



D6.1.1

System Architecture and Development Guidelines

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/SRFG/D6.1.1/D/PU/a1
Responsible editor(s):	Sebastian Schaffert, Sergio Fernández
Reviewer(s):	Florian Stegmaier, Patrick Aichroth
Contributing participants:	SRFG, FHG, UP, UMU, UOX, ZA, IO10
Contributing workpackages:	WP2, WP3, WP4, WP5
Contractual date of delivery:	30 April 2014

Abstract

This deliverable describes an initial architecture for the MICO platform based on high-level requirements known already at the beginning of the project. It focusses on the core architecture, i.e. infrastructure, communication, persistence, orchestration, and technological decisions, as well as on development infrastructure and best practices, and is supposed to give guidance on how to start development of the platform. Requirements currently gathered in WP7/WP8 and specifications under development in WP2-WP5 will be taken into account in a revised version of this document.

Keyword List

architecture, best practices, infrastructure

System Architecture and Development Guidelines

Sebastian Schaffert, Sergio Fernández

Salzburg Research, Austria
Email: {firstname.lastname}@salzburgresearch.at

30 April 2014

Abstract

This deliverable describes an initial architecture for the MICO platform based on high-level requirements known already at the beginning of the project. It focusses on the core architecture, i.e. infrastructure, communication, persistence, orchestration, and technological decisions, as well as on development infrastructure and best practices, and is supposed to give guidance on how to start development of the platform. Requirements currently gathered in WP7/WP8 and specifications under development in WP2-WP5 will be taken into account in a revised version of this document.

Keyword List

architecture, best practices, infrastructure

Contents

1	Introduction	2
2	System Architecture	4
2.1	Architecture Overview	4
2.2	Workflow	4
2.3	Data Model and Format	5
2.4	Persistence and Communication	5
2.5	Service Orchestration	5
2.6	Linked Data Publishing	6
2.7	Search and Recommendations	6
3	Data Model	7
3.1	Content Items	7
3.1.1	Structure	7
3.1.2	Content Item Metadata	8
3.1.3	Content Parts	9
3.1.4	Analysis Parts	10
3.2	Media Fragments URI	10
3.3	Ontology for Media Resources	11
3.4	Analysis Results	13
3.5	Data Provenance	15
4	Communication and Persistence	16
4.1	Distributed File System	16
4.2	Event-Based Communication	17
4.3	Metadata Storage	18
5	Service Orchestration	20
5.1	Component Types	20
5.2	Batch Processing	20
5.2.1	Injecting Content Items	20
5.2.2	Starting Analysis Process	21
5.2.3	Receiving Result Callback	21
5.3	Service Description	21
5.4	Execution Plans	21
5.5	Analysis Process	22
6	Development Infrastructure	24
6.1	Components	24
6.1.1	Website	24
6.1.2	Mailing lists	24
6.1.3	Documentation	25
6.1.4	Issue Tracking	25
6.1.5	Source Code Management	25
6.1.6	Continuous Integration	25
6.2	Licensing and Data Protection	26

7	Development Guidelines and Best Practices	27
7.1	Code Conventions	27
7.2	Code Practices	27
7.3	Documentation	28
7.4	Issues Management	29
7.5	Code Management	31
7.6	Build Management	32
7.7	Testing and Continuous Integration	33

List of Figures

1	MICO Distributed System Architecture	4
2	Content Item representation as RDF graph using the Ontology for Media Resources . .	9
3	Content Part representation as RDF graph using the Ontology for Media Resources . .	10
4	Apache Stanbol Enhancement Structure	14
5	Apache Hadoop HDFS Architecture	16
6	Apache Zookeeper Node Hierarchy	17
7	OpenRDF Sesame API stack	18
8	MICO Execution Plan (Example)	22
9	MICO Analysis Process (Example)	23
10	Development infrastructure overview	24
11	Projects in Jira for each WP	30
12	GitFlow overview	31
13	Example of the usage of smart commits in Jira	32
14	Overall deployment process	33

1 Introduction

This document describes an initial version of the MICO system architecture, which is based on initial high-level requirements that were known at the beginning of the project. As a result, the architecture described in this document just documents initial technology decisions and does not go into much detail regarding concrete APIs or data models. More fine-grained requirements and resulting architectural decisions will be included in a revised version of the system architecture once the requirements analysis in the use case work packages WP7 and WP8 and the state of the art analysis in WP2-WP5 has been finished and a first prototype has been implemented.

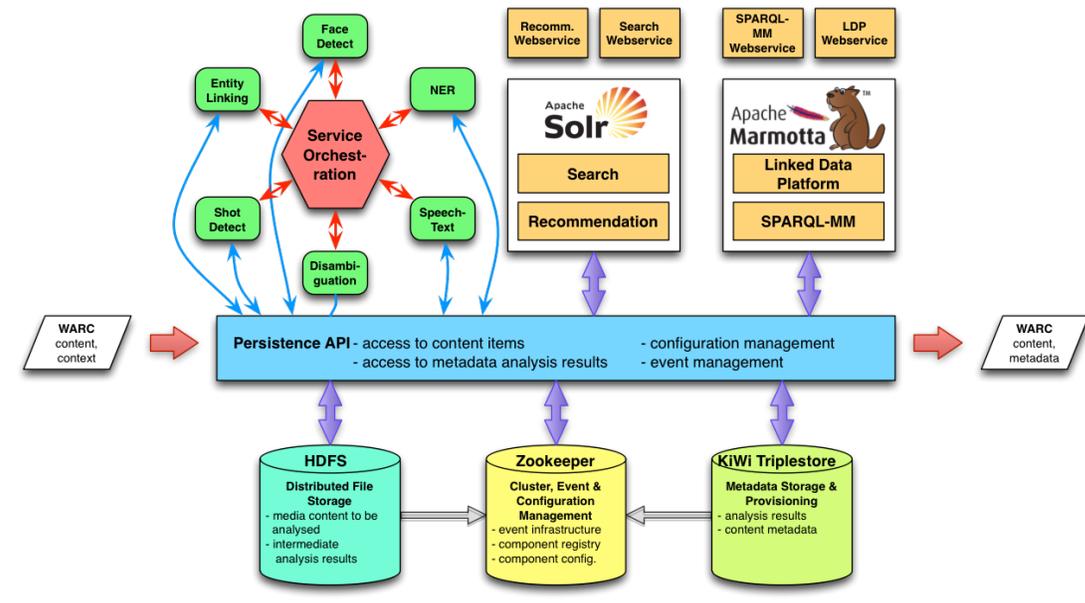
The MICO system architecture is a distributed service-oriented architecture supposed to run in a cluster of servers providing custom functionalities. The following sections provide an overview over the initial architecture that will be implemented in the first iteration of the platform development. The architecture is based on the discussions and (preliminary) outcomes of the requirements analysis process in deliverables D7.1.1 and D8.1.1. The following high-level requirements have been guiding principles of the development:

1. **large content items:** with media content, even individual content items can be very large; as a consequence, the computation needs to move at least partly to the data instead of the data to the computation
2. **composite content items:** content items will often consist of a collection of related elements, e.g. a textual document with embedded images or videos; as a consequence, a container format is required for representing content items
3. **independent analysis components:** there is a plethora of analysis components that will be used, each contributing a small part to the overall analysis results; these components will remain independent but require orchestration
4. **Java and C/C++:** analysis services might be implemented in different programming languages and even environments; it is therefore necessary to provide a communications and persistence layer accessible by all these environments
5. **user feedback:** users might be involved in the analysis process, e.g. by delegating certain decisions where the machine is not confident to crowdsourcing
6. **metadata publishing:** for each analysed content item, the metadata needs to be available using standard formats (e.g. Linked Data)
7. **querying of metadata, content, and context:** for each content item, it needs to be possible to query the metadata, the content, and related information through an extension of SPARQL
8. **search and recommendations:** content items need to be available to full-text search and recommendations based on extracted features

This document only covers the core architecture (i.e. initial technological decisions, infrastructure, communication, configuration, etc). Analysis components, data models, algorithms and interfaces that go beyond the core architecture are described in the state-of-the-art deliverables D2.1.1/D3.1.1/D4.1.1/D5.1.1 and later in the specifications and models in WP2-WP5. More detailed requirements are given in deliverables D7.1.1/D8.1.1 and will be taken into account in the revised version of this document.

The following sections will first introduce into the first version of the architecture (Section 2) and data model (Section 3), and then go into more detail for the most central aspects of communication and persistence (Section 4) and service orchestration (Section 5). Afterwards, we describe the development infrastructure (Section 6) and best practices used in the project (Section 7).

Figure 1 MICO Distributed System Architecture



2 System Architecture

2.1 Architecture Overview

The MICO framework needs to be able to provide configurable analysis of different kinds of media content using different kinds of mostly independent services. In addition, these services might be implemented in different programming languages and run on different servers in a cluster. To accommodate for all these technical requirements, the framework will use a **distributed service-oriented architecture**, depicted in Figure 1. This kind of architecture allows for a low coupling of components with a shared infrastructure for communication and persistence. In this section, we give a short overview over the system architecture. Sections 3, 4 and 5 then describe central components in more detail.

2.2 Workflow

The main workflow implemented by the framework is the analysis of media content items and subsequent publishing of analysis results, either for search, querying and recommendations, or just for further processing by the user who started the analysis. In Figure 1, the main workflow is depicted left-to-right:

1. a content item (with its context) is loaded into the framework, (temporarily) stored by the shared persistence component and a batch process for analysis started
2. the service orchestration component builds an execution plan, stores it with the content item metadata in the persistence component, and signals that a new content item is available for analysis
3. analysis components process the content item according to the execution plan, storing additional analysis results as part of the content item metadata in the persistence component, and signal other components once they are finished

4. the analysed content item is either exported for further processing, or made available for access, querying, search or recommendations inside the platform

2.3 Data Model and Format

The main objective of MICO is to analyse “media in context”. For this reason, it is necessary to represent the content to be analysed in a “container” that allows combining different elements, e.g. HTML, images and video. A format that is commonly used in the archiving community and by web crawlers for this purpose is WARC¹ (Web ARChive). WARC is a container format allowing to represent content items with different parts, including meta-data.

In MICO, WARC will be used as the foundation to develop a format for representing media content items in their context. As depicted in Figure 1, the analysis process takes an archive file as input, runs the different analysis components on its content, and allows retrieving the archive file, including a new part containing the analysis results.

The final format used by the MICO architecture will follow similar ideas to WARC but adapt them to the requirements of multimedia processing. Particularly, Section 3 describes a model for representing analysis results in RDF. An additional record using this model will be included in the archive file during analysis.

2.4 Persistence and Communication

Since the different analysis services potentially run on different servers in a cluster, the MICO framework will offer a shared persistence and communication component. This component has the following main functionalities:

- **access to content items:** provides access to all elements of a content item to the different services
- **access to metadata:** provides access to the initial content item metadata (e.g. retrieval information) as well as the current analysis results to the different services
- **configuration management:** provides a central storage for service configuration
- **event management:** provides a central facility for sending messages, either directly between sender and receiver or via publish/subscribe events

Persistence and communication will be implemented using a distributed file system for content item storage (e.g. Hadoop HDFS²) as well as a triple store for metadata storage (Apache Marmotta³). A candidate for configuration and event management is Apache Zookeeper⁴. Cluster-wide event management might also be implemented using Hazelcast⁵.

2.5 Service Orchestration

Since the analysis services are mostly independent components, a major component in the architecture is *service orchestration*. The service orchestration component is the central point for coordinating the

¹http://archive-access.sourceforge.net/warc/warc_file_format-0.16.html

²<http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>

³<http://marmotta.apache.org>

⁴<http://zookeeper.apache.org>

⁵<http://www.hazelcast.org>

execution of analysis components in the proper order. For this purpose, it will automatically build an *execution plan* for each analysis request (see Section 5.4). Execution plans are based on declarative *service descriptions* specifying the input and output dependencies of the service. A candidate for these service descriptions is a clearly defined subset of OWL-S⁶.

2.6 Linked Data Publishing

The analysis results and metadata about content items are made available as Linked Data⁷ for further access, using the data model developed in workpackage WP3. The Linked Data Publishing component uses Apache Marmotta, a reference implementation of the Linked Data Platform⁸. Apache Marmotta will be extended by the multimedia querying functionalities developed in workpackage WP4.

2.7 Search and Recommendations

To provide search and recommendation functionality, the platform also offers an installation of Apache SOLR⁹ where content items and their analysis results will be indexed for semantic search and full-text search, as well as for simple content-based recommendations. Indexing is triggered once the analysis of a content item has been completed.

⁶<http://www.w3.org/Submission/OWL-S/>

⁷<http://www.w3.org/DesignIssues/LinkedData.html>

⁸<http://www.w3.org/TR/ldp/>

⁹<http://lucene.apache.org/solr/>

3 Data Model

This section describes an initial version of the core data models used in the MICO platform, in particular for representing collections of media content and corresponding analysis results. Since the outcomes of work packages WP2 and WP3 are not yet available, this data model is just an initial version based mostly on existing standards and formats. It will be refined during the project.

3.1 Content Items

Content Items are the representation of media resources together with their context in the MICO platform. With *context* we mean all other resources that are directly related to the main object. For example, an HTML document might contain images and videos – the content item would in this case be the HTML document itself, as well as all images and videos it contains. Grouping this information is a necessary precondition for being able to analyse *media in context*.

Over the years, different container formats have been developed for grouping related media resources. These mostly fall into two categories:

- *media containers for players*: these are container formats developed for the purpose of playing a media resource in a media player, e.g. AVI, WebM, MKV. They are abstracting away from the technical details of encoding and compressing the media content and often allow to include additional data like metadata, subtitles, etc.
- *archive formats*: these are containers primarily designed for the purpose of archiving collections of arbitrary resources. In the context of Web Archiving, the format WARC¹⁰ has gained some traction. Also, general archive formats like ZIP, or specialised variants like JAR are often used. Particularly interesting are also open archive formats recently developed for office documents like the OpenDocument¹¹ format used e.g. by OpenOffice.

In the context of MICO, media containers are too specialised to be useful for our purposes because they are mainly concerned with the technical details required for delivering media resources to the end user (metadata in this sense includes codecs used, compression format, etc). On the other hand, archive formats are very promising, even though they need to be adapted/extended for our needs. In this initial version of the architecture, we therefore loosely build on ideas from the WARC format, which has been established as a de-facto standard in the Web Archiving and Crawling community. It is e.g. used by the Internet Archive and the Heritrix Crawler. However, instead of simply concatenating raw records as is done by WARC, we organize them in a file-system like structure, and for representing metadata we use a more flexible RDF representation. This considerably simplifies accessing individual records.

3.1.1 Structure

The general structure of content items in the MICO framework must satisfy a number of (technical) requirements:

- **multiple content parts**: it must be possible to store and provide access to multiple content parts of different media types as discussed above

¹⁰http://archive-access.sourceforge.net/warc/warc_file_format-0.16.html

¹¹<http://opendocumentformat.org/>

- **content part metadata:** it must be possible to store and provide access to the explicit metadata available for each content part, e.g. MIME type, creation time, provenance, existing manually added metadata (creator, title, rights, ...)
- **(intermediate) analysis results:** it must be possible to store, update and provide access to analysis data generated in the analysis process by different analysis components
- **easy access inside and outside the framework:** the content item data and metadata needs to be easy to access both inside the framework during analysis and outside the framework when packaging or returning for further processing

Based on these technical requirements, we use the following file system like structure for content items in the initial version of the MICO framework:

```
Content Item
|- metadata.rdf          content item metadata (provenance, creation, records, ...)
|- execution.rdf        execution plan and metadata (timing, dependencies, ...)
|- result.rdf           shared RDF graph for analysis results (annotations, ...)
|- content              folder containing content parts
|  |- 37a7270cf4301e4e  folder for content part 1 (hash value)
|  |  |- metadata.rdf  content part metadata (format, creation, links, ...)
|  |  +- content.html  content part binary data
|  |- a67117355425712d folder for content part 2
|  |  |- metadata.rdf  content part metadata (format, creation, links, ...)
|  |  +- content.jpg   content part binary data
|  ...
|- analysis             folder containing analysis parts
|  |- d0cafca15883337  folder for analysis part 1 (hash value)
|  |  |- metadata.rdf  analysis part metadata (format, creation, links, ...)
|  |  +- content.bin   analysis part binary data
|  |- dele53a28dda33f4 folder for analysis part 2
|  |  |- metadata.rdf  analysis part metadata (format, creation, links, ...)
|  |  +- content.bin   analysis part binary data
|  ...
...
```

In general, the structure consists of three parts: content item metadata (metadata.rdf, execution.rdf, result.rdf), content parts, and analysis parts. The following sections describe briefly the intention of each of the elements in this structure.

3.1.2 Content Item Metadata

The metadata of a content item (files metadata.rdf, execution.rdf and result.rdf above) are RDF graphs storing general metadata about the whole content item. They can be queried and updated by the different analysis components during the analysis process:

- metadata.rdf contains metadata available even before the analysis process started, i.e. information about the content item itself (provenance, creation time, etc) and its parts. In the initial version of the MICO framework, this file will mostly follow the Ontology for Media Resources as described in Section 3.3 below. In this context, the content item itself is a ma-ont:Collection,

Figure 2 Content Item representation as RDF graph using the Ontology for Media Resources



while the individual content parts are `ma-ont:MediaResource` with a `ma-ont:hasMember` relation to the content item (see Figure 2). The metadata might also include further available explicit information, e.g. extracted from other metadata formats like EXIF using the mapping tables provided by the Ontology for Media Resources.

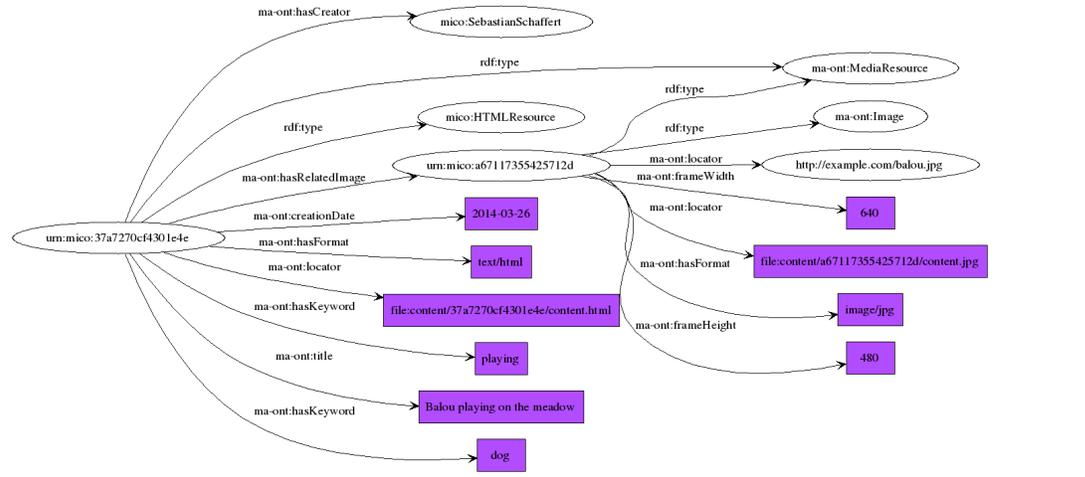
- `execution.rdf` contains a description of the execution plan as well as statistics about the actual execution. The execution plan as well as its RDF representation is described in more details in Section 5.
- `result.rdf` contains the (intermediate and final) structured results of the analysis process. In the initial implementation of the MICO framework, this structure will follow the Apache Stanbol enhancement structure described in Section 3.4 below.

3.1.3 Content Parts

Content Parts are individual content objects that are part of the original collection (content item). A content part can be of any type, e.g. an HTML text, an image, or a video part. Content parts are stored in the file system structure in the folder `content/`. Each content part consists of an identifier, a metadata description in RDF format, and a binary representation.

Identifier. Each content part is identified by a unique identifier (e.g. an MD5 hash of the content). This identifier is used as both, name of the folder where the content part is stored, and URI of the content part in RDF metadata descriptions. For example, the data for the content part with identifier `37a7270cf4301e4e` is stored in the folder `content/37a7270cf4301e4e`, and its URI is `<urn:mico:37a7270cf4301e4e>`.

Figure 3 Content Part representation as RDF graph using the Ontology for Media Resources



Metadata. Like for content items, the initial metadata for each content part is represented in RDF format using the Ontology for Media Resources in a file called `metadata.rdf` for each content part. The metadata contains additional content part specific information that might be relevant for analysis (e.g. format, title, creator, ...). An example of such metadata is given in Figure 3.

Binary Data. The binary data of the content part (e.g. the JPG image or the HTML text) is stored in a file called `content.xyz` in the folder for the content part. The file suffix can be chosen based on the content type of the data. A reference to the binary content should be given in the metadata using the `ma-ont:locator` property.

3.1.4 Analysis Parts

Analysis parts are individual content parts created during the analysis process, e.g. key frame images for a video track, feature vectors for images, POS tags for text, etc. Analysis parts are stored in the file system structure in the folder `analysis/`. Like content parts, each analysis part consists of an identifier, a metadata description in RDF format, and a binary representation (see above). Analysers need to update the main content item's `metadata.rdf` to refer to the additional analysis parts created during the analysis process.

3.2 Media Fragments URI

Media Fragments URI¹² is a recommendation recently issued by the World Wide Web Consortium (W3C). Its purpose is to allow not only addressing media objects on the Web themselves but also *regions* or *fragments* inside such media objects in a way that they can be both uniquely identified (e.g. for adding metadata) and directly addressed (e.g. in a browser).

Media Fragments URI builds on the main URI specification and just offers an additional convention for the *fragment part*. Consider for example the following URI

¹²<http://www.w3.org/TR/media-frag/>

`http://www.example.org/video.ogv#t=60,100`

The fragment part is the part following the # symbol in the URI (i.e. `t=60,100`, addressing the time fragment between seconds 60 and 100). This approach is already in wide-spread use in HTML to address sections inside a document. In the context of media documents, the following dimensions are supported:

- *temporal*, denoting a time range in the original media, e.g. “between seconds 60 and 100” as in the example above; Syntax:

`t=<start>,<end>`

- *spatial*, denoting a range of pixels in the original media, e.g. “a rectangle with size (100,100) with its top-left at coordinate (10,10)”; Syntax:

`xywh=<x-coord>,<y-coord>,<width>,<height>`

- *track*, denoting a qualified track in the original media, e.g. “the english audio and the video track”; Syntax:

`track=<track-name>`

- *id*, denoting an identified temporal fragment in the original media, e.g. “Chapter 2”; Syntax:

`id=<identifier>`

Dimensions can be arbitrarily combined using the & operator. Note that the syntax description given above is not complete. Please refer to the specification for further details. The MICO platform will use the Media Fragments URI specification to refer to regions (temporal and spatial) in media documents, both for adding metadata descriptions and for retrieval.

3.3 Ontology for Media Resources

The *Ontology for Media Resources*¹³ is a standardised vocabulary for describing media resources on the Web. It provides both commonly used metadata properties and mappings to the most wide spread media formats like XMP, ID3, and YouTube. The following table (from the specification) summarizes the core properties offered by the Ontology for Media Resources:

Name	Description
Identification	
identifier	A URI identifying a media resource, which can be either an abstract concept (e.g., Hamlet) or a specific object (e.g., an MPEG-4 encoding of the English version of "Hamlet").
title	A tuple that specifies the title or name given to the resource. The type can be used to optionally define the category of the title.

¹³<http://www.w3.org/TR/mediaont-10/>

language	The language used in the resource. We recommend to use a controlled vocabulary such as BCP 47.
locator	The logical address at which the resource can be accessed (e.g. a URL, or a DVB URI).
Creation	
contributor	A tuple identifying the agent, using either a URI (recommended best practice) or plain text. The role can be used to optionally define the nature of the contribution (e.g., actor, cameraman, director, singer, author, artist, or other role types).
creator	A tuple identifying the author of the resource, using either a URI (recommended best practice) or plain text. The role can be used to optionally define the category of author (e.g., playwright or author). The role is defined as plain text.
date	A tuple defining the date and time that the resource was created. The type can be used to optionally define the category of creation date (e.g., release date, date recorded, or date edited).
location	A tuple identifying a name or a set of geographic coordinates, in a given system, that describe where the resource has been created, developed, recorded, or otherwise authored. The name can be defined using either a URI (recommended best practice) or plain text. The geographic coordinates include longitude, latitude and an optional altitude information, in a given geo-coordinate system.
Content description	
description	Free-form text describing the content of the resource.
keyword	A concept, descriptive phrase or keyword that specifies the topic of the resource, using either a URI (recommended best practice) or plain text. In addition, the concept, descriptive phrase, or keyword contained in this element SHOULD be taken from an ontology or a controlled vocabulary.
genre	The category of the content of the resource, using either a URI from a controlled vocabulary or ontology (recommended best practice) or plain text.
rating	The rating value (e.g., customer rating, review, audience appreciation), specified by a tuple defining the rating value, an optional rating person or organization defined as either a URI (recommended best practice) or as plain text, and an optional voting range.
Relational	
relation	A tuple that identifies a resource that the current resource is related with (using either a URI -recommended best practice- or plain text), and optionally, specifies the nature of the relationship. An example is a listing of content that has a (possibly named) relationship to another content, such as the trailer of a movie, or the summary of a media resource.
collection	The name of the collection (using either a URI or plain text) from which the resource originates or to which it belongs. We recommend to use a URI, as a best practice.
Rights	
copyright	A tuple containing the copyright statement associated with the resource and optionally, the identifier of the copyright holder. Issues related to Digital Rights Management are out of scope for this specification, apart from the metadata supported by the copyright and policy attributes.

policy	A tuple containing a policy statement either human readable as a string or machine resolvable as a URI, and the type of the policy to provide more information as to the nature of the policy.
Distribution	
publisher	The publisher of a resource, defined as either a URI or plain text. We recommend, as a best practice, to define the publisher as a URI.
targetAudience	A tuple identifying the audience being addressed (demographic class, parental guidance group, or geographical region) and an optional classification system (e.g., a parental guidance issuing agency).
Fragment	
fragment	A tuple containing a fragment identifier and optionally, its role. A fragment is a portion of the resource, as defined by the [MediaFragment] Working Group.
namedFragment	A tuple containing a named fragment identifier and its label.
Technical Properties	
frameSize	A tuple defining the frame size of the resource (e.g., width and height of 720 and 480 units, respectively). The units can be optionally specified; if the units are not specified, then the values MUST be interpreted as pixels.
compression	The compression type used. For container files (e.g., QuickTime, AVI), the compression is not defined by the format, as a container file can have several tracks that each use different encodings. In such a case, several compression instances should be used. Thus, querying the compression property of the track media fragments will return different values for each track fragment. Either or both of two values may be supplied: a URI, and a free-form string which can be used for user display or when the naming convention is lost or unknown.
duration	The actual duration of the resource. The units are defined to be seconds.
format	The MIME type of the resource (e.g., wrapper or bucket media types, container types), ideally including as much information as possible about the resource such as media type parameters.
samplingRate	The audio sampling rate. The units are defined to be samples/second.
frameRate	The video frame rate. The units are defined to be frames/second.
averageBitRate	The average bit rate. The units are defined to be kbps.
numTracks	A tuple defining the number of tracks of a resource, optionally followed by the type of track (e.g., video, audio, or subtitle).

The Ontology for Media Resources is very useful as a format for describing common metadata properties of media resources, especially because of its extensive mapping tables to other such formats. It will therefore be used in the MICO system to describe this kind of metadata about media objects.

3.4 Analysis Results

One of the main problems in information extraction is the plethora of input and result formats and models used by the different analysis components. For example, a POS tagger might take plain text and return plain text where each word is augmented with its POS tag. This makes it very hard to integrate different analysis components for computing combined formats.

In MICO, one of the main goals is to make analysis components interoperable. For this reason, we are developing (in WP2, WP3 and WP6) a shared metadata model for representing analysis results and inputs. The format for analysis results will use the RDF model (and syntax) as data structure and will be

Entity `dbpedia:Bob_Marley`. However it is also stated that there is a disambiguation with an other person `dbpedia:Bob_Marley_(comedian)`.

The dashed relations are also important as they are used to formally describe the extraction context: which EnhancementEngine has extracted a feature from what ContentItem. If even more contextual information are needed, users can combine those information with the ExecutionMetadata collected during the enhancement process.

An important distinction introduced by Apache Stanbol is the difference between TextAnnotation and EntityAnnotation: whereas a TextAnnotation just describes a region of the content, an EntityAnnotation links such TextAnnotations to one or more resources from external datasets (e.g. Linked Data servers). The first version of the MICO platform keeps this main concept, but extends the structure with ways to describe also regions in media documents using the Media Fragments URI specification and Ontology for Media Resources introduced before. This model will be refined once results from the Metadata Publishing workpackage WP3 are available.

3.5 Data Provenance

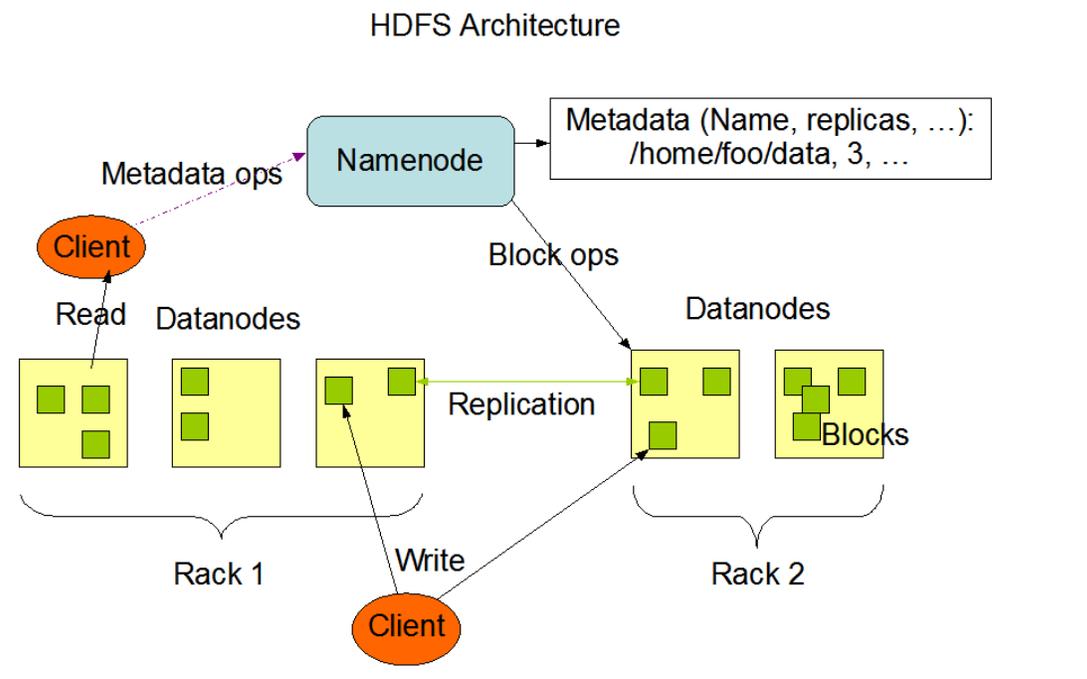
In the whole analysis process it is important to track which information is coming from which source. e.g. for deciding how trustworthy an analysis result is. Provenance information is stored as part of the RDF metadata for each content and analysis part as well as for the analysis result:

- **content item:** provenance information for the whole content item contains where the content item originally comes from (e.g. website URL), time when the content item has been created, who created the content item, when it has been modified and by whom, and when it has been uploaded / exported from the framework
- **content parts:** provenance information for each content part contains where the content part comes from (e.g. retrieval information from a crawling process), time when the content part has been created, and who created the content part
- **analysis parts:** provenance information for each analysis part contains information about the analysis components that created or modified the analysis part, time when this happened, and the content parts that were used in the analysis process
- **analysis result:** provenance information contains the execution plan, which analysis components were used in the analysis process at what times, what content parts were used by what analysis components, and statistics about the execution (e.g. timing, machine ids, memory usage)

Provenance information in the RDF metadata will be represented using the PROV vocabulary¹⁵, which already allows describing all the information mentioned above. A detailed description of PROV can be found in the MICO State-of-the-Art Deliverable, Section 3.1.1.

¹⁵<http://www.w3.org/TR/prov-o/>

Figure 5 Apache Hadoop HDFS Architecture



4 Communication and Persistence

The Persistence API (see Figure 1) offers a central point of access for all persistence and communication related activities. It will be implemented as both, a collection of Java methods for direct access, and REST web services for components not implemented in Java. Where possible, the MICO framework will also try to provide access through additional APIs, e.g. for C/C++. The core functionalities are

- access to content items (Apache Hadoop HDFS)
- event and configuration management (Apache Zookeeper, Commons Configuration)
- RDF metadata storage and querying (OpenRDF Sesame, Apache Marmotta)

The functionalities are described in more detail in the following sections.

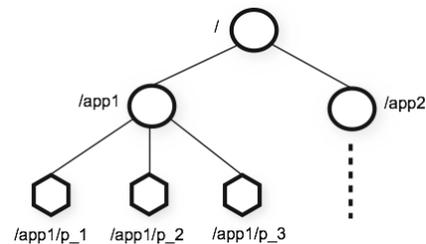
4.1 Distributed File System

The distributed file system API provides file-system like access to content items injected into the MICO framework. Content items are structured as described in Section 3.1. Access to content items is provided using the following APIs:

- **Java API:** the Java API will be implemented based on Apache Hadoop HDFS¹⁶. The advantage of using HDFS is that it already offers different backends and built-in distributed access and storage of large content items.

¹⁶<http://hadoop.apache.org/>

Figure 6 Apache Zookeeper Node Hierarchy



- **REST API:** the REST API will be implemented on top of the Java API, exposing the content item data to services that are not implemented in Java. The REST API will use the WebHDFS implementation already provided by Hadoop.
- **C API :** in case a service not implemented in Java needs more direct access to content items, it is also possible to use a native C API for accessing HDFS. Like the REST API, this is already provided by Hadoop.
- **NFS Gateway:** for third-party services that cannot be adapted to HDFS, the final option is to use the HDFS NFS gateway, which allows to mount a remote HDFS file system like a normal local file system and use normal Unix file operations to interact with content items.

Figure 5 shows the basic architecture of the HDFS filesystem. Data is stored on datanodes and can be replicated on different racks if needed for data locality. Building on HDFS allows to configure the MICO framework for many different kinds of scenarios, including a local file system implementation for testing, and a distributed cluster implementation for high performance demands. HDFS allows the MICO components to carry out I/O intensive computations close to the data instead of moving data from service to service, provided the service is implemented as a Hadoop Job.

4.2 Event-Based Communication

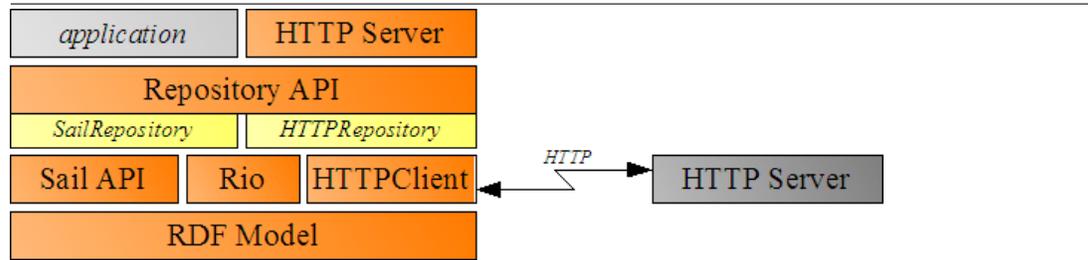
Since the MICO framework implements a service-oriented architecture, the main means of communication between components is *events*. Using an event-based mechanism allows components implemented in different environments and running on different machines to properly interact with each other. The event communication will be implemented based on Apache Zookeeper¹⁷, a centralized service for storing configuration and coordinating/synchronizing different services. The following Zookeeper functionalities will be used by the MICO framework

- **Configuration Management:** allows components and machines to access and store their configuration in a central service; using centralized configuration management, components can be relocated and executed on different machines without too much effort; for configuration management, the MICO framework will implement Zookeeper access through the Apache Commons Configuration API¹⁸
- **Synchronization:** allows components to create locks (e.g. on content items or content parts) in a central service to avoid other components running on different machines to interfere with an

¹⁷<http://zookeeper.apache.org/>

¹⁸<http://commons.apache.org/proper/commons-configuration/>

Figure 7 OpenRDF Sesame API stack



ongoing computation; for synchronization purposes the MICO framework will provide implementations of the Java Lock API backed by Zookeeper

- **Producer-Consumer Queues:** allow for simple distributed event communication where consumers can subscribe to a certain queue while producers (possibly running on different machines) can publish to queues; for event communication, the MICO framework will provide implementations of Java Listeners as well as of the Java Collection API backed by Zookeeper

Zookeeper structures data in a filesystem-like hierarchical structure that can be traversed from the root (see Figure 6). Entries (“nodes”) in this hierarchy can be either normal nodes, which are persistent across restarts, or ephemeral nodes, which exist as long as the client which created it is connected to the server. Nodes can store any kind of data, and clients can register listeners on nodes to be notified of changes to this data. Zookeeper gives certain guarantees that allow using these fundamental structures to implement the functionalities described above.

Using Zookeeper has several advantages for the MICO framework: first, it is a reliable and tested platform used by many enterprise users; second, there is a plethora of libraries for connecting and interacting with Zookeeper servers; and third, it transparently provides the main event communication and configuration management services in a distributed environment – components do not need to take care of the communication.

4.3 Metadata Storage

Content Item metadata and analysis results will be represented using the RDF model. To simplify working with RDF and provide efficient storage and access, the Persistence API will transparently offer access to OpenRDF Sesame¹⁹ repositories to components. Components will be able to request access to any of the RDF metadata and analysis parts of a Content Item. The stack of different APIs offered by Sesame is depicted in Figure 7. The following functionalities will be offered in Java and possibly through REST webservice:

- **RDF import/export:** allows storing and retrieving the RDF data contained in a repository in different serialization formats, e.g. Turtle, RDF/XML, CSV, or JSON-LD
- **Sesame Repository API:** provides Java API-level access to the statements and resources contained in the RDF data using the OpenRDF Sesame Repository API (Java only)

¹⁹<http://www.openrdf.org/>

- **SPARQL Query and Update:** allows querying and updating RDF data using the SPARQL 1.1 Query and Update specifications, either directly through a Java API or indirectly using the SPARQL 1.1 REST protocol

OpenRDF Sesame is the de-facto standard for processing RDF data. It offers many tools and higher-level libraries to work with RDF. Sesame supports many different backends for storing data physically. In MICO, we will use either the KiWi backend offered by Apache Marmotta²⁰ or the Hadoop-based Titan backend²¹. REST access to RDF data will be provided through Apache Marmotta.

²⁰<http://marmotta.apache.org>

²¹<http://thinkaurelius.github.io/titan/>

5 Service Orchestration

A central part of the MICO architecture is the service orchestration component. The service orchestration component computes for each input content item an appropriate individual execution plan that is then followed by the analysis components. Execution plans are build based on component descriptions and dependencies. The following sections illustrate the initial draft for this process.

5.1 Component Types

The MICO framework supports arbitrary analysis services that can work on content items. In general, there are three categories of analysis components:

- **native analysis components** are implemented inside the MICO framework in Java; they communicate with events through the Zookeeper communication API (Section 4.2) and have direct access to content items, their parts, and metadata
- **external analysis components** are implemented outside the MICO framework in any language; they communicate with events through the Zookeeper communication API and access content items, their parts and their metadata through the REST API
- **human interaction components** allow to involve humans in analysis tasks, e.g. to confirm intermediate results; they are implemented typically as web applications and are notified either through the Zookeeper communication API or through REST webservice callbacks; they access content items, their parts and their metadata through the REST API

5.2 Batch Processing

Since many analysis tasks carried out in the MICO framework will work on large amounts of data and will therefore be long-running processes, the execution model used by the framework is batch processing. This means a user can upload or inject a content item, trigger the analysis process, and is then notified once the analysis result (or possibly intermediate results) are available.

5.2.1 Injecting Content Items

Content items are injected into the system by storing them in the HDFS file system. HDFS offers a number of different ways to carry out this storage:

- **HDFS command line tools** allow interacting with the HDFS file system, including copying a file from the local file system to the HDFS
- **NFS server** allows mounting an HDFS file system like a normal network file system; content items can then be copied using the normal file system operations (e.g. using a GUI explorer)
- **WebHDFS** allows uploading content items using REST webservice interactions using any kind of HTTP client

5.2.2 Starting Analysis Process

The real analysis process is started explicitly by the client by calling either a method in the Java API or by calling a REST webservice. When starting the analysis process, the service orchestration component builds an execution plan, optionally asks the user for confirmation or changes to the execution plan, and signals analysers to start working. Users can observe the progress of execution while it is processing.

5.2.3 Receiving Result Callback

Clients can register callbacks to be notified when an analysis process for a content item has finished. Optionally, a client can also register to be notified whenever a certain result is available (e.g. a person detected) or when user input is required. Callbacks can be either Java Listeners in the Java API or external REST services that are called by the platform.

5.3 Service Description

The service orchestration component builds execution plans semi-automatically as described in the following section. For this purpose, each analysis component needs a (lightweight) description of its functionalities. The following minimal set of properties needs to be described:

- **input requirements:** requirements a content item needs to fulfill at least for the analysis component to be able to process it, e.g. “plain text” or “image”; input requirements can be fulfilled either directly by the initial content item or by the output generated by another analysis component
- **output characteristics:** the kind of information added by the analysis component to a content item after processing, e.g. “face regions” or “POS tags”; output characteristics can be either the final result format or intermediate results needed by other components
- **execution characteristics:** information about the execution of the component, e.g. involved computation costs, expected quality of results, etc.

In the initial implementation, we will start by deriving service descriptions from known formats like OWL-S²². However, analysis components are both more specific and more lightweight than general service descriptions, so we will refine service descriptions as needed. Input/output characteristics will be defined using a simple hierarchical vocabulary represented in SKOS format.

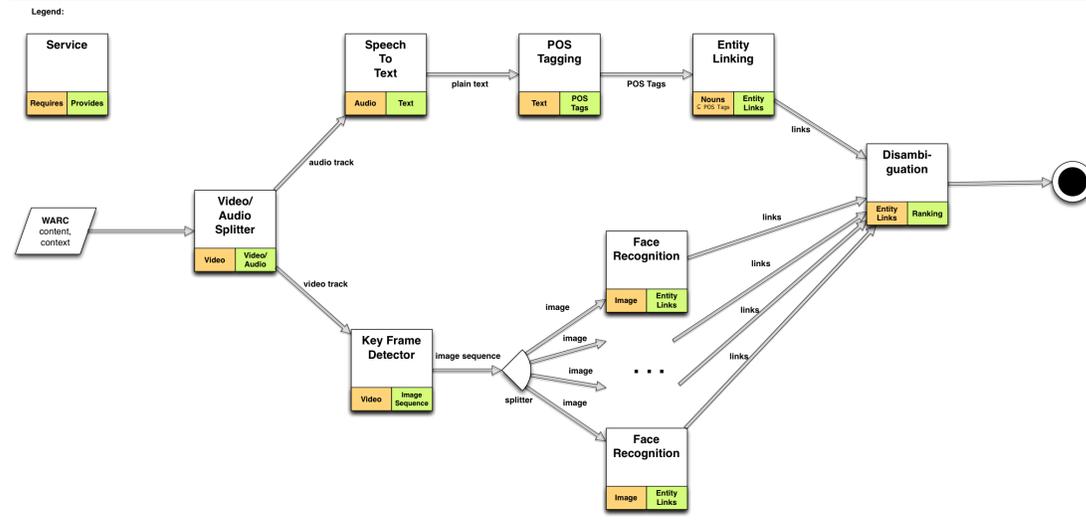
5.4 Execution Plans

When an analysis process is started by a client, the orchestration component first computes an execution plan for the content item and stores it as part of the content item metadata. Execution plans are essentially dependency graphs for a given input and output specification, e.g. “I have video (input) and I want to link to the persons occurring in the video (output)”. The nodes in this dependency graph are analysis components, and there is an edge from component A to component B if A provides as output information that B requires as input. An example of such a dependency graph is given in Figure 8. Execution plans are represented in RDF, an appropriate vocabulary will be specified during the implementation of the platform.

Execution plans will be computed based on dependency graphs using a topological sorting and eliminating graph components that do not contribute to the flow of information from the input content item

²²<http://www.w3.org/Submission/OWL-S/>

Figure 8 MICO Execution Plan (Example)



to the output result. When computing the execution plan for a content item and an output specification, one of several ambiguities might occur:

- **dependencies cannot be satisfied**, i.e. the output cannot be generated from the input because the framework does not contain a component to carry out the step from a certain input to a certain output (e.g. the input is “video”, the output “persons”, but there is no component to transform a video into images); in this case, the MICO framework should notify the user about the problem and suggest possible “minimum” fixes (e.g. “provide a component to transform video into images”, or “manually confirm persons detected in an image”)
- **there are multiple alternatives**, i.e. different chains of analysis components can deliver the same result; in this case, the user should be notified about the possible alternatives and be able to select one or more paths; in case the user chooses several alternatives, the execution plan must be capable of splitting and merging (intermediate) results, in the most simple case by working on the same data and waiting until all previous components are finished (as shown in Figure 8)

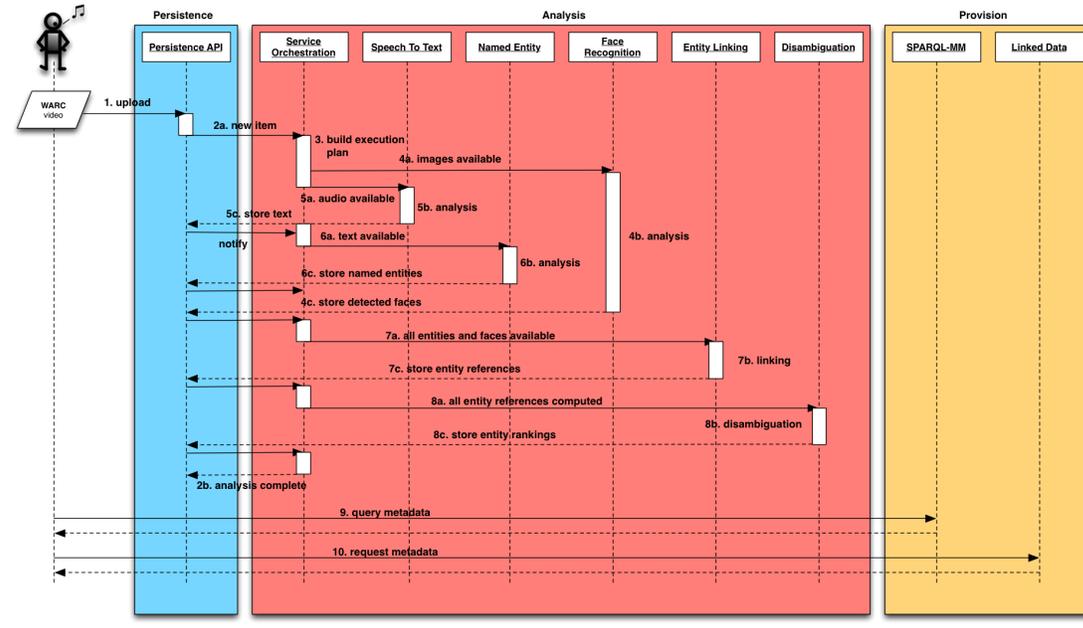
Dependency graphs will be implemented using the JGraphT graph library.²³ JGraphT offers a collection of useful algorithms over abstract graphs, including topological sorting and various other traversal algorithms.

5.5 Analysis Process

Analysis processes follow the execution plan in topologically sorted order, possibly executing in parallel and on different cluster nodes those components that are independent from each other. The orchestration component is the central point of coordination – it has full knowledge of the execution plan and triggers execution of the next components once preconditions are satisfied (e.g. because another component finished). Components communicate with the orchestration component through event notification (as

²³<http://jgrapht.org/>

Figure 9 MICO Analysis Process (Example)



described in Section 4.2). The outcome of the analysis process is represented in RDF as described in Section 3.4.

Figure 9 shows an example of a typical analysis process. In this example, a video is uploaded into the MICO framework, and the analysis outcome should be links from temporal and spatial video fragments to URIs representing persons. Once uploading is complete, the orchestration component builds the execution plan, and notifies both the speech-to-text and the face recognition components, as they can work in parallel. Once the speech recognition has finished transcribing spoken language into plain text, a named entity analysis is carried out on that text. When faces and named entities are available, a linking component connects these objects with background datasets. Finally, a disambiguation component is notified and eliminates unlikely candidates by taking into account the overall context. When the process is finished, the user can either query the analysis metadata using SPARQL or download the whole analysis result from the MICO platform.

6 Development Infrastructure

Every cooperation initiative, specially it is also geographically distributed, requires some infrastructure that allows people to work together as a team. In this section we will describe the main components we initially expect would be required for this project. Although it will probably evolve according to the needs in the future. Then the next section (7) outlines some best practices to follow in the project, and how this infrastructure would be actually used to support such methodologies and best practices.

6.1 Components

The design of this infrastructure has been already introduced in the project kick-off meeting²⁴, in Salzburg, and it will be under continuous evolution during the project execution.

6.1.1 Website

Although it is not strictly part of the development infrastructure, at some point the website will be the hub that the project will use to engage with other research and development communities. So in the end it is an important piece of the infrastructure.

The website²⁵ is online since the very first months of the project. It is home for all the latest news and documentation about the project. As stated below the project is driven by the desire to make the results as accessible as possible and so most of the documentation is available as public reports on the website for easy download in PDF format. The website will also provide access to running technology results in the section "System Demos". All partners are encouraged to document their activities in the "News" section. Further details at the dissemination plan deliverable (D9.2 [PS14]).

6.1.2 Mailing lists

Mailing list are a very common mechanism for effective communication in distributed teams. At this stage the project only requires two mailing lists²⁶: one for the project management office and another

²⁴<http://www.slideshare.net/Wikier/mico-development-infrastructure>

²⁵<http://www.mico-project.eu>

²⁶<http://lists.mico-project.eu>

Figure 10 Development infrastructure overview



one for all general discussions. Since the project is still defining its form, a specific mailing list for development purposes is not yet required, but we expect it would be required sooner than later.

6.1.3 Documentation

The main documentation of the project is hosted at Google Drive²⁷. In addition, technical or scientific documents will make use of the same infrastructure used for code (see Section 6.1.5 below).

6.1.4 Issue Tracking

Tasks tracking is crucial for measuring the actual progress of every project. For MICO, based on our previous experience, we have chosen Jira²⁸ for such need. Jira is a proprietary issue tracking product, developed by Atlassian; it provides bug tracking, issue tracking, and project management functions.

In addition to the regular Jira, two plugins have been enabled too: Jira Agile²⁹ for supporting agile development, and Jira Capture³⁰ for helping us to build and fix the user interfaces of the prototypes produced by the project.

6.1.5 Source Code Management

Source code management is a very important resource for a software-centered project like MICO. Probably every organization already has his own source code management tool. But in this context it is important to have a common tool accessible by all partners, but also capable of managing the variant workflows used within the project. For example, there will be cases where all partners involved in a concrete task will directly work on the same repository; in other cases, maybe because internal procedures, some organizations need to privately work using their own infrastructure, and share only after special conditions; and probably many others. Therefore it is clear a Distributed Version Control Systems (DVCS) provides the good solution to address all these requirements.

After evaluating the different options available, both hosted and managed, the project has created a team at Bitbucket³¹ for managing the code. There we can have unlimited repositories and collaborators, including private and public access when necessary. It also gives us the freedom of choosing between Git or Mercurial, although Git would be preferable as the default one as it will be explained later.

6.1.6 Continuous Integration

Even if there are plenty of open source tools for continuous integrations (Jenkins for example), mainly regarding integration criteria with the other tools (Issue Tracking 6.1.4, Source Code Management 6.1.5), we have selected Bamboo³² for satisfying this requirement. Because Bamboo does more than just run builds and tests; it connects issues, commits, test results, and deploys. So the full MICO picture can be provided for the project team, whatever role: from project managers, to developers and testers, to system administrators.

²⁷<http://drive.google.com/a/salzburgresearch.at/folderview?id=0Bz8MpuEDhMckbDc2N2hUeGNzZDA>

²⁸<https://www.atlassian.com/software/jira>

²⁹<https://www.atlassian.com/software/jira/agile>

³⁰<https://www.atlassian.com/software/jira/capture>

³¹<https://code.mico-project.eu>

³²<http://www.atlassian.com/software/bamboo>

The private instance is already set up and working³³. Although for the moment just building some of the open source projects that are part of the project technology stack, awaiting until actual software development starts in MICO.

6.2 Licensing and Data Protection

All the infrastructure that was described above is brought into the project by using different resources: many (servers' operating systems and so on) are open source; some tools are part of a Community License granted by Atlassian³⁴ to the project; and others (such as Google Drive from SRFG) are part of the common infrastructure of some organizations that they bring into the project. So the project has spent no budget on any software license.

In addition to the licensing details, according to the current European legislation about data privacy³⁵ it is also quite important to clarify where all this infrastructure is actually located. Most of the tools are hosted by SRFG, so they provide the legal framework required for the usage of this infrastructure, both at an individual and organizational level. But there are two exception, since project also uses two cloud services:

- The first one Google Drive, which is used as main document management system for collaborative edition. The account is part of the Google Apps for Business contract of SRFG, and hosted by Google, Inc. According to their general privacy policies³⁶ the data should stay in their European cloud infrastructure, but this is not warranted due internal processes. To ensure as much as possible the privacy of the project data, according to the legal advice from Fraunhofer, on April 17th of 2014 SRFG has extended the contractual conditions with Google, accepting the EU Model Contract Clauses³⁷ and the Data Processing Amendment³⁸. Such new clauses should be enough to warranty a proper collaboration environment according to the current European legislation.
- The second one is Bitbucket, which runs in the Atlassian cloud infrastructure. In this case, their business model does not support signing one-off documents for individual customers. Therefore by default the project has to rely on the general privacy policies³⁹. And, for those cases where individuals or organizations could be specially concerned about the privacy of the data stored in some repositories, we will implement a git-based encrypted solution⁴⁰.

You can not warranty full security and privacy when using telematic tools, specially with those hosted by third-party companies. But at least the project has put in place the main mechanism to assure an acceptable level.

³³<https://ci.mico-project.eu>

³⁴<https://www.atlassian.com/software/views/community-license-request>

³⁵http://ec.europa.eu/justice/data-protection/document/international-transfers/transfer/index_en.htm

³⁶<https://www.google.com/policies/privacy/> (December 20, 2013)

³⁷http://www.google.com/intx/en/enterprise/apps/terms/mcc_terms.html

³⁸https://www.google.com/intx/en/enterprise/apps/terms/dpa_terms.html

³⁹<https://www.atlassian.com/company/privacy> (March 26, 2014)

⁴⁰<https://www.agwa.name/projects/git-crypt/>

7 Development Guidelines and Best Practices

Every development effort requires some kind of agreements and best practices to produce something that can be maintainable in the mid-long term. And this is even more required in a cooperative project like MICO where seven different organizations distributed over Europe will collaborate together to a common technological goal.

7.1 Code Conventions

The technology stack that the consortium brings into the project is mainly based on two programming languages: Java and C/C++. Others might appear in the future for some concrete tasks, but those two conform the core languages that will be used.

Java Code Conventions: So far we are not following strict code conventions for Java coding, although we heavily base it on the Code Conventions for the Java Programming Language [Sun97] originally published by Sun Microsystems, now Oracle, in 1997.

C/C++ Code Conventions: As happens with Java, we are not following strict code conventions for C/C++ coding either. We recommend to common ones, such as the GCC C++ Coding Conventions⁴¹ or the Google C++ Style Guide⁴².

In addition to the two main programming languages are being used in the consortium, other may be used. For instance, front-end programming languages such as Javascript would be used for sure in the user interface building. But for the moment there is no need to specify concrete code conventions for those cases.

One important additional setting is to use **space instead of tabs** for indentation. This ensures a common and stable source code layout over all platforms and environments.

As a general rule, it is always important to assert that the official coding language is English, both for naming (classes, variables, etc) and commenting.

In future iterations of this deliverable we will go thorough with much more detail about to coding guidelines and conventions.

7.2 Code Practices

Every developer or every organization have their own coding practices. But, as a consortium, there are some few things that we should adopt for the benefit of the project. Some of them could look obvious, but it is always good to make explicit some best coding practices that will be recommended to follow in the project. Here we summarize some of them:

Bridge design pattern: The Bridge design pattern [GHJV94] should be extensively used to separate a class's interface from its implementation, allowing to vary or replace the implementation without changing related code. Interfaces should use a meaningful name, where the usage of any additional artificial naming, such as prefix 'I', is disallowed. Concrete implementations are named by adding a suffix to the implemented interface; by default 'Impl' is enough, but something more descriptive is required in case there are available alternative implementations.

⁴¹<http://gcc.gnu.org/wiki/CppConventions>

⁴²<http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>

Inversion of Control and Dependency Injection: Usage of IoC⁴³ makes much easier the potential integration of different components from different authors/partners. Because assuming a dependent object is coupled to the object it needs at run time, and not at construction time, simplifies application code and increases configuration flexibility, avoiding several issues at different scenarios, such as testing. In addition, IoC is very useful in conjunction with a Dependency Injection framework, such as JSR346 [Mui14] for JavaEE or Declarative Services⁴⁴ for OSGi [osg09]. Further, this is aligned with the Bridge pattern describe before.

Service-Oriented Architecture: Service-oriented architecture (commonly SOA) is a software design and software architecture design pattern based on discrete pieces of software providing application functionality as services to other applications, independent of any vendor, product or technology⁴⁵. Due the fact that the architecture outlined in this document requires that different and heterogeneous systems will work together, the usage of this pattern is very recommended for all components of the MICO platform.

RESTful: REST (Representational State Transfer) is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system [Fie00, FT02]. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. Given all these arguments, RESTful will be the preferable interaction pattern across different components when native interaction would not be possible.

Test-Driven Development is a very important approach as it encourages simple designs and inspires confidence. It is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactoring the new code to acceptable standards [Bec03]. Later, in Section 7.7, we will go is much more detail about this aspect.

7.3 Documentation

This section strictly targets the technical documentation of the project. Any other kind of documentation, such as dissemination material or scientific papers, is out of the scope of this document.

The documentation in the code should cover two different levels:

Technical documentation for allowing other developer to use the code/APIs produced. In Java this is done with Javadoc, for C/C++ could be done with Doxygen⁴⁶ or Doc++⁴⁷, and there are equivalent solutions for any other language.

Besides Javadoc/Doxygen, the technical documentation also includes information about the delivered software product, its system requirements and how it can be build and/or installed. This part is specifically intended for software and maintenance engineers.

⁴³<http://martinfowler.com/bliki/InversionOfControl.html>

⁴⁴http://wiki.osgi.org/wiki/Declarative_Services

⁴⁵<http://msdn.microsoft.com/en-us/library/bb833022.aspx>

⁴⁶<http://www.stack.nl/~dimitri/doxygen/>

⁴⁷<http://docpp.sourceforge.net/>

Table 2 Mappings of Jira with DoW's nomenclature

DoW	Jira
WP	Project
Milestone	Version
-	Component
Task	Epic
Deliverable	Deliverable
-	Task
-	Sub-Task
-	Bug

User documentation for providing some hints how to use the software, whatever it comes with a graphical user interface or a command-line one. Final users are those who will require this documentation.

7.4 Issues Management

According to the mapping specific in Table 2, we have installed and configured a private instance of Jira for MICO⁴⁸, which is being used both for project management and software development.

When reporting a new issue, please try to be as descriptive as possible. For instance, the issue summary should be a short and clear statement that indicates the scope of the issue; you are probably being too verbose if you exceed the length of the text field. Use the Environment and Description fields to provide more detailed information. Besides, use the right type and priority to have a better issue management.

With respect to the project itself, Table 2 outlines how Jira maps with the Description of Work (DoW). In Figure 11 it can be seen how it was structured, where each Work Package is a Project in Jira. So each WP Leader has total freedom to organize his WP as preferred, under some common agreements:

- Components are entirely a matter of what each WP requires or needs.
- Deliverables are already associated with their (Epic) Task according DoW.
- All new issues in the project should be linked with their correspondence Epic Task to track the effort and progress.
- Every Epic Task has total freedom for creating as many issues as needed.
- Assigner resolves an issue when work is complete.
- Work Package Leader closes (releases) tasks when are actually delivered (milestone accomplished).

In parallel to every individual working plan in the consortium, we also plan to organize sprints where approach a concrete goal in a more agile style. Scrum methodology [SB02] and Kanban [SKCU77] are

⁴⁸<https://issues.mico-project.eu>

Figure 11 Projects in Jira for each WP

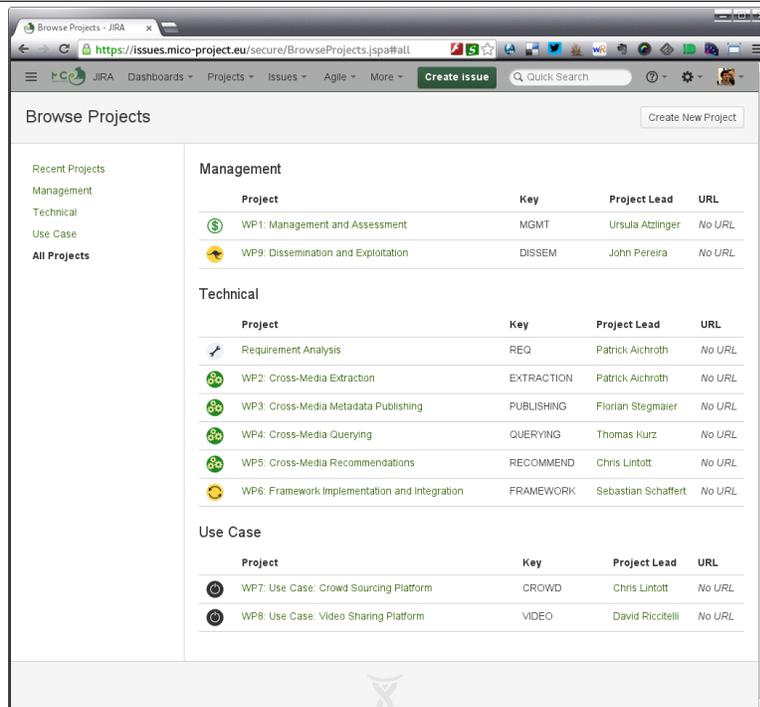
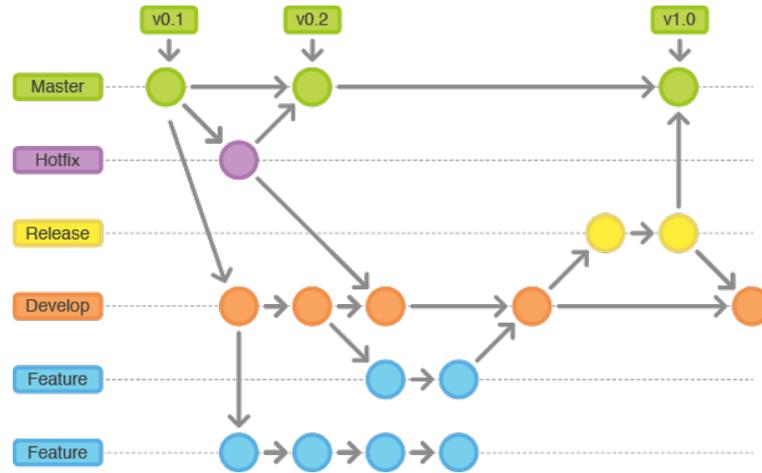


Figure 12 GitFlow overview



supported by the tool, so it is suitable not only for organizing physical hackathons, but remotely working too. Roles (product owner and scrum master) will be assigned to the member of the team in accordance to the goals of each sprint.

7.5 Code Management

The infrastructure described before (Section 6.1.5) could be freely used by all partners of the project. But for the best of it, there are few recommendations to follow:

- Compared to the traditional approach from old source control management tools, where all development happens directly in the `trunk` branch, in MICO we will try to follow a more effective and modern workflow. Particularly we recommend to follow Gitflow^{49,50}, depicted by Figure 12. GitFlow, instead of a single `master` branch, uses two branches to record the history of the project, where `master` stores the official release history, and the `develop` branch serves as an integration branch for features. The whole project development works around the distinction between these two branches. In addition, new features could optionally open feature branches for some topic/s/issues, which are typically identified with the issue key from our issue tracker. And maintenance or hotfix branches could be also used to quickly patch production releases without interrupting the current development cycles.
- For those using Git for the first time, it is very common to (ab)use `git pull`⁵¹. A `git pull` works by doing a `git fetch` followed by a `git merge`. This is OK if your local branch is in sync with the remote branch. Otherwise, the `git merge` will result in a commit graph that looks like a spaghetti because it will do a merge of your local branch with the remote one. A much better approach is to use `git fetch` followed by `git rebase`. That will find the earliest common ancestor between `master` and `origin/master`, move to a temporary space everything

⁴⁹<http://nvie.com/posts/a-successful-git-branching-model/>
⁵⁰<http://www.atlassian.com/git/workflows#!workflow-gitflow>
⁵¹<https://coderwall.com/p/jgn6-q>

Figure 13 Example of the usage of smart commits in Jira



in your local branch that comes after that, fast forward it to the latest commit on `origin/master` and then apply each of the commits that were put aside on top of it, one by one, stopping if conflicts are found so that you can fix them before going on with the `rebase`. But be careful, there are cases (specifically when rebasing a merge commit) when this approach might not be a good idea⁵², so we encourage to use `git` with a bit of understanding.

- Another important recommendation is to align the work done in code with actual tasks of the project. You can process JIRA issues through commit messages, transitioning issues to any status, commenting on issues, and recording time tracking information against issues. This feature is called "smart commits"⁵³. Figure 13 shows how commits are attached to the referenced issues in Jira.

7.6 Build Management

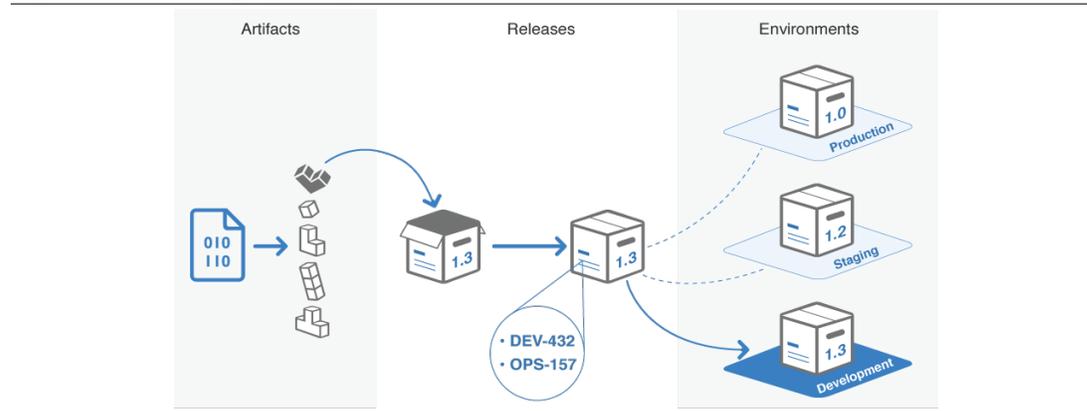
Build management is crucial, and having a fully automatized process in place a strong requirement. It should be possible to build all software artefacts developed within the project, no matter who and where. That means that the tool should:

- have a full build lifecycle management;
- manage dependencies, include transitive ones;
- be available for different operative systems (Windows, Linux and MacOS at least);
- have good support in other development tools (such as IDEs or CIs);
- scale, from small to large multi-module projects;

⁵²<https://coderwall.com/p/jiswdq>

⁵³<http://goo.gl/DZxW3>

Figure 14 Overall deployment process



- and optionally, be extensible.

So, attending all these requirements, the recommendation is to use Apache Maven⁵⁴. Apache Maven is a software project management and comprehension tool; based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. Since this build system is already being used by some of the projects which conform the core of the outlined architecture described in this document, so it looks an obvious decision. Maven is a kind of industry standard in the Java world, although it could be used also for other languages with some plugins, such as NAR for C/C++⁵⁵.

7.7 Testing and Continuous Integration

Even it can not be warranted, testing dramatically increases the quality of the software produced. Writing tests is not a nicest task, and few people have innate skills for it, but with some discipline the results are clear, specially when new features come in and may have lateral effect in other parts of the systems.

In MICO, the technical work packages (WP2, WP3, and WP4) will mainly focus on unit and integration testing. WP6 will do also some integration testing, but mainly component interface and system testing. And the use cases (WP7 and WP8) will be in charge on the acceptance testing for the project.

Following the same practices proposed for coding, agile testing could be used too. Agile testing is a software testing practice that follows the principles of agile software development, involving all members of a cross-functional agile team, with special expertise contributed by testers, to ensure delivering the business value desired by the customer at frequent intervals, working at a sustainable pace. Specification by example is used to capture examples of desired and undesired behavior and guide coding.

Just to anticipate the idea, although it is still a bit early for the project, the infrastructure described before in Section 6.1.6 offers much more than simple Continuous Integration. It actually provides all necessary for implementing Continuous Delivery for the project. The diagram depicted by Figure 14 introduces a workflow that enables it⁵⁶.

⁵⁴<http://maven.apache.org>

⁵⁵<http://maven-nar.github.io>

⁵⁶<https://confluence.atlassian.com/display/BAMBOO/Deployment+projects>

References

- [Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [Fie00] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, doctor in philosophy in information and computer science, University of California, Irvine, 2000.
- [FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, pages 115–150, May 2002.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [Mui14] Pete Muir. Jsr 346: Contexts and dependency injection for javaee 1.1. Java Specification Requests 346, Java Community Process, 2014.
- [osg09] Osgi service platform service compendium: Release 4, version 4.2. Technical report, OSGi Alliance, 2009.
- [PS14] John Pereira and Sebastian Schaffert. D9.2 MICO Dissemination Plan. Deliverable, MICO Project, April 2014.
- [SB02] Ken Schwaber and Mike Beedle. *Agilè software development with scrum*. 2002.
- [SKCU77] Y Sugimori, K Kusunoki, F Cho, and S Uchikawa. Toyota production system and kanban system materialization of just-in-time and respect-for-human system. *The International Journal of Production Research*, 15(6):553–564, 1977.
- [Sun97] Java code conventions. Technical report, Sun Microsystems, Inc, 1997.