 2-3 The same record as in Figure 2-2, formatted in MarcXML. Some parts are omitted. The MARC tags are red, the indicators blue and the subfields are green.....	5
Figure 2-4 A Simplified Invenio module overview. Some modules are omitted as they are not relevant for multimedia related records. Green modules are responsible for the ingestion of metadata and for the creation of records. Red modules are for the processing of metadata and fulltext information. Orange modules are for accessing metadata and records. Data storage pools are shown in blue colour.	6
Figure 2-5 Basis structure of the BibDoc hierarchy.	7
Figure 2-6 A picture of the BibSched (Bibliographic Task Scheduler) in the automatic scheduling mode with multiple waiting tasks and a running task.	8
Figure 4-1 ffprobe example output for one video stream within a video container.....	25
Figure 4-2 Mediainfo example output for one track within a video container.	28
Figure 4-3 Example HTTP Live Streaming M3U manifest file containing different bandwidth dependent references to other manifests.	33
Figure 4-4 Example for one of the referenced M3U manifests of Figure 4-3 containing the path to the video segments and their duration.	33
Figure 4-5 Example for an INI file storing configuration profiles for a video encoder.....	35
Figure 4-6 Example for an XML file storing configuration profiles for a video encoder.....	35
Figure 4-7 Example for a simplified configuration file using Attributes instead of Tags.	35
Figure 4-8 Example for a configuration file in JSON notation.....	36
Figure 4-9 Parsing INI in Python with ConfigParser.....	36
Figure 4-10 Parsing JSON in Python with the json module.....	37
Figure 4-11 Parsing XML with the minidom module.....	37
Figure 4-12 Basic structure of the Document, Instantiation and EssenceTrack relation in PBCore. For the leaf elements only some examples are presented.....	39
Figure 5-1 General concept for a chain of systems handling videos.....	41
Figure 5-2 First concept of an encoding infrastructure tied to Invenio based on the 'four systems' approach.....	42

Figure 5-3 An alternate concept for the encoding infrastructure, where the video master and the metadata are handled separately.	43
Figure 5-4 An alternate concept where the video master is uploaded to a file server and the processing is automatically initiated.....	43
Figure 5-5 Use-Case Diagram for the Video Workflow. Showing two human actors in the form of an end-user who submits a new video record through the Submission System and an administrator who manages the Processing System. In addition, a non-human actor is shown in the form of a Daemon that automatically launches processing jobs based on the results of the human actors.....	44
Figure 5-6 Use-Case Diagram for video consumers. The human actor only consumes video material from the Presentation System and never gets in touch with any of the processing related systems.....	45
Figure 6-1 A structural overview of the BibEncode module with its most important submodules and functions. Be aware that this is not a UML class diagram, as BibEncode is developed in a procedural style.....	47
Figure 6-2 Output of ffmpeg to the command line during the encoding of a video.....	49
Figure 6-3 Encoding status representation in the visual task manager of BibSched. The first bracket shows the overall task status. The 4th of 7 subtasks is in process. The second bracket shows a progress bar for the current subtask. The third bracket shows the sub process status of subtasks (e.g. for multi pass encoding). The right side shows the file that is currently processed.....	50
Figure 6-4 Basic workflow between the background modules during a video submission. (1) A video master is submitted to WebSubmit through the web interface. (2) WebSubmit calls BibConvert and BibUpload to inject a new video record into the database. (3) WebSubmit stores the master in a temporary file. (4) WebSubmit creates a batch job description file describing the necessary steps to generate all the required video formats for the record. The description is stored in a predefined job folder. (5) The BibEncode daemon is scheduled periodically in BibSched and analyses the job folder for new job descriptions. (6) If a new description is found, the daemon launches a BibEncode batch task with the path to the job as a parameter. The BibEncode batch tasks reads the job description and (7) calls BibDocFile to copy the master from the temporary directory to its final position in a new BibDoc folder. (8) BibEncode batch calls FFmpeg to transcode the slave versions directly into the previously created BibDoc folder. (9) After the transcoding is finished, the newly created files and their metadata are appended to the record through BibUpload.	53
Figure 6-5 Simple HTML form styled with CSS for video submissions with YouTube like fields and no advanced options. Two buttons are provided to upload a file and to submit the form.....	58
Figure 6-6 Detail of the form shown in 6-5, when the form is submitted, but mandatory fields are missing.....	58
Figure 6-7 A detail of the form shown in figure 6-5, after a video from the local file system was selected and the upload was started.	59
Figure 6-8 Detail of the form after a corrupted, non-video file or video of unsupported type has been uploaded.....	59
Figure 6-9 Detail of the form shown in figure 6-5 after the video upload was successful. The page shows a slideshow of images generated from the uploaded video. Additionally, the user can alter the aspect ratio of the uploaded video.....	60

Figure 6-10 The preview images shown in figure 6-5 will dynamically adapt in their appearance if the aspect ratio is changed. The user can also enter a custom ratio.	60
Figure 6-11 Example for a file containing extraction profiles in BibEncode with two profiles 'POSTERS' and 'SMALLTHUMBS' and C-style comments for additional explanations.	66
Figure 6-12 Example of a file containing an encoding profile for BibEncode with one profile 'MP4_720P' defined and C-style comments for additional explanations.	67
Figure 6-13 A compact batch description with mandatory parameters only.	68
Figure 6-14 A complete batch job description in JSON with extensive comments and optional parameters. The first part contains general options for all sub jobs and the video master. The seconds part contains the sub job of the batch process.	71
Figure 6-15 A batch description for updating a record with new video formats transcoded from a master video that is already appended to the record.	72
Figure 6-16 Wireframe for a YouTube like video player in Invenio.	74
Figure 6-17 Presentation of a video record in CDS. The video sources and thumbnail images are produced and hosted on the Media Archive server.	75
Figure 6-18 Visual design mock-up for a video You-Tube like video player in Invenio.	76
Figure 6-19 Actual appearance of the video player after its implementation.	78
Figure 6-20 Actual appearance of the download and popup element after its implementation in HTML, CSS and JS.	79
Figure 6-21 Option and Select Elements created by the BibFormat video record template to feed the HTML5 video player with video sources.	81
Figure 6-22: Multiple Invenio instances working together. (1) The main server, serving HTTP requests through Apache such as record submissions in WebSubmit and running BibSched for bibliographic tasks in the background. (2) A files server, serving the BibDocs and other files that are commonly used by all the instances. (3) A database server, containing the shared database between all hosts. (4) The encoding server, not running Apache, only running BibSched set to only execute BibEncode tasks. Both (1) and (4) are mounting some shared folders of (2) into their Invenio directory. Both share the same database (3), which means that all the instances of BibSched (1 + 4) are working on the same task queue.	84
Figure 7-1 The virtual machine configuration for testing BibEncode and Multinode. InvenioBuilder3 hosts an Apache webserver and a MySQL database server. This machine runs the web tier of the Invenio installation. InvenioBuilder4 does not run a webserver and shares the database with InvenioBuilder3. Both machines run BibSched but Builder4 only runs BibEncode tasks and Builder3 does only run non-BibEncode tasks. Both machines share the same filesystem on AFS.	92
Figure 7-2 Invenio-local.conf configuration file for the virtual machines.	93
Figure 7-3 BibSched running on InvenioBuilder3 with multiple running and waiting tasks. The cyan colored tasks are executed on InvenioBuilder3 or are not allowed on InvenioBuilder4.	95
Figure 7-4 BibSched running on InvenioBuilder4 with multiple tasks. The cyan colored tasks are not allowed on this machine or are running on InvenioBuilder3.	95
Figure 7-5 A concept of a video segmenter in BibEncode for HTTP Live Streaming.	100

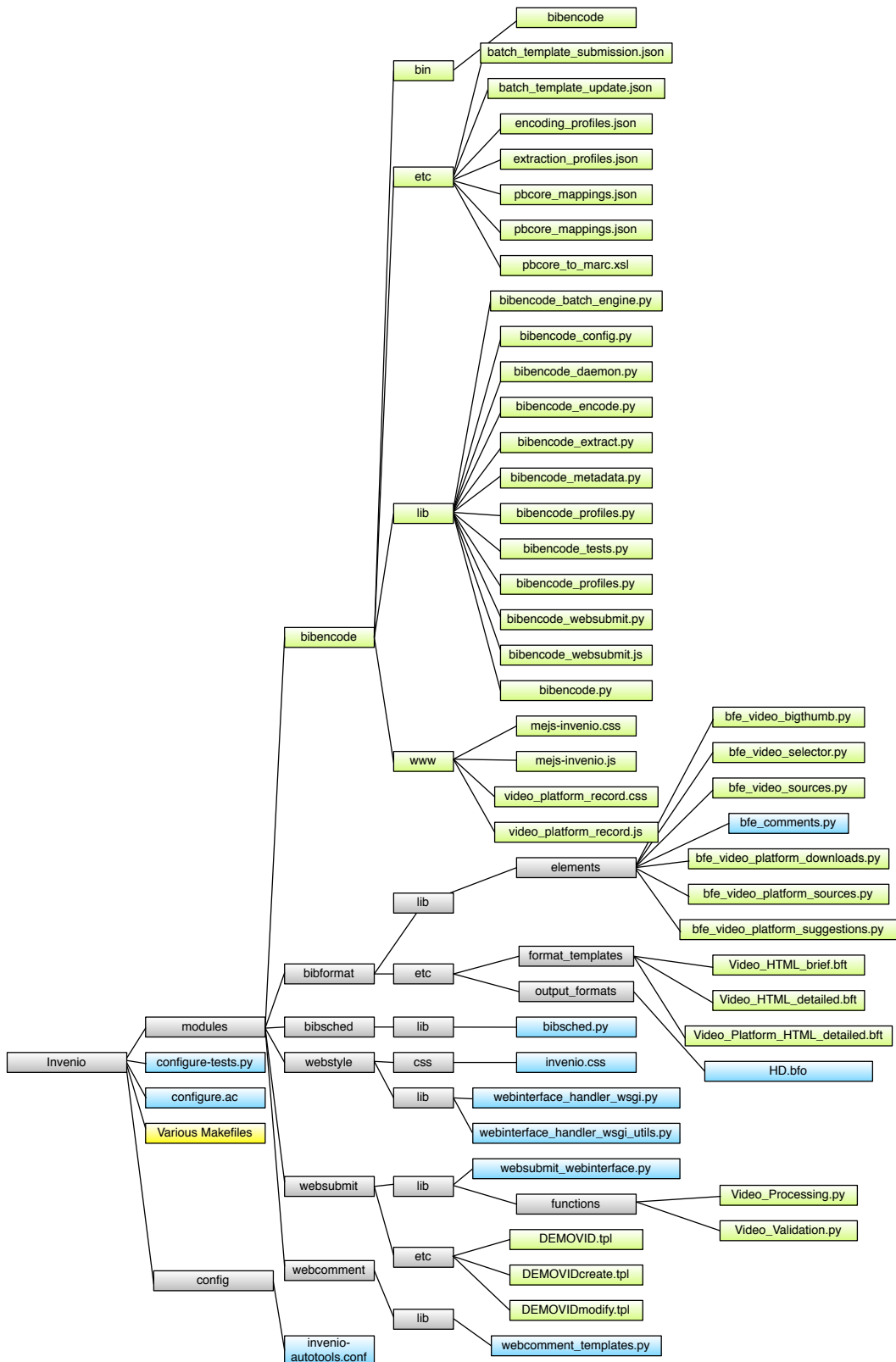
Figure 7-6 Serial, cyclic evaluation of the task queue with multiple BibSched instances.	102
---	-----

11 Table of Tables

Table 4-1: Browser support for video codecs Theora, H.264 and WebM in their latest versions (20/06/2011)	15
Table 4-2 Browser statistics by browser family based on Apache logs from CDS of May 2011, analysed with AWSTATS. The particular tables for one browser family show percentages based on the total hits in the top left table.	20
Table 4-3 HTML5 video compatibility based on the CDS statistics of May 2011. The table shows the overall compatibility including all the current HTML5 video codecs: H.264, Theora and WebM. Google Chrome is not counted towards the total percentage of H.264 because the support is going to be cut in future versions. Unknown browsers have been removed from the total number of hits to calculate the percentages.....	21
Table 4-4 HTML5 video compatibility based on StatCounter statistics of May 2011. The table shows the overall compatibility including all the current HTML5 video codecs H.264, Theora and WebM. Google Chrome is not counted for the total percentage of H.264 because the support is going to be cut in future versions. Safari includes all mobile versions with support for H.264.	22
Table 4-5 Average bitrates for different video resolutions of H.264 video with AAC audio on YouTube calculated by McFarland, P. [38].....	24

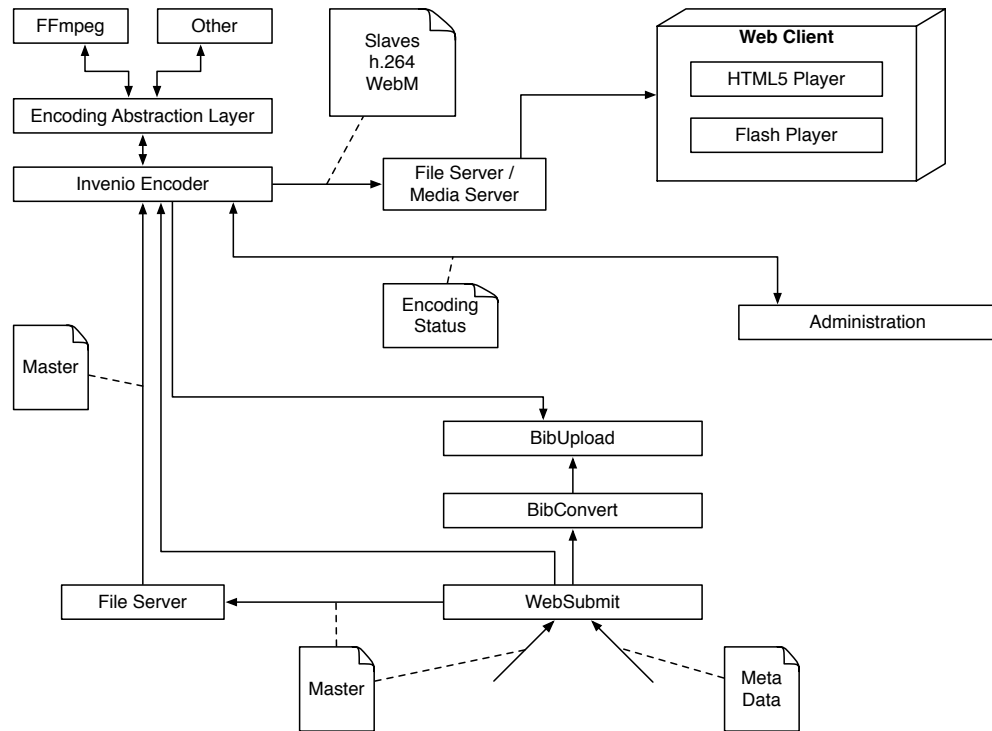
12 Appendices

Appendix A - Invenio Code Extension Overview

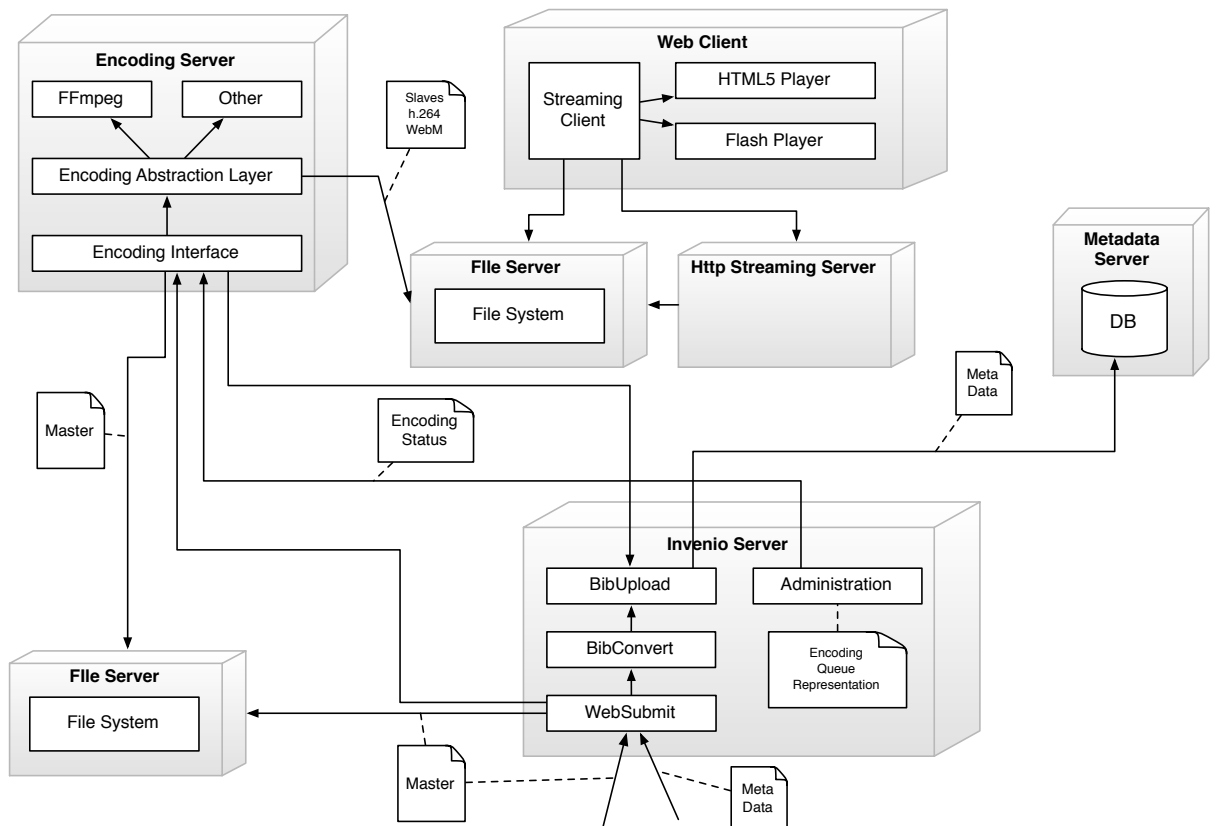


Overview of added (green) and changed (blue) files for BibEncode in the Invenio folder structure.

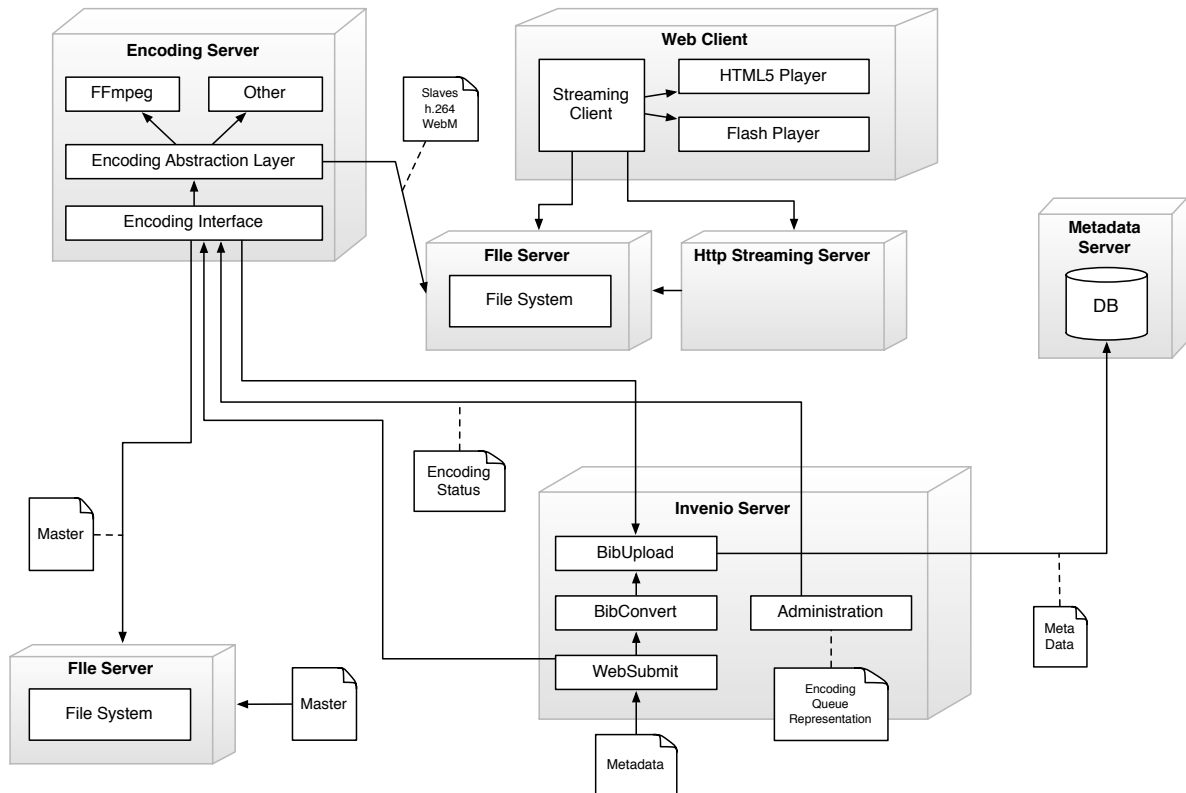
Appendix B - Early Analysis and Design Documents



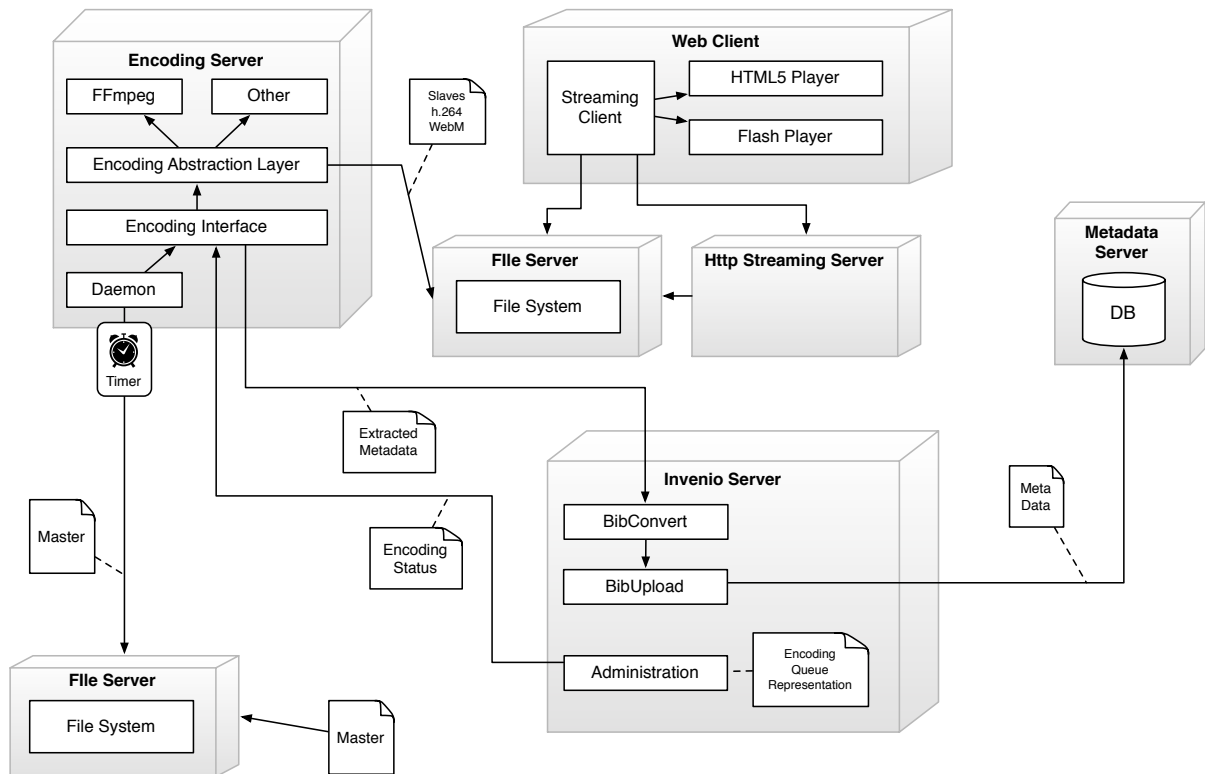
First concept of module interaction for the video workflow during the analysis and design phase.



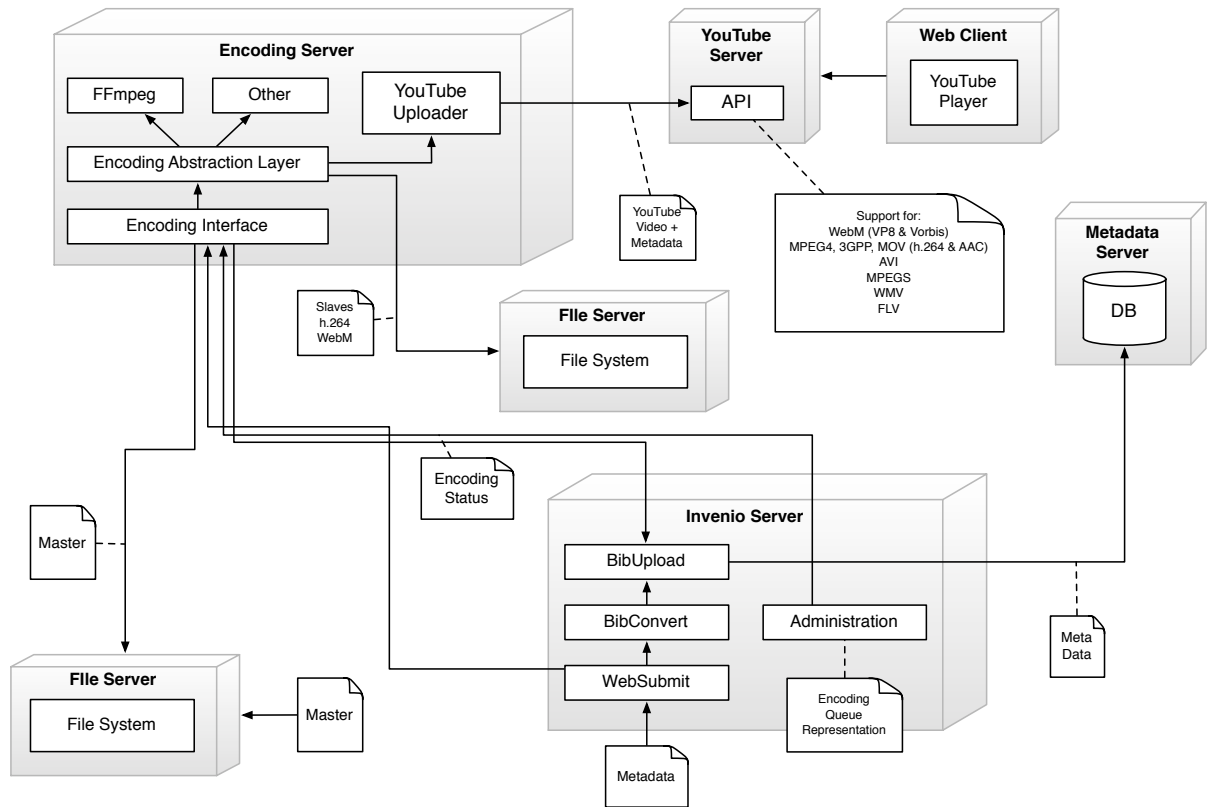
Refined concept of the module interaction shown in the last figure.



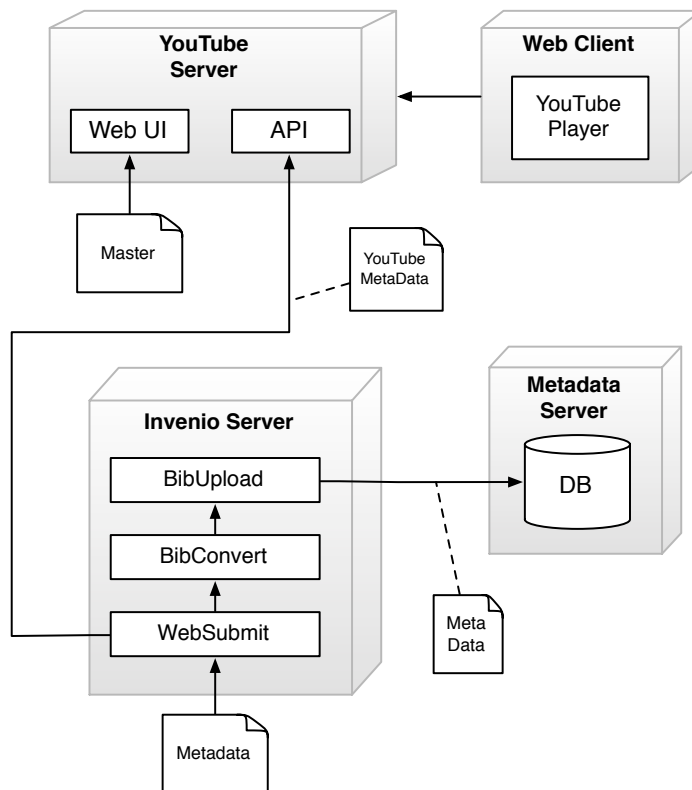
Alternate concept with manual file upload or shared file system for a video production and Invenio.



'Hot-Folder' concept with a daemon and automatic record creation.



Extended video workflow with automatic YouTube video upload module.



Completely YouTube based workflow where WebSubmit directly integrates into the YouTube API.

Appendix C – Use Case Descriptions

Use Case	Submit Video Record
Actor	Submitter
Precondition	System shows Invenio homepage.
Postcondition	A new record is created and a batch description is stored in the job storage directory.
Main Path	1: User selects 'Submit' 2: User select 'Video Submission' 3: User selects 'Submit New Record' 4: System shows the Submission Form 5: UC Upload Video 6: UC Enter Metadata 7: User selects 'Submit Video Record' 8: UC Store Job Description

Use Case	Upload Video
Actor	Submitter
Precondition	System shows the Submission Form
Postcondition	The video is uploaded to the Storage System and a Video Preview is shown to the user.
Main Path	1: User selects 'Upload Video' 2: User selects a file from local filesystem. 3: User selects 'OK' 4: UC Receive Metadata From Video 5: UC Store Master
Alternative A	5a: UC Correct Aspect Ratio

Use Case	Enter Metadata
Actor	Submitter
Precondition	System shows the Submission Form
Postcondition	Metadata is stored in the form
Main Path	1: User enters a title 2: User enters the year 3: User enters authors 4: User enters a description

Use Case	Correct Aspect Ratio
Actor	Submitter
Precondition	Video was uploaded and video preview is shown
Postcondition	Corrected aspect ratio is stored in the form
Main Path	1: User selects the appropriate aspect ratio

Use Case	Receive Metadata From Video
Actor	Submitter
Precondition	Video was uploaded and video preview is shown
Postcondition	Form is prefilled with extracted metadata from the video
Main Path	1: System tries to extract metadata from the video

Use Case	Receive Metadata From Video
Actor	Submitter
Precondition	Video was uploaded and video preview is shown
Postcondition	Form is prefilled with extracted metadata from the video
Main Path	1: System tries to extract metadata from the video

Use Case	Encode Video
Actor	Administrator
Precondition	Command Line Interface is accessed.
Postcondition	A new video encoding task is scheduled
Main Path	1: User selects encoding mode 2: User selects video Input 3: User selects encoding parameters 4: User selects video output 5: User acknowledges
Alternative A	5a: UC Alter Metadata 6a: User acknowledges

Use Case	Extract Frames
Actor	Administrator
Precondition	Command Line Interface is accessed
Postcondition	A new video extraction task is scheduled
Main Path	1: User selects extraction mode 2: User selects video input 3: User selects the number of frames to extract 4: User selects output format and location 5: User acknowledges
Alternative A	3a: User specifies a list of positions to extract frames from
Alternative B	4b: UC Substitute Filename

Use Case	Alter Metadata
Actor	Administrator
Precondition	Command Line Interface is accessed.
Postcondition	A new metadata task is scheduled.
Main Path	1: User selects metadata mode 2: User selects write submode 3: User selects input file 4: User defines metadata to write 5: User selects output file 6: User acknowledges
Alternative A	4: User selects file containing metadata

Use Case	Access Job Description
Actor	Daemon
Precondition	Job directory contains at least one job
Postcondition	The job is scheduled for execution
Main Path	1: Daemon accesses job folder 2: Daemon retrieves list of jobs 3: Daemon evaluates jobs 4: Daemon launches jobs for execution

Use Case	Create Batch Job
Actor	Administrator, Daemon
Precondition	Command Line Interface is accessed
Postcondition	A new batch job is created in the job storage directory
Main Path	1: User selects batch mode 2: User selects batch job 3: User acknowledges
Alternative A	2a: User selects batch template 3a: User selects records by collection 4a: User acknowledges
Alternative B	2b: User selects batch template 3b: Users selects records by search pattern 4b: User acknowledges

Use Case	Dump Metadata
Actor	Administrator
Precondition	Command Line Interface is accessed
Postcondition	Metadata is dumped to a file
Main Path	1: User selects input file 2: User selects type of metadata to dump 3: User selects output file 4: User acknowledges

Appendix D – Flow Chart of the Batch Processing Engine

