D2.3.1/D3.3.1/D4.3.1/D5.3.1

Enabling Technology Modules: Initial Version

Grant Agreement No:	610480
Project title:	Media in Context
Project acronym:	MICO
Document type:	D (deliverable)
Nature of document	R (report)
Dissemination level:	PU (public)
Document number:	610480/FHG, UMU, UP, SRFG, ZAIZ-I/D2.3.1/D3.3.1/D4.3.1/D5.3.1/D/PU/a1
Responsible editor(s):	Patrick Aichroth, Johanna Björklund, Kai Schlegel, Thomas Kurz, Antonio Perez
Reviewer(s):	J. Pereira
Contributing participants:	SRFG, FHG, UP, UMU, ZA
Contributing workpackages:	WP2, WP3, WP4, WP5
Contractual date of delivery:	30 April 2014

Enabling Technology Modules: Initial Version

Emanuel Berndl, Kai Schlegel, Andreas Eisenkolb, Christian Weigel, Patrick Aichroth, Johanna Björklund, Thomas Kurz, Antonio Perez, Marcel Sieland, Luca Cuccovillo

berndl@dimis.fim.uni-passau.de, schlegel@dimis.fim.uni-passau.de,
eisenkolb@dimis.fim.uni-passau.de, christian.weigel@idmt.fraunhofer.de,
patrick.aichroth@idmt.fraunhofer.de, johanna@cs.umu.se, schlegel@dimis.fmi.uni-passau.de,
thomas.kurz@salzburgresearch.at, aperez@zaizi.com, marcel.sieland@idmt.fraunhofer.de,
luca.cuccovillo@idmt.fraunhofer.de

June 8, 2015– 16:17

Abstract
This documents outlines the initial implementation of the MICO Enabling Technology Modules. The modules consist of a basic set of extractors dynamically orchestrated according the content as required for the use case evaluation.

Keyword List
Specification, Cross-Media Analysis, Metadata Publishing, Querying, Recommendations

Contents

1 Executive Summary **2**

2 Enabling Technology Models for Extractors & Orchestration Components **4**
2.1 MICO Extractors . . . 4
2.1.1 Extractor Setup . . . 4
2.1.2 Extractor Output Implementation . . . 5
2.2 Object and Animal Detection - OAD (TE-202) . . . 6
2.2.1 Deviations from and additions to D2.2.1 . . . 7
2.2.2 Specific comments . . . 7
2.3 Face detection - FDR (TE-204) . . . 8
2.3.1 Deviations from and additions to D2.2.1 . . . 8
2.3.2 Specific comments . . . 9
2.4 Temporal Video Segmentation - TVS (TE-206) . . . 10
2.4.1 Deviations from and additions to D2.2.1 . . . 10
2.4.2 Specific comments . . . 10
2.5 Audiovisual Quality - AVQ (TE-205) . . . 12
2.5.1 Deviations from and additions to D2.2.1 . . . 12
2.5.2 Specific comments . . . 12
2.6 Speech-to-text (TE-214) . . . 14
2.6.1 Deviations from and additions to D2.2.1 . . . 14
2.6.2 Specific comments . . . 15
2.7 Sentiment analysis (TE-213) . . . 16
2.7.1 Deviations from and additions to D2.2.1 . . . 16
2.7.2 Specific comments . . . 16
2.8 Chatroom cleaner (TE-216) . . . 17
2.8.1 Deviations from and additions to D2.2.1 . . . 17
2.8.2 Specific comments . . . 17
2.9 Phrase Structure Parser (TE-217) . . . 18
2.9.1 Specific comments . . . 18
2.10 Audio Cutting Detection (TE-224) . . . 19
2.10.1 Deviations from and additions to D2.2.1 . . . 19
2.10.2 Specific comments . . . 19
2.11 MICO Broker . . . 22
2.11.1 Orchestration . . . 22
2.11.2 Integration . . . 22
2.11.3 Early lessons learned, outlook . . . 23

3 Enabling Technology Models for Cross-media Publishing **25**
3.1 Introduction to Multimedia Metadata Model . . . 25
3.2 Introduction to Metadata Model API . . . 28
3.2.1 Design Decisions . . . 28
3.2.2 Technologies . . . 29
3.2.3 Anno4j . . . 29
3.2.4 API Overview . . . 30
3.2.5 Source Code . . . 31

3.3 Extending Metadata Model API . . . 31
3.3.1 Step 1: Creating the POJOs . . . 31
3.3.2 Step 2: Annotating the created POJOs . . . 33
3.3.3 Step 3: Creating the Annotation object . . . 35

4 Enabling Technology Models for Cross-media Querying 37
4.1 Introduction to SPARQL . . . 37
4.2 Introduction to SPARQL-MM . . . 42
4.3 Sparql-MM in Action . . . 47

5 Enabling Technology Models for Cross-media Recommendations 50
5.1 Recommendation architecture and technologies . . . 50
5.1.1 PredictionIO . . . 51
5.2 How to install PredictionIO . . . 51
5.3 Content-Based recommendation engine . . . 52
5.3.1 Overview . . . 52
5.3.2 How it works . . . 52
5.3.3 How to install, configure and deploy it . . . 53
5.3.4 Example . . . 54
5.4 MICO recommender system outlook . . . 57

6 MICO Platform Installation 58
6.1 VirtualBox Image . . . 58
6.1.1 Network settings . . . 59
6.2 Debian Repository . . . 60
6.2.1 Setup Debian Jessie . . . 61
6.2.2 Add MICO repository . . . 62
6.2.3 Install HDFS . . . 62
6.2.4 Install the MICO Platform . . . 66
6.2.5 Access the MICO Platform . . . 66
6.3 Compiling the MICO Platform API yourself . . . 67
6.3.1 Prerquisites . . . 67
6.3.2 Building . . . 68

List of Figures

1 Broker model . . . 22
2 Broker messaging . . . 23
3 Base concept of an annotation . . . 26
4 Annotation example of a facerecognition result . . . 27
5 Modules and general design of the multimedia metadata model . . . 28
6 Client-side RDF model generation . . . 29
7 Squebi SPARQL Editor . . . 48
8 Recommender System Architecture . . . 51
9 Import VirtualBox Image . . . 58
10 VirtualBox Network Setting . . . 59
11 Start Screen . . . 60
12 Overview page . . . 61

List of Tables

1 TE-204 implementation: HOG-based animal group blank image detection using OpenCV 6
2 TE-204 implementation: libccv face detection . . . 8
3 TE-206 implementation: Fraunhofer tvs . . . 11
4 TE-205 implementation: Fraunhofer bemvisual . . . 13
5 TE-214 implementation: ASR implementation with the Kaldi library . . . 14
6 TE-213 implementation: Implementation of sentiment analysis with Stanford CoreNLP 16
7 TE-216 implementation: Implementation of Chatroom cleaner based on CoreNLP . . . 17
8 TE-217 implementation: Implementation of the Phrase Structure parser using Stanford CoreNLP . . . 18
9 TE-224 implementation: Audio Cutting Detection via Inverse Decoder . . . 20
10 TE-224 implementation: Audio Cutting Detection via Stable Tone Analysis . . . 21
11 TE-224 implementation: Audio Cutting Detection via Microphone Discrimination . . . 21

1 Executive Summary

This documents outlines the initial implementation of the MICO Enabling Technology Modules. The modules consist of a basic set of extractors dynamically orchestrated according to the content as required for the use case evaluation. The deliverable will focus primarily on adapting existing extractors. It will include both open source as well as proprietary extractors. The document summarises the following results and progress in the project:

- First version of the prototype for orchestrating extraction components, based on D2.2.1 and D2.3.1.
- Prototype implementation of cross-media publishing in Apache Marmotta, including analysis results and media metadata representation, based on D2.3.1.
- Prototypical implementation of the cross-media query language in Apache Marmotta based on D4.2.1.
- Prototype implementation of the cross-media recommendations based on D5.2.1.

A central part of the MICO project is the service orchestration for extractors and components. The service orchestration component is expected to compute each input content item towards an appropriate individual execution plan that is then followed by the available analysis components. That high-level goal is approached in MICO following an iterative approach, where this first iteration delivers a broker with the idea of learning by experimenting, and then proposing changes and evolution towards providing a full orchestration solution by the next iteration. Multimedia analysis components typically operate in isolation as standalone applications and therefore do not consider the context of other analyses of the same media resource. We document here a multimedia metadata model in conjunction with a metadata API. The model will be introduced in section 3.1, the decision taking and a description of the API in section 3.2. Section 3.3 will cover the extensibility of the model and the API, allowing the model to be adapted for third parties and their extractors. With the introduction of the complex annotation model both annotators and humans can put media and concepts in context. The model represents information using RDF, which makes SPARQL a good candidate for an adequate retrieval mechanism / query language. Nevertheless some functionalities that are necessary for proper media asset and fragment retrieval are missing, like described in [KS14]. Here we document Multimedia Extension for SPARQL named SPARQL-MM. The extension includes mainly relation and aggregation functions for media fragments but is under continuous development so this description is just a snapshot of the current version. Recommender systems have changed the way people find information. Based on behaviour patterns and content analysis, items can be presented to the users that might be completely new to them, but match their expectations. Within MICO, the goal and opportunity is to use various metadata types, and apply recommendation algorithms to further enrich and extend metadata, creating richer models which can be used for various recommendation purposes. The underlying theme of both showcases in MICO, is the need for cross-media recommendation: Automatic and manual annotations, contextual information and user interaction data related to various media types can be used as input to the algorithms, resulting in recommendations for all relevant media types as output, thereby crossing media borders. For instance, a user preference for images with lions (which may have been identified by automatic image analysis or manual annotation, and usage information) can be used to recommend related images, documents, posts, videos or video fragments. Consequently, one of the key challenges for this domain is to define which and how to use information sources to calculate similarity as needed for a specific use case. This document also provides instructions on how to setup and run the MICO Platform. The MICO Platform

Server runs on a Linux server providing the following MICO relevant services: Apache Marmotta with contextual extensions, Broker, RabbitMQ, Hadoop HDFS server for binary content storage. For development and testing purpose, we provide a ready-to-use virtual image (see section 6.1). If you would like to install the MICO Platform on your own machine, take a look at section 6.2.

2 Enabling Technology Models for Extractors & Orchestration Components

A central part of the MICO project is the service orchestration for extractors and components. The service orchestration component is expected to compute each input content item towards an appropriate individual execution plan that is then followed by the available analysis components. That high-level goal is approached in MICO following an iterative approach, where this first iteration delivers a broker with the idea of learning by experimenting, and then proposing changes and evolution towards providing a full orchestration solution by the next iteration.

2.1 MICO Extractors

MICO extractors are software components that analyse multi media content in order to produce new content representations and meta data about that content to be fed into the MICO data model. They are one or different implementations of the Technology Enablers that had been identified and defined in the context of WP2 in the joint deliverable D2.2.1 [AKW14].

This deliverable and specifically this section will describe the used implementations in detail. It aims at giving a deeper understanding on how the extractor implementations work, how they can be parametrized and what performance can be expected. Thereby it will serve as a reference for both, future extractor development as well as the evolution of the broker (cmp. section 2.11). In addition, if there has been changes with respect to D2.2.1 we will also describe them here.

This deliverable also describes the basic process of the extractor set-up in this software release.

2.1.1 Extractor Setup

All available MICO extractors are provided as packages by the MICO Debian package repository. While some extractors are publicly available some contain closed source components and are only available through a protected repository. If you require access to this section of the repository, you need to request an account[1]. Details on how to install the platform and extractor packages can be found in section 6.

In their runtime environment MICO extractors are executed as system daemon services. The current broker implementation uses a simple mime type system in order to connect inputs and outputs of extractors and orchestrate processing. Thus, running all extractor services at once would produce unnecessary processing even of extractors that are not needed by the current processing task. Therefore we provide an easy to use shell script that start or stops specific extractor services along with their proper parameter configuration. In order to start an specific extractor configuration simply use[2]:

```
1 mico_config_extractors [extractor_config_file] start
```

After starting a specific configuration you should check the status in the MICO broker web frontend. Before starting a new extractor chain you should stop the previous one:

```
1 mico_config_extractors [extractor_config_file] stop
```

Configuration files for different show case user stories are located in the `/usr/share/mico/config` directory. Since configurations and parameter settings are under continuous development and often subject to change, please refer to `https://bitbucket.org/mico-project/extractors-public` for latest information and new configurations.

[1] Simply ask on the project mailing list for username/password: office@mico-project.eu

[2] Script is available once the platform has been installed

2.1.2 Extractor Output Implementation

This platform release provides a new API for persisting meta data(see 3.2). While this API is available directly for Java extractors, currently there is no support for C++ implementations. Therefore MICO extractors implemented in C++ use an additional extractor implemented in Java for the sole purpose of persisting the native extractor annotations (usually in a proprietary XML format) into RDF. It consumes the new content part containing the native annotation as an input and creates the RDF annotations using the new API. Since we've decided to keep the native annotations in the unstructured storage, too, this step introduces little overhead to an extractor chain.

2.2 Object and Animal Detection - OAD (TE-202)

The object and animal detection technology enabler serves for both, detecting blank image and detecting specific groups of animals. Within MICO we've decided to support a HOG-based (Histograms of Oriented Gradients) [DT05] approach as a baseline implementation. In order to make the extractor publicly available we've used OpenCV for the implementation. The extractor has been integrated using the MICO native C++ extractor API.

Table 1 TE-204 implementation: HOG-based animal group blank image detection using OpenCV

Name	TE-204_OAD_hog_detector
External dependencies	OpenCV (=>2.4.3) compiled with libpng, libjpeg (image support)
Original licenses	BSD(OpenCV), zlib/libpng license (libpng), BSD-like (libjpeg)
MICO integration license	Apache License 2.0
Input data	image (png, jpeg)
Output data	One xml file with annotations of animal regions per image according to D2.2.1 [AKW14]
RDF persistence	Persistence into the MICO data model (cmp. 3.1) via a separate Java extractor that converts the native xml annotation content item. See section 2.1.2 for details.
External Parameters	`hit_threshold` : threshold for the distance between features and SVM classifying plane ([default: required to be set]). While this is a rather "algorithmic" parameter its effect is just: the higher the less animals get detected (false negatives) but the more gets detected accurately (true positives). Usual values are in the range of $0.8 \ldots 1.5$. In future version that parameter will be transformed into a better understandable one (e.g. three stages of accuracy)
Internal Parameters	`scale0` : Coefficient of the detection window increase. ([default:1.05]). Basically affects accuracy of detection. `group_threshold` : Coefficient to regulate the similarity threshold. When detected, some objects can be covered by many rectangles. 0 means not to perform grouping ([default:2])
Additional requirements	Training model file needs to be provided (OpenCV's yaml persistence format is used). A preliminary work in progress model is shipped along with the extractor.

2.2.1 Deviations from and additions to D2.2.1

There is no deviation from D2.2.1. In this deliverable the baseline HOG implementation is provided. Within the next development phase we plan to integrate a deformable part based model (DPM) approach [Fel+10].

2.2.2 Specific comments

Detectable species / Training Model: In this first implementation of the extractor we've trained a model for hoofed animals with horns. Manual annotation of a comprehensive training set consisting of several hundreds of image for each class (animal / no animal) is a quite time consuming task. We've opted for hoofed animal since a vast number of species captured in wild by the Snapshot Serengeti project are such animals (gazelles, hartebeests. etc.). For both the HOG as well as the future DPM detector new models will be trained in further meaningful groups in order to cover the majority of the Snapshot Serengeti animal types.

RDF persistence: While the current implementation uses a "proxy" extractor written in Java in order to produce RDF annotations the future version will use the native API.

2.3 Face detection - FDR (TE-204)

In order to make the face detector publicly available we opted for the integration of an open source implementation. We chose the recent version of *libccv*, that includes a comprehensive state-of-the-art face detection algorithm based on a modified version of SCD (SURF Cascade Detector) [LZ13]. The library is written in C and we have integrated it into the C++ extractor API of MICO.

Table 2 TE-204 implementation: libccv face detection

Name	TE-204_FDR_ccv_facedetection
External dependencies	libccv 0.7 (`http://libccv.org/`), ffmpeg (video support), libpng, libjpeg (image support)
Original licenses	BSD 3-clause (libccv software), Creative Commons Attribution 4.0 International License.(libccv data models), GPLv2 (ffmpeg), zlib/libpng license (libpng), BSD-like (libjpeg)
MICO integration license	Apache License 2.0
Input data	video (mp4 container) or image (png, jpeg)
Output data	One xml file per video with annotations of face regions per frame according to D2.2.1 [AKW14]
RDF persistence	Persistence into the MICO data model (cmp. 3.1) via a separate Java extractor that converts the native xml annotation content item. See section 2.1.2 for details.
External Parameters	`size` : the smallest face size to be detected ([default:48x48 pixels]) `min_neighbors` : groups objects that intersect each other (options: [default:yes, no, min. number])
Internal Parameters	`step_through` : step size of detection ([default:4]) `interval` : number of interval images between full and half size (pyramid) ([default:5])
Additional requirements	training model file needs to be provided (sqlite3 format). A model for frontal faces is shipped with the extractor package.

2.3.1 Deviations from and additions to D2.2.1

In D.2.2.1 [AKW14] we planned to use a proprietary licensed Fraunhofer component for face detection. One reason was the superior performance in terms of speed and accuracy compared to standard open source implementations (e.g. Viola-Jones Boosted Cascades in OpenCV). With the availability of *libccv* state-of-the art accuracy is available as open source implementation at the cost of speed (see next

section).

2.3.2 Specific comments

Performance: Although the accuracy of the SCD is competitive the used implementation is not real time capable (i. e. running the analysis at the time of the video length). In a short test we've estimated approx. $8\times$ the video length on a Core 2 Duo processor, single threaded where already reduced to frame size to $480px$ width. There might be space for tuning the algorithm parameters towards better performance. *libccv* also provides a BBF(Brightness Binary Feature) based detector and model which might be employed which increases the speed by factor 2 by sacrificing accuracy. On the other hand BBF detection has been rendered obsolete by the main author of the library and may not be available in future versions.

Temporal Face Detection: Another drawback of using *libccv* is its design towards images. It does not provide any mean of tracking a face over time directly. Therefore in the first version the annotations will be on a per frame basis not having any ids. We may combine the *libccv* object tracker with the face detector in order to get temporal face tracks.

Training Model: The training model that comes with *libccv* is trained for frontal faces. If the detection of rotated or tilted faces is required, a new model needs to be trained with manually annotated data which is a quite elaborative task and may only be considered when there is no other option.

RDF persistence: While the current implementation uses a "proxy" extractor written in Java in order to produce RDF annotations the future version will use the native API. Currently we are only able to put the region information only on a per frame basis which would – in real showcase scenarios – flood the triple store with annotations when videos are used as input. If we succeed in creating face tracks from the library we consider to use smarter annotation models such as interpolated key framed regions in order to reduce the amount of data persisted in the triple store.

2.4 Temporal Video Segmentation - TVS (TE-206)

Temporal video segmentation provides detection of edited shot boundaries and key frame within these boundaries. Since there is no ready to use comprehensive open source library available for that purpose, within MICO we use the proprietary Fraunhofer tvs library. Table 3 gives the overview.

2.4.1 Deviations from and additions to D2.2.1

None.

2.4.2 Specific comments

Performance: Depending on the processing settings the library is capable of processing a video in real time or faster.

Output: The TVS extractor provides outputs of different kinds of media: annotations in structured text and/or thumbnail images. In the current MICO platform system the desired behaviour must be set via CLI parameter during extractor process start-up. In future version it might be configured during runtime using new platform functionalities.

RDF persistence: While the current implementation uses a "proxy" extractor written in Java in order to produce RDF annotations the future version will use the native API. We keep the amount of data for the native extractor annotations small by consequently removing redundant information by grouping frames that are neither shot boundaries nor key frames).

Table 3 TE-206 implementation: Fraunhofer tvs

Name	TE-206_TVS_temporalvideosegmentation
External dependencies	Fraunhofer tvs, ffmpeg (video support), ImageMagick++ (key frame output)
Original licenses	proprietary license (tvs), GPLv2 (ffmpeg), ImageMagick License
MICO integration license	Apache License 2.0 / proprietary
Input data	video (mp4 container)
Output data	One XML file per video with annotations of temporal regions for shots, scenes and key frames according to D2.2.1 [AKW14], alternatively the key frames can be delivered as images (png, jpeg)
RDF persistence	Persistence into the MICO data model (cmp. 3.1) via a separate Java extractor that converts the native xml annotation content item. See section 2.1.2 for details.
External Parameters	`tempDir` - Process writeable directory for temporary files) `outputFormat` - Output mode of the extractor: [default:XML, JSON, JPEG, PNG]) `thumbType` - Specifies for which frames thumb nails are created: [default:KEYFRAMES, SHOT_BOUNDS]) `thumbSize` - Specifies the pixel size of the longest thumbnail edge: [default:160])
Internal Parameters	`KEYFRAME_MINIMUM_DISTANCE` - min. distance of key frames [default: 5] `KEYFRAME_MAXIMUM_DISTANCE` - max. distance of key frames [default: 1000] `KEYFRAME_MINIMUM_DIFFERENCE` - similarity measure for key frames [default: 0.2f] `DOWNSCALE_WIDTH/HEIGHT` AND `FASTPROCESSING` - Speeds up processing time but may sacrifice accuracy `SHOT_DETECTION_WORKING_WINDOW/_OVERLAP` - temporal window used for shot detection decision and its overlap [default: 100/30] `KEYFRAME_EXTRACTION_WORKING_WINDOW` - temporal window used for key frame detection decision [default: 300]
Additional requirements	none

2.5 Audiovisual Quality - AVQ (TE-205)

The AVQ extractor provides measures for visual (and in later versions also auditive) quality using no-reference algorithms that is algorithms that do not need a video for comparison. We use the comprehensive Fraunhofer IDMT Broadcast Error Monitoring (BEM) library within this MICO extractor.

2.5.1 Deviations from and additions to D2.2.1

None.

2.5.2 Specific comments

Performance: Depending on the processing settings the library is capable of processing a video in real time or faster.

Output: The BEM extractor provides outputs that comprises a very low level semantic level of annotations using scalar technical measures for each frame of a video. This creates two challenges we address in MICO.

1. We will combine the single technical measures into one traffic light kind of measure we call media quality. The media quality represents a accumulated, more understandable quality measure. It is the weighted combination of selected quality features / events. In order to find the right selection and weighting we will conduct further experiments within MICO.

2. We need to reduce the amount of data when storing video annotations to the RDF model. We may target a similar approach as proposed for the face detection extractors (cmp. section 2.3) by using SVG animation kind of description for the temporal description of the quality of a video.

RDF persistence: While the current implementation uses a "proxy" extractor written in Java in order to produce RDF annotations the future version will use the native API.

Table 4 TE-205 implementation: Fraunhofer bemvisual

Name	TE-205_AVQ_bemvisual_image and TE-205_AVQ_bemvisual_video
External dependencies	Fraunhofer bem (Broadcast Error Monitoring)
Original licenses	proprietary license (bem), GPLv2 (ffmpeg)
MICO integration license	Apache License 2.0 / proprietary
Input data	video (mp4 container), image (portable pixmap)
Output data	One xml file per video with annotations of quality measures per frame according to D2.2.1 [AKW14]. For each type of quality measure a scalar value per frame is provided.
RDF persistence	Persistence into the MICO data model (cmp. 3.1) via a separate Java extractor that converts the native xml annotation content item. See section 2.1.2 for details.
External Parameters	`events` - The kind of error to be detected [BLUR, BLOCKING, RINGING, INTERLACE, FREEZE, OVEREXPOSURE, UNDEREXPOSURE, BLACKFRAME, FIELDORDER_TFF/BFF, HBLACKBARS, VBLACKBARS]. By default, all events are set to be active. `FREEZE_MIN_DURATION` - Number of frozen frames before sending a freeze event [default: 5] `BLOCKING_RESOLUTION` - Processing region size on which the blocking detection is performed.[default: 8]
Internal Parameters	`BLUR_RESOLUTION` - Processing region size on which the blur detection is performed. [default: 5] `FREEZE_THRESHOLD` - Freeze threshold to adjust the sensitivity. [default: 3.0] `INTERLACE_PREFILTER_THRESHOLD` - [default:] `INTERLACE_THRESHOLD` - Threshold to adjust the active region for interlace processing. [default: 30] `BLUR_LEVEL` Blur processing level allowing coarse blur detection. [default: 2] `RINGING_LEVEL` - Ringing processing level allowing coarse ringing detection. [default: 2] `BLOCKING_LEVEL` - Blocking processing level allowing coarse blocking detection. [default: 2]
Additional requirements	none

2.6 Speech-to-text (TE-214)

Automatic speech recognition (ASR) takes as input an audio stream with speech and outputs a transcription as plain text. ASR is useful in itself, e.g., to generate subtiles or make content indexable via text-driven approaches, but also as an intermediate step towards sentiment analysis, named entity recognition and topic classification. In the context of the MICO Showcases, ASR is relevant for the News Media scenario (US-18), to make videos searchable through keywords.

In the initial phase of the MICO project, we evaluated several open-source and proprietary ASR libraries. The outcome of the experiments are published on the project's webpage[3]. Given the results, we decided to base MICO's ASR capabilities on the C++ library Kaldi, as it outperformed the other open-source alternatives and was at the level of some of the commercial systems. Kaldi also has the advantages of being modern and extensible, and having a business-friendly license.

The major disadvantage of Kaldi is that the system supports fewer languages than the more established alternatives, but this is will likely be remedied over time, as more language-models become available. As an effort in this direction, we are currently converting language models for Arabic and Italian from the CMU Sphinx format to one suitable for Kaldi.

Table 5 TE-214 implementation: ASR implementation with the Kaldi library

Name	Kaldi (`http://kaldi.sourceforge.net`)
Original license	Apache license, version 2.0
MICO integration license	Apache license, version 2.0
External dependencies	Debian package libav-tools
Input data	Video mp4
Output data	Time-stamped transcriptions in the SRT format
RDF persistence	Currently not supported

2.6.1 Deviations from and additions to D2.2.1

- Deviation: In the initial requirements, the assumption was a focus was on American English. This has now shifted towards Italian and Arabic after input from the use-case partners.
- Addition: Improved language support to support the News Media use-case (US-18)
- Addition: Adjusted configuration to save system resources

[3] http://www.mico-project.eu/experiences-from-development-with-open-source-speech-recognition-libraries/

2.6.2 Specific comments

ASR is still in its infancy, and requires time and effort to mature. Leading commercial systems achieve an accuracy of approx. 85 percent in the best possible setting, that is, with a single male speaker working from a manuscript, with no back-ground noise and high-quality audio. Open-source systems tend to lie 10-15 percent lower on the same content. Under less favorable circumstances, for example a typical YouTube clip, the accuracy can easily drop to 50%. Despite these facts, the output can still be useful to search content by keywords, or to sort it by topic.

Another aspect worth mentioning is that ASR systems are very resource intensive. The language-models used to represent language typically requires a couple of gigabytes of disk, and the computations involved are CPU intensive. Future systems will likely have similar requirements, but this will be mitigated by faster processing kernels, higher parallelization, and lower costs of disk and memory.

2.7 Sentiment analysis (TE-213)

Sentiment analysis consists in identifying and extracting subjective opinion in written text. This form of analysis is relevant for the Zooniverse showcases (US-27,US-28,US-29,US-54,US-58), where we develop tools that automatically notify researches and draw their attention to questions, controversies, or scientifically relevant discussions in the forum.

The implementation of sentiment analysis in MICO is written in Java and based on the Stanford CoreNLP library. We chose this solution because it is extensible, supports a wide-range of natural-language processing tasks, and because of its general high quality.

Table 6 TE-213 implementation: Implementation of sentiment analysis with Stanford CoreNLP

Name	CoreNLP 3.4 (`http://nlp.stanford.edu/software/corenlp.shtml`)
Original license	GNU General Public Licence, version 3+
MICO integration license	GNU General Public Licence, version 3+
External dependencies	-
Input data	Normalized text (normalization is offered by TE-216)
Output data	Tuples of sentences and associated polarity value, represented as plain text or JSON-LD, depending on configuration
RDF persistence	Currently not supported
External Parameters	-
Internal Parameters	-
Additional requirements	-

2.7.1 Deviations from and additions to D2.2.1

TE-213 is largely unaltered compared to D2.2.1. If time allows, future versions will also take syntactical information into account, so as to improve the extractor's accuracy.

2.7.2 Specific comments

The output of TE-213 can grow quite large, because it represents each sentence in the input file together with a numerical value indicating the sentence's polarity. For this reason, the files are kept in the binary storage.

2.8 Chatroom cleaner (TE-216)

The purpose of the chatroom cleaner is to normalize textual content, e.g., from the Zooniverse user forums, to make it more suitable for MICO's textual analysis tools. The normalisation consists in e.g., removing xml-formatting, non-standard characters, superfluous white spaces, and so forth.

Like the sentiment-analysis tool (TE-213), the implementation is written in Java and based on the Stanford CoreNLP library. CoreNLP is an attractive option because it is extensible, supports a wide-range of natural-language processing tasks, and because of its general high quality.

Table 7 TE-216 implementation: Implementation of Chatroom cleaner based on CoreNLP

Name	CoreNLP 3.4 (`http://nlp.stanford.edu/software/corenlp.shtml`)
Original license	GNU General Public Licence, version 3+
MICO integration license	GNU General Public Licence, version 3+
External dependencies	-
Input data	Plain or xml/html-formatted text
Output data	Plain text
RDF persistence	Currently not supported
External Parameters	-
Internal Parameters	-
Additional requirements	-

2.8.1 Deviations from and additions to D2.2.1

Since D2.2.1, functionality for parsing has been broken out in a separate component, i.e., the phrase-structure parser.

2.8.2 Specific comments

The current version of the chatroom cleaner can handle xml formatted input. Future versions will have improved support for handling non-standard characters and other normalisation tasks.

2.9 Phrase Structure Parser (TE-217)

The purpose of the phrase structure parser is to extract syntactic information from natural language text. For each sentence in the text, the parser produces a phrase structure tree.

The implementation is written in Java and based on the Stanford CoreNLP library. CoreNLP is an attractive option because it is open source, extensible, supports a wide-range of natural-language processing tasks, and because of its general high quality.

Table 8 TE-217 implementation: Implementation of the Phrase Structure parser using Stanford CoreNLP

Name	CoreNLP 3.4 (`http://nlp.stanford.edu/software/corenlp`)
Original license	GNU General Public Licence, version 3+
MICO integration license	GNU General Public Licence, version 3+
External dependencies	-
Input data	Plain text (can handle XML markup)
Output data	XML
RDF persistence	Currently not supported
External Parameters	-
Internal Parameters	-
Additional requirements	-

2.9.1 Specific comments

The current version of the phrase structure parser provides much more information than the bare parse trees in its output. This includes tokenization, named entity recognition, etc. In future versions, this extra information should be optional.

2.10 Audio Cutting Detection (TE-224)

All extractors for Audio Cutting Detection (ACD) are based on pre-existing proprietary C++ components by Fraunhofer IDMT, which were provided as a single library. In order to improve modularity, we decided to implement and integrate within the C++ extractor API of the MICO system one component for each ACD method described in D2.1.2, Section 5.8.

The first implementation of ACD, addressing the *Inverse decoder* algorithm, is detailed in Table 9, the second implementation of ACD, based on *Stable tone analysis*, is detailed in Table 10, and the third implementation of ACD, addressing the *Microphone discrimination* method, is detailed in Table 11.

2.10.1 Deviations from and additions to D2.2.1

The three implemented extractors for TE-224 adhere to the description in D2.2.1. The only two deviation, concerning ST-ACD and MD-ACD, are the following:

- Deviation: In the initial description, ST-ACD addresses stable tones with arbitrary frequencies, but the current implementation is restricted to the analysis of the Electrical Network Frequency, i.e. 50 Hz of 60 Hz.
- Deviation: In the initial description, MD-ACD receives an arbitrary input segmentation to be verified. Due to the current API being unable to receive this external input, the extractor looks for a change of the recording device every 7 seconds.

More information concerning the rationale of the current deviation of ST-ACD is reported in section 2.10.2.

2.10.2 Specific comments

In the current implementation, the extractors for TE-224 are able to run concurrently on the same or on different input audio files. Moreover, each extractor is able to analyze more than one file at once. It is not recommended, however, to register the same extractor on two *independent* MICO platforms at once: In this very specific case, the output of the duplicated extractor may be corrupted.

The ST-ACD extractor, in general, requires to verify the *presence* of the target stable tone, before being executed. This verification, however, is yet to be modeled and included in the complex processing of the MICO Broker. At the present, we foresee three possible approaches:

1. **User verified** The user can verify the presence of the stable tone in a high resolution spectrogram, and notify the outcome to the MICO broker
2. **Semi automated** The user asks for a specific stable tone, and the system starts the analysis after verifying the presence of the stable tone in an automated fashion
3. **Fully automated** The system automatically searches for and detects candidate stable tones, and starts the analysis accordingly

While the first approach may already be implemented and included, the second and third one require some basic research before being included. The current choice is to already allow the analysis of stable tones related to the Electrical Network Frequency (50 Hz or 60 Hz), which can be considered as one of the most likely stable tones. The verification of its presence is left to the user using external tools, until the MICO work flow is better determined.

The MD-ACD extractor currently requires a pre-segmentation, which may be provided by any extractor determining locations where a change of recording device may be expected, e.g. ID-ACD and ST-ACD (TE-224), or Temporal Video Segmentation (TE-206). A future extension of MD-ACD is going to leverage the current algorithm, and to automatically determine cutting points, instead of confirming previously suspect ones as in the current version.

Table 9 TE-224 implementation: Audio Cutting Detection via Inverse Decoder

Name	Inverse Decoder ACD (ID-ACD)
Original license	Proprietary license
MICO integration license	Public MICO extractor / proprietary C++ library
External dependencies	-
Input data	One audio (wav) file. This file is supposed to be a decoded - and possibly modified - version of a previously encoded audio file
Output data	One xml file
RDF persistence	Persistence into the MICO data model (cmp. 3.1) via a separate Java extractor that converts the native xml annotation content item. See section 2.1.2 for details.
External Parameters	-
Internal Parameters	Target set of encoding schemes to be addressed, i.e. mp3, aac, mp3pro, he-aac
Additional requirements	-

Table 10 TE-224 implementation: Audio Cutting Detection via Stable Tone Analysis

Name	Stable tone analysis ACD (ST-ACD)
Original license	Proprietary license
MICO integration license	Public MICO extractor / proprietary C++ library
External dependencies	-
Input data	One audio (wav) file
Output data	One xml file
RDF persistence	Persistence into the MICO data model (cmp. 3.1) via a separate Java extractor that converts the native xml annotation content item. See section 2.1.2 for details.
External Parameters	Target analysis frequency
Internal Parameters	Allowed variance of the stable tone
Additional requirements	-

Table 11 TE-224 implementation: Audio Cutting Detection via Microphone Discrimination

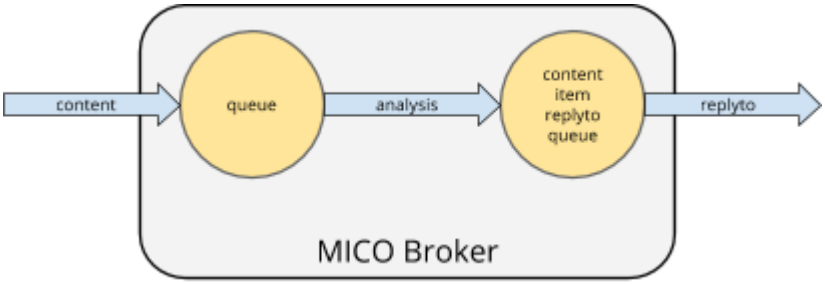
Name	Microphone discrimination ACD (MD-ACD)
Original license	Proprietary license
MICO integration license	Public MICO extractor / proprietary C++ library
External dependencies	-
Input data	One audio (wav) file
Output data	One xml file
RDF persistence	Persistence into the MICO data model (cmp. 3.1) via a separate Java extractor that converts the native xml annotation content item. See section 2.1.2 for details.
External Parameters	-
Internal Parameters	Allowed variance of the correlation of the estimated microphone frequeny responses, granularity of the detected segmentation
Additional requirements	Presence of a pre-trained model, distributed together with the closed source library

2.11 MICO Broker

2.11.1 Orchestration

The MICO Message Broker takes care of orchestrating the communication between analysis services and the analysis workflow for content. It's implemented on top of RabbitMQ[4], following the principles of AMQP[5]. The model[6] is very simple and outlined by figure 1. The main business of the broker is

Figure 1 Broker model



to orchestrate different analysis services (extractors) when content is submitted. That is implemented using two different queues: the "content item input queue" for receiving new content and a temporary queue called "content item reply to queue" where each analysis service sends its results.

The broker also provides the basic infrastructure for service registration. That requires two more queues: a "registry queue" for handling analysis service registration, and each service discovery event creates a temporary "reply-to queue" for taking care of service registration and analysis scheduling. In addition two RabbitMQ exchanges are needed: service_registry for handling new service registration events and service_discovery for sending discovery requests on startup time.

To keep the overhead minimal, especially in distributed systems, meta-data is directly stored to Marmotta and the content parts are read or written directly accessing the storage layer. It turned out, that configuring each component manually is very error prone. Therefore the necessary configuration information is globally distributed by the broker. This is accomplished using a queue named "configuration discovery", where each client can request the necessary configuration information. The reply is sent via a temporary queue, created by the questioner and sent together with the request.

2.11.2 Integration

The main idea is to decouple the extractors from their orchestration. The advantages of this approach are the freedom of the programming language the extractors are written in, and the potential distribution of the orchestration tasks. Therefore the integration of the broker in the platform is done via AMQP for communication and Protobuf[7] for serializing data. For convenience purposes the Event API, available in Java and C++, implements the basic set of instructions required by the extractors.

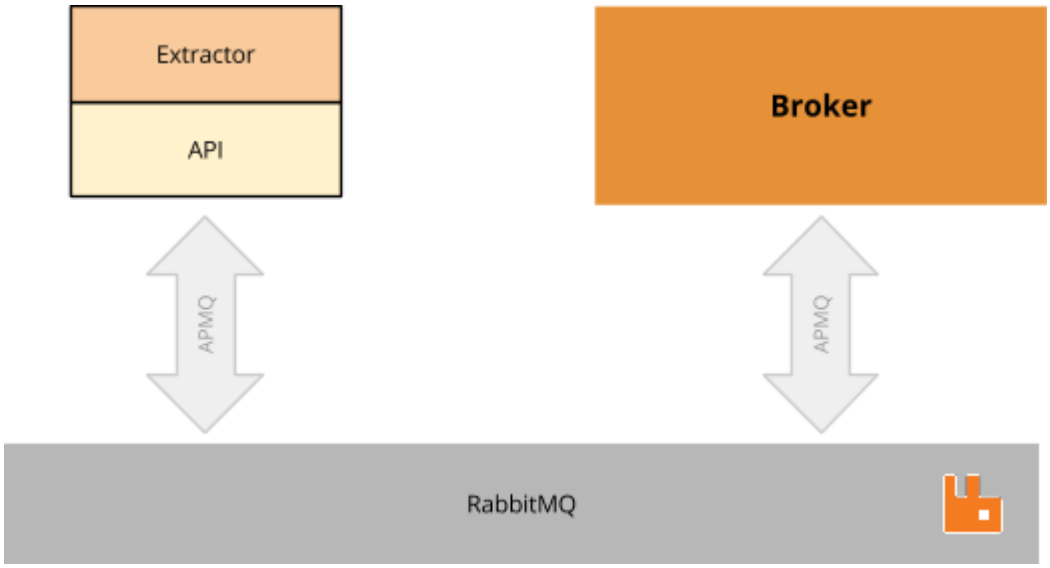
[4] https://www.rabbitmq.com/

[5] Advanced Message Queuing Protocol 1.0, https://www.amqp.org/

[6] For a better comprehension it may be required to be familiar with the AMQP Model: https://www.rabbitmq.com/tutorials/amqp-concepts.html

[7] Protocol Buffers, https://developers.google.com/protocol-buffers/

Figure 2 Broker messaging



2.11.3 Early lessons learned, outlook

The alpha version was quickly assembled around the end of the first year of the project. We have invested quite substantial work on stabilising and making the broker robust enough for being used inside the MICO Platform. Most parts of the setup actually work quite well, especially RabbitMQ as messaging infrastructure and Protobuf should be kept if possible. But there are still some critical aspects that we have realized cannot be solved by applying engineering methods on the current broker, but require a major structural re-design for the next iteration: The orchestration of the different extractors is a much more complex task than the solution we outlined by this first prototype is able to cope. Here we try to summarize the key findings in order to feed the next requirements phases:

- The assumption that each extractor generates one single output has turned out to be insufficient. During our experimentation, we identified several extractors where more than one output in different formats will be produced; an audio-video splitter as the most obvious one. The same applies to the input part; i.e., an extractor may require different inputs to be processed together. In the end, the assumption of the microservices architecture[8], where each extractor is responsible for only one single element of functionality, may also need to be revised. A new model and broker design supporting multiple inputs and outputs, extractor parametrization and possibly even several functionalities within a single component is currently investigated to address these issues.

- The simple MIME type-based system for describing what extractors are able to provide or consume is not enough. It could be extended adding qualifiers to avoid some of the issues detected, but this would only solve a minor part of the real problem. Our conclusion was that a more sophisticated language to describe the meaning of the exchanged data is required. The core element to this is a separation of syntactic extractor input and output definitions (provided per extractor) versus semantic extractor input and output (provided later, on top of syntactic descriptions, thereby

[8] http://martinfowler.com/articles/microservices.html

considering extractor interdependencies as well as content and use case specifics). This is foreseen for the next broker version, and will be supported not only by an extension of the extractor model (see aforementioned points), but also with a dedicated extractor registration and discovery service.

- It is clear that a more high-level approach to plan and execute jobs will be required, including limited support for dynamic routing based on computational conditions within extractor workflows. There is relevant work in the area of EIP (Enterprise Integration Patterns) that can be reused for solving this requirement, and we are currently investigating the integration of Apache Camel into the MICO platform for this purpose. This would be implemented by specific MICO auxiliary components to support decision-based routing based on information retrieved from the knowledge base. Moreover, a dedicated workflow planner which is able to support the manual and semi-automatic creation of workflows (using the aforementioned model and extractor registration and discovery service) is planned for future platform releases.
- Besides the aforementioned major points, there are several minor issues to be checked, including status tracking of the jobs, failure tolerance, etc.

3 Enabling Technology Models for Cross-media Publishing

Multimedia analysis components typically operate in isolation as standalone applications and therefore do not consider the context of other analyses of the same media resource. Furthermore, the context and the analysis results are also present in different proprietary formats, hence not only the lack but also the interpretation of them poses possible obstacles. As a solution to these issues, we designed a multimedia metadata model in conjunction with a metadata API. The model will be introduced in section 3.1, the decision taking and a description of the API in section 3.2. Section 3.3 will cover the extensibility of the model and the API, allowing the model to be adapted for third parties and their extractors.

3.1 Introduction to Multimedia Metadata Model

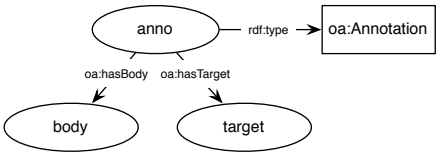
The main purpose of the MICO multimedia metadata model lies on the semantic combination of different multimedia items. As extraction and analysis processes (in general) produce a manifold of different results and output formats, a unified way of storing and requesting the content is needed. Our model will combine many different formats and open possibilities for unified querying. Furthermore, the utilisation on the other hand gets facilitated. Metadata as well as provenance can be persisted in close relation to the multimedia content. The applied use cases in combination with the vision of the platform generate different requirements that were posed to the model:

- **Cross-Multimedia Application**: The platform must be able to process different multimedia formats, such as text, audio, video, or images. Besides this fact, every output of the extractors will be of different syntactical and semantical format, producing a multitude of different results. All of this has to be funnelled into one uniform metadata model.
- **Different Annotation Targets**: Next to the diversity of extraction results, it must also be possible to express the input of an extractor in a sufficient manner. Additionally, as not every extraction process is using the full multimedia object, it must also be possible to address only a temporal or spatial part or fragment of the object.
- **Extensibility**: The platform will support a baseline of extractors combined with the corresponding vocabulary for their results. But as the platform is supposed to be used outside our own application context, the platform as well as the model need to be extensible in order to incorporate new extractors and their results.
- **Provenance**: All the aforementioned items need to be backed up with provenance information, as it is important to refer to the workflows at the level of every result. At all extraction steps, the executing instance and the timestamp will be stored. The contextual background for extractors is saved. This is necessary in order to receive a full provenance chain for extraction workflows. Additionally, versioning as well as the calculation of trust and confidence values is required.

Our model is based on the Resource Description Framework RDF[MM04], which is a common standard for linking, structuring, merging, and interchanging data. It enables for semantic interlinking and comprehensive querying through SPARQL[PS08] (see section 4 for a full description of SPAQRL and its extension SPARQL-MM). RDF (and its schema RDFS[BG14]) in combination with the ontology specification given by the Web Annotation Working Group[9] WADM (see `http://www.w3.org/TR/annotation-model/`, which is a adopted and extended version of the Open Annotation Data Model

[9] `http://www.w3.org/annotation/`

Figure 3 Base concept of an annotation



OADM[SCS13]) are the core ontologies used. This process and the other involved ontologies have been covered in the deliverable [SB14]. Ultimately, a combined ontology for the MICO use case has been created. The specification can be found at `http://mico-project.bitbucket.org/vocabs/platform/1.0/`.

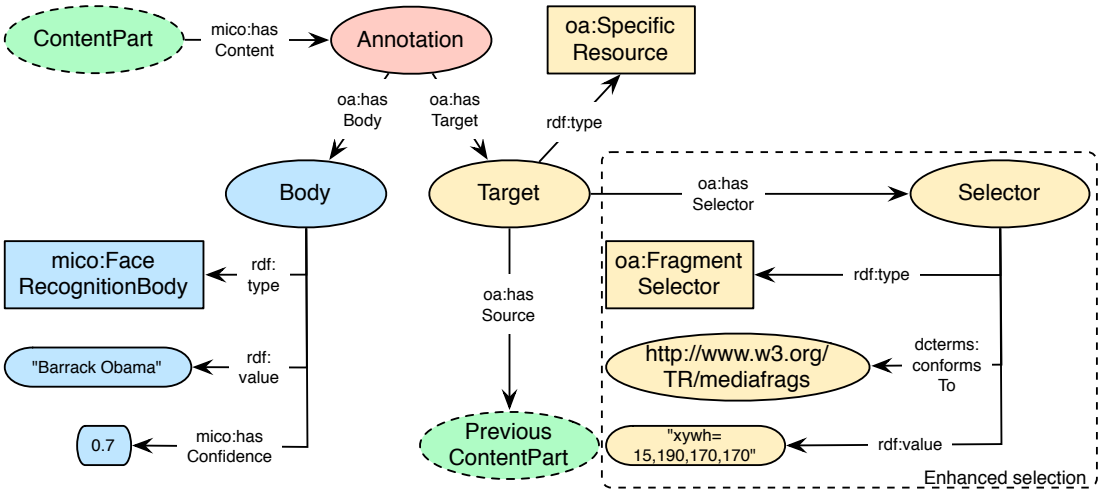
The main workflows implemented by the MICO platform deal with the analysis of media information units, in this context called **content items**, and subsequent publishing of analysis results, either for search, querying, and recommendations. Outcome is all types of structured metadata about the content and its context. **Content items** are the representation of media resources together with their context in the platform. With context we mean the collection of multiple **content parts**, typically results of analysis components with different media types, that are directly related to the main content item. In other words, a content item is a semantic grouping of information objects (**content parts**) considering a specific multimedia asset. In our context, this concept is subsumed under the name composition (see figure 5, RDF entities of this type are coloured in green).

The metadata that is created at the very end of every extraction will form an **annotation**, an entity that further describes a media item or an intermediary result. These annotations will form the core of our model and will consecutively be designed in a rich and extensible manner. By adapting those annotations to the WADM, they can find use in other contexts or use cases outside of the MICO platform. An annotation consists of three core components: the **body**, the **target**, and a generic **annotation entity** that joins both of the former entities. Figure 3 shows the basic shape of an annotation. Annotations are always of the type (`rdf:type`) `oa:Annotation`.

The **body** contains the actual content of the annotation. Every type of extractor in conjunction with its result will be broken down into RDF triples. Additionally, by supporting an own distinctive body class for the various extraction results, the bodies (and consecutively the whole annotation) gets query-able in a straightforward fashion. The **target** of an annotation specifies what media resource is the input for the given annotation, and it can be specified that only a subpart of the media item is to be selected. This selection can be made conform to different existing specifications, for example the W3C Media Fragments[Tro+12] or an SVG vector[Fer01]. An annotation entity of the type `oa:Annotation` connects the body and target via the relationships `oa:hasBody` and `oa:hasTarget` respectively. Figure4 shows an example of an annotation. It is focused on the annotation itself, as it is the central point for the metadata API, following in section 3.2. Connections to other features of the metadata model are only indicated. For further information on the composition of the extraction results of a whole workflow, refer to [SB14]. The colours of the nodes refer to the modules of the metadata model (annotation in red, body/content in blue, target/selection in yellow - see figure 5).

The annotation in figure 4 is a result of an extractor that runs a face recognition algorithm. Therefore, its body is typed (`rdf:type`) as an instance of the class `mico:FaceRecognitionBody`. The content of the extraction result contains a name (corresponding to the person that is recognised) and a confidence value (which reflects how sure the extractor is about its findings). These two parts of the result are also stored as triples connected to the body node. The relationships are `rdf:value` and

Figure 4 Annotation example of a facerecognition result



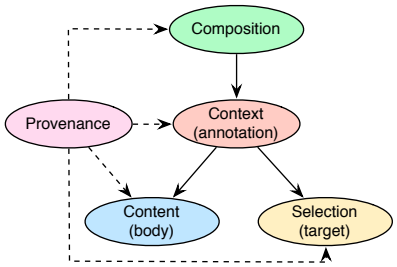
`mico:hasConfidence`. The target of the annotation refers to the input for the extraction process. In this case, the target links to its preceding content part (see [SF14] for the design of content items and content parts, [SB14] for the implementation of corresponding model features), which itself contains the picture that the recognition is done on. As the face recognition algorithm also supports the coordinates of the located face, a link to the sole media item is not enough. An extension to the target can be seen on the right side of figure 4. The specified selector (via the relationship `oa:hasSelector`) of the type `oa:FragmentSelector` specifies a rectangle of pixels around the face. This selection is conform to (`dcterms:conformsTo`) the W3C Media Fragments specification[Tro+12], and gets its value assigned via the relationship `rdf:value`.

In some of the annotations, both the body and/or the target can be ”empty”. An empty body means, that the extraction result does not have any further input that needs to be triplified. In those cases, the sole typing of the body has enough significance and allows for querying. An example for such an annotation would be a face detection. It is sufficient to type the annotation accordingly, the body does not need further specification, as all the information - **where** a face is detected - can be supported by an enhanced selection target.

Ultimately, we created an ontology that fits the MICO purpose and meets the requirements defined above in a fully satisfying way. With our model, it is possible to support a cross-multimedia platform with rich context among its results. All types of multimedia items such as text, audio, video, images, and linked data in combination with its metadata background are available. Additionally, different proprietary output formats of various extractors are enabled. By specifying selectors for annotations, the input for an extractor can be defined accurately, subparts of multimedia items can be selected. The model also allows for extensions, which enables for the addition of extractors into the MICO platform. This is enabled by the definition of new body/content constructs. New target and selector specifications can also be implemented. Provenance features for the extractors, agents, and resulting annotations, are supported in a rich fashion.

Figure 5 shows the final design of the multimedia metadata model. It has been designed around five modules. Content, context, and selection refer to body, annotation, and target of the WADM, the

Figure 5 Modules and general design of the multimedia metadata model



modules for composition has been added for MICO purposes, provenance features are adopted and extended from the WADM specification.

3.2 Introduction to Metadata Model API

The MICO platform stores and publishes all results of extractors as RDF statements using the previously explained MICO metadata model. This common data model allows semantic interlinking on the level of single resources and provides comprehensive querying with SPARQL. At the first glance, the use of Semantic Technologies like RDF an SPARQL can entail complexity for non-experts, because most of them are used to relational- or document-oriented persistence approaches. To overcome this issue, a Data Model API was envisaged to provide an easy entry point for non Semantic Web developers to persist extractor results and access the content and context of the platform workflows.

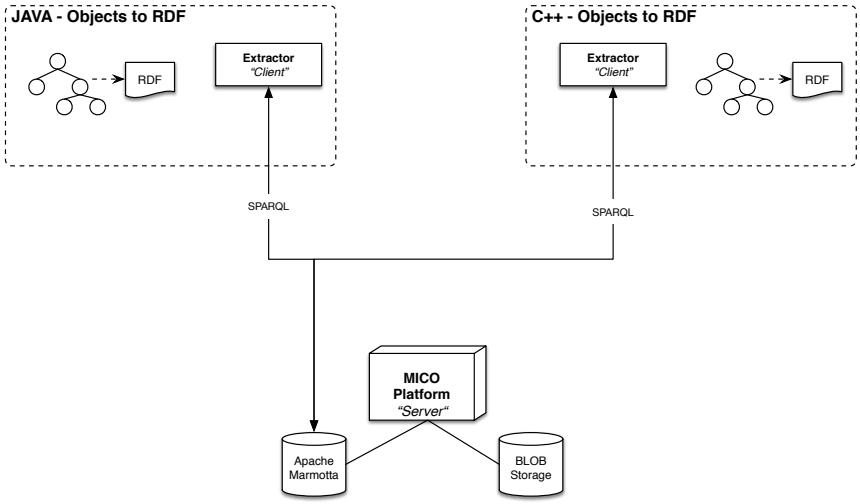
3.2.1 Design Decisions

Developers should be able to use a predefined Java/C++ API to create their content, rather than dealing with triples and SPARQL queries. From a software engineering point of view and the vision of the MICO platform, which includes a generic and future-proof integration of 3rd party extractors, the API shouldn't be limited and tailored to the workflows and extractors of the MICO Use-Case scenarios. Therefore the API has to be extensible and should allow 3rd parties to easily integrate own extractors and connect to existing workflows as well as create custom workflows, elevating the platform in a big scale and connected context.

To offer most flexibility and extensibility, the API adopted to a client-side RDF model generation (see figure 6), thereby resting on the standardised SPARQL technology and only behave as a facade instead of introducing a restricted and proprietary transport format. The model API serves as a utility module that facilitates the generation of RDF and SPARQL communication.

As a downside, this option would yield an redundant implementation in Java and C++ and consequently requires higher initial development efforts. As a consequence, we decided to mainly focus a fully functional and extensible Java Model API for 3rd parties and additionally support a lightweight C++ API for our technical partners, which is based on generic SPARQL templates. Therefore, the next sections will always refer solely to the Java API.

Figure 6 Client-side RDF model generation



3.2.2 Technologies

The objective of the model API is to allow easy-to-use generation of RDF statements for non-experts. Therefore the API is based on the Alibaba[10] library, which provides simplified RDF store abstractions to accelerate development and facilitate application maintenance. This means that developers work with well-known object-oriented POJOS (Plain Old Java Object) which are automatically mapped to RDF statements and persisted to our Triple Store (Apache Marmotta[11]). The utilisation of Java annotations facilitate maintenance, development and even extensions by 3rd parties (see section 3.3). Additionally Alibaba provides a reverse-mapping from RDF to Java objects which will form the cornerstone of a fluent interface API to access stored data form the MICO triple store without writing SPARQL queries. Besides Alibaba, several other Java libraries like Empire[12], Som(m)er[13], jennabean[14] were also investigated. It is noticeable that most libraries are no longer maintained or lack full documentation. A summary report of the investigation was published as MICO blog post[15].

3.2.3 Anno4j

Since our model is based on the Web Annotation Data Model / Open Annotation Data Model we decided to extract the MICO Model API core to build an independent Open-Source project for general purpose annotations. As a result the Anno4j project[16] was born and can be used as a library to provide programmatic access for read and write W3C Web Annotation Data Model / W3C Open Annotation Data Model from and to local/remote SPARQL endpoints. Although Anno4j will be developed independently in the future, the MICO Model API will be directly based on Anno4j and features will be merged regularly.

[10] https://bitbucket.org/openrdf/alibaba
[11] http://marmotta.apache.org/
[12] https://github.com/mhgrove/Empire/
[13] https://github.com/bblfish/sommer
[14] https://code.google.com/p/jenabean/
[15] http://www.mico-project.eu/selecting-an-rdf-mapping-library-for-cross-media-enhancements/
[16] https://github.com/anno4j/anno4j

Detailed information about Anno4j, and therefore about the implementation of the MICO Model API, can be seen directly on the GitHub page of the project at `https://github.com/anno4j/anno4j`.

3.2.4 API Overview

The following section gives an overview over the MICO API and how to use it. It can be used for extractor services to create metadata about analysis results. The amendments extend in particular the already available *MICO Persistence API* which is responsible for creating, accessing, and updating content items and content parts. Basically, the `Content` class was enriched with the method `createAnnotation` to create and persist model annotations. The overloaded method `createAnnotation` can be called with up to four parameters, where the first three parameters are mandatory:

1. An implementation of a (com.github.anno4j.model) `Body` object containing the actual body content of the annotation
2. An instance of the parent (eu.mico.platform.persistence.model) `Content` object to model the target of the annotation
3. An instance of a (eu.mico.platform.persistence.metadata) `MICOProvenance` object containing provenance information about the extractor service
4. (optional) A implementation of a (com.github.anno4j.model) `Selector` object describing a more detailed selection of the annotation

After invoking the `createAnnotation` method the ContentPart itself creates the model annotation, generates a corresponding SPARQL update query, and automatically writes the result to the MICO triple store. The current model API includes a set of predefined implementations for Body, Selection and Provenance components, regarding the identified extractor services in specification D3.2.1. These implementation are bundled in the packages `eu.mico.platform.persistence.impl.BodyImpl`, `eu.mico.platform.persistence.impl.SelectorImpl` and `com.github.anno4j.model.impl` and contain for example implementations like `AVQBody`, `FaceDetectionBody`, `FaceRecognitionBody`, `LowLevelFeatureBody`, `FragmentSelector` or `SVGSelector`. Listing 1 shows an overall example how to create a face recognition annotation using predefined Java classes and automatically generate and persist an annotation model using the just introduced Model API.

Listing 1: Creating a new model annotation in a extractor service

```
1  // Create Body for a recognised face with a confidence value
2  FaceRecognitionBody body = new FaceRecognitionBody("Barrack Obama", 0.7);
3
4  // Specifying the coordinates of the detected face (xCor,yCor, width, height)
5  FragmentSelection selection = new FragmentSelection(15, 190, 170, 170);
6
7  // Provenance information about the extractor itself
8  MICOProvenance prov = new MICOProvenance();
9  prov.setExtractorName("http://www.mico-project.eu/extractors/FaceRecExtractor");
10 prov.setRequires("text/xml");
11 prov.setProvides("application/rdf-model");
12
13
```

```
14 // Create a new content part for the analysis result and create a new annotation with all
       components
15 Content cp = contentItem.createContentPart();
16 cp.createAnnotation(body, parentContentPart, prov, selection);
```

3.2.5 Source Code

As Deliverable D3.3.1 is a software deliverable, the actual source code and implementation is not a supplementary deliverable, but also part of the overall platform deliverable.

3.3 Extending Metadata Model API

The presented multimedia metadata model in combination with the API allows developers of multimedia extractors to easily connect them to a given MICO platform. This enables the results to be included into a contextual background, integrating the process into given workflows allows for even more flexible and rich outcomes. The utilisation of the API is designed in an easy-to-use manner. When integrating a new extractor, the developer does not have to deal with any RDF, SPARQL, composition, or provenance details. Considering figure 5, "only" the content module and the selection module (to a small degree) has to be extended by the extractor owner. An own body type, complemented with its specific attributes and values, has to be defined. When the input multimedia item is taken in its entirety, the target does not have to be altered. Otherwise, the simple addition of a selector uses a subpart of the multimedia item.

In the following, a step by step example will guide through the process of implementing the necessary steps to extend an extractor to work with the platform. Therefore the RDF graph from Figure 4 will be used as example. An important AliBaba feature is worth noting: It is necessary to provide an empty `''META-INF/org.openrdf.concepts''` in the root directory (or JAR) of the annotated classes to be able to persist the data. If this file does not exists the API will throw an exception, telling you to create this file.

3.3.1 Step 1: Creating the POJOs

In the first step we want to create the particular Java classes, for the FaceRecognitionBody and the FragmentSelector. At the beginning we create an ordinary Java object (POJO) for the Body as shown in Listing 2. Comparing the given code with the example graph, it can be seen, that the FaceRecognitionBody class defines a String value for the recognised person (line 6) and a Double value for the confidence (line 11), according to the RDF graph from Figure 4. In line 13 we generated the default constructor. This is not important for persisting the class, but if we want to query the objects later on, AliBaba needs the default constructor to automatically map the queried triples to a specific object. Besides the FaceRecognitionBody a POJO for the FragmentSelector is required for the given example, as shown in Listing 3.

Listing 2: POJO - FaceRecognitionBody

```
1 public class FaceRecognitionBody {
2
3   /**
4    * The name of the person that was detected
5    */
6   private String detection;
7
```

```
    /**
     * Confidence value for the detected face
     */
    private Double confidence;

    public FaceRecognitionBody() {
    }

    public FaceRecognitionBody(String detection, Double confidence) {
        this.detection = detection;
        this.confidence = confidence;
    }

    public String getDetection() {
        return detection;
    }

    public void setDetection(String detection) {
        this.detection = detection;
    }

    public Double getConfidence() {
        return confidence;
    }

    public void setConfidence(Double confidence) {
        this.confidence = confidence;
    }
}
```

Listing 3: Listing 2: POJO - FragmentSelector

```
public class FragmentSelector {

    // The x−coordinate of the fragment
    private int xCoord;

    // The y−coordinate of the fragment
    private int yCoord;

    // The width of the fragment
    private int width;

    // The height of the fragment
    private int height;

    // String representation of the x−coordinate, the y−coordinate, the width and the height
    private String fragmentData;

    private final String CONFORMS_TO = "http://www.w3.org/TR/mediafrags";

```

```
20    public FragmentSelector() {
21    }
22
23    public FragmentSelector(int xCoord, int yCoord, int width, int height) {
24       this.xCoord = xCoord;
25       this.yCoord = yCoord;
26       this.width = width;
27       this.height = height;
28       this.fragmentData =
29          "xywh=" + String.valueOf(xCoord) + "," +
30          String.valueOf(yCoord) + "," +
31          String.valueOf(width) + "," +
32          String.valueOf(height);
33    }
34
35    /**
36     * Getter for the String representation of the x-coordinate,
37     * the y-coordinate, the width and the height in the form:
38     * <p/>
39     * "x-coordinate, y-coordinate, width, height"
40     *
41     * @return
42     */
43    public String getFragmentData() {
44       return fragmentData;
45    }
46
47    // Some setter and getter...
48 }
```

3.3.2 Step 2: Annotating the created POJOs

The next step is to take the FaceRecognitionBody from Listing 2 and annotate it using the *@Iri* annotation which is provided by AliBaba. Therefore, all we have to do is to add this annotation to all class tags and fields that should be stored in the repository. Considering the example from Figure 4, the first thing we have to specify is the triple for the *rdf:typef* of the body:

<http://mico-project.eu/exampleBody> rdf:type mico:FaceRecognitionBody

Assigning the type to the object is rather simple, all we have to do is to add the `@Iri` annotation with the type String directly above the class tag, as shown in Listing 4 (line 1). In line 7 and line 13, the other two triples (`<http://mico-project.eu/exampleBody> rdf:value "some person"` , `<http://mico-project.eu/exampleBody> mico:hasConfidence some_confidence_value`) are annotated by using the annotation and the respective String values for the predicates.

Another noticeable thing is the fact, that the FaceRecognitionBody extends the abstract class (com.github.anno4j.model) `Body`. There is also a corresponding abstract class for selections called (com.github.anno4j.model) texttSelector. Extending the particular classes with these abstract classes prevents AliBaba to treat these objects as blank nodes when persisting.

Listing 4: Annotated FaceRecognitionBody

```java
@Iri(Ontology.FACE_RECOGNITION_BODY_MICO)
public class FaceRecognitionBody extends Body {

    /**
     * The name of the person that was detected
     */
    @Iri(Ontology.VALUE_RDF)
    private String detection;

    /**
     * Confidence value for the detected face
     */
    @Iri(Ontology.HAS_CONFIDENCE_MICO)
    private Double confidence;

    public FaceRecognitionBody() {
    }

    public FaceRecognitionBody(String detection, Double confidence) {
        this.detection = detection;
        this.confidence = confidence;
    }
    ...
}
```

Listing 5: Annotated FragmentSelector

```java
@Iri(Ontology.FRAGMENT_SELECTOR_OA)
public class FragmentSelector extends Selector {

    // The x−coordinate of the fragment
    private int xCoord;

    // The y−coordinate of the fragment
    private int yCoord;

    // The width of the fragment
    private int width;

    // The height of the fragment
    private int height;

    // String representation of the x−coordinate, ...
    @Iri(Ontology.VALUE_RDF)
    private String fragmentData;

    @Iri(Ontology.CONFORMS_TO_DCTERMS)
    private final String CONFORMS_TO = "http://www.w3.org/TR/mediafrags";

    public FragmentSelector() {
    }
```

```
25    ...
26 }
```

3.3.3 Step 3: Creating the Annotation object

In the third and last step we want to use use the annotated FaceRecognitionBody and the FragmentSelector object to create an Annotation object, using the MICO Metadata Model API. Listing 1 shows how a possible implementation can look like. First of all, we create new instances of these two objects and fill them with the data from Figure 4 (line 2 and 5). After that, we create a Provenance object with some example values, like the required input format or the name of the extractor as URL (line 8 - 11). Immediately afterwards, we create a new Content object by invoking the `createContentPart` function of the contentItem. Now we can call the `createAnnotation` function. This method will automatically create the Annotation object and its triples. In addition, the `createAnnotation` function will also write the Annotation object to the underlying triple store. An example, how to use this function is shown in Listing 1.

As mentioned, executing the `createAnnotation` function will result in persisting the Annotation object and its components. The actual RDF statements, which are persisted on the MICO triple store, are presented in Figure 4 and are conform to the described MICO Metadata Model from subsection 3.1. Listing 6 shows how this result would look like using the Turtle syntax.

Listing 6: Example Annotation object using the Turtle syntax

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX mico: <http://www.mico-project.eu/ns/platform/1.0/schema#>
3  PREFIX oa: <http://www.w3.org/ns/oa#>
4  PREFIX dcterms : <http://purl.org/dc/terms/>
5
6  <http://www.mico-project.eu/exampleAnnotation>
7     rdf:type <http://www.w3.org/ns/oa#Annotation> ;
8     <http://www.w3.org/ns/oa#annotatedAt> "2015-06-03 09:52:28.65" ;
9     <http://www.w3.org/ns/oa#hasBody> <http://www.mico-project.eu/exampleBody> ;
10    <http://www.w3.org/ns/oa#hasTarget> <http://www.mico-project.eu/exampleTarget> ;
11    <http://www.w3.org/ns/oa#annotatedBy>
          <http://www.mico-project.eu/exampleProvenance> .
12
13 <http://www.mico-project.eu/exampleBody>
14    rdf:type mico:FaceRecognitionBody ;
15    rdf:value "Barrack Obama";
16    mico:hasConfidence "0.7"^^xsd:double .
17
18 <http://www.mico-project.eu/exampleTarget>
19    rdf:type <http://www.w3.org/ns/oa#SpecificResource> ;
20    <http://www.w3.org/ns/oa#hasSource>
          <http://www.mico-project.eu/exampleContentItem> ;
21    oa:hasSelector <http://www.mico-project.eu/exampleSelector> .
22
23 <http://www.mico-project.eu/exampleSelector>
24    rdf:type oa:FragmentSelector ;
25    dcterms:conformsTo <http://www.w3.org/TR/mediafrags> ;
26    oa:hasSource "xywh=15,190,170,170" .
```

```
27
28 <http://www.mico-project.eu/exampleProvenance>
29    rdf:type <http://www.w3.org/ns/prov/SoftwareAgent> ;
30    foaf:name <http://www.mico-project.eu/extractors/FaceRecExtractor> ;
31    mico:provides "application/rdf-model" ;
32    mico:requires "text/xml" .
```

4 Enabling Technology Models for Cross-media Querying

With the introduction of the complex annotation model both annotators and humans can put media and concepts in context. The model represents information using RDF, which makes SPARQL a good candidate for an adequate retrieval mechanism / query language. Nevertheless some functionalities that are necessary for proper media asset and fragment retrieval are missing, like described in [KS14].
In this chapter we give an introduction to the Multimedia Extension for SPARQL named SPARQL-MM. The extension includes mainly relation and aggregation functions for media fragments but is under continuous development so this description is just a snapshot of the current version. An up-to-date information of all functions and it's underlying logic can be found on `http://github.com/tkurz/sparql-mm`.
Before we introduce SPARQL-MM we give an overview on the SPARQL 1.1 specification [HS13] in a very basic tutorial in section 4.1. In section 4.2 we give an overview on all the functions, which is just a wrap up of the all functions specified in [KS14]. Furthermore we provide a hands-on on section 4.3 where we introduce the SPARQL query interface Squebi as well as a complex example query.

4.1 Introduction to SPARQL

As the standard mentions "RDF is a directed, labeled graph data format for representing information in the Web. RDF is often used to represent, among other things, personal information, social networks, metadata about digital artifacts, as well as to provide a means of integration over disparate sources of information.This [The SPARQL, editor's note] specification defines the syntax and semantics of the SPARQL query language for RDF. The SPARQL query language for RDF is designed to meet the use cases and requirements identified by the RDF Data Access Working Group." [HS13]. In this part we introduce SPARQL as a query language for RDF and do not consider their data manipulation functions that has been introduces in the newest standard version. Under this conditions, a SPARQL query can be separated in the several construction parts, that are already outlined in [Kur14a]:

PREFIX Prefixes allow to shorten URLs. They are optionally defined on the top of a SPARQL query.

PROJECTION This block represents the projection part of the language. SPARQL allows 4 different kind of projection: SELECT, CONSTRUCT, DESCRIBE, and ASK.

DATASET This block allows to specify the context(s) in which the query is evaluated.

SELECTION This block (WHERE) may contain triple patterns, optional clauses, existence checks and filters.

LIST_OPS This block allows result ordering (ORDER BY) and segmentation (OFFSET,LIMIT).

AGGREGATION This block allows result aggregation (GROUP BY, HAVING).

The type of result of a SPARQL query depends on the projection type. In case of a SELECT (by far the mostly used type) it is a table, whereby every projected variable relates to one column. The set of all rows is the set of all possible bindings that fulfill the query. If the projection is of type CONSTRUCT or DESCRIBE, the result is RDF. In case of ASK the result is a boolean value, that is true, if at query has at least one binding, false otherwise.
In the following part we give examples for almost all constructs and describe their function based on a simple test example:

```
@prefix mxo: <http://mico-project.org/sparqlmm/examples/onthology/> .
@prefix mxd: <http://mico-project.org/sparqlmm/examples/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix schema: <http://schema.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

mxo:Lion a rdf:Class ;
rdfs:label "Lion"@en, "Löwe"@de .

mxd:Simba a mxo:Lion ;
schema:name "Simba" ;
mxo:sex mxo:Male .

mxd:Scar a mxo:Lion ;
schema:name "Scar" ;
mxo:sex mxo:Male .

mxd:Mufasa a mxo:Lion ;
schema:name "Mufasa" ;
mxo:sex mxo:Male ;
mxo:has_partner mxd:Sarabi .

mxd:Nala a mxo:Lion ;
schema:name "Nala" ;
mxo:sex mxo:Female .

mxd:Sarabi a mxo:Lion ;
schema:name "Sarabi" ;
mxo:sex mxo:Female ;
mxo:has_partner mxd:Mufasa .
```

The graph (expressed with TURTLE[17] syntax) contains the description and relation of animals using two different ontologies. The tutorial is inspired by the Jena basic SPARQL tutorial. [18]

Query 1: Find any instance of a lion

This is a very basic query where we want to return anything that has the relation `rdf:type ex:Lion`. Like in TURTLE a type test can also be short-cutted by using the keyword `a`, so both listed queries return the same result. In the evaluation process the variables in the query are bound so the result (list) is the cross product of all possible bindings.

```
PREFIX mxo: <http://mico-project.org/sparqlmm/examples/onthology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?lion WHERE {
  ?lion rdf:type mxo:Lion .
}
```

[17] `http://www.w3.org/TR/turtle/`
[18] `https://jena.apache.org/tutorials/sparql.html`

```
7
8  SELECT ?lion WHERE {
9    ?lion a mxo:Lion .
10 }
11
12 ==========
13 ?lion
14 ----------
15 mxd:Simba
16 mxd:Scar
17 mxd:Mufasa
18 mxd:Nala
19 mxd:Sarabi
```

It has to be mentioned, that prefixes has to be defined vor every SPARQL query. You can look up all the used prefixes (and more) under `http://prefix.cc/` which is also used by the Squebi Autocompletion.

Query 2: Select all lion names

In this query we combine two triple patterns that are bound together via variable `?x`. If two patterns starts with the same variable of resource they can be combined (like in TURTLE syntax), so again both queries deliver the same result.

```
1  PREFIX mxo: <http://mico-project.org/sparqlmm/examples/onthology/>
2  PREFIX schema: <http://schema.org/>
3
4  SELECT ?name WHERE {
5    ?x a mxo:Lion .
6    ?x schema:name ?name .
7  }
8
9  SELECT ?name WHERE {
10   ?x a mxo:Lion ;
11       schema:name ?name .
12 }
13
14 ==========
15 ?name
16 ----------
17 "Simba"
18 "Scar"
19 "Mufasa"
20 "Nala"
21 "Sarabi"
```

Query 3: List all lions that have a name starting with S (upper or lower case)

For this query we use a regular expression filter. Filters in SPARQL start with the keyword FILTER and are used for testing, which means that they are evaluated to a boolean value. Standard SPARQL supports many filter operations like boolean and mathematical functions, comparison methods, type testing (e.g.

isURI), accessors for literals (e.g. lang) and functions (e.g. regex). An exhausting description of all filters can be found in the standard. Filter functions build one of the extension points for SPARQL (that is used by SPARQL-MM). The regex function from the example is defined as `regex(value v,string p, string f)`, whereby v is the value to test, p is the regex pattern and f is a flag (that e.g. allows to filter in-case-sensitive).

```
PREFIX mxo: <http://mico-project.org/sparqlmm/examples/onthology/>
PREFIX schema: <http://schema.org/>

SELECT ?lion WHERE {
 ?lion a mxo:Lion ;
       schema:name ?name .
 FILTER regex(?name, "^s", "i")
}

==========
?lion
----------
mxd:Simba
mxd:Scar
mxd:Sarabi
```

Query 4: Find the german label of the species Simba belongs to

For this query we use a SPARQL FILTER, too. To access the language value of the literal we use an accessor function `lang(literal)`. The language similarity function `langMatches(value,string)` is used to compare language strings. This function is should be used for language testing because in comparison to simple string equals (=) it is robust against language type facets like e.g. "de-by", "en-au", etc.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mxd: <http://mico-project.org/sparqlmm/examples/resource/>

SELECT ?label WHERE {
 mxd:Simba a ?type .
 ?type rdfs:label ?label .
 FILTER langMatches( lang(?label), "de")
}

==========
?label
----------
"Löwe"
```

Query 5: Find all lionesses and optionally their partner

For this query we use an OPTIONAL clause. It is embedded within the WHERE clause and allows optional bindings (bindings that can be empty in the result set). In OPTIONAL clauses any combination of statements (like Triple Patterns, Filters, Existence checks, etc.) that is allowed within a WHERE clause is accepted. It is also possible to use more the one OPTIONAL block within one WHERE clause.

```
PREFIX mxo: <http://mico-project.org/sparqlmm/examples/onthology/>

SELECT ?lioness ?partner WHERE {
 ?lioness a mxo:Lion ;
        mxo:sex mxo:Female .
 OPTIONAL {
  ?lioness mxo:has_partner ?partner .
 }
}

======================
?lioness   | ?partner
----------------------
mxd:Nala   |
mxd:Narabi | mxd:Mufasa
```

Query 6: Return all lions ordered ascending by sex and descending by name

Ordering in SPARQL can be done by using an the ORDER BY block. For ordering it is possible to use variable values that are defined but not necessarily bound in the SELECT block. The default comparison operator in SPARQL is based on ascending string ordering. Additionally to variables SPARQL also allows to use functions in the ORDER BY clause.

```
PREFIX mxo: <http://mico-project.org/sparqlmm/examples/onthology/>
PREFIX schema: <http://schema.org/>

SELECT ?lion WHERE {
 ?lion a mxo:Lion ;
      mxo:sex ?sex ;
      schema:name ?name .
}
ORDER BY ASC(?sex) DESC(?name)

==========
?lion
----------
mxd:Sarabi
mxd:Nala
mxd:Simba
mxd:Scar
mxd:Mufasa
```

It is obvious that SPARQL provides a much broader functionality (e.g. EXISTS operator, property paths, etc.) that we will not introduce here for the matter of simplicity. For further reading we suggest the tutorial of the W3C working group *SPARQL by example* [19] and the official W3C recommendation[20].

[19] http://www.cambridgesemantics.com/semantic-university/sparql-by-example

[20] http://www.w3.org/TR/sparql11-query/

4.2 Introduction to SPARQL-MM

In this section we describe the current state of SPARQL-MM (version 1.0) including spatio-temporal accessor, relation and aggregation functions. SPARQL-MM has been presented to the scientific community within the Linked Media workshop at the World Wide Web Conference 2015 [KSK15]. To get a better understanding we give an example of a SPARQL-MM query in the next section and recommend the presentation slides of the workshop[21]. Currently SPARQL-MM supports MediaFragments and MediaFragmentURIs to identify spatio-regional fragments. In further versions we will extend it to SVG Selectors and/or to an extended version of Media Fragment URIs, which is currently under discussion in the W3C working group [22].
The SPARQL-MM function set follows the specifications in D4.2.1-SPEC (Table 3). The base URI for the SPARQL-MM vocabulary is `http://linkedmultimedia.org/sparql-mm/ns/1.0.0/function#`. When abbreviating terms the suggested prefix is `mm`. Each function in this function set has a URI constructed by appending a term name to the vocabulary URI. For example: `http://linkedmultimedia.org/sparql-mm/ns/1.0.0/function#getBoxArea`. There are machine readable function description using *SPARQL Extension Description Vocabulary*[23] e.g. in RDF/XML[24]. In this section we give only a very short, human readable description of all functions. A theoretical foundation can be found in [KS14].

Spatial Relations

Name	mm:intersects
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true if p1 has at least one common point with p2, else false.

Name	mm:within
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true if p1.shape contains all points of p2.shape an p1.shape.edge has not point in common with p2.shape.edge, else false.

Name	mm:above
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true p1 if is above p2 (based on model m), else false.

Name	mm:below
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true p1 if is below p2 (based on model m), else false.

Name	mm:coveredBy
Properties	*SpatialEntity*, *SpatialEntity*
Description	is the inverse function to covers.

[21] `http://slideshare.net/thkurz1/www2015-lime-sparqlmm`
[22] `https://lists.w3.org/Archives/Public/public-media-fragment/2015May/0003.html`
[23] `http://www.ldodds.com/schemas/sparql-extension-description/`
[24] `https://raw.githubusercontent.com/tkurz/sparql-mm/master/ns/1.0.0/function/index.rdf`

Name	mm:covers
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true if all points of p1.shape are points of p2.shape, else false.

Name	mm:crosses
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true if p1.shape and p2.shape have common points and p1.shape.edge and p2.shape.edge has common points, else false.

Name	mm:leftAbove
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true p1 if is left above p2 (based on model m), else false.

Name	mm:leftBelow
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true p1 if is left below p2 (based on model m), else false.

Name	mm:leftBeside
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true p1 if is left beside p2 (based on model m), else false.

Name	mm:rightAbove
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true p1 if is right above p2 (based on model m), else false.

Name	mm:rightBelow
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true p1 if is right below p2 (based on model m), else false.

Name	mm:rightBeside
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true p1 if is right beside p2 (based on model m), else false.

Name	mm:spatialContains
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true if p1.shape contains p2.shape

Name	mm:spatialDisjoint
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true is p1.shape has no common points with p2.shape, else false.

Name	mm:spatialEquals
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true if p1.shape == p2.shape, else false.

Name	mm:crosses
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true if p1.shape and p2.shape have common points, else false.

Name	mm:touches
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns true if p1.shape.edge has at least one common point with p2.shape.edge and p1.shape.interior has no common point with p2.shape.interior, else false.

Spatial Aggregations

Name	mm:spatialBoundingBox
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns new MediaFragment / MediaFragmentURI with spatial fragment out of existing resources p1 and p2, so that x = min(p1.x, p2.x) and y = min(p1.y, p2.y) and w = max(p1.x + p1.w, p2.x + p2.w) and h = max(p1.y + p1.h, p2.y + p2.h).

Name	mm:spatialIntersection
Properties	*SpatialEntity*, *SpatialEntity*
Description	returns new MediaFragment / MediaFragmentURI with spatial fragment out of existing resources p1 and p2, so that x = max(p1.x, p2.x) and y = max(p1.y, p2.y) and w = min(p1.x + p1.w, p2.x + p2.w) - max(p1.x, p2.x) and h = min(p1.y + p1.h, p2.y + p2.h) - max(p1.y, p1.x)

Spatial Accessors

Name	mm:getBoxArea
Properties	*SpatialEntity*
Description	returns the area of BoundingBox of a shape, null if there is none.

Name	mm:getBoxHeight
Properties	*SpatialEntity*
Description	returns the height of a shape, null if there is none.

Name	mm:isSpatialFragment
Properties	*SpatialEntity*
Description	returns true is value is a spatial fragment.

Name	mm:getBoxWidth
Properties	*SpatialEntity*
Description	returns the width of a shape, null if there is none.

Temporal Relations

Name	mm:overlappedBy
Properties	*TemporalEntity*, *TemporalEntity*
Description	is the inverse function of overlaps.

Name	mm:overlaps
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns true if p1.start ¡ p2.start ¡ p1.end ¡ p2.end or p2.start ¡ p.start ¡ p.end ¡ p.end, else false.

Name	mm:precedes
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns true if p1.end ¡ p2.start, else false.

Name	mm:after
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns *true* if resource1.start ¿= resource2.end, else *false*.

Name	mm:contains
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns true if p1.start ¡ p2.start and p1.end ¿ p2.end, else false.

Name	mm:during
Properties	*TemporalEntity*, *TemporalEntity*
Description	inverse function of contains.

Name	mm:equals
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns true if p1.start == p2.start and p1.end == p2.end, else false.

Name	mm:finishedBy
Properties	*TemporalEntity*, *TemporalEntity*
Description	is the inverse function of finishes.

Name	mm:finishes
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns true if p1.end == p2.end and p1.start ¿ p1.start , else false.

Name	mm:temporalMeets
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns true if resource1.start = resource2.end or resource1.end = resource2.start, else false.

Name	mm:metBy
Properties	*TemporalEntity*, *TemporalEntity*
Description	is the inverse function of meets.

Name	mm:startedBy
Properties	*TemporalEntity*, *TemporalEntity*
Description	is the inverse function of starts.

Name	mm:starts
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns true if p1.start == p2.start and p1.end ¡ p2.end , else false.

Temporal Aggregations

Name	mm:temporalBoundingBox
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns new MediaFragment / MediaFragmentURI with temporal fragment (Min(p1.start, p2.start), Max(p1.end, p2.end)).

Name	mm:temporalIntermediate
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns new MediaFragment / MediaFragmentURI with temporal fragment (Min(p1.end, p2.end), Max(p1.start, p2.start)) if intersection not exists, else null.

Name	mm:temporalIntersection
Properties	*TemporalEntity*, *TemporalEntity*
Description	returns new MediaFragmentURI with temporal fragment (Max(resource1.start, resource2.start), Min(resource1.end, resource2.end)) if intersection exists, else null.

Temporal Accessors

Name	mm:getDuration
Properties	*TemporalEntity*
Description	returns the duration of an interval, null if there is none.

Name	mm:getEnd
Properties	*TemporalEntity*
Description	returns the end of an interval, null if there is none.

Name	mm:getStart
Properties	*TemporalEntity*
Description	returns the start of an interval, null if there is none.

Name	mm:isTemporalFragment
Properties	*TemporalEntity*
Description	returns true is value is a temporal fragment.

Combined Aggregations

Name	mm:boundingBox
Properties	*SpatialTemporalEntity*, *SpatialTemporalEntity*
Description	returns new MediaFragment / MediaFragmentURI with spatial and temporal fragment. It it works like spatialFunction:boundingBox, temporalFunction:boundingBox or both together.

Name	mm:intersection
Properties	*SpatialTemporalEntity*, *SpatialTemporalEntity*
Description	returns new MediaFragment / MediaFragmentURI with spatial and temporal fragment. It works like spatialFunction:boundingBox, temporalFunction:intersection and both.

Other Accessors

Name	mm:isMediaFragment
Properties	*URI*
Description	returns if value is a MediaFragment

Name	mm:isMediaFragmentURI
Properties	*URI*
Description	returns if value is a MediaFragmentURI

SPARQL-MM 1.0 is implemented as Sesame[25] function set and can be dynamically integrated in any existing Sesame triplestore via Java Class Loader. The release is available on Maven Central and thus can be used as Maven dependency.

```
<dependency>
 <groupId>com.github.tkurz</groupId>
 <artifactId>sparql-mm</artifactId>
 <version>1.0</version>
</dependency>
```

4.3 Sparql-MM in Action

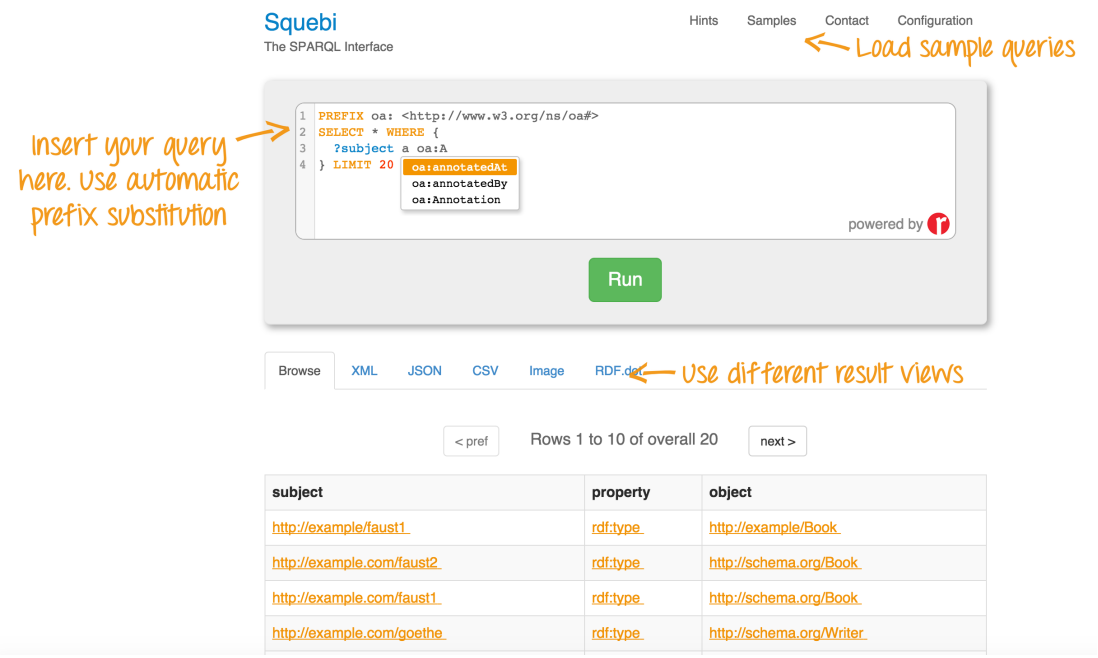
In this section we give an overview, how SPARQL-MM can be used for Media Fragment Retrieval. As example we use a an annotation set for the movie ”Lion King” that has been persisted using the MICO Annotation Schema described in Section 3. To make it more convenient for the user to query and display results we also introduce the SPARQL Editor *Squebi* [Kur14b].

Squebi - a flexible SPARQL Editor

Squebi is a SPARQL editor and SPARQL result visualizer. It is very flexible and can be adapted to specific use cases very easy. The feature set include:

[25] http://rdf4j.org/

Figure 7 Squebi SPARQL Editor



- customization of SPARQL result visualization
- support for SPARQL 1.1 (update and select)
- bookmarkable uris that define queries and the visualization type
- support for SPARQL query editing (URIs, ontologies and prefixes)
- fast backend switch (quite useful for demo)
- nice GUI

Figure 7 shows a screenshot of the application. The input fields supports prefix, type and property autocompletion for private and public ontologies. In combination with the color highlight it flats the learning curve for SPARQL query construction and helps to prevent mistakes. As outlined in the figure, Squebi allows various views on query results, starting from simple tables up to very use cases specific visualizations like charts or media player. Squebi is integrated in several software systems like Apache Marmotta and the MICO platform. For more information we recommend to have a look at the project description on `https://github.com/tkurz/squebi`.

Hands-On - Find the lions

We extend the example from section 4.1 with certain media assets and fragments, so that parts of the ”Lion King” video are linked to animals and concepts. As example we want to formalize the natural language query:

”Find scenes where Simba is left beside his father during the Lion King opening scene ordered by length.”

The following SPARQL query shows, how we can use SPARQL-MM together with other SPARQL functionalities to fulfill the request.

```
1  SELECT ?result WHERE {
2   ?a3    oa:hasBody     ?scene;
3          oa:hasTarget   ?s3.
4   ?scene a              movie:Scene;
5          schema:summary ?description.
6
7   FILTER regex(str (?description), ”opening scene”,”i”)
8
9   ?a2    oa:hasBody     :Simba;
10         oa:hasTarget   ?s2.
11  ?a1    oa:hasBody     ?p1;
12         oa:hasTarget   ?s1.
13  ?p1    :father_of     :Simba.
14
15  FILTER mm:leftBeside(?s2,?s1)
16
17  FILTER mm:intersects(?s3,?s2)
18
19  BIND (mm:boundingBox(?s1,?s2) AS ?result)
20
21 } ORDER BY DESC(mm:duration(?result))
```

Lines 2-7 define that there has to be a scene with description that includes the string opening scene. Lines 9-13 define the resources Simba and Simbas father. Line 15 filters all resources where Simba is left beside his father. Line 17 filters all resources where Simba appears at the same time like the resource annotated with ”opening scene”. In line 19 we construct a spatio-temporal bounding box with Simba and his father. Line 21 uses the ORDER BY block to get the longest scenes first. And in line 1 we project the bounding box with the result.

In this chapter we described the current State of SPARQL-MM implementation. As mentioned, the work is constantly aligned to discussions within the scientific community, which form the following focus of the next months:

- Improve performance by implementing SPARQL-MM on a deeper Layer (Database)
- Extend Media Fragments (within the W3C Media Fragment working group) to more complex shapes (including moving objects)
- Extend SPARQL to graph based similarity functions

5 Enabling Technology Models for Cross-media Recommendations

Recommender systems have changed the way people find information. Based on behaviour patterns and content analysis, items can be presented to the users that might be completely new to them, but match their expectations. Within MICO, the goal and opportunity is to use various metadata types, and apply recommendation algorithms to further enrich and extend metadata, creating richer models which can be used for various recommendation purposes.

The two MICO showcases have fairly different requirements with respect to recommendation: The first showcase, *News Media*, will require the recommendation of similar content, e.g. based on the similarity of topics, and content features. The second showcase, *Snapshot Serengeti* is mostly requiring use of past user interactions and classification accuracy scores, and extracted metadata, to provide an personalized, optimized mix of classification tasks to promote user involvement and engagement.

The underlying common theme of both showcases, however, is the need for *cross-media* recommendation: Automatic and manual annotations, contextual information and user interaction data related to various media types can be used as input to the algorithms, resulting in recommendations for all relevant media types as output, thereby crossing media borders. For instance, a user preference for images with lions (which may have been identified by automatic image analysis or manual annotation, and usage information) can be used to recommend related images, documents, posts, videos or video fragments. Consequently, one of the key challenges for this domain is to define which and how to use information sources to calculate similarity as needed for a specific use case.

5.1 Recommendation architecture and technologies

The technology behind recommender systems has evolved over the past 20 years into a rich collection of tools that enable the practitioner or researcher to develop effective recommender systems. These systems use different kinds of information in order to try to build specific models useful for the respective task.

Regarding recommendation technologies in MICO, we have applied an iterative approach, starting with the support for **content-based recommendations**, using PredictionIO and a monitoring API for data collection. After some researching and testing, the proposed architecture to satisfy the recommendation use cases is presented in figure 8:

The architecture consists of the following components:

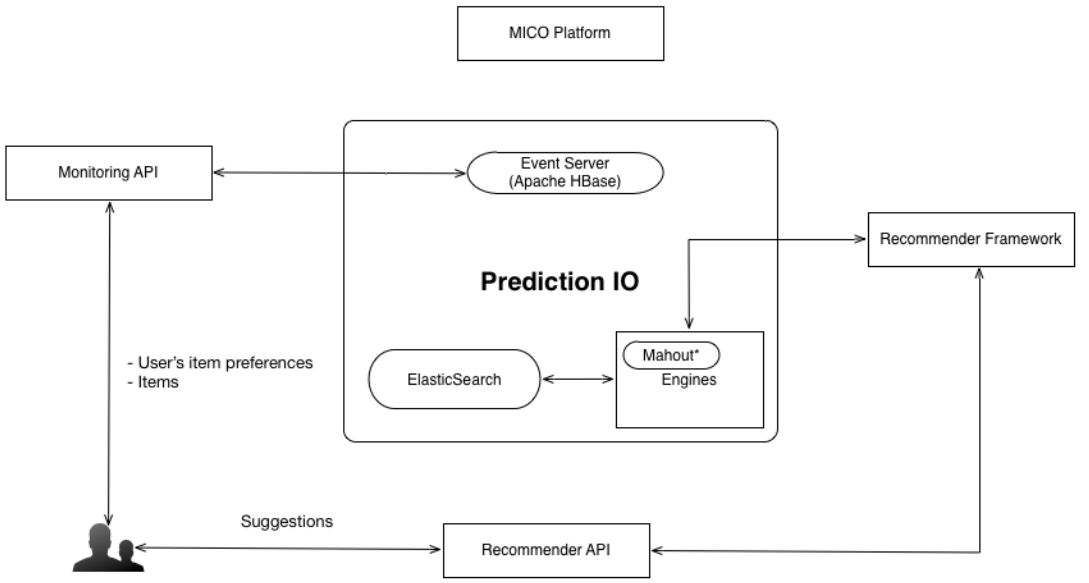
- Monitoring API: Responsible of collect information that will be used later by the recommendation algorithms.
- Recommender API: Receives client queries and give recommendations
- Recommender Framework: Responsible of execute the algorithms that respond to recommendation queries

In this overview, the MICO Platform is considered an external component, because the recommendation system will be fed not only by information extracted within the MICO platform, but also by information collected via the monitoring API.

For this first version of the recommendation system, these components are implemented using **PredictionIO**, applying several customizations:

- The Monitoring API is implemented using the Event Server

Figure 8 Recommender System Architecture



- The Recommendation Framework is implemented using PredictionIO platform
- The Recommendation API is implemented by engine endpoints

In the future, these components will be refined and complemented by additional functionality.

5.1.1 PredictionIO

PredictionIO is an OSS machine-learning server. It claims to ”empower developers and data scientists to build smart applications with data productively”. PredictionIO provides many components that can be used on top of the underlying core framework. Its architecture is based on different applications, which use various engines (the base of recommendation algorithms) consisting of a set of pluggable components that compose the recommendation algorithm. When an engine is built and configured (data source, algorithm, model, ...), it has to be trained in order to generate the recommendation model. After the training, the engine is deployed which creates an HTTP endpoint ready to receive queries and return recommendations based on the trained model. Each application has a separate space to store data that can be used by the engines in the recommendation algorithm.

For all the metadata PredictionIO manages, it uses an ElasticSearch[26] instance, which can be also used for other purposes.

5.2 How to install PredictionIO

There are several ways to install PredictionIO.

[26] `https://github.com/elastic/elasticsearch`

Automatic installation

On Linux / Mac OS X, PredictionIO can be installed with a single command:

```
$ bash -c "$(curl -s https://install.prediction.io/install.sh)"
```

The above script will complete the installation of PredictionIO, downloading and installing the necessary components to make use of PredictionIO and its tools

Manual installation

In order to install PredictionIO manually, you can choose between installing it from source code, using Vagrant machine, etc. Check `http://docs.prediction.io/install/#method-3:-manual-install` for a complete list of manual installation methods.

After the installation, an application ID and access key has to be generated. Check `http://docs.prediction.io/templates/recommendation/quickstart/#3.-generate-an-app-id-and-access-key` to see how to do that.

Each application will have a separated space of data in the Event Server, so that sending information to the Event Server requires to send also the application key along with the data in order to the data be bound to that application. Each client application should have a PredictionIO instance in order to not mix information between applications.

5.3 Content-Based recommendation engine

As described earlier, support for content-based recommendation was a priority for the initial version of the MICO recommendation system. Considering that PredictionIO does not provide built-in algorithms to deal with this, a dedicated engine to meet these use cases has been developed, which is described in the following.

5.3.1 Overview

The Content-Based recommendation engine within PredictionIO uses different features from the content of the item to suggest similar items. A commonly applied technique is to use the co-occurrence of words / tags to find similar items. However, there are items like media content (images, videos, etc) where this approach is not applicable as such - manual tags would have to be added first, but it is a tedious and costly task. However, by using the MICO platform, automatic annotations can be extracted from media content, and can then be exploited for the aforementioned approaches.

5.3.2 How it works

The engine uses the PredictionIO Event Server as data source, so all the item information has to be stored in the Event Server in order to the engine be able to access the information contained on it. The engine is configured to work using some fields of the item (filled with MICO extracted information) and an ElasticSearch instance. It uses the ElasticSearch More Like This feature (based on TF-IDF) to suggest similar items to a given one. Basically, what TF-IDF does, is to collect the most relevant terms in the fields (which can have different boost) of the given item and then execute a query over the rest of items using those terms.

5.3.3 How to install, configure and deploy it

The first version of the engine is located at `https://bitbucket.org/mico-project/recommendation`. Once the code has been downloaded (either using git or getting the zip file) and put in a folder, the structure is as follows:

- src/ : Source code of the engine in Java
- build.sbt : File for sbt (Scala Build Tool) used when building the engine
- manifest.json : Metadata information used internally by PredictionIO
- engine.json : Engine metadata used to configure the engine for training and deploying

With the source code in place, PredictionIO command line tools are used to build, train and deploy the engine.

Building the engine is as simple as execute a single command inside the folder:

```
$ pio build
```

The engine will be built and ready for training. Before training the engine to build the recommendation model (in the case of this engine, is an ElasticSearch index), it has to be configured. To do that, engine.json file has to be edited:

The next piece of code shows the content of the file:

```
{
 "id": "content-based-mlt-test",
 "name": "Content-based MLT recommendation engine",
 "description": "An engine to perform content-based recommendations using ElasticSearch
     MoreLikeThis feature using Events stored in EventServer to feed ES",
 "engineFactory": "eu.mico.recommendation.content.mlt.engine.EngineFactory",
 "datasource": {
    "params" : {
                "eventServerUrl": "EVENTSERVER_URL",
                "accessKey": "APPLICATION_ACCESS_KEY",
                "entityType" :  "ENTITY_TYPE",
                "properties":  ["PROPERTY1", "PROPERTY2", ...]
    }
 },
 "algorithms": [
    { "name": "MltBasedAlgorithm",
      "params" :  { "server" : "SERVER",
                    "port"     : 9300,
                    "index"    : "ELASTICSEARCH_INDEX",
                    "type"     : "ENTITY_TYPE",
                    "properties": ["PROPERTY1", "PROPERTY2"]
                }
    }
 ]
}
```

In order to configure the engine, the *params* key of datasource and algorithms have to be configured:

- Datasource parameters
- eventserverUrl: The url of the PredictionIO event server (`http://localhost:7070` is the default url when PredictionIO event server is started)
- accessKey: The application access key obtained when the application was created
- entityType: The entity type of the events from that application to be used (each event sent to event server has an entity type)
- properties: Array of properties from the events that will be used by this engine (each event sent to event server can have properties)
- Algorithm parameters
- server: The ElasticSearch server. If empty, local PredictionIO ElasticSearch instance will be used
- port: The ElasticSearch server port. By default, ElasticSearch is running on port 9300
- index: The ElasticSearch index that will be created for this engine
- properties: Array of properties to be used from the events to feed the index. They should be the same as datasource.

After that, to train the engine the next command has to be executed inside the engine folder:

```
$ pio build
```

Once the training has finished successfully, the engine can be deployed to an endpoint executing:

```
$ pio deploy
```

The default port where the engine is deployed is 8000
Now the engine is deployed and can receive queries to suggest similar items.

5.3.4 Example

Here is an example for about how the engine works: The event server has been fed with 12 items representing web pages, from which MICO platform has extracted entities from different parts of the web and they have been put in two properties: *entities* and *about*. An example of an event sent to the event server has the following form:

```
{
  "eventId": "328q7oheSB9YX0enrZ_z_gAAAUvgAMVBgri8-ApL8Eo",
  "event": "io_event",
  "entityType": "io_item",
  "entityId": "http://data.redlink.io/91/be2/post/Editoriale_-_La_rotta",
  "properties": {
  "about":
    ["http://data.redlink.io/91/be2/entity/OGM",
    "http://data.redlink.io/91/be2/entity/cambiamenti_climatici",
    "http://data.redlink.io/91/be2/entity/petrolio",
    "http://data.redlink.io/91/be2/entity/COP21"
```

```
    ],
  "entities":
    ["http://data.redlink.io/91/be2/entity/petrolio",
    "http://data.redlink.io/91/be2/entity/cambiamenti_climatici",
    "http://data.redlink.io/91/be2/entity/Cina",
    "http://data.redlink.io/91/be2/entity/COP21",
    "http://data.redlink.io/91/be2/entity/OGM"]
  },
  "eventTime": "2015-03-03T14:18:41.857Z"
}
```

where:

- eventId: The id of the event. If empty, the EventServer will create a new one
- eventType: The type of the event. Useful if we want to manage different type of information within the same application
- entityType: The type of the entity representing the event
- entityId: The id of the entity. In the example, the URL of a web page
- properties: properties attached to the event. In the example, the entities and about entities extracted by MICO

During the training, the engine uses these events to create a specific ElasticSearch index that will be used later to get the suggestions. When the engine is deployed, it can receive POST queries to return suggestions recommending items at http://localhost:8000/queries.json (assuming default server and port). The query to be sent is a JSON object with the following form:

```
{
  id: "http://data.redlink.io/91/be2/post/Editoriale_-_La_rotta",
  fieldsInfo: [
    {field: 'entities', boost: '10'}
  ]
}
```

where:

- id: The id of the entity to get similar items to it
- fieldsInfo: an array containing the fields to be used to get similar items and the boost (weight) of that field. The boost is useful when the recommendations are based on several fields to give more weight to one field than another

When sending that query to the engine, the suggestions for that id are returned:

```
$ curl -X POST -d '{id: "http://data.redlink.io/91/be2/post/Editoriale_-_La_rotta", fieldsInfo:
    [{field: 'entities', boost: '10'}]}' http://localhost:8000/queries.json
```

Response:

```
{
  "recommendations":[
    {"iid": "http://data.redlink.io/91/be2/post/Federica_Ferrario_-_intervista",
     "value":1.9876279},
    {"iid": "http://data.redlink.io/91/be2/post/L'anno_pi", "value":1.4310836},
    {"iid": "http://data.redlink.io/91/be2/post/MAS_-_Parole", "value":1.4142134},
    {"iid":
     "http://data.redlink.io/91/be2/post/OGM__prolungato_il_bando_per_il_MON810_-_breve",
     "value":0.08944272},
    {"iid": "http://data.redlink.io/91/be2/post/Un_santuario_per_6_milioni_-_articolo_Artico",
     "value":0.07154754}
  ]
}
```


where:

- iid: The item representing the entity id
- value: a similarity value (without normalization) where the higher value means the most similar item

Taking the first suggestion, the event information stored is:


```
{
  "eventId": "opQqhdE2ds6jauoth2_oBgAAAUvgAKyqnN1zEGP8FRM",
  "event": "io_event",
  "entityType": "io_item",
  "entityId": "http://data.redlink.io/91/be2/post/Federica_Ferrario_-_intervista",
  "properties": {
  "about":
    ["http://data.redlink.io/91/be2/entity/Expo_2015",
    "http://data.redlink.io/91/be2/entity/MAS",
    "http://data.redlink.io/91/be2/entity/OGM",
    "http://data.redlink.io/91/be2/entity/cambiamenti_climatici"
    ],
  "entities":
    ["http://data.redlink.io/91/be2/entity/Expo_2015",
    "http://data.redlink.io/91/be2/entity/OGM",
    "http://data.redlink.io/91/be2/entity/MAS",
    "http://data.redlink.io/91/be2/entity/cambiamenti_climatici"
    ]
  },
  "eventTime": "2015-03-03T14:18:35.562Z"
}
```


Since the query was done to check similar items based on "entities" property, it can be seen that this item is the most similar one because they are sharing two entities.

Improvements that can be done in future iterations involve thresholds for suggestion values, query parameters to limit the number of suggestions, parameters to exclude some items from the response, etc.

5.4 MICO recommender system outlook

There are several limitations to the current recommendation system, which will be worked on in the next project phase:

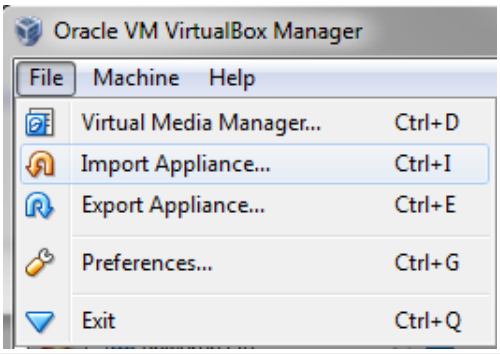
Apart from content-based recommendation, MICO showcases also require support for recommendation based on collaborative filtering of user-item interaction, e.g., to recommend users to projects. For example, different users could have watched different videos and rated them in the News Video showcase, which could be exploited by collaborative filtering algorithms. For this purpose, the current PredictionIO-based framework will be extended with Apache Mahout or similar tools.

Beyond that, a key challenge and opportunity for more accurate cross-media recommendations in MICO lies in the fact that different recommendation approaches and information sources have to be combined: For instance, collaborative filtering works fine on an item level such as in the example above. However, it does not work on the level of media fragments, e.g., to recommend specific news segments (related to one specific topic) within news show items, which will require the use of automatic or manual annotation on a fragment level. The same goes for the linking of different media types, which requires additional metadata, e.g., describing that various documents, images and videos are actually about the same species. Another example is the identification of persons in images (using face detection and recognition). The goal is to find related textual, audio or video content based on the person identified in the image and other (social) interactions by the the same user.

In order to enable the aforementioned approaches, the data model is paramount. Although the user interactions can be represented in a generic way like ”user,item,action/value”, the data model of the items being used are very important to refine the suggestions. Same items can be compared in many different ways, using different features, so that depending on the nature of the feature being used, two items can be similar or dissimilar. That is to say, a video can be similar to another based with respect to technical parameters (duration, resolution) or content analysis (picture of a landscape) but not with respect to the related topic or persons. Hence, the definition of consistent data models to represent the information which will be used later in conjunction with the classic collaborative filtering algorithms is very important at this stage. Moreover, data model definitions are very useful in order to decide which are the most important features to be used by the algorithms to refine the results based on user interactions. Depending on the use case to solve, it will be very helpful to use a subset of features as specified in the data model. The data model should contain all the necessary data required by the use cases which can range from information extracted by MICO platform to user profiles information provided by the specific applications. Of course, combining these models with the user interactions is fundamental.

The main objective will be to take advantage of these data models to enable cross-media recommendations. Richer algorithms will be provided, which give more accurate recommendations by exploiting a broad set of information sources. For this purpose, new engines that are customized to meet the related showcase requirements will be developed and integrated into the proposed PredictionIO architecture.

Figure 9 Import VirtualBox Image



6 MICO Platform Installation

The MICO Platform Server runs on a Linux server providing the following MICO relevant services:

- Apache Marmotta with contextual extensions
- Broker
- RabbitMQ
- Hadoop HDFS server for binary content storage

For development and testing purpose, we provide a ready-to-use virtual image (see section 6.1). If you would like to install the MICO Platform on your own machine, have a look at section 6.2.

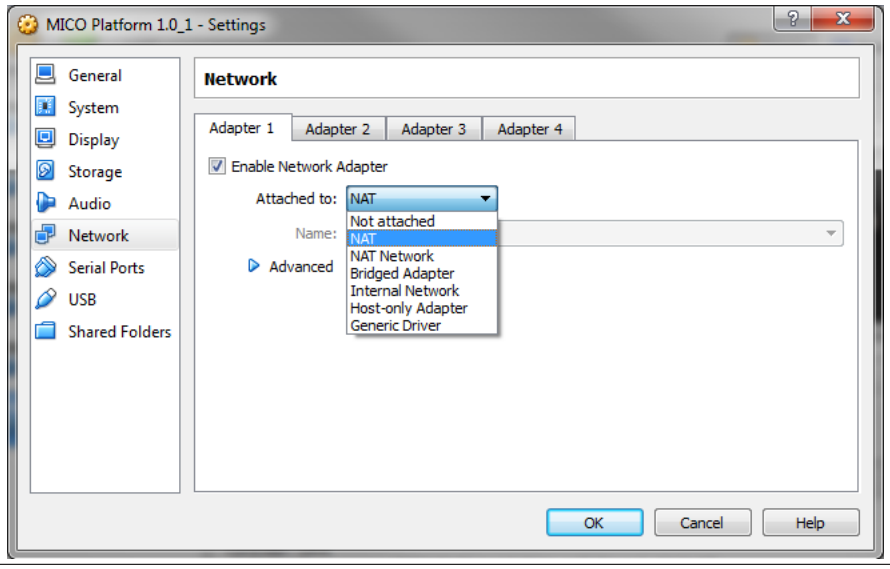
6.1 VirtualBox Image

First of all you have to download and install VirtualBox[27] on the machine the MICO Platform should run on. Then you have to download the VirtualBox Image[28]. The username to log into the machine (directly or via SSH) is *user* with the password *user*. Therefore you sould not use this image in public accessible environments.

To import the image start VirtualBox and select *Import Appliance...* as shown in figure 9. A dialog will show up, where you have to select the VirtualBox Image you downloaded (file extension is *.ova* as the image is provided as Open Virtualization Format). On the next screen you can change the appliance settings, if needed. Continue clicking the *Import* button. Before you start the virtual machine, you should have a look at the network settings (see figure 10) by selecting the virtual machine and clicking *Settings*.

[27] https://www.virtualbox.org/wiki/Downloads
[28] http://apt.mico-project.eu/download/MICO%20Platform%20v1.1.1.ova

Figure 10 VirtualBox Network Setting



6.1.1 Network settings

NAT

If you run the virtual machine (VM) on your local computer, and all interaction with the MICO Platform happens within the VM, choosing *NAT* should be fine. In the shipped VM configuration the following port forwardings are already set up:

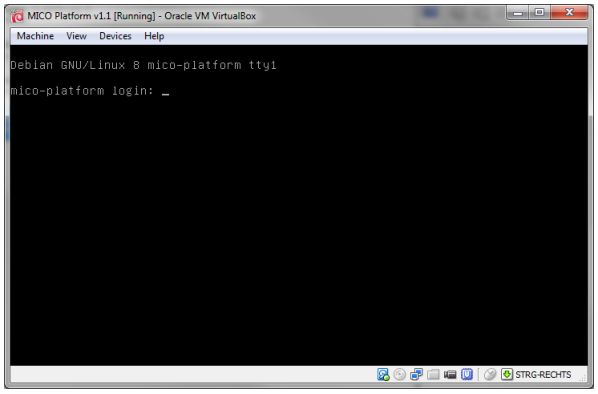
VM host port	VM guest destination port	Description
22080	80	Welcome & Info page
8080	8080	Tomcat (broker and Marmotta)
15672	15672	RabbitMQ administration webinterface (username and password is *mico*)
22022	22	SSH server (username *user*)
50070	50070	HDFS NameNode web interface
50075	50075	HDFS DataNode web interface
8020	8020	HDFS NameNode metadata service
50090	50090	HDFS SecondaryNamenode
50010	50010	HDFS DataNode data transfer
50020	50020	HDFS DataNode metadata operations

Bridged Adapter

If you need to connect to the virtual machine via network from another computer (even if it is the computer that runs the virtual machine) you might choose *Bridged Adapter*, but take care, as the default passwords are insecure and therefore anyone (depending on your network configuration) might have access to the virtual machine. You also have to adopt the */etc/hosts* file in the virtual machine. Therefore start the VM and login as *user* (directly or via SSH). Then open the file with *root* privileges:

```
1 sudo nano /etc/hosts
```

Figure 11 Start Screen



and replacing the line *127.0.0.1 mico-platform* with *x.x.x.x mico-project*, where *x.x.x.x* is the IP address of the platforms network interface. Now you need to restart the HDFS server so the change takes effect:

```
1 sudo /etc/init.d/hdfs stop; sudo /etc/init.d/hdfs start
```

Setup client system to access the platform VM
To access the platform VM from other machines, you need to make sure that the DNS/hostname *mico-platform* resolves to the public IP of the VM. *Accessing the platform via IP-Address or any other hostname may result in unexpected behaviour of the system!*

The easiest way is to configure *mico-platform* in the *hosts* file:

- For (most) *NIX systems you have to edit the file */etc/hosts* (e.g. `sudo nano /etc/hosts`)
- For Windows systems the file is *%SystemRoot%\system32\drivers\etc\hosts* (where *%SystemRoot%* is the Windows directory, normally it is *C:\Windows*). Make sure you open the file with administrative rights.

The line to add is the same for *NIX as well as Windows: `x.x.x.x mico-platform`, where *x.x.x.x* is *127.0.0.1* of your network configuration is *NAT*, or it has to be the IP address of the platform machine in any other case (e.g. *10.0.2.15*).

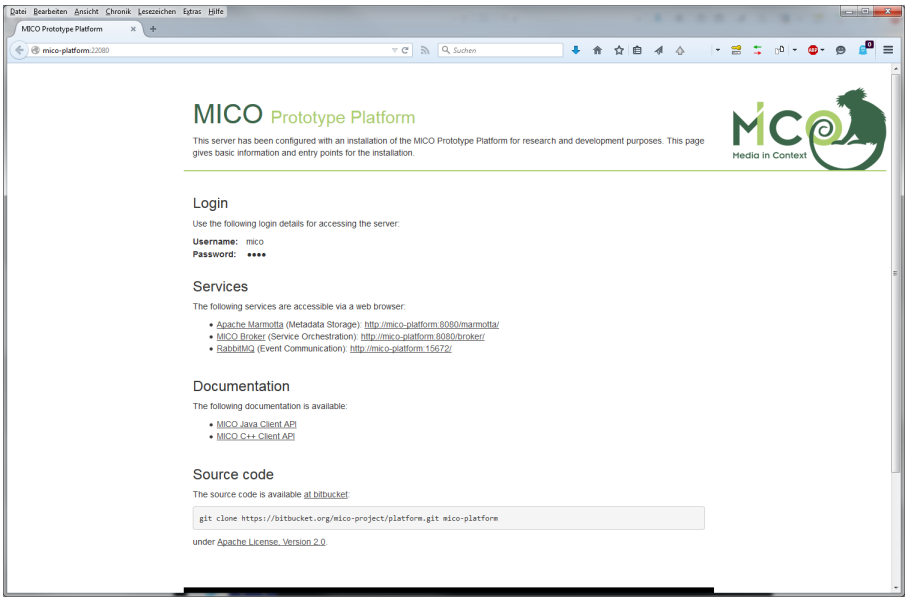
Run the VM
If you have not started the virtual machine until now, do so by clicking the *Run* button and wait until you can see the start screen as shown in figure 11. Now all services are up and running and you can begin to work with the MICO Platform. Open a browser and type *http://mico-platform/* (or *http://mico-platform:22080/* if your network is configured as *NAT*) and you get the overview page as in figure 12.

Also have a look at section 6.2.5 for a description on starting extractors and injecting content.

6.2 Debian Repository

This section explains how to setup a MICO Platform Server yourself based on Debian Linux. All required components for developing and running the MICO ecosystem are provided as custom Debian

Figure 12 Overview page



packages and available from a public available repository. If you want to setup a development server or virtual image, this is the easiest way to get up and running.

The default installation provides the following services:

- Apache Marmotta with contextual extensions (can be found in the *marmotta/* directory), running on *http://<hostname>:8080/marmotta*
- Broker, simple user interface running on *http://<hostname>:8080/broker*
- RabbitMQ, running on *<hostname>:5672*, the status and administration user interface is accessible via *http://<hostname>:15672/*
- Hadoop HDFS server running on *hdfs://<hostname>/*

As long as you did not change the user credentials you are asked for during installation, the default username for all components is *mico* with password *mico*.

6.2.1 Setup Debian Jessie

The MICO packages are provided to work with Debian Jessie amd64 arch (at the time of writing Jessie is in *testing* state). For MICO, a plain installation is sufficient, no special package preselection is needed. You can start with the latest Debian Network Installation Image[29].

Please note, that you should not create a user named *mico* as this is the default user that will be created during the setup process of the MICO platform later.

[29] http://cdimage.debian.org/debian-cd/8.0.0/amd64/iso-cd/

6.2.2 Add MICO repository

To add the MICO repository create the file */etc/apt/sources.list.d/mico.list* with *root* previleges and add the following line:

```
## APT Repository for the MICO FP7 Project

## MICO main and contrib sections
deb http://apt.mico-project.eu/ mico main contrib
## If you need access to the restricted non-free section add the
## following line with your username and password:
#deb http://user:password@apt.mico-project.eu/ mico non-free
```

All packages are signed with a key[30]. To avoid warnings during installation and updating of the MICO packages either install the *mico-apt-repository* package or add the key manually by typing the following command in a shell:

```
wget -O- http://apt.mico-project.eu/apt-repo.key | sudo apt-key add -
```

To retrieve the new list of packages execute:

```
sudo apt-get update
```

Restricted / Non-Free extractors Some extractors contain restricted libraries that cannot be freely distributed. Those are available in the non-free section of the mico repository. If you require access to this section of the repository, you need to request an account[31] and update the *mico.list* file created in the beginning of this section.

6.2.3 Install HDFS

Currently, HDFS is not available from the repositories. You need to install it manually. For development and testing server, this is done in a few steps which mainly follow the *Singe Node Setup*[32] with "Pseude-Distributed Operation". HDFS is part of the Apache Hadoop project[33] so we will download the binary Hadoop release[34], but only use the HDFS service. Most of the following steps need *root* privileges, so make sure to prepend the *sudo* program to the following commands or run it as *root*.

As HDFS is written in JAVA install a JAVA runtime environment without GUI components and make sure SSH is installed:

```
apt-get install default-jre-headless ssh
```

First, create the directories to store the hdfs data, the *hadoop* user home and the log directory:

```
mkdir -p /var/lib/hadoop/datanode
mkdir -p /var/lib/hadoop/namenode
mkdir -p /var/lib/hadoop/home
mkdir -p /var/log/hadoop
```

[30] Key-ID: AD261C57
[31] Simply ask on the project mailinglist for username/password: office@mico-project.eu
[32] http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SingleCluster.html
[33] At the time of writing, the current stable version of Hadoop was 2.6.0
[34] http://www.apache.org/dyn/closer.cgi/hadoop/common/hadoop-2.6.0/hadoop-2.6.0.tar.gz

Next, create the user and group the HDFS service should run as and fix the permissions:

```
useradd -d /var/lib/hadoop/home -K UID_MIN=100 -K UID_MAX=500 -K GID_MIN=100
    -K GID_MAX=500 -K PASS_MAX_DAYS=-1 -U hadoop
chown hadoop:hadoop /var/lib/hadoop/*
chmod 700 /var/lib/hadoop/home
chown hadoop:hadoop /var/log/hadoop
```

The HDFS start and stop scripts are using SSH to execute the proper commands. Therefore it is necessary to setup authentication using keys:

```
sudo -u hadoop ssh-keygen -t dsa -P '' -f /var/lib/hadoop/home/.ssh/id_dsa
sudo -u hadoop cp /var/lib/hadoop/home/.ssh/id_dsa.pub
    /var/lib/hadoop/home/.ssh/authorized_keys
```

and accept the keys by running both commands

```
echo exit | sudo -u hadoop ssh mico-platform
echo exit | sudo -u hadoop ssh 0.0.0.0
```

Now it is time to download the Apache Hadoop binary. We will install it to the */opt* directory

```
cd /opt ; wget
    http://tweedo.com/mirror/apache/hadoop/common/hadoop-2.6.0/hadoop-2.6.0.tar.gz
tar -zxf hadoop-2.6.0.tar.gz
```

The default directory for the configuration files is */etc/hadoop*:

```
ln -s /opt/hadoop-2.6.0/etc/hadoop/ /etc/hadoop
```

To configure HDFS for our needs, some config files have to to be adopted. First open */etc/hadoop/hadoop-env.sh* and look for:

```
export JAVA_HOME=${JAVA_HOME}
```

Set the variable *JAVA_HOME* to the location of the JRE (e.g.export JAVA_HOME=/usr/lib/jvm/default-java) . To get the location of your JRE home directory run:

```
dirname $(readlink -f `which java` | sed s:/bin/java::)
```

And also add export HADOOP_LOG_DIR=/var/log/hadoop to */etc/hadoop/hadoop-env.sh*.

Make sure the file */etc/hadoop/core-site.xml* contains the following settings:

```
<configuration>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/tmp</value>
  </property>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://mico-platform:8020</value>
  </property>
  <property>
    <name>dfs.client.read.shortcircuit</name>
```

```
    <value>false</value>
  </property>
</configuration>
```

Now configure the HDFS daemon by adding the following to */etc/hadoop/hdfs-site.xml*:

```
<configuration>
  <property>
    <name>dfs.name.dir</name>
    <value>file:///var/lib/hadoop/namenode</value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>file:///var/lib/hadoop/datanode</value>
  </property>
  <property>
    <value>0.0.0.0</value>
    <name>dfs.namenode.rpc-bind-host</name>
  </property>
  <property>
    <value>0.0.0.0</value>
    <name>dfs.namenode.servicerpc-bind-host</name>
  </property>
  <property>
    <name>dfs.datanode.address</name>
    <value>0.0.0.0:50010</value>
  </property>
  <property>
    <name>dfs.datanode.hostname</name>
    <value>mico-platform</value>
  </property>
  <property>
    <name>dfs.client.use.datanode.hostname</name>
    <value>true</value>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.permissions.enabled</name>
    <value>false</value>
  </property>
  <property>
    <name>dfs.namenode.acls.enabled</name>
    <value>false</value>
  </property>
  <property>
    <name>fs.permissions.umask-mode</name>
    <value>000</value>
  </property>
  <property>
```

```
    <name>dfs.webhdfs.enabled</name>
    <value>true</value>
  </property>
  <property>
    <name>dfs.client.read.shortcircuit</name>
    <value>false</value>
  </property>
</configuration>
```

/etc/hadoop/slaves just contains one line:

```
mico-platform
```

After that, initialize the hdfs storage by running

```
sudo -u hadoop /opt/hadoop-2.6.0/bin/hdfs namenode -format
```

start the service

```
sudo -u hadoop /opt/hadoop-2.6.0/sbin/start-dfs.sh
```

and set the permission:

```
sudo -u hadoop /opt/hadoop-2.6.0/bin/hdfs dfs -chmod 777 /
```

After that HDFS is installed and ready to be used.
To start HDFS on boot you can use a simple start script like this one:

```
#! /bin/sh
### BEGIN INIT INFO
# Provides:          Hadoop HDFS
# Required-Start:    sshd
# Required-Stop:
# Default-Start:     2 3 4 5
# Default-Stop:
# Short-Description:
### END INIT INFO

PATH=/sbin:/bin

USER=hadoop
export HADOOP_INSTALL=/opt/hadoop

case "$1" in
 start)
    su -c ${HADOOP_INSTALL}/sbin/start-dfs.sh $USER
    ;;
 stop)
    su -c ${HADOOP_INSTALL}/sbin/stop-dfs.sh $USER
    ;;
 restart|reload|force-reload|status)
    echo "Error: argument '$1' not supported" >&2
    exit 3
    ;;
```

```
*)
    echo "Usage: $0 start|stop" >&2
    exit 3
    ;;
esac
```

6.2.4 Install the MICO Platform

Now you can start the installation of the MICO platform by installing the *mico-platform* package and dependencies. Type the following to initiate the installation:

```
sudo apt-get install mico-platform
```

This process might take several minutes, depending on your Internet connection bandwidth and your computer. When it is finished, the MICO platform is up and running.

6.2.5 Access the MICO Platform

Web Interface
The Debian installation comes with a single entry-point for accessing the Web interfaces of the running services. It is available at *http://<hostname>/* (e.g. *http://localhost/* if you are working on the server itself). If the server is accessible from outside the development environment, please make sure to further protect this page by e.g. a firewall or changes to the lighttpd configuration, as it contains the login details for the MICO user.

Sample Service
The Debian installation also includes a sample service implemented in C++ that is used for demonstrating the platform functionality. This service is capable of transforming text contained in JPEG and PNG images into plain text using the tesseract OCR library. To test it you have to start the service in a shell as *root*:

```
mico_ocr_service <host> <user> <password>
```

Restricted / Non-Free extractors Some extractors contain restricted libraries that cannot be freely distributed. Those are available in the non-free section of the MICO repository. If you require access to this section of the repository, you need to request an account[35] and update the *mico.list* file created in the beginning of this section.

where *<host>*, *<user>* and *<password>* have to be replaced by the values you provided during the installation. The default would be:

```
mico_ocr_service localhost mico mico
```

To check if the service and its dependencies are running you can access the broker web interface *http://<hostname>:8080/broker*.

In order to run specific extractor chains please refer to section 2.1.1.

[35] Simply ask on the project mailinglist for username/password: office@mico-project.eu

Inject Content
To inject content for analysis you can open the broker inject page *http://<hostname>:8080/broker/inject.html* or use the provided simple command line tool as follows:

```
1 mico_inject <host> <user> <password> <files...>
```

where *<files...>* is a list of files on the local file system. The call will inject a single content item, with creating a content part for each files given as argument.

6.3 Compiling the MICO Platform API yourself

If you would like to compile the platform yourself, the necessary steps are described in this section. These instruction assume you are using a Debian Jessy Linux with the required build tools installed (see below).

First you have to fetch the source from the public available repository[36]. This repository contains the source code of the MICO platform API and modules. It provides implementations of the API for both Java and C++ (version 11). The API is used by analysis services to register with the platform and by other applications to interact with the MICO platform (e.g. inject and export content items).

6.3.1 Prerquisites

The Java API is built using Apache Maven and will therefore retrieve all its dependencies automatically. Therefore, only the following prerequisites need to be satisfied:

JAVA API

- JDK 7
- Apache Maven

C++ API
Building the C++ API has additional requirements for native libraries. In particular, these are:

- GNU Autotools, GCC >= 4.8 with C++11 support
- cURL library for HTTP requests (apt−get install libcurl4−gnutls−dev)
- expat library for XML parsing (apt−get install libexpat1−dev)
- Boost 1.55 libraries for additional C++ functionalities
  (apt−get install libboost1.55−dev libboost−log1.55−dev libboost−system1.55−dev)
- xxd for inlining SPARQL queries in C++ (part of VIM, apt−get install vim−common)
- protobuf for the event communication protocol (apt−get install libprotobuf−dev)

[36] https://bitbucket.org/mico-project/platform

- AMQP-CPP for communication with RabbitMQ (manual install[37] or available from the MICO apt repository (cf. 6.2.2))
- libhdfs3 for HDFS access (manual install[38] or available from the MICO apt repository (cf. 6.2.2), depends on further libraries: `apt-get install libxml2-dev libkrb5-dev libgsasl7-dev uuid-dev protobuf-compiler`)
- Doxygen for building the documentation (`apt-get install doxygen`)
- libdaemon for running extractors (`apt-get install libdaemon-dev`)

For building the C++ binary tools (*mico_inject*, etc.), there are the following additional dependencies:

- magic library for guessing MIME type (`apt-get install libmagic-dev`)

For building the C++ sample analyzers (*mico_ocr_service* etc.), there are the following additional dependencies:

- tesseract library for OCR with English database (`apt-get install libtesseract-dev tesseract-ocr-eng`)
- leptonica library for image processing (`apt-get install libleptonica-dev`)

6.3.2 Building

The API is built using the standard toolset for the respective environment (i.e. Apache Maven or GNU make). When running tests, make sure the MICO Platform Server installation is started, and point to its host name or IP address by setting the test.host environment variable appropriately, for example:

```
export test.host="127.0.0.1"
```

Building JAVA

The complete platform is built using Maven. To build and install the current version, run

```
mvn clean install
```

on the command line. This will compile all Java source code, run the test suite, build JAR artifacts and install them in the local Maven repository.

Binary Maven artifacts are periodically published to our development repositories:

```
<repositories>
  ...
  <repository>
    <id>mico.releases</id>
    <name>MICO Relesases Repository</name>
    <url>http://mvn.mico-project.eu/content/repositories/releases/</url>
  </repository>
  <repository>
    <id>mico.snapshots</id>
```

[37] https://github.com/CopernicaMarketingSoftware/AMQP-CPP
[38] https://github.com/PivotalRD/libhdfs3/releases/latest

```
    <name>MICO Snapshots Repository</name>
    <url>http://mvn.mico-project.eu/content/repositories/snapshots/</url>
  </repository>
</repositories>
```

Building C++

The C++ bindings of the platform are built using CMake. To build the C++ API, create a new directory (can be located anywhere) and in that directory run:

```
cmake /path/to/repository/api/c++
```

This will generate the required Makefile to compile and install the C++ API.

After the configuration succeeds (i.e. all dependencies are found), the C++ libraries can be built and automatically tested using GNU make as follows:

```
make
```

To create a complete API documentation of the MICO Platform API in the api/c++/doc directory, run

```
make doc
```

To install the C++ libraries and headers to the predefined prefix, run

```
make install
```

References

[AKW14] Patrick Aichroth, Thomas Kurz, and Christian Weigel. *D2.2.1 First Specifications in Cross-Media Analysis*. Deliverable. MICO, 2014. URL: http://www.mico-project.eu/wp-content/uploads/2014/12/D2.2.1-D3.2.1-D4.2.1-D5.2.1_First_specs_07112014.pdf.

[BG14] Dan Brickley and R.V. Guha. *RDFS RDF Schema 1.1*. W3C Recommendation. http://www.w3.org/TR/rdf-schema/. W3C, Feb. 2014.

[DT05] N. Dalal and B. Triggs. "Histograms of Oriented Gradients for Human Detection." In: *Computer Vision and Pattern Recognition*. 2005.

[Fel+10] P.F. Felzenszwalb et al. "Object Detection with Discriminatively Trained Part Based Models." In: *IEEE Pattern Analysis and Machine Intelligence* 32.9 (2010), pp. 1627–1645.

[Fer01] Jon Ferraiolo. *Scalable Vector Graphics (SVG) 1.0 Specification*. W3C Recommendation. http://www.w3.org/TR/2001/REC-SVG-20010904. W3C, Sept. 2001.

[HS13] Steve Harris and Andy Seaborne. *SPARQL 1.1 Query Language*. 2013. URL: http://www.w3.org/TR/sparql11-query/.

[KS14] Thomas Kurz and Kai Schlegel. *D4.2.1 First Specifications in Cross-Media Querying*. Deliverable. MICO, 2014.

[KSK15] Thomas Kurz, Kai Schlegel, and Harald Kosch. "Enabling access to Linked Media with SPARQL-MM." In: *Proceedings of the 24nd international conference on World Wide Web (WWW2015) companion (LIME15)*. 2015. DOI: 10.1145/2740908.2742914.

[Kur14a] Thomas Kurz. *D4.1.1 State of the Art in Cross-Media Querying*. Deliverable. MICO, 2014.

[Kur14b] Thomas Kurz. *Squebi*. Presentation at the ISWC Developer Workshop (ISWC2014). 2014.

[LZ13] Jianguo Li and Yimin Zhang. "Learning SURF Cascade for Fast and Accurate Object Detection." In: *2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2013, pp. 3468–3475. ISBN: 1063-6919. DOI: 10.1109/CVPR.2013.445.

[MM04] Frank Manola and Eric Miller. *RDF RDF Primer*. W3C Community Draft. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/. W3C, Feb. 2004.

[PS08] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. W3C, Jan. 2008.

[SB14] Kai Schlegel and Emanuel Berndl. *D3.2.1 First Specifications in Metadata Publishing*. Deliverable. MICO, 2014. URL: http://www.mico-project.eu/wp-content/uploads/2014/12/D2.2.1-D3.2.1-D4.2.1-D5.2.1_First_specs_07112014.pdf.

[SCS13] Robert Sanderson, Paolo Ciccarese, and Herbert Van de Sompel. *OADM Open Annotation Data Model*. W3C Community Draft. http://www.openannotation.org/spec/core/. W3C, Feb. 2013.

[SF14] Sebastian Schaffert and Sergio Fernández. *D6.1.1 System Architecture and Development Guidelines*. Deliverable. MICO, 2014. URL: http://www.mico-project.eu/wp-content/uploads/2014/06/Del-6.1.1-MICO-Architecture.pdf.

[Tro+12] Raphaël Troncy et al. *Media Fragments URI 1.0 (basic)*. W3C Recommendation. http://www.w3.org/TR/2012/REC-media-frags-20120925/. W3C, Sept. 2012.