D6.2.1
Platform: Initial Version

Grant Agreement No:	610480
Project title:	Media in Context
Project acronym:	MICO
Document type:	D (deliverable)
Nature of document	P (prototype)
Dissemination level:	PU (public)
Document number:	610480/SRFG/D6.2.1/D/PU/a1
Responsible editor(s):	Sebastian Schaffert, Sergio Fernández
Reviewer(s):	Antonio Pérez, Jakob Frank
Contributing participants:	SRFG, FHG, UP, UMU, UOX, ZA, IO10
Contributing workpackages:	WP2, WP3, WP4, WP5, WP6
Contractual date of delivery:	31 October 2014

Abstract
This deliverable contains the initial implementation the MICO platform and integration of the enabling technology components from WP2-5. The current version is the first of three iterations. It provides an implementation of the initial architecture proposed in D6.1.1, as well as the early evolution of some aspects within the project.

Keyword List
implementation

Platform: Initial Version

Sebastian Schaffert, Sergio Fernández

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

6 November 2014

Abstract
This deliverable contains the initial implementation the MICO platform and integration of the enabling technology components from WP2-5. The current version is the first of three iterations. It provides an implementation of the initial architecture proposed in D6.1.1, as well as the early evolution of some aspects within the project.

Keyword List
implementation

Contents

List of Figures

List of Listings

1 Introduction

This deliverable contains the initial implementation the MICO platform and integration of the enabling technology components from WP2-5. The current version is the first of three iterations. It provides an implementation of the initial architecture proposed in D6.1.1 [SF14], as well as the early evolution of some aspects within the project.

This document is only the basic description pointing to the actual implementation, as well as some other complementary technical documents.

1.1 Audience

The audience this document target is mainly technical people interested in working with the MICO platform. At this stage of the project developers in the consortium. The foundations of many aspects are not described to make this concise and useful document for that target group.

1.2 Source Code

Deliverable D6.2.1 is a software deliverable, so the main part is not this report but the actual source code and implementation. Also, a lot of the documentation is available online to make access easier for developers.

See also:

- C++/Java API Documentation at `http://mico-project.bitbucket.org/api/1.0/`
- Source Code at `https://bitbucket.org/mico-project/platform`

2 Architecture evolution (from D6.1.1)

The MICO Architecture (D6.1.1) provided a first conceptual overview over the different modules, services and technologies that together build the MICO platform. During the development of the first platform prototype, some of the decisions made in D6.1.1 had to be adapted to fit the prototype environment and the emerging requirements of the use cases. A revised version of the architecture deliverable will be published as D6.1.2. The following list gives a preliminary summary of the changes:

RabbitMQ Messaging. Deliverable D6.1.1 originally suggested to use Apache Zookeeper for event messaging and configuration storage. During the development, we decided to switch instead to RabbitMQ[1], as it provides more high-level communication infrastructure, is better suited for the main purpose of communication, and is generally more lightweight and easier to administer than Zookeeper. The conceptual architecture remains the same, and we might change the messaging backend in later iterations of the prototype if needed.

URL-based File Access. Deliverable D6.1.1 suggested using Hadoop HDFS as distributed file system for accessing content items, particularly large binary data like videos. Since the requirements on content storage are not yet completely clear and need to take into account the storage systems used by the use case partners, we instead implemented a more simple approach in the first prototype where the different content parts of a content item are retrieved using URL-based methods (HTTP or FTP). In that way, we remain flexible for future changes to the storage system instead of binding ourselves to a particular technology. In the current prototype, the content items are stored on a simple FTP server [PR85].

Simplified Servive Orchestration. As the service orchestration has not yet been developed in WP2, we instead provided a much simpler service orchestration component as initial prototype for being able to experiment with the system. This orchestration component is not capable of building individual execution plans for each analysis task. Instead, it has a shared service dependency tree that it uses for controlling the current state of the analysis process. Analysis of a content item is finished when there are no more possible state transitions in the dependency graph (see Section 4).

[1] http://www.rabbitmq.com/

3 MICO Client and Extractor API

See full C++/Java API Documentation at http://mico-project.bitbucket.org/api/1.0/

3.1 API Overview

The following sections give an overview over the MICO API and how to use it. The MICO API exists in implementations in both Java and C++ and can be used for implementing both, analysis services and client applications. The MICO API covers the following functionalities:

- **Persistence:** creating, accessing, and updating content items and content item metadata; creating, accessing and updating analysis results.
- **Event Messaging:** notifying other components about created or updated content items; waiting for notifications from other components
- **Service Skeletons:** abstract base classes that can be used for easily implementing custom services and running them as Unix services or Java components

As much as possible and reasonable, we tried to keep the C++ and Java APIs identical. In cases where this would break common language patterns, we chose a specific implementation instead. A full documentation of the C++ version of the API, automatically generated from the source code, is available in the online documentation[2].

In the prototype system, the MICO API uses a common username and password for accessing all platform services (i.e. persistence and messaging). This username and password need to be provided by all services that register with the platform (see e.g. 3.2.2 below).

3.2 Writing Analyzers

3.2.1 Analyzer Implementation

Analyzers for different content types are implemented by deriving from the base class `AnalysisService`. This base class requires five methods to be implemented:

- **getServiceID():** needs to return a URI uniquely identifying this service; this information is used internally as well as in provenance information for analysis results
- **getProvides():** needs to return a symbolic representation of the type of output produced by this analyzer; in the most simple case, this can be a MIME type, but more complex symbols are also feasible if needed (e.g. encoding the language of a text)
- **getRequires():** needs to return a symbolic representation of the type of input consumed by this analyzer; same conditions as for getProvides(); the service orchestration component uses this information to build its dependency graph
- **getQueueName():** can optionally return the RabbitMQ queue name to be used by this service; in case it returns null or the empty string, a random queue name is used

[2] http://mico-project.bitbucket.org/api/1.0/

- **call(function callback, ContentItem item, URI part):** contains the actual implementation of the analysis service; passes identifiers for the content item and part that is supposed to be analysed, as well as a callback function to use for notifying the service broker that it has created new results

The prototype source code contains example analysers in C++ and Java that show how AnalysisServices can easily be implemented. The following C++ excerpt shows an analyser that calls an OCR library implemented in C. Note that the code has been simplified for readability by omitting error and exception handling:

Listing 1: Example analysers in C++

```
class OCRAnalysisService : public AnalysisService {

private:
    tesseract::TessBaseAPI api;

public:
    OCRAnalysisService()
      : AnalysisService("http://www.mico-project.org/services/OCR-png",
                        "image/png", "text/plain", "ocr-queue-png") {
        api.Init(NULL, "eng");
    }

    ~OCRAnalysisService() {
        api.End();
    }

    void call(std::function<void(const ContentItem& ci, const URI& object)> callback,
              ContentItem& ci, URI& object) {
        Content* imgPart = ci.getContentPart(object);

        if(imgPart != NULL) {
            // read data from content part into in-memory buffer
            std::istream* in = imgPart->getInputStream();
            std::vector<char> buf = std::vector<char>(std::istreambuf_iterator<char>(*in),
                                                      std::istreambuf_iterator<char>());
            delete in;

            Pix* pic = pixReadMem((const unsigned char*)buf.data(),buf.size());

            // let tesseract do its magic
            api.SetImage(pic);
            char* plainText = api.GetUTF8Text();

            // write plain text to a new content part
            Content *txtPart = ci.createContentPart();
            txtPart->setType("text/plain");

            // set some metadata properties (provenance information etc)
            txtPart->setRelation(DC::creator, getServiceID());
            txtPart->setRelation(DC::provenance, getServiceID());
            txtPart->setProperty(DC::created, getTimestamp());
            txtPart->setRelation(DC::source, object.stringValue());

            std::ostream* out = txtPart->getOutputStream();
            *out << plainText;
            delete out;
            // notify broker that we created a new content part by calling the callback
            // function passed as argument
            callback(ci, txtPart->getURI());

            delete imgPart;
            delete txtPart;
            delete pic;
            delete [] plainText;
        }
    };
};
```

3.2.2 Analyzer Registration

Analyzers need to be registered with the MICO platform to be taken into account when planning the analysis of a content item. Registration is done via the EventManager instance that is part of the MICO API:

Listing 2: Analyser registration in C++

```
// create service instance
AnalysisService* service = new ...

// create event manager instance with given server name and credentials and
// register service instance
EventManager eventManager(server, user, password);
eventManager.registerService(service);

// wait for service to be stopped, e.g. by a SIGTERM
...

// unregister service instance before shutting down
eventManager.unregisterService(service);
delete service;
```

To avoid repetitively writing the same registration and signal handling code, the C++ version of the API contains a library (libmico_daemon) that developers can link against to create a Unix daemon to be started like other Unix services. In this case they just need to start the daemon, handing over the service instances.

3.3 Writing Client Applications

Client applications are applications that interact with the MICO platform in various ways. Client applications use the MICO platform mainly for two purposes:

- injecting new content items for analysis
- accessing the analysis results (content and metadata) when analysis is finished

Since analysis results and metadata are stored in a Apache Marmotta triple store, applications that only intend to access these results can simply use the standard facilities offered by Apache Marmotta (Linked Data, LDP, and SPARQL query language). Applications that require more interaction (e.g. for creating content items) need to use the MICO API instead. In this case, a client application is not very different from an analysis service. It will usually first create an EventManager instance, and then access the MICO platform through the persistence service managed by the EventManager, and finally notify the EventManager about any content items it changed:

Listing 3: Creating a new content part from C++

```
// create event manager instance
EventManager eventManager(server, mico_user, mico_pass);

// create new content item
ContentItem* item = eventManager.getPersistenceService().createContentItem();

// read a file from disk and add it as part
int fd = open(filename, O_RDONLY);
if(fd >= 0) {
    struct stat st;
    fstat(fd,&st);

    size_t len = st.st_size;
    char* buffer = (char*)mmap(NULL, len, PROT_READ, MAP_SHARED, fd, 0);

    // create new content part in content item and set metadata properties
    Content* c = item->createContentPart();
    c->setType(getMimeType(buffer,len));
    c->setProperty(DC::source, argv[i]);
    c->setRelation(DC::creator, URI("http://www.mico-project.org/tools/mico_inject"));
    c->setProperty(DC::created, getTimestamp());

    // write file contents to content part
    std::ostream* os = c->getOutputStream();
    os->write(buffer, len);

    // clean up
    delete os;
    delete c;
    munmap(buffer, len);
}

// notify event manager that new content item is available
eventManager.injectContentItem(*item);

// clean up
delete item;
```

4 System Implementation

The following sections describe the major server-side components of the MICO platform and how they interact with each other.

4.1 Overview

4.2 Triple Store: Apache Marmotta

4.2.1 Metadata Representation

Apache Marmotta is used for storing RDF metadata and analysis results for content items and their parts. Content items and content parts are uniquely identified using URIs. Metadata for content items is stored in separate named graphs according to the following patterns:

- `<URI>-metadata` identifies the named graph used for storing general content item metadata (e.g. provenance information, title, author, keywords, . . .)
- `<URI>-execution` identifies the named graph used for storing information about the execution of analysers, e.g. statistical information about the performance of analysers, the dependency graph. . . .
- `<URI>-result` identifies the named graph used for storing extracted analysis results; these are usually higher level descriptions suitable for querying by client applications and not raw analyser output

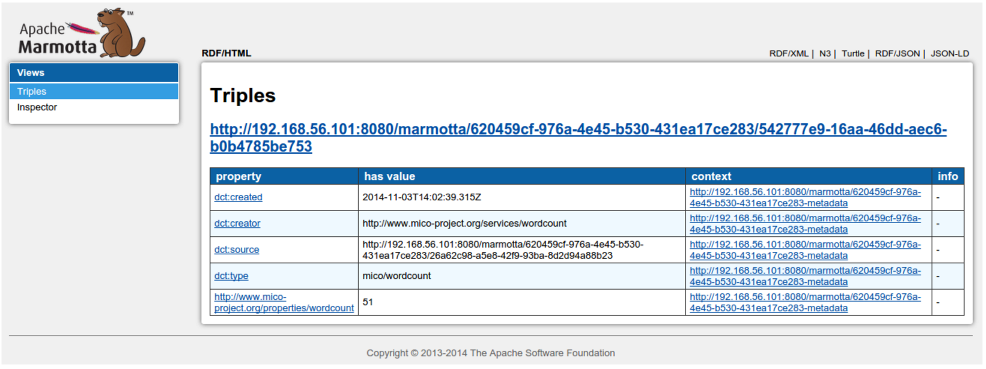
where `<URI>` represents the URI of the content item. All URIs are created using the hostname and port of the server as prefix, followed by a randomly generated UUID. For example, if the server is running on `192.168.56.101`, then an automatically generated URI could be `http://192.168.56.101:8080/marmotta/b4eede30-6359-11e4-9803-0800200c9a66`. Metadata for content parts is stored in a single separate named graph, using the URI of the content part as identifier. The concrete schemas used for representing metadata are in the first prototype still up to the analysis components. Once WP3 provides first versions of the schemas to use, the MICO API will provide convenient API calls for creating correct metadata.

4.2.2 Interaction

Content items, parts, and analysis results stored in Apache Marmotta can be accessed in one or more of the following ways:

- **SPARQL endpoint:** a SPARQL 1.1 endpoint [FWCT13] is provided by Marmotta and can be used for both, querying and updating metadata; the SPARQL endpoint is accessible in the browser through the Apache Marmotta admin interface, and as webservice conforming to the SPARQL protocol at `/marmotta/sparql/select` and `/marmotta/sparql/update`. Using SPARQL is the most powerful method of interacting with content item metadata, but might be too complex for simple cases. SPARQL is described at `http://www.w3.org/TR/sparql11-query/`.
- **Linked Data:** content item and content part metadata can also be accessed using the Linked Data principles by directly accessing the URI of the content item or part and using HTTP content negotiation for retrieving the data in the proper format. The Linked Data principles are described at `http://linkeddatabook.com/editions/1.0/`

Figure 1 Content Part Metadata shown in Apache Marmotta's Metadata Explorer



Triples

http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283/542777e9-16aa-46dd-aec6-b0b4785be753

property	has value	context	info
dct:created	2014-11-03T14:02:39.315Z	http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283-metadata	-
dct:creator	http://www.mico-project.org/services/wordcount	http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283-metadata	-
dct:source	http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283/26a62c98-a5e8-42f9-93ba-8d2d94a88b23	http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283-metadata	-
dct:type	mico/wordcount	http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283-metadata	-
http://www.mico-project.org/properties/wordcount	51	http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283-metadata	-

Copyright © 2013-2014 The Apache Software Foundation

- **Linked Data Platform:** the Linked Data Platform extends the Linked Data principles with containers and with methods for creating and updating RDF metadata [SAM14]. This technology is currently developed in Apache Marmotta.

4.2.3 RDF Schema

The data stored in Marmotta for representing content item metadata currently uses properties from several well-known RDF vocabularies:

- **LDP Containers**: the membership of content parts in their content item is modeled using the vocabulary defined for containers as part of the Linked Data Platform specification [SAM14]; most importantly, the `ldp:contains` property defines the relation between a content part and the content item it is part of

- **Dublin Core**: basic descriptions and provenance information is provided using the Dublin Core vocabulary [DCT12]; the properties currently used are `dc:title` (name of content part), `dc:created` (creation date of content part), `dc:creator` (creator of content part, either user, extractor, or tool), `dc:source` (URI of content part this part was derived from) and `dc:type` (symbolic type representation)

Figure 1 shows the representation of a content part (result of an analysis proces) in the Apache Marmotta Metadata Explorer. Note the URI schema used for generating the content part URI and the contexts, as well as the Dublin Core properties used to describe provenance information. Beyond these two basic vocabularies, implementations of extractor services are generally free to choose any schema that is necessary. Extractor services should, however, strive to use the vocabularies defined by WP3[3]. The MICO API will provide specific support for these vocabularies once they have been completed.

4.2.4 SPARQL Query Extensions

As preparation for implementing the SPARQL-MM [KSS+14, PBT+] query language in WP4, the SPARQL support in Apache Marmotta[4] has been significantly improved. All SPARQL queries are now

[3] http://www.mico-project.eu/ns/platform/1.0/schema#
[4] http://marmotta.apache.org/kiwi/sparql.html

directly translated into SQL queries in the underlying database following the relational schema used internally by Apache Marmotta. This gives the following benefits:

- existing geo and multimedia extensions of the underlying database can be used from SPARQL (e.g. PostGIS for working with geometric shapes)
- filter queries over large amounts of result data (e.g. region information about each key frame in a video) can be efficiently evaluated
- aggregation queries needed for recommendation in WP5 can be efficiently evaluated

The relational schema of Apache Marmotta consists of two main tables, `NODES` and `TRIPLES`. The `NODES` table stores an entry for each RDF node (URI, blank node, or literal) used in the triple store. In case the node is typed (e.g. a number or date), the typed value is also stored in a separate column. The `TRIPLES` table contains quadruples referring to the nodes table for subject, predicate, object, and context.

Table 1 gives an overview over how SPARQL constructs are mapped into SQL. This information might be useful to estimate evaluation performance or debugging. Note that the evaluation of certain SPARQL constructs can lead to considerable evaluation time (e.g. unnecessary use of DISTINCT, ORDER BY, GROUP BY or subqueries). Therefore, care should be taken when writing complex queries.

SPARQL Extensions are generally implemented as new SPARQL functions (recommended) or so-called virtual predicates (not recommended).

- a **SPARQL function** needs to be mapped into a corresponding SQL expression by deriving from the interface `NativeFunction` provided by Apache Marmotta. The most important method to implement is the translation to SQL; in many cases, this translation has to be done separately for different SQL dialects.
- a **virtual predicate** needs to be handled separately by extending the SQL mapping functionality; as this is both confusing for the end user and can easily break the mapping implementation, this is not recommended.

4.3 Content Store: FTP

Binary content of content parts is currently stored using a simple FTP server. This decision has been taken because it is not yet clear in which way use cases will be able to provide access to large amounts of contents like videos. FTP offers an URL-based file access that can easily be changed to a more sophisticated storage system at a later stage (see also Section 2). The FTP implementation used by the prototype system is proftpd[5], but any other FTP server can be used.

The binary content of a content part is stored using FTP URLs conforming to the following pattern:

```
ftp://<user>@<password>:<hostname>/<content_item-uuid>/<content_part-uuid>.bin
```

For example, the content part with UUID `b4eede30-6359-11e4-9803-0800200c9a66`, part of content item with UUID `5eb9d140-635a-11e4-9803-0800200c9a66` on server `192.168.56.101` and username/password mico/mico will use the following URI for accessing and writing binary content:

```
ftp://mico@mico:192.168.56.101/5eb9d140-635a-11e4-9803-0800200c9a66/↩
                                              b4eede30-6359-11e4-9803-0800200c9a66.bin
```

All binary content is accessed in a streaming fashion by the MICO API (both Java and C++).

[5] http://www.proftpd.org/

Table 1 Apache Marmotta SPARQL to SQL Mapping

SPARQL Construct	**SQL Construct**
Node Value Use (in SELECT part)	Node ID projection and subsequent lookup in nodes table
Node Value Use (outside SELECT part)	JOIN of triples table with nodes table
Aggregation, GROUP BY, HAVING	SQL Aggregation, GROUP BY and HAVING
LIMIT, OFFSET	LIMIT, OFFSET
ORDER BY	ORDER BY
DISTINCT	DISTINCT
Triple Patterns	FROM and JOIN over triples table
OPTIONAL	LEFT JOIN over triples table
FILTER	WHERE conditions or LEFT JOIN conditions (in case of OPTIONAL)
Subquery	Subquery
NOT EXISTS	NOT EXISTS and subquery
UNION	two subqueries with balanced projection to same columns and a UNION
MINUS	two subqueries with NOT EXISTS
BIND	internally resolved with aliases when building the query
type casting (e.g. xsd:integer)	type coercion (selecting appropriate column from nodes table for variables, casting for constants)
SPARQL function	SQL function (for all built-in SPARLQL functions; extensions need to be mapped separately)
SPARQL comparison	SQL comparison with appropriate type coercion
SPARQL arithmetics	SQL arithmetics with appropriate type coercion

4.4 Messaging: RabbitMQ and Protocol Buffers

4.4.1 Queues and Exchanges

RabbitMQ[6] is used as a platform and language independent system for message and event exchange between components in the MICO platform. The basic building blocks of RabbitMQ are *queues* and *exchanges*:

- a *queue* is a publisher-consumer style messaging infrastructure where publishers write messages into the queue and consumers read (and consume) messages from the queue; optionally, a publisher can also register a callback queue to be notified when the consumer has finished processing the message (one-to-one messaging)
- an *exchange* is a publish-subscribe style messaging infrastructure where consumers can register for certain messages, and producers then send their message to all registered consumers (one-to-many messaging)

The MICO platform uses both styles of messaging, albeit for different purposes:

- *service queues:* each analyser service sets up its own queue when registering with the MICO platform; this queue is used to notify the analyser service that a new content part has to be analysed; when several implementations of the same analyser are running (e.g. in load balancing), messages are consumed first-come first-serve, i.e. the first available analyser will carry out the task
- *service registration and discovery exchanges:* the broker sets up a registry exchange and a discovery exchange; whenever a new analyser service registers with the platform, it notifies all MICO brokers through the registry exchange; whenever a new MICO broker is started, it sends out a service discovery request on the discovery exchange
- *processing callback queues:* the broker, when processing a new content item, notifies analysers through their queues according to its execution plan; for each call to a service, the broker sets up a temporary callback queue on which it is notified when the processing is finished
- *input/output queues:* these queues are used to notify the broker that a new content item is available (input), and by the broker to notify that processing a content item is finished (output)

An example of the list of queues of a running MICO platform is given in Figure 2. Queues starting with ”amq” are auto-generated temporary queues used as service callbacks. Note in particular the content input, content output, and service notification queues for the services ”ocr-jpeg”, ”ocr-png” and ”wordcount”.

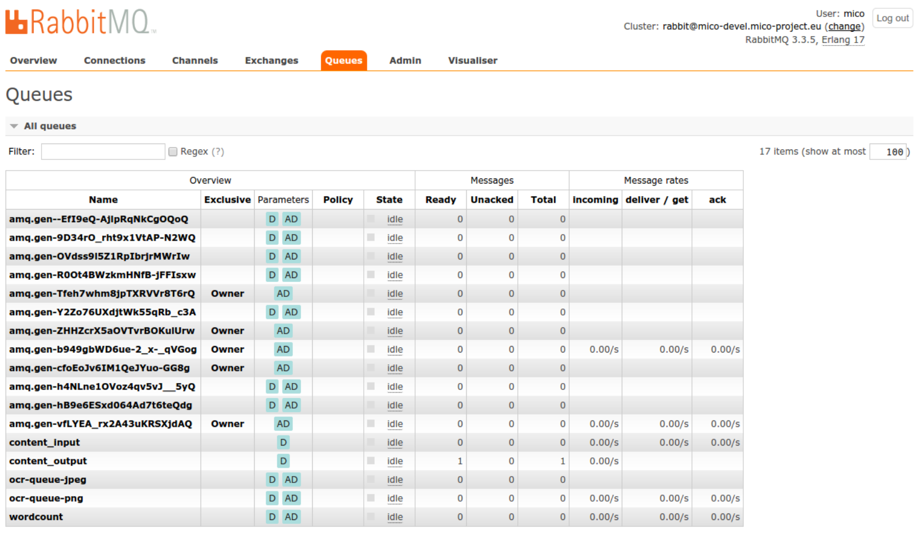
4.4.2 Messaging Protocol

All messages exchanged in the MICO platform use the Protocol Buffer (Protobuf)[7] binary format. Protobuf allows a language independent definition of message schemas that is then compiled into a highly efficient binary representation. The following three message types are currently used by the system:

- *registration event*: sent by a service to the service registration exchange when it starts up

[6] http://www.rabbitmq.com/
[7] https://code.google.com/p/protobuf/

Figure 2 RabbitMQ queue overview for the MICO platform.



RabbitMQ

User: mico Log out
Cluster: rabbit@mico-devel.mico-project.eu (change)
RabbitMQ 3.3.5, Erlang 17

Queues

All queues

Filter: Regex (?) 17 items (show at most 100)

Overview					Messages			Message rates		
Name	Exclusive	Parameters	Policy	State	Ready	Unacked	Total	incoming	deliver / get	ack
amq.gen--EfI9eQ-AjlpRqNkCgOQoQ		D AD		idle	0	0	0			
amq.gen-9D34rO_rht9x1VtAP-N2WQ		D AD		idle	0	0	0			
amq.gen-OVdss9l5Z1RpIbrjrMWrIw		D AD		idle	0	0	0			
amq.gen-R0Ot4BWzkmHNfB-jFFIsxw		D AD		idle	0	0	0			
amq.gen-Tfeh7whm8jpTXRVVr8T6rQ	Owner	AD		idle	0	0	0			
amq.gen-Y2Zo76UXdjtWk55qRb_c3A		D AD		idle	0	0	0			
amq.gen-ZHHZcrX5aOVTvrBOKulUrw	Owner	AD		idle	0	0	0			
amq.gen-b949gbWD6ue-2_x-_qVGog	Owner	AD		idle	0	0	0	0.00/s	0.00/s	0.00/s
amq.gen-cfoEoJv6IM1QeJYuo-GG8g	Owner	AD		idle	0	0	0			
amq.gen-h4NLne1OVoz4qv5vJ__5yQ		D AD		idle	0	0	0			
amq.gen-hB9e6ESxd064Ad7t6teQdg		D AD		idle	0	0	0			
amq.gen-vfLYEA_rx2A43uKRSXjdAQ	Owner	AD		idle	0	0	0	0.00/s	0.00/s	0.00/s
content_input		D		idle	0	0	0	0.00/s	0.00/s	0.00/s
content_output		D		idle	1	0	1	0.00/s		
ocr-queue-jpeg		D AD		idle	0	0	0			
ocr-queue-png		D AD		idle	0	0	0	0.00/s	0.00/s	0.00/s
wordcount		D AD		idle	0	0	0	0.00/s	0.00/s	0.00/s

```
message RegistrationEvent {
  required string serviceId          = 1; // service identifier of service
  required string queueName          = 2; // input queue name used by service
  required string provides           = 3; // type of output this service provides
  required string requires           = 4; // type of input this service provides
  optional Implementation language   = 5 [default = JAVA];      // or CPP for C++
  optional RegistrationType type     = 6 [default = REGISTER];  // or UNREGISTER
}
```

- *analyis event*: sent by the broker to the next service in the execution plan, and sent by an analysis service to the broker to signal that it has created a new content part as result; can be sent multiple times, e.g. when a service generates several content parts as result

```
message AnalysisEvent {
  required string serviceId      = 1; // identifier of the service generating the event
  required string contentItemUri = 2; // URI of the content item being processed
  optional string objectUri      = 3; // URI of the content part being processed
}
```

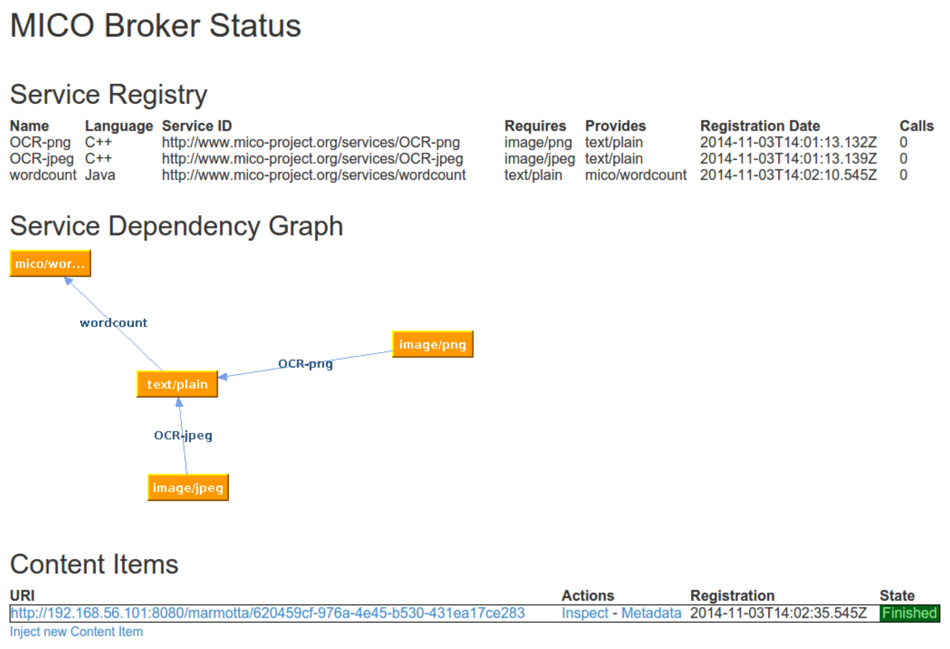
- *content item event*: notifies the broker that a new content item is available; also used for signaling that processing a content item has been finished:

```
message ContentEvent {
  required string contentItemUri = 1;
}
```

The complete definition of all messages can be found in the source code file `platform/api/shared/event/Event.proto`[8].

[8] https://bitbucket.org/mico-project/platform/src/1.0.0/api/shared/event/Event.proto

Figure 3 MICO Broker user interface with dependency graph and content item processing status.



MICO Broker Status

Service Registry

Name	Language	Service ID	Requires	Provides	Registration Date	Calls
OCR-png	C++	http://www.mico-project.org/services/OCR-png	image/png	text/plain	2014-11-03T14:01:13.132Z	0
OCR-jpeg	C++	http://www.mico-project.org/services/OCR-jpeg	image/jpeg	text/plain	2014-11-03T14:01:13.139Z	0
wordcount	Java	http://www.mico-project.org/services/wordcount	text/plain	mico/wordcount	2014-11-03T14:02:10.545Z	0

Service Dependency Graph

Content Items

URI	Actions	Registration	State
http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283	Inspect - Metadata	2014-11-03T14:02:35.545Z	Finished

Inject new Content Item

4.5 MICO Broker

The current prototypical implementation of the MICO broker offers the following main functionalities:

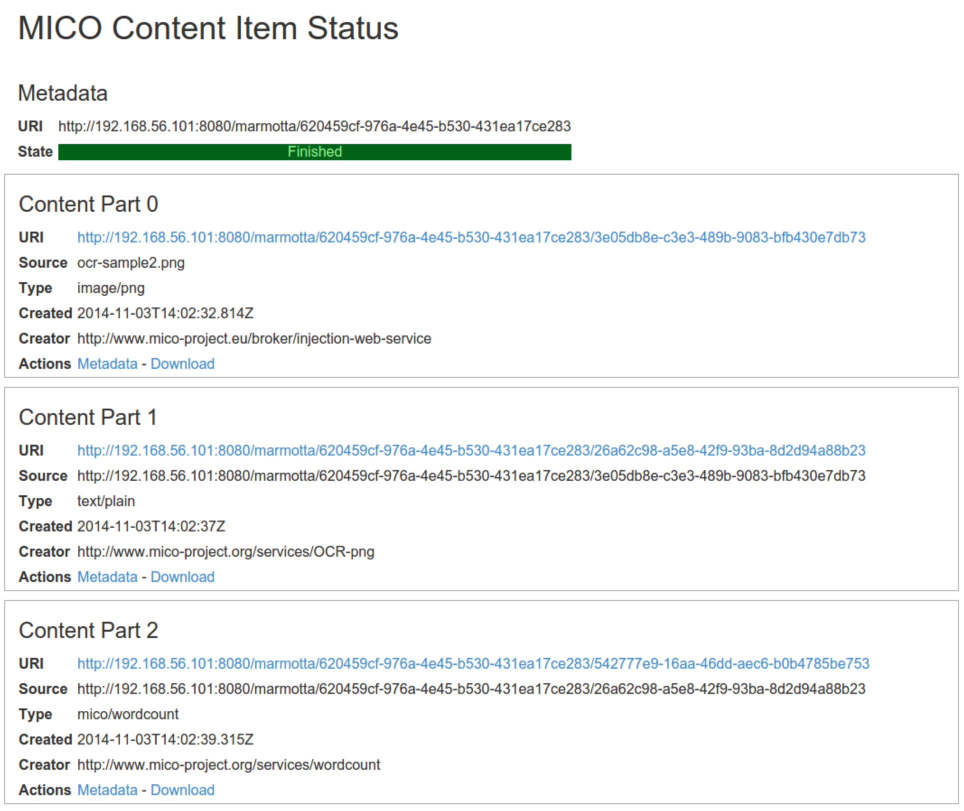
- *service registration*: whenever an analysis service connects or disconnects from the platform, the MICO broker is notified; at any point in time, it knows about the state and availability of analysis services; it also builds up a static dependency graph based on the requires/provides specifications of the services, which it uses as simple execution plan

- *service orchestration*: when a new content item has been created, the MICO broker is notified and starts executing the different analysis steps based on a simple execution plan; it does so by starting at an initial state in the dependency graph that matches with the type of one of the content parts of the created content item and then notifies analysers in turn along the edges of the dependency graph until no further transition is possible

- *debugging user interface*: the broker also provides a simple user interface for development purposes which allows developers to get some insight about the analysis process; it shows the dependency graph, the current processing state of content items, and allows inspecting the results of analysis; it also allows creating and uploading new content items

Figure 3 shows a screenshot of the main MICO broker user interface. Note that three services have been registered (OCR-png, OCR-jpeg and wordcount) and how the dependency graph is constructed. At the bottom of the screenshot you can also see that one content item has been processed.

Figure 4 MICO Broker user interface with content item inspection.



MICO Content Item Status

Metadata

URI http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283

State Finished

Content Part 0

URI http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283/3e05db8e-c3e3-489b-9083-bfb430e7db73

Source ocr-sample2.png

Type image/png

Created 2014-11-03T14:02:32.814Z

Creator http://www.mico-project.eu/broker/injection-web-service

Actions Metadata - Download

Content Part 1

URI http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283/26a62c98-a5e8-42f9-93ba-8d2d94a88b23

Source http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283/3e05db8e-c3e3-489b-9083-bfb430e7db73

Type text/plain

Created 2014-11-03T14:02:37Z

Creator http://www.mico-project.org/services/OCR-png

Actions Metadata - Download

Content Part 2

URI http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283/542777e9-16aa-46dd-aec6-b0b4785be753

Source http://192.168.56.101:8080/marmotta/620459cf-976a-4e45-b530-431ea17ce283/26a62c98-a5e8-42f9-93ba-8d2d94a88b23

Type mico/wordcount

Created 2014-11-03T14:02:39.315Z

Creator http://www.mico-project.org/services/wordcount

Actions Metadata - Download

Figure 4 shows the inspection page for a content item. Developers get detailed insights about the different content parts, can download the binary content and display the generated metadata.

5 System Usage

instructions how to setup the system, from README; tools

5.1 Building from the source

Build the MICO Platform from the source code is only strictly required for those extending or patching it. This section is part of this deliverable to document the process. Therefore, for proceeding with a regular installation, please continue at Section 5.3 for further details.

The source code of the platform can be obtained from the project account at bitbucket[9]:

Listing 4: Cloning the git repository with the platform source code

```
$ git clone https://bitbucket.org/mico-project/platform.git mico-platform
```

The API is built using the standard tools for the respective environment:

5.1.1 Building (Java)

The complete platform is built using Maven[10], including the server components (Marmotta and Broker) as well as the Java bindings. To build and install the current development version, the following command needs to be executed:

Listing 5: Using Maven to build the platform

```
$ mvn clean install
```

This command will compile all Java source code, run existing unit tests, build JAR artifacts, and install them in the local Maven repository. Binary Maven artifacts are periodically published to our development repositories[11], so they can be directly used from dependant projects (e.g., analyzers):

Listing 6: Maven repositories configuration

```
<repositories>
    ...
    <repository>
        <id>mico.public</id>
        <name>MICO Maven Proxy</name>
        <url>http://mvn.mico-project.eu/content/groups/public/</url>
        <releases><enabled>true</enabled></releases>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>
</repositories>
```

5.1.2 Building (C++)

The C++ bindings of the platform are built using the CMake[12], where the following steps need to be done (each step corresponds with the same number of line at Listing 7:

[9] http://code.mico-project.eu/platform
[10] http://maven.apache.org
[11] http://mvn.mico-project.eu
[12] http://www.cmake.org

1. Configure a build directory and create the Makefiles necessary to build the platform.
2. In case configuration succeeds (i.e. all dependencies are found), the C++ libraries can be built and automatically tested using GNU Make.
3. Optionally create a complete API documentation of the MICO Platform API at the `api/c++/doc` directory.
4. Finally install the C++ libraries and headers to the predefined prefix.

Note that the C++ version has a number of library dependencies that are checked when calling CMake. Most notably, these are cmake itself, doxygen, protobuf, boost, libexpat, libcurl, libtesseract, libmagic, libdaemon.

Listing 7: Build with CMake

```
$ cmake api/c++
$ make
$ make install
```

5.2 Packaging

The distributed service-oriented architecture proposed [SF14] is implemented as a UNIX-based system, where components are independent POSIX processes [BW01]. A revised version of the architecture deliverable will be published as D6.1.2 providing much more detail.

More precisely the MICO Platform is implemented as a Debian system [Mur94]. Each server component of the platform is provided as a package for that system (`.deb`), taking advantage of the advanced packaging system Debian offers for managing versions, dependencies, etc.

Listing 8: Package Debian components with Maven

```
$ mvn package -Pdebian
```

Currently there are two complentary method to package the components: The Java components (Marmotta and Broker) are packaged, as Listing 8 show, with a special Maven profile using jdeb[13], a cross-platform Maven plugins. While all the remaining components make use of regular Debian Development tools[14]. Packages are periodically published to the project Debian repository[15].

5.3 Installation

A complete binary installation for development can be setup using custom-built packages that we offer in the MICO Debian repository. For those who need to setup a development server, this is the easiest way to get up and running.

This setup is also provided as a preinstalled VirtualBox[16] image for the members of the consortium. We are currently working on providing a more easy-to-use vagrant[17] version of this image.

[13] http://github.com/tcurdt/jdeb
[14] http://www.debian.org/doc/manuals/maint-guide/build.en.html
[15] http://apt.mico-project.eu
[16] https://www.virtualbox.org
[17] https://www.vagrantup.com/

To install these packages, first setup a basic installation of Debian Jessie[18] (currently in *testing*). Regarding the requirements of the MICO Platform, a plain installation is sufficient, i.e. no special package preselection is needed.

Listing 9: Repository configuration

```
deb http://apt.mico-project.eu/ mico main non-free contrib
```

The first step is to add the MICO Repository (Listing 9 to the `/etc/apt/sources.list` file. All packages are signed with a with a gpg-key (`Key-ID: AD261C57`). To avoid warnings by `apt-get` either install the `mico-apt-key` package or manually fetch the key [19] as Listing 10 shows.

Listing 10: GPG key installation

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

The you can already proceed to install the MICO platform, fetching the most recent package list and install the package `mico-platform` executing the following commands:

Listing 11: Platform installation

```
apt-get update
apt-get install mico-platform
```

The installation will interactively ask you a few questions regarding a MICO user to be created and the hostname to use for accessing the system. Please take your time to carefully configure these values. To see how to interact with the MICO Platform please refer to Section 5.4.

5.4 Usage

The MICO platform installation comes with a single entry-point at `http://<host>/` for accessing the Web interfaces of those services that provide it.

By default installation comes with some sample services, implemented in C++, that are used for demonstrating the platform functionality. For instance, Listing 12 shows a service capable of transforming text contained in JPEG and PNG images into plain text using the Tesseract OCR library[20]. Any content can be injected to the platform, which will orchestrate a extraction process among all registered extractors, whic a command like the one shown by Listing 13; the call will inject a single content item, with a content part for each file given as argument.

Listing 12: Example how to extract text with Tesseract OCR

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

Listing 13: Example how to inject content to the MICO Platform

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

It should be taken into account that for the moment the integrated MICO Platform is a prototypical state, where many components are still unstable or missing for some cases.

[18] https://www.debian.org/releases/jessie/
[19] http://apt.mico-project.eu/apt-repo.key
[20] http://tesseract-ocr.googlecode.com

6 Perspectives

This version of the MICO platform is a first prototype mainly intended to serve as a platform for the development of extractors and starting to work on the implementation of the use cases. While the API should remain mostly stable, several components are only proof-of-concept and might be replaced in future versions:

6.1 MICO Broker: Service Orchestration

The current version of the MICO broker implements a simplistic service orchestration that uses a static execution plan for all analysis tasks and will not work well with cyclic dependencies, missing transitions, unavailable services, etc. Future versions will implement a much more sophisticated service orchestration as described in D6.1.1 [SF14] and developed in work package WP2. Future versions need to take into account:

- individual execution plans for each content item, based on available input data and required output type, as well as available analysis services and their quality and costs
- cyclic dependencies, alternative execution paths, and missing transitions
- user interaction in the analysis process, e.g. to confirm certain results and thus improve the quality and reliability of the overall annotations

6.2 MICO API: Vocabulary API

The current implementation of the MICO API had to be done without knowledge of the RDF schemas that would be used for representing annotations and analysis results (work package WP3). For this reason, it currently only offers generic low-level metadata support where the schema to be used is completely up to the developer. While metadata will still be represented in RDF, this makes it more difficult to integrate and combine the results of different analysers and let one analyser build on another. The next version of the MICO API will therefore provide specific high-level support for the schemas and vocabularies developed in WP3 so developers need not be concerned about working with RDF directly.

6.3 MICO Persistence: More Elaborate Storage

Using FTP to store the binary content of content parts is only a least common denominator and therefore only a temporary solution. Once it is known how the use cases will be able to provide their binary content, we will implement a more powerful storage system, e.g. based on technology previously developed by MICO partner Fraunhofer for this purpose. The programming API will remain unchanged, so that existing extractors need not be adapted to a different storage system.

References

[BW01] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[DCT12] DCMI Metadata Terms. Technical report, Dublin Core Metadata Initiative, 2012.

[FWCT13] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. Sparql 1.1 protocol. Recommendation, W3C, March 2013.

[KSS$^+$14] Thomas Kurz, Sebastian Schaffert, Kai Schlegel, Florian Stegmaier, and Harald Kosch. Sparql-mm-extending sparql to media fragments. In *The Semantic Web: ESWC 2014 Satellite Events*, 2014.

[Mur94] Ian Murdock. Overview of the debian gnu/linux system. *Linux Journal*, 1994(6es):15, 1994.

[PBT$^+$] Valentina Presutti, Eva Blomqvist, Raphael Troncy, Harald Sack, Ioannis Papadakis, and Anna Tordai. *The Semantic Web: ESWC 2014 Satellite Events*. Springer.

[PR85] JB Postel and J Reynolds. Rfc959: File transfer protocol. *Network Information Center*, 1985.

[SAM14] Steve Speicher, John Arwe, and Ashok Malhotra. Linked data platform 1.0. Last call working draft, W3C, September 2014.

[SF14] Sebastian Schaffert and Sergio Fernández. D6.1.1 System Architecture and Development Guidelines. Deliverable, MICO, 2014.