A Tutorial on the MILOS Multimedia Content Management System

Giuseppe Amato, Paolo Bolettieri, Franca Debole, Fabrizio Falchi, Claudio Gennaro, Fausto Rabitti, Pasquale Savino

ISTI-CNR, Pisa, Italy
{firstname.secondname}@isti.cnr.it

Abstract
In this paper we present the MILOS[1] Multimedia Content Management System. MILOS supports the storage and content based retrieval of any multimedia documents whose descriptions are provided by using arbitrary metadata models represented in XML. It provides developers of digital library applications with functionalities for dealing with heterogeneous digital documents, heterogeneous metadata, and metadata schema mapping.
This paper shows how to configure and use all MILOS components.

1. Introduction

The Digital Library (DL) technology emerged during mid 90ties as a combination of different technological results, mainly in the area of Database Management and Information Retrieval, and as an application of these technologies to the management of libraries.
Regrettably, nowadays several Digital Library Applications (DLA) are monolithic software modules where the application itself, the content management software, and the digital library are merged together. Many systems were built having in mind a specific application and, in many cases, a specific document collection, thus resulting in an ad-hoc solution: all components of the DLA -- the data repository, the metadata manager, the search and retrieval components, etc. -- are specific to a given application and cannot be easily used in other environments. In these cases there is not possibility of service personalization/specialization, and adaptation to new user requirements.
We believe that this situation is mainly due to the lack of basic building components, tailored to DL application design, which are standard and general purpose. For instance, database applications generally rely on Database Management Systems (DBMS) which simplify substantially the complexity of designing and implementing them. It is possible to do the same in the DL field: it is possible to build a general purpose Multimedia Content Management System (MCMS) which offers functionalities specialized for DL applications (see Figure 1). Different DL applications, requiring the management of different types of document format, described by using arbitrary metadata description models, searchable in many different modes, can be built on top of such an MCMS. This MCMS should be able to manage not only formatted data, like in databases, but also textual data, using Information Retrieval technology, semi-structured data, mixed-media data, like structured presentations, and multimedia data, like images and audio/video.

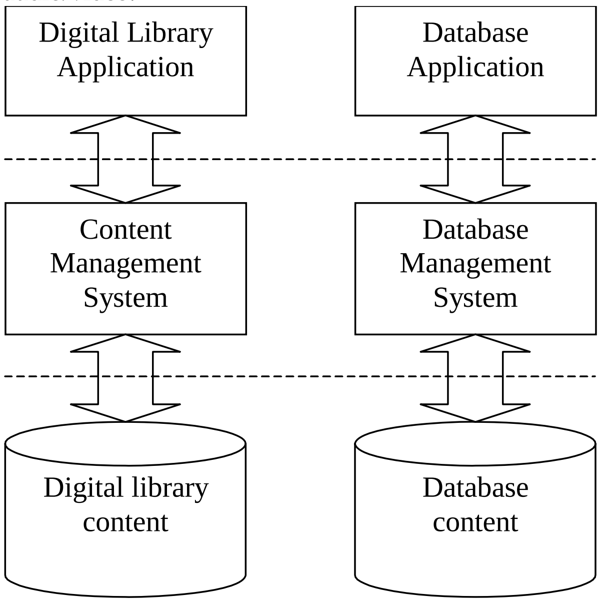
Digital Library Application
Database Application
Content Management System
Database Management System
Digital library content
Database content

Figure 1 Relationships among layers of DLs and databases

2. Characteristics of a MCMS for digital library applications

Digital library applications are document intensive applications where heterogeneous documents and their metadata have to be managed efficiently and effectively.
More in detail, a general purpose MCMS for digital library applications has to satisfy the following three basic characteristics:

1. capability of managing different documents embodied in different media and stored with different strategies;
2. capability of describing documents by way of arbitrary, and possibly heterogeneous, metadata;

1 This work was partially supported by the DELOS NoE and the MultiMatch STREP project both funded by the European Commission under FP6 (Sixth Framework Programme).

3. capability of providing DL applications with custom/personalised views on the metadata schema actually handled.

Point 1) implies that no assumption should be taken on the types of media and encoding used to represent documents, and especially on the specific strategy used to store them. This allows applications to be unaware of the technical details related to multimedia document management. For instance, textual documents can be stored in the file system and served to the users using a normal web server. However, video documents might need to be maintained in a video server that uses various storage devices, as for example digital tapes stored in silos, optical disks, and/or temporary storage space on arrays of hard disks [10]. In addition, video documents might be served exploiting specific real-time continuous media streaming strategies to avoid hiccups during playback. The DL application should be designed independently of these issues, which should be managed transparently by the MCMS. For instance, changes on the storage strategies should be possible without changing the DL application software.

Point 2) states that a content management system should be able to deal with arbitrary metadata. This is required by the fact that different DL applications, according to their specific requirements, might need to use different metadata. Consider that existing archiving organizations have already their own metadata schemas, and hardly want to modify them to be compatible with a specific system. Therefore, a DL management system should be able to support any metadata schema without requiring metadata translation or restrictions on the functionality offered. There are also cases where the same application needs to deal with different metadata at the same time. These different metadata might be needed because the documents have redundant descriptions in terms of different metadata, or because the DL application is dealing with document collection described with heterogeneous metadata. The last case might occur, for instance, in case of integration/merging of archives managed by different organization.

Point 3) makes it possible that the metadata schema seen by the DL application is different from the metadata schemas actually stored in the repository of the content management system. Suppose that an application was built to deal just with a specific metadata schema. The MCMS should be able to serve requests of such an application even if metadata stored in the repository comply to different schemas. Metadata schema independence can be obtained by exploiting techniques of schema mapping. This feature is especially useful in case of heterogeneous metadata available at the same time in the repository: the DL application will refer to just one metadata schema, relying on the multiple schema mapping performed on the fly by the MCMS. In addition, this feature allows different DL applications, which require different metadata schemas, to share the same MCMS transparently.

3. MILOS

MILOS is a multimedia content management system with special functionalities for multimedia document intensive applications, which satisfies the requirements discussed in Section 2. The MILOS MCMS has been developed by using the Web Service technology and, as depicted in Figure 2, is composed of three main components: the XML Search Engine (XMLSE) component, the Multi Media Server (MMS) component, and the Multi Media Digital Library service (MMDLS) component. All these components are implemented as Web Services and interact by using SOAP.

The XMLSE manages the metadata of the DL. It relies on our technology for native XML databases [8], and guarantees the possibility of dealing with arbitrary and heterogeneous metadata.

The MMS manages the multimedia documents used by the DL applications, and guarantees the possibility of transparently dealing with arbitrary document formats and access strategies.

The MMDLS implements the service logic of the repository providing developers of DL applications with a uniform and integrated way of accessing MMS and XMLSE. In addition, it supports the automatic and transparent mapping between different metadata schemas.

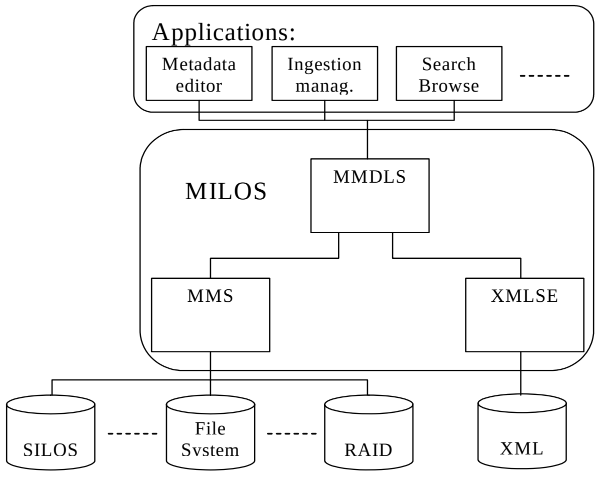

Figure 2 General Architecture of MILOS

4. MILOS quick start:

MILOS distribution runs on Windows 2000, Windows XP and Linux.

It already contains a web server (Tomcat 5.0 for JWSDP), however it requires a Java Developer Kit (JDK 1.5 or above) to run.

Installing MILOS just requires

a) copying the main MILOS directory in your preferred location of your hard disk;
b) installing the Java Developer Kit (JDK 1.5 or above)
c) set the JAVA_HOME environment variable to your JDK home (For instance: "JAVA_HOME=C:\Programs\Java\jdk1.5.0_05")

Examples of entries in the `mms.properties file`:
```
test.image_jpeg = jpg,c:\\my-application\\media-objects,http://my.web.server/images/
test.video_mov = mov,c:\\my-application\\video-server-folder,rtsp://my.video.server/videos/
```

Examples of entries in the `mimeFile.properties`
```
video/mpeg = video_mpeg,mpg
application/vnd.ms-powerpoint = application_powerpoint,ppt
video/x-ms-wmv = video_wmv,wmv
```

Table 1: Examples of configuration of MMS

Once you have installed MILOS, you enter the main MILOS directory. From this directory you can execute the following batch files[2]:

- "run.bat" to run the MILOS system
- "stop.bat" to shutdown the MILOS system

From the `milos/etc` subdirectory you can execute the following service batch files:

- "DBBackup.bat"
  to create a backup of the XML database into the `milos/milos/.milos/dbBackup` directory (First you have to shutdown MILOS). Note: It doesn't create a backup of the Multimedia documents.
- "DBRestore.bat"
  to restore the XML database from a previous database backup (First you have to shutdown MILOS and destroy the current database by using "DBDestroy_Metadata.bat"). From the command line you must specify the database backup file name; you can find the database backup files into the `milos-milos/milos/.milos/dbBackup` directory
- "DBDestroy_Metadata.bat"
  to destroy (delete) the content of the XML database (First you have to shutdown MILOS). Note: it doesn't destroy the multimedia files.
- "DBDestroy_All.bat"
  to destroy (delete) all data: XML and multimedia files. (First you have to shutdown MILOS)

If you run MILOS for the first time you might want to configure some indexes for the XML search engine (see Section 6 on XMLSE), configure the mapping rules for the MMDSL (see Section 7 on MMDLS), and configure the mapping rules for the URN (see Section 5 on MMS). Then you are ready to write your applications or run your existing MILOS application.
In the `MILOS-sdk` folder of the distribution you can find some example applications that you can compile and run. The MILOS distribution is already configured to support these applications (XMLSE, MMS and MMDLS are already correctly configured for them).

[2] On Linux the batch files are named <name>.sh while on windows they are named <name>.bat

5. MultiMediaServer: MMS

Different DL applications may have different storage and access needs. For example, very small DLs might store documents on standard hard disks, while more mission critical applications might need to store documents on arrays of disks, possibly duplicating and distributing content on several sites. Digital libraries dealing with huge archives of video documents, might need to store them on digital tapes maintained in silos, and to have arrays of disks used as temporary storage for frequently used documents. In addition, we must consider that a DL may scale over time, when the number of documents grows over a certain limit or faster access is needed.
DL applications might also use different delivery strategies. For example, a small DL might serve documents using a normal web server, while heavily accessed DLs might need to use replication and load balancing strategies to guarantee high performance access to content. A video DL might use high performance video servers to stream videos in real time to users [10].
The MMS allows the programmers of the DL applications to be unaware of all these issues. The key idea is that the DL application should deal with documents in a uniform way, independently of the specific strategy used to manage them. Thus, the MMS identifies all documents with an URN and maintains a mapping table to associate URNs with actual storage locations. Applications use the URN to get or store documents from the MMS, which behaves as a gateway to the actual repository that stores the document. The system administrator can define rules that make use of MIME types, to specify how the MMS has to store a document of a specific type. For example, the rule may specify that an MPEG-2[4] video has to be stored in a tape of a silos, while an image will be stored in an array of disks.
A special care is taken to deal with the actual access protocols offered to retrieve the documents. An application will refer a specific document always using its URN. However, the retrieval of the document should be done using an access protocol compatible with the storage and delivery strategy associated with the document. For instance, when the document is stored in a web server it will be retrieved with an HTTP request. On the other hand, suppose that a video document is served through a commercial video server

Packages to be imported to use MMS:

```
import java.rmi.RemoteException;                    //RMI Exceptions
import javax.xml.rpc.Stub;                          //Stubs for web services
import it.cnr.isti.milos.dataLogic.MMS.MMS_IF;      //Interface of the MMS web service
import it.cnr.isti.milos.dataLogic.MMS.MMS_Impl;    //Tool for creating a stub for MMS
import javax.activation.FileDataSource;             //Datasources for files
import javax.activation.DataHandler;                //To wrap media objects when inserting
```

Actions to be executed to obtain an instance of MMS (a statefull session aware stub for MMS):

```
MMS_IF mms = (MMS_IF) new MMS_Impl().getMMS_IFPort();
((Stub)mms)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, “http://localhost:8080/milos-MMS/MMS”);
((Stub)mms)._setProperty(Stub.SESSION_MAINTAIN_PROPERTY, new Boolean(true));
```

Example of use of an instance of MMS:

```
String guid="0a783ee65ab33d679aab0cb5e322e64d";     //a GUID to be used to create an URN
String urn="urn:milos:test:image_jpeg:"+guid;       //an URN
//create a Data source for the image to be inserted:
FileDataSource fds=new FileDataSource("E:/MILOS-sdk/test-data-sets/images/"+guid+".jpg");
//create a data hander for the object to be inserted:
DataHandler dh=new DataHandler(fds);
//insert, get, update, and delete a digital object in MMS
try{
       mms.insert(dh,urn);                          //insert a digital object
       String url=mms.getUrl(urn));                 //get the URL from the URN
       mms.update(dh_1,urn);                        //update a digital object
       mms.delete(urn);                             //delete a media object
}catch(RemoteException e){e.printStackTrace();}
```

Table 2: How to use MMS

such as the Helix Universal Server [3]; in this case the real time streaming of the video will be obtained using RTSP [5]. When an application requires to retrieve a document, the MMS will translate the given URN into a specific handle (for instance an RTSP URL) that the application will use to access the document.

5.1. Configuration of MMS

MMS performs basically two tasks:

1) Store media objects
2) Given the URN of a media objects returns the URL to access it.

The MMS configuration file `mms.properties`, which can be found in the `milos/milos/.milos` directory specifies the directory where objects should be stored and the rules used to generate the URL to access them.
MMS arranges media objects in collections of heterogeneous files. Different collections are stored in different directories. Different types of files are distinguished by different mime types. Different mime types of the same collection are stored in different directories of the same collection.
When an object is requested to MMS, it returns the URL that should be used to access it.

MMS is configured inserting in `mms.properties` rules of the following form

```
<collection>.<mimetypeID>                            =
<File_extension>,<Media_Objects_base_dir>,<Med
ia_Objects_base_URL>
```

where:

- the `<collection>` is an identifier for a collection of (heterogeneous) media objects
- `<mimetypeID>` is the MILOS identifier for a mime type (see mimeFile.properties).
- the `<file_extension>` is the file extension for the mime type (e.g. jpg, avi, etc...)
- the `<Media_Objects_base_dir>` is the path of the directory where these media objects are stored by MMS
- the `<Media_Objects_base_URL>` is the base URL that will be used to access these objects

When a media objects is asked to be inserted in MMS using the insert method, which we will discuss later, an URN for the object should be provided as well.
The URN must have the form:

```
urn:<application>:<collection>:<some_optional_
elements>:<mimetypeID>:<GUID>.
```

Accordingly MMS stores the media object in the subdirectory `<collection>/<some_optional_elements>/<mimetypeID>` of the base directory associated to the rule corresponding to the `<collection>.<mimetypeID>` entry. The file is given name `<mimetypeID><GUID>.<File_extension>`.

To request an object to MMS, with methods that we will discuss later, an URN should be specified. MMS gives back an URL built as follows:

```
<Media_Objects_base_URL>/<collection>/<some_op
tional_elements>/<mimetypeID>/<mimetypeID><GUI
D>.<File_extension>
```

N.B.: if `<Media_Objects_base_dir>` and/or `<Media_Objects_base_URL>` are set "auto", media objects are stored under the `MMSWeb/MMS` folder of the

MMS web service application of MILOS and accessed trough the same web server where MMS is running. Elsewhere a web server or a media server that provide access to the objects according to the specified URL should be installed and configured.
An example of entries for the `mms.properties` file are given in Table 1.

An additional configuration file should be used to configure mime types to be used by MMS. The configuration file is `mimeFile.properties` and can be found in the `milos/milos/.milos/` It sets a relation between a mime type, the `<mimetypeID>` (required for MILOS URN), and file extensions. Only the objects specified in this configuration file can be managed by MILOS.

Use the following syntax to add new mime types:

```
<mimetype> = <mimetypeID>, <file extension>
```

where:

- <mimetype> is the new mime type of the media object to add (e.g. "image/jpeg", "application/vnd.ms-powerpoint", etc...)
- <mimetypeID> is a mime type ID used into the MILOS URN to identify the media objects (e.g. "image_jpeg", "application_powerpoint", etc...)
- <file extension> if the file extension of the media object (e.g. "jpg", "ppt", etc...)

An example of entries for the `mimeFile.properties` file are given in Table 1

5.2. Using MMS

A java application in order to access the MMS web service has to import a number of packages. These are listed in Table 2. An application should import the exceptions that can be raised during network operations, the stubs[3] for the web services, the interface of the MMS, the tools for crating stubs of MMS. Finally, given that the MMS basically stores and retrieve file, we need tools for dealing with file data sources and data handlers. Their use will be discussed below.
In order to create an instance (a stub actually) of MMS we should create a stub using the MMS stub tools, and we have to set it to be connected to the correct endpoint (the web server where the web service is deployed) and set it to be session aware. These actions are shown in Table 2.
Table 2 also shows an example of using MMS. In the example a digital object is inserted in MMS. The digital object is first represented as a `FiledataSource` and then handled as a `DataHandler`. After the digital object is inserted the URL for accessing it can be obtained by using the `getUrl` method. A digital object can be updated and deleted by using respectively the `update` and `delete` methods.
In the next two sessions we report all methods supported by MMS distinguishing basic methods and advanced methods.

[3] The MMS is a remote web service. In order to use it, an application has to create a local surrogate, often called stub, of the web service. A stub has the same interface of the remote service it represents. It receives requests from the application and forwards them (protecting the application from all details concerning the network communication) to the web service.

5.3. Basic MMS methods

insert

```
void insert(DataHandler dh,
        java.lang.String urn)
        throws java.MMDLS.RemoteException
```

Insert a media object into the multimedia repository. The media object must be wrapped in a `DataHandler` object

Parameters:

`dh` - Media object to insert

`urn` - Related media object URN

Throws:

`java.MMDLS.RemoteException`

getUrl

```
java.lang.String getUrl(java.lang.String urn)
                        throws
java.rmi.RemoteException
```

Get the URL of a media object

Parameters:

`urn` - media object URN

Returns:

return the URL of the media object

Throws:

`java.MMDLS.RemoteException`

delete

```
void delete(java.lang.String urn)
        throws java.rmi.RemoteException
```

Delete a media object

Parameters:

`urn` - media object URN

Throws:

`java.rmi.RemoteException`

update

```
void update(DataHandler dh,
        java.lang.String urn)
        throws java.rmi.RemoteException
```

Update a media object into the multimedia repository.

Parameters:

`dh` - Media object to insert

`urn` - URN of the previous media object

Throws:

`java.rmi.RemoteException`

5.4. Advanced MMS methods

insert2

```
void insert2(DataHandler dh,
         java.lang.String urn,
         java.lang.String version)
         throws java.rmi.RemoteException
```

Insert a media object into the multimedia repository. The media object must be wrapped in a `DataHandler` object. A version identifier is associated with the document. Several version of the same object can be associated to the same URN. Versions of objects are physically stored in a subdirectory of the directory assigned to the given URN. The name of the subdirectory is `version`.

Parameters:

`dh` - Media object to insert

`urn` - Related media object URN

`version` - Media object version (e.g. big, small, thumbnail, etc...)

Throws:

`java.rmi.RemoteException`

getUrl2

```
java.lang.String getUrl2(java.lang.String urn,

java.lang.String version)
                                  throws
java.rmi.RemoteException
```

Get the URL of a specific version of a media object

Parameters:

`urn` - media object URN

`object` - version (e.g. big, small, thumbnail, etc...) of the media object (if it exists)

Returns:

return the URL of the media object

Throws:

`java.rmi.RemoteException`

delete2

```
void delete2(java.lang.String urn,
             java.lang.String version)
             throws java.rmi.RemoteException
```

Delete a specific version of a media object

Parameters:

`urn` - media object URN

`object` - version (e.g. big, small, thumbnail, etc...) of the media object (if it exists)

Throws:

`java.rmi.RemoteException`

update2

```
void update2(DataHandler dh,
             java.lang.String urn,
             java.lang.String version)
             throws java.rmi.RemoteException
```

Update a specific version of a media object into the multimedia repository.

Parameters:

`dh` - Media object to insert

`urn` - URN of the previous media object

`version` - Media object version (e.g. big, small, thumbnail, etc...)

Throws:

`java.rmi.RemoteException`

insertKeyframe

```
void insertKeyframe(DataHandler dh,
                    java.lang.String videoUrn,
java.lang.String keyframeId)
                    throws
java.rmi.RemoteException
```

Insert a video's keyframe object into the multimedia repository. The video's keyframe object must be wrapped in a `DataHandler` object. Keyframes are physically stored in a subdirectory of the directory assigned to the URN of the main video.

Parameters:

`dh` - Video's keyframe object to insert

`videoUrn` - Video's URN

`keyframeId` - ID of the video's keyframe

Throws:

`java.rmi.RemoteException`

getKeyframeUrl

```
java.lang.String
getKeyframeUrl(java.lang.String videoUrn,

java.lang.String keyframeId)
                                        throws
java.rmi.RemoteException
```

Get the URL of a video's keyframe object

Parameters:

`urn` - video's URN

`keyframeId` - ID of the video's keyframe

Returns:

return the URL of the video's keyframe object

Throws:

`java.rmi.RemoteException`

deleteKeyframe

```
void getKeyframeUrl(java.lang.String videoUrn,

java.lang.String keyframeId)
                                        throws
java.rmi.RemoteException
```

Delete a video's keyframe object

Parameters:

`urn` - video's URN

`keyframeId` - ID of the video's keyframe

Throws:

`java.rmi.RemoteException`

invalidateSession

```
void invalidateSession()
                               throws
java.rmi.RemoteException
```

Invalidate a HTTP Session

Throws:

`java.rmi.RemoteException`

6. XML Search Engine: XMLSE

A typical search in a DL is performed on metadata which describe the document content end their bibliographic information. Three different approaches have been adopted until now to support document retrieval in digital libraries: (a) use of relational databases; (b) use of information retrieval engines; (c) full sequential scan of metadata records. Unfortunately, these approaches did not prove to be effective for DL applications: designers had to face the problem of

Examples of entries in the `index.properties file`:

```
MediaUri = pathindex
Abstract = fulltext
Image = similarityindex
```

Table 3: Examples of configuration of XMLSE in MILOS version 1.0

choosing the right compromise between efficiency of the search systems and complexity of the metadata schema. The result of this compromise is that in many cases DLs use very simple and flat metadata schemas such as Dublin Core [1].

Solution (a) requires that metadata should be converted into relational schemas. This is easy for simple flat metadata schemas, such as Dublin Core, but it far more difficult for complex and descriptive metadata schemas. Moreover, a query on these metadata must be translated into complex SQL queries at relational level, resulting in many expensive joins to implement tree structure traversals. Thus, the resulting search performance is often unacceptable. However, even with flat metadata schemas, pure relational databases do not offer all functionalities needed for an effective retrieval, such as full text search.

Solution (b) uses full text search engines to index metadata records. In this case the main emphasis is devoted to the textual information contained in metadata fields. Many text search engines offer the fielded indexing capability, where text contained in different fields is independently indexed. However, applications are limited to relatively simple and flat metadata schemas. In addition, it is not possible to search by specifying ranges of values.

Solution (c) is very trivial and inefficient. It is not practicable in applications that pretend to be more than toy systems. In this case no indexing is performed on the metadata and the custom search algorithms always scan the entire metadata set to retrieve searched information.

MILOS uses a different approach: we have designed and implemented an enhanced native XML database/repository system with special features for DL applications [8]. This is especially justified by the well known and accepted advantages of representing metadata as XML documents. Metadata represented with XML might have arbitrary complex structures, which allows to deal with complex metadata schemas, and might be easily exported and imported. Our XML database can store and retrieve any valid XML document. No metadata schema or XML schema definition is needed before inserting an XML document, except optional index definitions for performance boosting. Once an arbitrary XML document has been inserted in the database it can be immediately retrieved using XQuery. This allows DL applications to use arbitrary (XML encoded) metadata schemas and to deal with heterogeneous metadata, without any constraint on schema design and/or overhead due to metadata translation.

The MILOS XML search engine supports high performance search and retrieval on heavily structured XML documents, relying on specific index structure [7][12], as well as full text search[11], and feature similarity search [9]. The system administrator can associate an index to a specific XML element. For instance, the tag `<abstract>` can be associated with a full text index and to an automatic topic classifier that automatically indexes it with topics chosen from a controlled vocabulary. On the other hand, the MPEG-7 [4] `<VisualDescriptor>` tag can be associated with a similarity search index structure and with an automatic visual content classifier. The XQuery language has been extended with new operators that deal with approximate match and ranking, in order to deal with these new search functionality.

In our database every XML document is identified by an URN. Therefore, relationships and links among documents - even if they are stored in different repositories - can be easily and unambiguously represented.

6.1. Configuration of XMLSE

When you insert an XML file in MILOS, the underlying XML database indexes it to support efficient query execution on it.

By default the XML database only indexes the structure of the XML file for efficient structure query processing. Faster query processing can be obtained by asking the XML database to also index the content of some specific elements.

The way in which indexes are configured depends on the version of MILOS you are using. Version 1.0 offers some predefined indexes and very simple configuration options. Version 2.0 in addition to the predefined indexes offers a plug-in mechanism that enables new index to be easily implemented, installed, and configured.

6.2. Version 1.0

Version 10. of MILOS offers three different types of indexes:

a) content index for exact match of XML elements (named "pathindex")
b) content index for text retrieval of XML elements (named "fulltext")
c) content index for content based retrieval of elements containing MPEG-7 visual descriptors (named "similarityindex"). The `<image>` element should be associated to the index.

Examples of entries in the `index.properties file`:

```
/Mpeg7/Description/MultimediaContent/Image/MediaLocator/MediaUri = PATHINDEX/YAPI
/newsitem/text = FULLTEXT/LUCENE
/Mpeg7/Description/MultimediaContent/Image = SIMILARITY/AMTREE[GC,EH]
```

Examples of entries in the `indexes.properties file for the AMTREE`:

```
GC = MPEG7CLD
EH = MPEG7EHD
```

Examples of entries in the `modulesFactory.properties file`:

```
FULLTEXT/LUCENE = it.cnr.isti.milos.dataLogic.fulltext.Lucene_IRIndex
SIMILARITY/AMTREE  = it.cnr.isti.milos.dataLogic.similarity.SimilarityIndex
```

Table 4: Examples of configuration of XMLSE in MILOS version 2.0

Configuration for the indexes can be set in the file "`index.properties`" file in the `milos/milos/.milos` folder, with the following syntax:

```
<XML element name> = <index name>
```

For example, to speed-up exact match search on the `<name>` element you need to add the following entry

```
        name = pathindex
```

To support text search on the `<abstract>` element you have to add the following entry

```
        abstract = fulltext
```

To use the MPEG-7 visual descriptor index on the `<Image>` element you have to add the following entry

```
        Image = similarityindex
```

An index for exact match search should be used to speed-up queries where exact match conditions on specific elements are used. For instance:

```
for $a in /article
where $a//name= 'John Smith'
return $a/title
```

runs faster if an exact match index is defined on the `<name>` element.

An index for text search should be used to support queries where text search on content of specific elements are used. For instance:

```
for $a in /article
where $a//abstract ~ 'information retrieval'
return $a/title
```

searches for titles of articles whose `<abstract>` element is related to information retrieval. If no index is defined, the ~ operator performs a sub-string match search using a trivial sequential text scan algorithm.

An index for similarity search should be used to support queries where content based retrieval on content of specific elements containing MPEG-7 visual descriptors are used. For instance:

```
for $a in /Mpeg7/Description/MultimediaContent
where $a/Image ~ ‘—a visual descriptor--’
return $a/Image/MediaLocator
```

searches for pictures whose content is similar to the given visual descriptor. If no index is defined, the ~ operator performs a sub-string match search using a trivial sequential scan algorithm.

The fulltext and pathindex indexes can be associated with several elements (by using several entries in the `index.properties` file). Similarity can be just associated with one element. Table 3 shows an example of configuration of indexes.

6.3. Version 2.0

In MILOS version 2.0 index management was improved. In addition to the predefined indexes provided in version 2.0, it is possible to easily implement, install, and configure new fulltext and similarity indexes.
In this version indexes are no longer associated with an element name, now indexes are associated with paths.
The syntax to be used in the "`index.properties`" file in the `milos/milos/.milos` folder is the following:

```
<path> = <typeOfIndex>/<IndexName>
```

where `<typeOfIndex> can be PATHINDEX, FULLTEXT, or SIMILARITY. <IndexName>` indicates the name of the data structure which implement the index.

For instance

```
/article/abstract = FULLTEXT/LUCENE
```

specifies that the elements `<abstract>` contained in elements `<article>` should be indexed by LUCENE.

The predefined indexes of version 2.0 are PATHINDEX/YAPI for exact match, FULLTEXT/LUCENE for full text search, and SIMILARITY/AMTREE for mpeg7 similarity search.

In several cases it might be useful to have multiple instances of the same index structure, and/or to pass some parameters to the index to customize its behaviour. In the first case, for instance, one might want to use different physical indexes (using the same implementation) to index the content of different elements. In the second case, for instance, one might want to specify in a specific query the visual descriptor

Packages to be imported to use XMLSE:

```
import java.rmi.RemoteException;                    //RMI Exceptions
import javax.xml.rpc.Stub;                          //Stubs for web services
import it.cnr.isti.milos.dataLogic.XmlSE.XmlSE_IF;         //Interface of the XMLSE web service
import it.cnr.isti.milos.dataLogic. XmlSE.XmlSE_Impl;      //Tool for creating a stub for SMLSE
```

Actions to be executed to obtain an instance of XMLSE (a statefull session aware stub for XMLSE):

```
XmlSE_IF xmlse=(XmlSE_IF) new XmlSE_Impl().getXmlSE_IFPort();
((Stub)xmlse)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,         "http://localhost:8080/milos-
XmlSE/XmlSE");
((Stub)xmlse)._setProperty(Stub.SESSION_MAINTAIN_PROPERTY,  new Boolean (true));
```

Example of use of an instance of XMLSE:

```
String guid="0a783ee65ab33d679aab0cb5e322e64d";     //a GUID to be used to create an URN
String urn="urn:milos:test:"+guid;   //an URN
//a dummy XML document:
String dummyXML = "<a> <b>hello world</b> <c>How are you</c></a>";
//insert, and search for xml documents:
String xquery= "for $a in /a where $a//b='hello world' return $a/c";      //an XQuery query
try{
        xmlse.insert(dummyXML, urn);                        //insert an XML document in XMLSE
        xmlse.search(xquery);                               //run the query
        while(xmlse.next()){                                //browse results
                System.out.println(xmlse.getContent());     //get a result
        }
}catch(RemoteException e){e.printStackTrace();};
```

Table 5: How to use XMLSE

that has to be used to measure the image similarity. To support this, the syntax used in the "`index.properties`" file in the `milos/milos/.milos` folder, to define FULLTEXT and SIMILARITY indexes can also be the following:

```
<path>=<typeOfIndex>/<IndexName>[(instanceName
OrParameter,)*]
```

where `instanceNameOrParameter` can be a label known to the index structure identified by `<typeOfIndex>/<IndexName>`. An example of entries in the `index.properties` file is shown in Table 4.
Among the predefined indexes, this possibility currently can be used only with the SIMILARITY/AMTREE predefined index. For instance

```
path1 = SIMILARITY/AMTREE[GC,EH]
```

specifies that the content of `path1` should be indexed by the AMTREE and that the “GC”, and “EH” labels are sent to the index as well, to decide what to do with the received content. The specific behaviour of the AMTREE in correspondence of the label MUST be configured by using the “`indexes.properties`” file contained in the `milos/milos/.milos/similarity/amtree` folder. The syntax that can be used here is the follwing:

```
<label> = <feature>
```

where `<feature>`can be `MPEG7CLD (colour), MPEG7CSD (colour structure) , MPEG7EHD (edge histogram), MPEG7HTD (homogeneous texture), MPEG7SCD (scalable color), MPEG7COMB (fixed combination of descriptors).`

For instance

```
GC = MPEG7CLD
```

specifies that a separate physical index is built to consider the colour descriptor, when the label GC is received.

In a query the labels can be used as well. For instance:

```
for $a in /Mpeg7/Description/MultimediaContent
where $a/Image ~ (‘<descr>’, EH)
return $a/Image/MediaUri
```

sends the label GC to the index in addition to the content to be matched. In this case the physical index that uses the colour descriptor is used to perform similarity search. If no label is specified in the query and labels are specified in the `index.properties` file, then the first label is used, elsewhere no label are passed. The index implementation has to decide what to do in this case.
The AMTREE index implementation uses the first entry in the “`indexes.properties`” file contained in the “`milos/milos/.milos/similarity/amtree`” directory. An example of entries in the `indexes.properties` file is shown in Table 4.

It is possible to implement new indexes that use particular strategies and that satisfy particular needs. The association between the pair `<typeOfIndex>/<IndexName>` and a specific index implementation is set in the "`modulesFactory.properties`" file in the `milos/milos/.milos` folder. The syntax that can be used is

```
<typeOfIndex>/<IndexName>= <Java Class>
```

where `<Java Class>` is the class that implement the index.
An example of entries in the `modulesFactory.properties` file is shown in Table 4.
You can implement new indexes for full text search and for similarity search. Therefore valid values for `<typeOfIndex>` are `FULLTEXT` and `SIMILARITY`.

It is possible to implement new indexes to be used in MILOS 2.0. Section 8 discussed this.

6.4. Using XMLSE

A java application in order to access the XMLSE web service has to import a number of packages. These are listed in Table 5. An application should import the exceptions that can be raised during network operations, the stubs for the web services, the interface of the XMLSE, the tools for crating stubs of XMLSE.
In order to create an instance of XMLSE we should create a stub using the XMLSE stub tools, and we have to set it to be connected to the correct endpoint and set it to be session aware. These actions are shown in Table 5.
Table 5 also shows an example of using XMLSE. In the example an XML string is inserted in XMLSE. Then, an xquery query is submitted and the results are printed.
In the next two sessions we report all methods supported by XMLSE.

6.5. Basic XMLSE methods

insert

```
void insert(java.lang.String xml,
            java.lang.String urn)
            throws java.rmi.RemoteException
```

Insert a XML document into the database
Parameters:
`xml` - XML document to insert
`urn` - Related XML document URN
Throws:
`java.rmi.RemoteException`

delete

```
void delete(java.lang.String urn)
            throws java.rmi.RemoteException
```

Delete a XML document
Parameters:
`urn` - XML document URN
Throws:
`java.rmi.RemoteException`

search

```
void search(java.lang.String query)
            throws java.rmi.RemoteException
```

Searching of XML documents. It allow to search for documents in the XML database using the XQuery syntax.
This method creates a result set that can be browsed using a cursor that can be moved by using the `next()` and `absolute()` methods. Initially the cursor is positioned before the first result.
Parameters:
`query` - xxquery query
Throws:
`java.rmi.RemoteException`

next

```
boolean next()
        throws java.rmi.RemoteException
```

Move the cursor at the next position of the current result set
It can be used to access a result set with a fine grained control, together with `absolute(int index)`, `getUrn()`, `getContent()`, `getScore()`,
A more high level and more efficient manner to access to the query resultset is to call `getResults(int startFrom, int numElements)` instead
Returns:
Return true if it has a result at the next position
Throws:
`java.rmi.RemoteException`

absolute

```
boolean absolute(int index)
          throws
java.rmi.RemoteException
```

Set the value of the index position of the query resultset.
It can be used to access a result set with a fine grained control, together with `next()`, `getUrn()`, `getContent()`, `getScore()`,
A more high level and more efficient manner to access to the query resultset is to call `getResults(int startFrom, int numElements)` instead
Parameters:
`index` - Starting position
Returns:
Return true if it has a result at the `index` position
Throws:
`java.rmi.RemoteException`

getResults

```
java.lang.String[][] getResults(int startFrom,

int numElements)
                                        throws
java.rmi.RemoteException
```

Get the query results
Parameters:
`startFrom` - Starting point of the resultset
`numElements` - Number of results to return
Returns:
Return a matrix of results. Each row in the matrix contain: [document `score`, document `URN`, and XML `content`] of a query result

If the resultset length is smaller than

`numElements`, then the matrix of the results will has `null` values in the related result rows
Throws:
`java.rmi.RemoteException`

getUrn

```
java.lang.String getUrn()
                        throws
java.rmi.RemoteException
```

Get the document URN of the current query result.
Returns:
The document URN of the current query result
Throws:
`java.rmi.RemoteException`

getContent

```
java.lang.String getContent()
                            throws
java.rmi.RemoteException
```

Get the XML content of the current query result.
Returns:
The XML content of the current query result
Throws:
`java.rmi.RemoteException`

getScore

```
float getScore()
               throws java.rmi.RemoteException
```

Get the document score of the current query result.
Returns:
The document score of the current query result
Throws:
`java.rmi.RemoteException`

6.6. Advanced XMLSE methods

beginBulkInsert

```
void beginBulkInsert()
                     throws
java.rmi.RemoteException
```

Start a bulk insert session.
To insert the XML documents in a bulk insert session call the `bulkInsert(String xml, String urn)` method.
To end a bulk insert session call the `commitBulkInsert()` method.
Throws:
`java.rmi.RemoteException`

commitBulkInsert

```
void commitBulkInsert()
                      throws
java.rmi.RemoteException
```

Commit a bulk insert session.
To start a bulk insert session call the `beginBulkInsert()` method.
To insert the XML documents in a bulk insert session call the `bulkInsert(String xml, String urn)` method.
Throws:
`java.rmi.RemoteException`

bulkInsert

```
void bulkInsert(java.lang.String xml,
                java.lang.String urn)
                throws
java.rmi.RemoteException
```

Insert the XML documents into the database in a bulk insert session. This is typically much faster than using the `Insert()` method: connections to indexes are opened when `beginBulkInsert()` is executed and closed when `commitBulkInsert()` is executed.
Parameters:
`xml` - XML document
`urn` - Related XML document URN
Throws:
`java.rmi.RemoteException`

getDocument

```
java.lang.String
getString(java.lang.String urn)
                       throws
java.rmi.RemoteException
```

Get a XML document from the database and return it in a String object
Parameters:
`urn` - XML document URN
Returns:
Return XML document in a String object
Throws:
`java.rmi.RemoteException`

invalidateSession

```
void invalidateSession()
                       throws
java.rmi.RemoteException
```

Invalidate a HTTP Session
Throws:
`java.rmi.RemoteException`

7. Multi Media Digital Library Service: MMDLS

The Multi Media Digital Library Service (MMDLS) of MILOS, also called Repository Metadata Integrator (RMI) somewhere, manages the accesses to the MMS and XMLSE. In addition it supports metadata mapping to guarantee metadata independence. Therefore, in addition to some functionalities related to the mapping features, it also offers the functionalities of MMS and XMLSE.
The mapping of application requests into requests compatible to the metadata schema actually managed by the MCMS is accomplished by defining a set of schema mapping rules. The main purpose of this mapping is to translate application requests into XQuery queries compliant to the stored metadata. This mechanism allows the MMDLS to translate names of fields (such as Title, Author, etc.) known to the DL application, into requests to the MSR without the need

Examples of entries in the `milos.properties file`:

```
hostMMS = http://213.26.247.189:8080/milos-MMS
hostXmlSE = auto
```

Examples of entries in the `mapping.properties file`:

```
dc.title = /lom, general/title/langstring
dc.description = /lom, general/description/langstring
```

Table 6: Examples of configuration of MMDLS in MILOS

of knowing the specific schema model adopted. When a new XML schema is introduced, the system administrator must specify the mappings for the new metadata.

7.1. Configuration of MMDLS

MMDLS has to know the addresses of XMLSE and MMS. This can be specified in the file `milos.properties` in the `milos/milos/.milos` folder. It accepts rules of the following form:

<endpoint>= <URL>

where <endpoint> can be `hostMMS` and `hostXmlSE`. When MMDLS is managed by the same Tomcat instance that manages MMS and XMLSE, the keyword auto can be used in place of an URL. In This case MMDLS tries to guess the actual address looking at the Tomcat configuration files. Table 6 shows an example of configuration of MMDLS.
MMDLS can be configured to support different mappings between metadata known by the application and those stored in the XMLSE by using the `mapping.properties` file located in the `.milos` directory.
The `mapping.properties` file contains a set of mapping rules. Each mapping rule specifies how to translate the name of a metadata field, known to the application, into an XPath expression that specifies the XML path names that should be used to access that metadata field in the target metadata schema. A generic mapping rule has the following structure:
<metadataType>.<FieldName> = <RE_XPath>,< SE_XPath> where

1. The *<metadataType>* field identifies the metadata model used by the application e.g. DublinCore, SCORM [6], MPEG-7 [4] etc;
2. *<FieldName>* is the name of a metadata field requested by the application e.g., Title, Author, etc.;
3. <*RE_XPath*> (Retrieved Element XPath) is the XPath corresponding to the XML element that will be retrieved with this rule;
4. <*SE_XPath*> (Searched Element XPath) is the XPath, under <*RE_XPath*>, of the element that contains the value of the metadata field used for searching. <*SE_XPath*> can also be empty and in this case just the <*RE_XPath*> is used.

As an example Table 6 shows some mapping rules that can be used for DL for e-Learning applications, where the Learning Objects in the repository have a complex metadata structure, based on SCORM. The mapping rules map Dublin Core requests into SCORM.

They specify that the Dublin Core metadata fields `dc.title` and `dc.description` can be searched in SCORM respectively by means of the XPath string `lom/general/title/langstring`, and `lom/genaral/description/langstring`. The whole `<lom>` element will be retrieved when `<langstring>` contains the desired value. Note that, the `<title>` and `<description>` SCORM XML elements do not contain the title text of the document, but the element `<langstring>`, which in turn contains the real text.
Let us now explain how the mapping directives are used by MMDLS to generate the XQuery query. The MMDLS allows applications to search on metadata by using the *search* method:

search(**String** MetadaType, **String[]** values, **String[]** fields, **String[]** operators, **String** returnField),

This method searches for a set of metadata records of the specified *MetadaType*. The *fields* parameter is an array of (application known) names of metadata fields, of the *MetadaType*, to search for. The *values* parameter specifies the values that the fields must match (the different fields are searched by using the boolean connective **AND**). The *operators* parameter specify the matching operator to be used. Finally, *returnField* specifies the field of the retrieved records (i.e. *RE_X_Path*) that the application wants to know. The method translates the request into an XQuery query as follows:

1. for each triple <*MetadaType*, $value_i$, $field_i$, op_i >, specified in the *search*, MMDLS searches the mapping rules matching *MetadataType.*$field_i$ to fetch the corresponding XPath strings $RE\_X\_Path_i$ and $SE\_X\_Path_i$;
2. given the pair <*MetadaType, ReturnField*>, specified in the *search*, MMDLS searches the mapping rule matching *MetadataType.returnField* to fetch the corresponding XPath strings $RE\_X\_Path_{ret}$ and $SE\_X\_Path_{ret}$.
3. checks that all the strings $RE\_X\_Path_i$ and $RE\_X\_Path_{ret}$ are the same string and call that

Packages to be imported to use MMDLS:
```
import java.rmi.RemoteException;                      //RMI Exceptions
import javax.xml.rpc.Stub;                            //Stubs for web services
import it.cnr.isti.milos.businessLogic.MMDLS.MMDLS_IF;     //Interface of the MMDLS web service
import it.cnr.isti.milos.businessLogic.MMDLS.MMDLS_Impl;   //Tool for creating a stub for MMDLS
import javax.activation.FileDataSource;               //Datasources for files
import javax.activation.DataHandler;                  //To wrap media objects when inserting
```

Actions to be executed to obtain an instance of MMDLS (a statefull session aware stub for MMDLS):
```
MMDLS_IF mmdls = (MMDLS_IF) new MMDLS_Impl().getMMDLS_IFPort();
((Stub)mmdls)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, "http://localhost:8080/milos-
MMDLS/MMDLS");
((Stub)mmdls)._setProperty(Stub.SESSION_MAINTAIN_PROPERTY, new Boolean(true));
```
Example of use of an instance of MMDLS:
```
String object_guid="0a783ee65ab33d679aab0cb5e322e64d";     //a GUID to be used to create an URN
String object_urn="urn:milos:test:image_jpeg:"+guid;       //an URN for the digital object
String metadata_guid="7b983ee65ab33d679aab0cb5e3a2e694";   //a GUID to be used for metadata
String metadata_urn="urn:milos:test:image_jpeg:"+guid;     //an URN for metadata
//create a Data source for the image to be inserted:
FileDataSource fds=new FileDataSource("E:/MILOS-sdk/test-data-sets/images/"+guid+".jpg");
//create a data hander for the object to be inserted:
DataHandler dh=new DataHandler(fds);
String[] operators = {"="};  //operators to match fields against values
String return_field = "image_handler";      //field to be returned
String metadata = "test";    //metadata schema to be searched

//insert, search and retrieva using MMDLS
try{
       mmdls.insertObject(dh,object_urn);                 //insert a digital object
       mmdls.insertMetadata(metadata,metadata_urn,null);  //insert metadata
       mmdls.search(metadata, values,fields, operators, return_field);   //search using mapping
       String results[][] = mmdls.getResults(0,1);         //get the first result
       String url=mmdls.getObjectUrl(object_urn);                 //get the URL from the URN
}catch(RemoteException e){e.printStackTrace();}
```

Table 7: How to use MMDLS

string *RE_X_Path*, otherwise fail and stop;

4. finally, combines the XPath strings *RE_X_Path*, *RE_X_Path*$_i$, and *SE_X_Path*$_{ret}$ and *op*$_i$ to generate the XQuery query, as follows:

for *$a* **in** *RE XPath*
where $*a*=*SE XPath*$_1$ *op*$_1$ *value*$_1$ **and** *: : :* **and** $*a op*$_n$ *SE XPath*$_n$ = *value*$_n$
return *$a/SE XPath*$_{ret}$

Example: Suppose that an application wants to use Dublin Core to search SCORM metadata having a specific title, and wants to have back the corresponding descriptions. In this case we have *MetadataType = dc, field*$_1$ *= title, returnField*$_1$ *= description*. Applying the previous mapping rules we obtain:

for *$a* **in** /lom
where $a/general/title/langstring = *value*$_1$
return $a/general/description/langstring

7.2. Using MMDLS

A java application in order to access the MMDLS web service has to import a number of packages. These are listed in Table 7. An application should import the exceptions that can be raised during network operations, the stubs for the web services, the interface of the MMDLS, the tools for creating stubs of MMDLS. Finally, we need tools for dealing with file data sources and data handlers.

In order to create an instance (a stub actually) of MMDLS we should create a stub using the MMLDS stub tools, and we have to set it to be connected to the correct endpoint (the web server where the web service is deployed) and set it to be session aware. These actions are shown in Table 7.

Table 7 also shows an example of using MMDLS. In the example both a digital object and a metadata are inserted in MMDLS. The example also shows how to search for metadata using the mapping functionalities and how to retrieve results and digital objects.

In the next two sessions we report all methods supported by MMDLS distinguishing basic methods and advanced methods.

7.3. Basic methods

insertMetadata

```
void insertMetadata(java.lang.String xml,
                    java.lang.String urn,
                    java.lang.String wrap)
                    throws
java.rmi.RemoteException
```
Inserts a XML document into XMLSE

Parameters:

`xml` - XML document to insert

`urn` – URN to be assigned to the XML document

`wrap` - metadata category (optional) (add wrap as a root element of the document)
Throws:
`java.rmi.RemoteException`

deleteMetadata

```
void deleteMetadata(java.lang.String urn)
                    throws
java.rmi.RemoteException
```

Deletes a XML document into the XMLSE
Parameters:
`urn` - XML document URN
Throws:
`java.rmi.RemoteException`

updateMetadata

```
void updateMetadata(java.lang.String xml,
                    java.lang.String urn,
                    java.lang.String wrap)
                    throws
java.rmi.RemoteException
```

Updates a XML document into the XMLSE
Parameters:
`xml` - XML document to insert
`urn` - URN of the previous XML metadata
`wrap` - metadata category (optional)
Throws:
`java.rmi.RemoteException`

search

```
void search(java.lang.String metadataType,
        java.lang.String[] values,
        java.lang.String[] fields,
        java.lang.String[] operators,
        java.lang.String returnField)
        throws java.rmi.RemoteException
```

Searching for XML documents. This is a high level searching method and it doesn't requires to express an XQuery query. The XQuery query is built by internal routines using the "mapping.properties" file. Be careful to add the required entries into the "mapping.properties" mapping file in the ".milos" directory before to run the search. The `operators` parameter allow the following operator types:
the standard xquery operators `"="`, `">"`, `"<"`, `">="`, `"<="`, `"!="` and the new approximate operator `"~"` (for similarity and fulltext search query)
Results are accessed using the `getResults()` method
Parameters:
`metadataType` - metadata schema known by the application
`values` –array of values to be matched
`fields` – array of application known fields to be matched
`operators` – array of operators to be used for matching
`returnField` - field whose content has to be returned
Throws:
`java.rmi.RemoteException`

For example, suppose that the application executes:

```
search("MT",{"value_1",…,"value_n"},
{"field_1",…"field_n"},          {"=",…,"="},
"return_field")
```

Suppose there are the following mapping rules in mapping.properties

```
        MT.field_1=/my_XML,/xml_field_1
        ...
        MT.field_n=/my_XML,/xml_field_n
        MT.return_field=/my_XML,/xml_field_retu
rn
```

MMDLS generates and submit to XMLSE the following Xquery

```
        for $a in /my_XML
        where
               $a/xml_field_1=value_1 and
               ...
               $a/xml_field_n=value_n
        return $a/xml_field_return
```

getResults

```
java.lang.String[][] getResults(int startFrom,

int numElements)
                                 throws
java.rmi.RemoteException
```

Gets the query results
Parameters:
`startFrom` - Starting position in the result set. The first element in the result set is at position 0
`numElements` - Number of results to be returned
Returns:
Return a matrix of results. Each row in the matrix contain: [document `score`, document `URN`, and XML `content`] of a query result

If the result set length is smaller than `numElements`, then the matrix of the results will have `null` values in the related result rows
Throws:
`java.rmi.RemoteException`

insertObject

```
void insertObject(DataHandler dh,
                  java.lang.String urn)
                  throws
java.rmi.RemoteException
```

Inserts a media object into the multimedia repository. The media object must be wrapped in a `DataHandler` object
Parameters:
`dh` - Media object to insert
`urn` - Related media object URN
Throws:
`java.rmi.RemoteException`

getObjectUrl

```
java.lang.String
getObjectUrl(java.lang.String urn)
                                   throws
java.rmi.RemoteException
```

Gets the URL of a media object

Parameters:

`urn` - media object URN

Returns:

return the URL of the media object

Throws:

`java.rmi.RemoteException`

deleteObject

```
void deleteObject(java.lang.String urn)
                     throws
java.rmi.RemoteException
```

Deletes a media object into the multimedia repository

Parameters:

`urn` - media object URN

Throws:

`java.rmi.RemoteException`

updateObject

```
void updateObject(DataHandler dh,
                     java.lang.String urn)
                     throws
java.rmi.RemoteException
```

Updates a media object into the multimedia repository.

Parameters:

`dh` - Media object to insert

`urn` - URN of the previous media object

Throws:

`java.rmi.RemoteException`

7.4. Advanced methods

beginBulkInsert

```
void beginBulkInsert()
                        throws
java.rmi.RemoteException
```

Starts a bulk insert session. To insert the XML documents in a bulk insert session call the `bulkInsert(String xml, String urn, String wrap)` method. To end a bulk insert session call the `commitBulkInsert()` method.

Throws:

`java.rmi.RemoteException`

commitBulkInsert

```
void commitBulkInsert()
                         throws
java.rmi.RemoteException
```

Commits a bulk insert session. To start a bulk insert session call the `beginBulkInsert()` method. To insert the XML documents in a bulk insert session call the `bulkInsert(String xml, String urn, String wrap)` method.

Throws:

`java.rmi.RemoteException`

bulkInsert

```
void bulkInsert(java.lang.String xml,
                java.lang.String urn,
                java.lang.String wrap)
                throws
java.rmi.RemoteException
```

Inserts the XML documents into the database in a bulk insert session. This is typically much faster than using the `Insert()` method: connections to indexes are opened when `beginBulkInsert()` is executed and closet when `commitBulkInsert()` is executed.

Parameters:

`xml` - XML document to insert

`urn` - Related XML document URN

`wrap` - metadata category (optional) (add `wrap` as a root element of the document)

Throws:

`java.rmi.RemoteException`

insertObject2

```
void insertObject2(DataHandler dh,
                      java.lang.String urn,
                      java.lang.String version)
                      throws
java.rmi.RemoteException
```

Inserts a media object into the multimedia repository. The media object must be wrapped in a `DataHandler` object. A version identifier is associated with the document. Several version of the same object can be associated to the same URN. Versions of objects are physically stored in a subdirectory of the directory assigned to the given URN. The name of the subdirectory is `version`.

Parameters:

`dh` - Media object to insert

`urn` - Related media object URN

`version` - Media object version (e.g. big, small, thumbnail, etc...)

Throws:

`java.rmi.RemoteException`

getObjectUrl2

```
java.lang.String
getObjectUrl2(java.lang.String urn,

java.lang.String version)
                                         throws
java.rmi.RemoteException
```

Gets the URL of a media object

Parameters:

`urn` - media object URN

`object` - version (e.g. big, small, thumbnail, etc...) of the media object (if it exists)

Returns:

return the URL of the media object

Throws:

`java.rmi.RemoteException`

deleteObject2

```
void deleteObject2(java.lang.String urn,
                      java.lang.String version)
                      throws
java.rmi.RemoteException
```

Delete a media object into the multimedia repository
Parameters:
`urn` - media object URN
`object` - version (e.g. big, small, thumbnail, etc...) of the media object (if it exists)
Throws:
`java.rmi.RemoteException`

insertKeyframe

```
void insertKeyframe(DataHandler dh,
                    java.lang.String videoUrn,

java.lang.String keyframeId)
                    throws
java.rmi.RemoteException
```

Inserts a video's keyframe object into the multimedia repository. The video's keyframe object must be wrapped in a `DataHandler` object. Keyframes are physically stored in a subdirectory of the directory assigned to the URN of the main video.
Parameters:
`dh` - Video's keyframe object to insert
`videoUrn` - Video's URN
`keyframeId` - ID of the video's keyframe
Throws:
`java.rmi.RemoteException`

getKeyframeUrl

```
java.lang.String
getKeyframeUrl(java.lang.String videoUrn,

java.lang.String keyframeId)
                                throws
java.rmi.RemoteException
```

Gets the URL of a video's keyframe object
Parameters:
`urn` - video's URN
`keyframeId` - ID of the video's keyframe
Returns:
return the URL of the video's keyframe object
Throws:
`java.rmi.RemoteException`

getMetadata

```
java.lang.String
getMetadata(java.lang.String urn,

boolean removeWrap)
                             throws
java.rmi.RemoteException
```

Gets a XML document from XMLSE
Parameters:
`urn` - XML document URN
`removeWrap` - If `removeWrap` is true, the root element of the XML document is removed. (To be used if the XML document has been inserted with a wrap)
Returns:
Return the XML document in a `String` object
Throws:
`java.rmi.RemoteException`

query

```
void query(java.lang.String query)
           throws java.rmi.RemoteException
```

Implements the XML documents searching functionality. It allows us to search for documents in the XML database using the XQuery syntax. Results are accessed by using the `getResults()` method.
Parameters:
`query` - xxquery query
Throws:
`java.rmi.RemoteException`

updateObject2

```
void update2(DataHandler dh,
             java.lang.String urn,
             java.lang.String version)
             throws java.rmi.RemoteException
```

Update a specific version of a media object into the multimedia repository.
Parameters:
`dh` - Media object to insert
`urn` - URN of the previous media object
`version` - Media object version (e.g. big, small, thumbnail, etc...)
Throws:
`java.rmi.RemoteException`

deleteKeyframe

```
void getKeyframeUrl(java.lang.String videoUrn,

java.lang.String keyframeId)
                                    throws
java.rmi.RemoteException
```

Delete a video's keyframe object
Parameters:
`urn` - video's URN
`keyframeId` - ID of the video's keyframe
Throws:
`java.rmi.RemoteException`

invalidateSession

```
void invalidateSession()
                       throws
java.rmi.RemoteException
```

Invalidate a HTTP Session
Throws:
`java.rmi.RemoteException`

8. Building new index plug-ins (MILOS v2.0)

Version 2.0 of MILOS offers the possibility of implementing and installing new full text and similarity indexes.

To implement a new index you have to write a java class that implements the `Indexer_IF` interface, contained in the package `it.cnr.isti.xmlrep.xxquery.engine.modules`. This interface contains all methods that XMLSE needs to interact with the index. It is possible to implement new indexes that use particular strategies and that satisfy particular needs. The association between the pair `<typeOfIndex>/<IndexName>` and a specific index implementation is set in the

```
import it.cnr.isti.xmlrep.xxquery.engine.modules.QueryDescriptor_IF;
import it.cnr.isti.xmlrep.xxquery.engine.modules.IndexerException;
import it.cnr.isti.xmlrep.xxquery.engine.modules.Indexer_IF;
import it.cnr.isti.xmlrep.xxquery.engine.modules.ResultList;
import it.cnr.isti.xmlrep.xxquery.engine.modules.Result;
import it.cnr.isti.xmlrep.xxquery.engine.modules.XMLDescriptor_IF;
import it.cnr.isti.xmlrep.xxquery.env.XmlRepEnvironment_IF;
```

Table 8: Packages to be imported to create new index plug-ins

`"modulesFactory.properties"` file in the `.milos` folder. The syntax that can be used is

```
<typeOfIndex>/<IndexName>= <Java Class>
```

where `<Java Class>` is the class that implement the index. An example of entries in the `modulesFactory.properties` file is shown in Table 4.

You can implement new indexes for full text search and for similarity search. Therefore valid values for `<typeOfIndex>` are `FULLTEXT` and `SIMILARITY`.

To implement a new index to be used in MILOS you have to

1. implement the “`Indexer_IF`” interface.
2. Give a name to your index in the `"modulesFactory.properties"` file in the `milos/milos/.milos` folder
3. Specify which elements have to be indexed with the new index by configuring the `"index.properties"` file in the `milos/milos/.milos` folder

8.1. Using the Indexer_IF interface

In order to use (implement) the Indexer_IF interface you have to import the packages shown in Table 8. To do that, you have to add the `xxquery.jar` file to your classpath. `xxquery.jar` can be found in the `milos/milos/web/XmlSEWeb/WEB-INF/lib` folder.

8.2. Index life cycle

In order to correctly implement a new index it is very important to understand the life-cycle of an index.

1. When an instance of XMLSE is asked to execute an operation for the first time it
   a. Creates and instance of all indexes
   b. For every index it calls the method `setEnvironment(…)` (See Below), to tell all indexes about their running environment
   c. For every index it calls the method `open(…)`(See Below) to perform some index initialisation
2. When XMLSE decides that something has to be inserted or searched in an index, it calls the appropriate methods
3. When the XMLSE instance is closed (for instance when the session expires) all indexes are closed with the `close()` method to release all resources.

Summarizing we have:

$$\begin{tikzcd} \texttt{setEnvironment(…)} \arrow[d] \\ \texttt{open(…)} \arrow[d] \\ \texttt{insert(…)} \arrow[d, bend left] \\ \texttt{search(…)} \arrow[u, bend left] \arrow[d] \\ \texttt{close(…)} \end{tikzcd}$$

8.3. Methods of Indexer_IF:

setEnvironment

```
void
setEnvironment(it.cnr.isti.xmlrep.xxquery.env.
XmlRepEnvironment_IF env)
                  throws IndexerException
```

Sets the index environment through the XmlRepEnvironment_IF parameter, like the working directory of XMLSE or some others information.

Parameters:

`env` - the data structure containing information passed by the XMLSE.

XmlRepEnvironment_IF has a number of methods for accessing XMLSE information. Currently just the getStartDirectory() methods gives useful information. It returns the directory where all index working (sub-)directory should be created.

Throws:

`IndexerException`

open

```
void open()
     throws IndexerException
```

Initializes the index. It can be used to perform any index initialisation.

Throws:

`IndexerException`

close

```
void close()
      throws IndexerException
```

Closes the index. It can be used to release all resources used by the instance.

Throws:

`IndexerException`

insert

```
void insert(XMLDescriptor_IF xmlDescriptor)
        throws IndexerException
```

Inserts an entry in the index. It has to be implemented as an atomic operation. The inserted entry should be available to all sessions immediately.
XMLDescriptor represents the entry and contains data to be indexed and data that are needed by the XMLSE when an entry is retrieved.
The content of the entry can be obtained by using the methods getElementXML();The content of an entry is basically an XML element (for instance, the abstract element, or the Image element).
The entry contains additional information useful to the XMLSE:
- the id of the document which the element belongs to. It can be obtained using getDocumentId();
- the internal id of the name of the element. It can beobtained using getEidName()
- the internal id of the element being indexed. It can be obtained by using getEiid()
- the start and end position of the element in the XML tree hierarchy. It can be obtained by using getElementStart() and getElementEnd();
- the index labels associated with the index definition in the index.properties file. This information can be used by the index to decide how to index the entry. It cen be obtained by using getIndexNames();

Parameters:
`xmlDescriptor` - the descriptor of the XML element.

Throws:
`IndexerException`

search

```
ResultList search(QueryDescriptor_IF query)
             throws IndexerException
```

Returns the list of results of the query described by the QueryDescriptor_IF.
The QueryDescriptor_IF contains:
- the value key to be matched against entries of the index. It can be obtained by getIndexQuery();
- the index label used for this query. It should be one among the ones defined in index.properties. It can be obtained by getIndexName();
- the internal identifier of the searched element name. It can be obtained by getEnid();
- the identifier of the element to be matched against the key value. It can be obtained by getEiid(). **This is not used at the moment.**
The result should be returned as a ResultList. A result can be inserted in the ResultList using the addResult(Result) method. A result cab be built using the Result constructor: Result(`float score, long eiid, int start, int end`), where
-score is the score assigned to this entry,
-eiid, is the internal identifier of the returned element
-start and end are the star and end position of the element in the XML tree hierarchy.

Parameters:
`query` - the descriptor of the query.

Returns:
the list of results.

Throws:
`IndexerException`

getResultSize

```
int getResultSize()
          throws IndexerException
```

Returns the size of the most recent result.

Returns:
the dimension of the result.

Throws:
`IndexerException`

deleteDocument

```
void deleteDocument(long docId)
           throws IndexerException
```

Deletes the document with identifier DocId

Parameters:
`docId` - the identifier of the XML document

Throws:
`IndexerException`

deleteElement

```
void deleteElement(long elementInstanceId)
          throws IndexerException
```

Deletes the XML element with identifier elementInstanceId. **Currently this is not used by the XML SE.**

Parameters:
`elementInstanceId` - the identifier of the XML element

Throws:
`IndexerException`

beginBulkInsert

```
void beginBulkInsert()
            throws IndexerException
```

Initializes the resource to do the bulk insert.

Throws:
`IndexerException`

commitBulkInsert

```
void commitBulkInsert()
            throws IndexerException
```

Database changes are committed. Bulk insertions must be accessible to everybody after this.

Throws:
`IndexerException`

bulkInsert

```
void
bulkInsert(XMLDescriptor_IF xmlDescriptor)
        throws IndexerException
```

Inserts the XML documents into the database in a bulk insert session. This is typically much faster than using the `Insert()` method. It is not necessary that bulk insertions are visible by everybody until commitBulkInsert() is executed.

Parameters:

`xmlDescriptor` - a description of the XML element to insert.

Throws:

`IndexerException`

produceStatistics

```
void produceStatistics()
                        throws IndexerException
```

An utility to produce index statistics. This is not currently used by XMLSE. An offline procedure should be provided that uses it.

Throws:

`IndexerException`

9. References

[1] Dublin Core Metadata Initiative. http://dublincore.org/.

[2] Echo: European CHronicles On-line. http://pc-erato2.iei.pi.cnr.it/echo/.

[3] Helix Universal Server. http://www.realnetworks.com/products/server/index.html .

[4] Moving picture experts group. http://www.chiariglione.org/mpeg/ .

[5] Real Time Streaming Protocol. http://www.rtsp.org/ .

[6] Shareable content object reference model initiative (scorm), the xml cover pages, October 2001. http://xml.coverpages.org/scorm.html .

[7] G. Amato, F. Debole, F. Rabitti, and P. Zezula. YAPI: Yet another path index for XML searching. In *ECDL 2003, 7th ECDL Conference, Trondheim, Norway, August 17-22, 2003*.

[8] G. Amato, F. Debole, A Native XML Database Supporting Approximate Match Search, in *ECDL 2004, 8th ECDL Conference, Vienna, Austria, September 18-23, 2003*

[9] C. B¨ohm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[10] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, and L. A. Rowe. Multimedia storage servers: A tutorial. *IEEE Computer*, 28(5):40–49, May 1995.

[11] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company, 1983.

[12] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree signatures for xml querying and navigation. In *Database and XML Technologies, XSym 2003*, volume 2824 of *LNCS*, pages 149–163. Springer, 2003.