CYCLADES IST-2000-25456
An Open Collaborative Virtual Archive Environment

Detailed System Specification Report
D3.0.1

Delivery Type: R
Number: D3.0.1
Contractual Date of Delivery: month 12
Actual Date of Delivery: February 11, 2002
Task: WP3

Name of Responsible: Umberto Straccia

Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche
56124 Pisa
Italy

E-Mail: straccia@iei.pi.cnr.it

Contributors

CNR-IEI: Leonardo Candela, Donatella Castelli, Pasquale Pagano, M. Elena Renda, Fabrizio Sebastiani, Umberto Straccia
University of Dortmund: Gudrun Fischer, Norbert Fuhr
FORTH: Dimitris Plexousakis, Nick Papadopoulos
GMD-FIT: Tom Gross, Thomas Kreifelts

Abstract: This report defines the detailed specifications of the CYCLADES software components.

Contents

Chapter 1

Introduction

1.1 Rationale of Cyclades

The main goal of Cyclades is to provide an integrated environment for scholars and groups of scholars that want to use, in a highly personalized and flexible way, *open archives*, i.e. electronic archives of documents compliant with the Open Archives Initiative[1] (OAI) standard.

Cyclades users

- will be able to search documents relevant to their interests from electronic archives;
- will be made aware of new documents relevant to their interests as they become available. Relevance to the interests of a user will be assessed by estimating the semantic similarity between candidate documents and other documents the same user have previously expressed interest in;
- will be provided with highly flexible and personalized tools for organizing references to accessed documents into a personal workspace. A reference to a document consists of a record containing a description of the document by means of metadata;
- will be made aware of other Cyclades users with similar interests;
- will be able to create virtual "communities" of Cyclades users sharing common scientific or professional interests, and will thus be allowed to use a community-specific workspace, where it will also be possible to exchange information among members in the form of annotated documents;
- will be able to create virtual "projects" (i.e. project groups) of Cyclades users working at a common task, and will thus be allowed to use a project-specific workspace, where it will also be possible to exchange information in the form of annotated documents or any other kind of external documents;
- will be able to ask for recommendations of records, users, communities relevant to their interests. Relevance to the interest of a user will be estimated based on implicit or expliting ratings that other Cyclades users have given to documents.

Cyclades is thus an integrated environment for the access to scholarly information. By exchanging information in the form of annotated documents, creating and joining communities and projects, and rating documents, Cyclades users collectively contribute to building "common knowledge" for scientific cybercommunities.

[1] http://www.openarchives.org

The CYCLADES system will use the protocol specified by the Open Archives Initiative to harvest metadata from any number of archives that support the OAI standard. As the harvesting will be done by one of several interoperable services, CYCLADES is not restricted to using open archives, as additional services can be added later, supporting other kinds of electronic archives that are not compliant with the OAI standard.

1.2 Users

Scholars become CYCLADES users by registering to the CYCLADES environment. Once registered, there are different roles that the user can play while interacting with the system. The user may switch from one role to another during the same CYCLADES session, provided she has the rights needed for these roles. In particular

- Every user can act as a *single user*, i.e. only by virtue of having registered into the CYCLADES environment. In this capacity, a CYCLADES user has access to all and only the folders of which she is the owner and can also rearrange the folder hierarchy that contains them.
- A user who is a member of a community can additionally perform actions in the context of this community, thus acting as a *community member*. In this capacity, a CYCLADES user has access to all and only the folders owned by the community.
- A community member with appropriate rights can also administrate a community, thus acting as a *community administrator*. In this capacity, not only she has access to all and only the folders owned by the community, but can also rearrange the folder hierarchy that contains them.
- A user who is a member of a project can perform additional actions in the context of this project. Then, the user is acting as a *project member*. In this capacity, a CYCLADES user has access to all and only the folders owned by the project.
- A project member with appropriate rights can administrate a project and its folders. Then, the user is acting as a *project administrator*. In this capacity, not only she has access to all and only the folders owned by the project, but can also rearrange the folder hierarchy that contains them.
- A user with the appropriate rights can create a collection and manage the collection she has created. Then, the user is acting as a *collection administrator*.
- A user with the appropriate rights can register or unregister an archive or edit the registration information concerning an archive. Then, the user is acting as an *archive administrator*.
- There is a special user who has the rights to grant other users the rights to register archives and collections. This user is called the CYCLADES *system administrator*.

1.3 Functionality

The functionality of CYCLADES may be summarized by looking (i) at the actions that the users can perform, and (ii) at the actions that the system performs in the background.

Concerning (i), the main classes of actions that CYCLADES users can perform are:

- *Register and login.* In order to access the CYCLADES services, a user must register into the CYCLADES environment. The effect of a successful registration is that the user has a CYCLADES account consisting of a user name, a password, and a home folder, and may now log into the CYCLADES system using user name and password.

- *Manage records and folders.* Once a user has logged in, she can perform basic record and folder management actions on records and folders she owns. Record management ooperations include deleting a record from a folder and moving a record from a folder to another folder. Folder management operations include creating a new folder as a child of an existing folder, deleting a folder, moving a subfolder from an existing parent folder to a new parent folder.

- *Search and browse records.* At any time from login to logout, the user performs her operations with respect to a current folder; at login, the users's home folder is the current folder. Given a current folder, the user can issue queries, whose results will be stored in the current folder. Queries can be formed from scratch, or can be formed by editing an older query. Queries can be associated to folders, so that the same query can be issues again and again with respect to the same folder.

- *Get recommendations.* Recommendations made by the system to a user are associated with a given user folder. The entities that can be recommended are records, collections, users, and communities. The system recommends an entity to a user's folder when it deems that entity interesting for the user in the context of that folder. Recommendation are made by the system based on the searching behaviour of other users who are deemed similar to this user.

- *Manage collections.* A collection is a set of records, and is defined by a set of archives and an optional selection criterion on these archives. Collections provide an abstract view of archives. Users may define collections and associate them with specific search and browse formats.

- *Manage archives.* Archives can be registered into CYCLADES, so that collections may be defined that use them.

Concerning (ii), the main classes of actions that the CYCLADES system performs behind the curtains are:

- *Harvest and index records from archives.* Metadata records are harvested and indexed offline from the underlying electronic archives, and a search method for the other services is provided. Additionally, information about the used archives and the available metadata formats is maintained.

- *Filter records.* As a result of (either implicit or explicit) user queries, records are filtered by means of a "folder profile", i.e. a representation of the user's interests as shown by the contents of that folder. The folder profile is automatically maintained by the system by exploiting implicit user feedback.

1.4 Services

The CYCLADES system provides the user with different environments, according to the actions the user wants to perform.

The main environment in which the user will carry out her work will be her folder hierarchy, i.e. a set of hierarchically organized folders, each of which is a repository of metadata records retrieved from open archives. Each of these records refers to a physical document, usually stored in the same archive where the record is stored.

Each folder typically corresponds to one subject (or discipline, or field) the user is interested in. However, in order to accomplish a truly personalized interaction between user and system, this correspondence is implemented in a way which is fully idiosyncratic to the user; this means that e.g. a folder named `Nuclear Waste Disposal` and owned by user `Sue` will not correspond to any "objective" definition or characterization of what nuclear waste disposal is, but will correspond

to what Sue means by nuclear waste disposal, i.e. to her personal view of (or interest in) nuclear waste disposal. This user-oriented view of folders is realized by learning the "semantics of folders" from the current contents of the folders themselves.

Folders can be private folders (in which case they can be accessed only by the individual user who owns the folder), or community folders (in which case they can be accessed by all members of the community who owns the folder), or project folders (in which case they can be accessed by all members of the project who owns the folder). The folder hierarchy of a given user will thus include her private folders and the community/project folders owned by the communities/projects she is a member of, freely intermixed.

As all the actions that the user performs in her folder hierarchy are closely related to each other, they will be implemented by the same system component, called the *Collaborative Work Service* (CWS). This service will also be in charge of basic user management operations, such as user registration into the CYCLADES system, and folder management operations, such as creation of a home folder. This service also includes a *rating management component* (RMS), which manages all aspects related to the gathering and maintainance of user-provided document ratings.

Searching and browsing is an activity only loosely connected to a folder (since it is performed "from within" a folder, i.e. with a given folder as the current folder), but otherwise autonomous. Thus, it will be implemented by a separate system component, called the *Search and Browse Service* (SBS).

Although in the beginning the CYCLADES system will deal with open archives only, it is meant to be extensible to other data sources in the future. Thus, access to the actual metadata is implemented in a separate system component, the *Access Service* (AS). In this way, in order to use data from other kinds of archives in the future, only an appropriate Access Service will have to be implemented.

The management of collections (i.e. their definition, creation, and update) is a separate task that will be implemented by yet another separate system component, the *Collection Service* (CS). This will be in charge of providing users with a conceptual view of how documents are organized, i.e. with a view in which documents are organized into topically coherent sets of independent interest to one or more users/communities/projects.

In principle, the CYCLADES system could work without recommendation and personalization facilities. As a consequence, the recommendation and personalization functionality will also be implemented by a separate system component, the *Filtering and Recommendation Service* (FRS). This separation of concerns will also enable the experimentation with different recommendation and personalization paradigms. The FRS will be in charge of enabling users to interact with the system in a personalized way, i.e. in a way that takes into consideration the user's interests, and to do so without any added load for the user (i.e. without requesting the user to explicitly specify her interests).

All of these services will interoperate in a distributed environment. Security and system administration will be provided for centrally. Therefore, a *Mediator Service* (MS) will mediate between the individual services and will manage user sessions. The CYCLADES services can run on different machines, and will only need a HTTP connection to communicate and collaborate.

1.5 Architecture

The CYCLADES system consists of the following services, as outlined in Section 1.4:

- Collaborative Work Service
- Search and Browse Service
- Access Service

- Collection Service
- Filtering and Recommendation Service
- Mediator Service

The Collaborative Work Service provides a folder-based environment for managing metadata records, queries, collections, external documents, and annotations. Furthermore, it supports collaboration between CYCLADES users by way of folder-sharing in communities and projects. One component of this service is the Rating Management Service, which manages ratings.

The Search and Browse Service supports the activity of searching records from the various collections, of formulating and reusing queries, and browsing schemas, attribute values, and metadata records.

The Access Service is in charge of interfacing with the underlying metadata archives. In this project, only archives adhering to the Open Archives specification will be accounted for; however, the system is extensible to other kinds of archives by modifying the Access Service only.

The Collection Service manages collections, thus allowing a partinioning of the information space according to the users' interests, and making the individual archives transparent to the user.

The Filtering and Recommendation Service provides personalized filtering of queries and query results, provides recommendations of records, collections, users, and communities deemed relevant to the user's interests, and provides personalized folder management.

The Mediator Service acts as a registry for the other services and provides security, i. e. it checks if a user is entitled to use the system, and ensures that the other services are only called after proper authentication.

Each service registers itself with the Mediator Service. Whenever a service needs to communicate with another service, it asks the Mediator Service for a list of services of the appropriate type. In this version, when asked for, say, Search and Browse Services, the Mediator will return a list of Search and Browse Services, and the requesting service can be sure that each Search and Browse Service of this list will provide the same functionality.

In the first version, i. e. the system as it will be implemented in the CYCLADES project, the infrastructure will provide for replication of the services, but not for distribution. The registry is extendable, so that in future versions also distribution may be supported.

1.6 Communication protocol

The services of the CYCLADES system will communicate via HTTP, using XML-RPC [2]. XML-RPC is a simple protocol for implementing cross-platform, distributed applications. As its name suggests, communication between the distributed applications is done via remote procedure calls. The XML-RPC protocol is based on Internet standards: method calls and responses are transmitted using HTTP, and the bodies of the calls and responses are encoded in XML.

An XML-RPC message is an HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML. Procedure parameters can be scalars, numbers, strings, dates, etc.; and can also be complex record and list structures.

1.7 User interface

Most of the services (i. e. the Collaborative Work Service, the Search and Browse Service, the Access Service (for archive management), and the Collection Service (for collection management)) will

[2] http://www.xmlrpc.com/spec

provide their own user interface. The Mediator service itself will provide the registration and login interface, and a system administration interface (for assigning access rights, etc.). Additionally, the Mediator Service will integrate the user interfaces of the other services, and make sure that those services and their interfaces are called only for authorized users, and only via the Mediator Service.

Chapter 2

Access Service

2.1 Functionality

The Access Service harvests and indexes metadata records from the underlying electronic archives and provides a search method for the other services. Additionally, it maintains information about the used archives and the available metadata formats which the user can edit as needed.

Thus, the functionality of the Access Service consits of two parts:

- information retrieval on metadata records
- management of archive information

2.1.1 Information retrieval

Metadata records are a central issue in Cyclades, as they are the entities that the users will search for, share, request as recommendation, and comment. All the metadata records in Cyclades are provided by the Access Service. Thus, information retrieval on these metadata records is one of the Access Service's main tasks. It consists of:

- harvesting metadata records
- indexing metadata records
- retrieving metadata records

Harvesting metadata records

Cyclades uses metadata records from electronic archives. As many archives do not provide a search function, Cyclades collects and stores the metadata records from the corresponding archives itself, so that it can then implement a search function on its own local copies of the records.

Indexing metadata records

To allow information retrieval on the harvested metadata records, they have to be indexed. The Access Service will implement several levels of indexing.

Retrieving metadata records

The user will not directly access the Access Service to retrieve metadata records. However, some of the other CYCLADESservices need to search for records. Thus, the Access Service provides them a search function, using an internal query language that allows to specify both mandatory conditions (e. g. the author *must* equal 'Fuhr'), and weighted conditions.

2.1.2 Management of archive information

The Access Service has to know which electronic archives it should harvest. That means that archives have to be registered first. The Access Service will provide a means for a user to manage archive information, and it will also make this information available to the other CYCLADESservices.

To the user, the Access Service will provide the following functions:

- register an archive
- edit archive information
- unregister an archive

Register an archive

This function allows the user to register a new archive for future harvesting by the Access Service. The user enters the archive URL, the system then automatically collects information about the archive that the archive itself provides, and then presents this information to the user to edit. The user can also specify information that could not be automatically detected, e. g. a textual description of the content, or more details about the metadata schemas used. After this registration, the user will be responsible for the archive, from the system's perspective. Thus, from the system point of view, this user is an *archive administrator*.

Edit archive information

The content and the schemas of an archive may change with time, there may be more topics available, more languages, additional metadata schemas and so on. Thus, the *archive administrator* has to be able to change the information about the archive at any time.

Unregister an archive

It may happen that an archive is no longer available for public use, or that the owner of an electronic archive does not want its contents to be accessible via CYCLADES. In these cases, the *archive administrator* can unregister the archive. After unregistering, this archive will not be harvested any more, and no records from this archive will be available in CYCLADES.

2.2 Process flow

2.2.1 Information retrieval component

This is the core part of the Access Service's functionality. It does not require user interaction and consists of functions running periodically in the background.

Harvesting

In the first version of CYCLADES, the system will rely only on those electronic archives that adhere to the Open Archives specification [1]. This specification defines a harvesting protocol to harvest all open archives in a uniform way, without having to write a special harvesting wrapper for every single archive. According to the specification, all archives provide at least unqualified Dublin Core metadata, but any other metadata format (*schema*) can be used additionally. The Access Service will harvest metadata records in all formats that it will be able to treat.

The harvesting comes in two flavours:

- a full harvest retrieves all records available from an archive
- an incremental harvests only those records from an archive that are newer than a specified date

Both kinds of harvesting will run periodically, however, full harvests will be less frequent.

Indexing

After a set of records has been harvested, the individual metadata fields are analyzed and their content split into terms, stemming is applied, stopwords are eliminated, certain terms are normalized (e. g. languages, if unambiguously identified). Then, the frequencies of the individual terms are calculated on the field level and on the document level. Furthermore, the lengths of the individual fields and the document lengths, both in number of terms, are calculated.

Retrieval

For retrieval, the corresponding query must be analyzed. Mandatory conditions are evaluated. For the resulting set of records, the relevance for the query is estimated using the weighted conditions from the query and the weights of the concerning terms, and the records are ordered by this estimated relevance.

2.2.2 Management of archive information

This part of the Access Service's functionality allows the user to register an archive to be included in the AS's automatic harvesting mechanism, to manually edit the archive information, and to unregister an archive, if it should not be available via CYCLADESany longer.

Register an archive

- **User:** enters the archive registration environment and chooses the function to add an archive
- **System:** asks the user for the primary URL of the archive
- **User:** enters the primary url for the archive
- **System:** collects initial archive information
- **System:** presents the metadata schemas available for the new archive
- **User:** chooses the schemas to be used in the record gathering process. Only records in these formats will be available to the CYCLADESsystem from this archive.

[1] http://www.openarchives.org

- **System:** presents the plain text description, language information and temporal coverage as far as these could be determined automatically
- **User:** edits the plain text description, languages and dates covered, enters additional urls (mirrors), keywords and so on
- **System:** asks the user to confirm the registration
- **User:** confirms or cancels the archive registration
- **System:** in case of confirmation, saves the changes
- **System:** presents a success message or a failure report

Edit archive registration information

- **User:** enters the archive registration environment and chooses the function to edit the registration information for an archive
- **System:** presents a list of archives the user is allowed to edit (in case of a CYCLADESadministrator, all archives, else only those that the user has registered herself)
- **User:** selects an archive to edit
- **System:** presents the concerning archive information, i. e. available schemas and languages, temporal coverage, mirror URLs, a textual description, keywords and so on
- **User:** edits the presented archive information
- **System:** asks the user to confirm the changes
- **User:** confirms or cancels the changes
- **System:** in case of confirmation, saves the changes
- **System:** presents a success message or a failure report

Unregister an archive

- **User:** enters the archive registration environment and chooses the function to unregister an archive
- **System:** presents a list of archives the user is allowed to manage (in case of a CYCLADESadministrator, all archives, else only those that the user has registered herself)
- **User:** selects an archive to unregister
- **System:** presents the archive information
- **System:** asks the user to confirm the unregistering action
- **User:** confirms or cancels the unregistering action
- **System:** in case of confirmation, deletes all entries for this archive from all tables in the database
- **System:** presents a success message or a failure report

2.3 Internal architecture

2.3.1 Overview

Figure 2.1 shows the internal architecture of the Access Service. It consists of the following layers, respectively components:

- a relational database
- an information retrieval component
- an implementation layer
- an API
- the archive management GUI
- a communication module

In the following, we describe these individual components.

2.3.2 Database

The Access Service has to store data, i. e. records and archive information, persistently. In the first version of the system, we use the Postgres database system for this purpose. The Access Service methods, however, will be easily adaptable to any other relational database system as well.

2.3.3 Information retrieval component

This part of the Access Service contains the harvesting and indexing of metadata records, and the retrieval function. Harvesting and indexing is implemented by Perl scripts that have to be run periodically, e. g. as Unix cron jobs. Retrieval is implemented as a static Java function that can be used by the Access Service classes, and also independently from CYCLADES. The information retrieval component communicates with the database via the Perl database interface, and with external electronic archives via the Open Archives protocol.

2.3.4 Access Service implementation classes

This layer consists of the Java classes implementing the Access Service's internal and public APIs. The class implementing the concept of an archive communicates with external electronic archives via the Open Archives protocol, all classes have access to the database where their data is stored persistently.

2.3.5 Access Service API

This layer is the API to all Access Service functions that are accessible to the GUI or to other CYCLADESservices. It consists of Java interfaces specifying the available methods. The individual classes (respectively, from the implementation point of view, Java interfaces) are described below in section 2.4.

2.3.6 Archive management GUI

This component of the Access Service interacts with the user to manage the information about the individual archives. It does not use the database directly, but retrieves and saves the archive data using the Archive and Schema classes contained in the Access Service API. The GUI itself is implemented by a Java servlet running in the local Web server.

2.3.7 Communication with other services

The Access Service provides data access methods for other CYCLADESservices (see next section, 2.4) as the public part of its API. However, the API itself specifies only the available methods, not how the other services can access them. This is the task of the communication module. In the first version of CYCLADES, the communication between the different services is carried out via XML-RPC, thus, the Access Service's communication module is implemented by an XML-RPC server.

2.4 Data and method specification

2.4.1 Access Service

In the following paragraphs, we describe the classes that are defined by the Access Service API. The persistent data of this service are registered archives and their metadata schemas.

AccessService

public:

- id
  Description: the unique ID of the service, string
- archives
  Description: a list of Archive objects that the service manages
- (term,weight)* getIndexedTermsAndWeights(recordId*)
  Description: this method exports the indexed terms and their respective weights
  Input. list of recordId: the list of ids of the records for which the indexed terms and weights are to be deter
  Output. list of pairs (term, weight)
- (Record, (term,weight)*)* getIndexedTermsAndRecords(recordId*)
  Description: this method exports the indexed terms and their respective weights for each record specified
  Input. list of recordId: the list of ids of the records for which the indexed terms and weights are to be deter
  Output. list of records, each record with a list of pairs (term,weight) associated
- (Record,(term,weight)*)* search(query,maxRecordNo,maxTermNo,timeStamp)
  Description: this method determines the records corresponding to *query* and returns at most *maxRecordNo* records gathered after the time specified by *timeStamp*, and for each record, *maxTermNo* indexed terms with their weights.
  Input. query: the query string
  maxRecordNo: the maximum number of records to be returned
  maxTermNo: the maximum number of terms to be returned with each record
  timeStamp: a timestamp specifying a date and time, only records gathered after this time will be
  Output. list of records, each with a list of pairs (term, weight) associated

- (oaiId,textualDescription,keyword*,schema*,language*,temporalCoverage) getArchiveDescription(oaiId)
  Description: this method returns a complete description of the archive specified by *oaiId*
  Input. oaiId: the id of the archive
  Output. oaiId: the id of the archive (string)
  textualDescription: a string describing the archive
  list of keywords: a list of strings that describe the archive's topic
  list of schemas: a list of schema names (strings) of the metadata schemas available in the archiv
  list of language: a list of languages (strings) available in the archive
  temporalCoverage: a list of pairs (fromYear,toYear) describing the intervals of time covered by the

- (term,weight)* getMetadataAttributeTerms(oaiId,schema,attributeName,maxTerm)
  Description: this method returns for the archive with the ID *oaiId*, for the *schema* and the specified *attribute*, at most *maxTerm* indexed terms with their weights
  Input. oaiId: the id of the archive
  schema: the name of the schema (string)
  attributeName: the name of the metadata attribute (string)
  maxTerm: the maximum number of terms to be returned
  Output. a list of pairs (term,weight)

- Schema* getSchemas(collectionId*)
  Description: this method exports a list of all schemas that the archives that the specified collections are based on supply (if no collection ids are specified, then all metadata schemas of all archives are listed)
  Input. list of collectionId: the list of collection ids (strings)
  Output. a list of Schema objects

- value* getAttributeValues(archiveId*,schemaName,attributeName,maxNo)
  Description: this method returns a list of *maxNo* attribute values from the archives specified, and for the given *schemaName* and *attributeName*
  Input. list of archiveId: the list ids of the archives (strings)
  schemaName: the name of the metadata schema
  attributeName: the name of the metadata attribute
  maxNo: the maximum number of values to be returned
  Output. a list of values, their type according to the type of the attribute

- Record* getRecords(recordId*)
  Description: this method returns the full records for the given record ids
  Input. list of recordId: the list ids of the requested records
  Output. a list of records

- HTML initiateArchiveManagement(userId)
  Description: this method returns the Access Service's interface for archive management
  Input. userId: the id of the user who initiated the archive management
  Output. the initial HTML page of the archive management GUI

service internal:

- void saveSchemas(Schema*)
  Description: saves the schemas' data to the database
  Input. list of Schemas: the list of Schema objects that are to be saved

- Archive initiateArchiveRegistration(url)
  Description: creates a new archive object and collects initial data from the data provider specified by *url*
  Input. url: the URL of the new archive's data provider
  Output. a new Archive object

Archive

An instance of this class represents one single open archive. **public:**

- id
  Description: this is the unique ID of the archive, a string
- textualDescription
  Description: a textual description of the archive, entered by the archive registrator (string)
- schemas
  Description: a list of Schema objects, containing the metadata schemas that the archive exports
- url
  Description: the main URL of the archive (string)
- mirrors
  Description: a list of further URLs for the archive (list of strings)
- languages
  Description: a list of languages that the archive data covers, specified by the archive registrator (list of strings)
- temporalCoverage
  Description: a list of pairs (year,year) that specify the temporal coverage of the archive, the archive registrator enters this data during registration
- keywords
  Description: a list of keywords describing the archive content, entered by the archive registrator (list of strings)

service internal:

- void saveToDB()
  Description: saves this archive's data to the database
- void readFromDB()
  Description: reads this archive's data from the database

Schema

An instance of this class describes one metadata schema.

- name
  Description: the unique name of the schema (string)
- url
  Description: the URL of a DTD or namespace for the schema (string)
- attributes
  Description: a list of the attributes of the schema, each attribute being a tuple (name,datatype)
- void saveToDB()
  Description: saves this schema's data to the database
- void readFromDB()
  Description: reads this schema's data from the database

2.4.2 Access Service tables

The Acces Service uses a relational database and creates the following tables:

- **schemas**
  Store information about a the available metadata schemas.

name	char(50)	the unique name of the schema.
url	Varchar	the url of a DTD where the schema is described.
namespace	Varchar	the url where the schema namespace is described.

- **schemasAndAttributes**
  Store the attributes belonging to the available metadata schemas.

schemaName	char(50)	the name of the schema the attribute belongs to.
attribute	char(200)	the unique name of the attribute.
type	char(50)	the type of the attribute.

- **archives**
  Store information about a the archives indexed by the Access Service.

archiveId	char(50)	the unique archive identifier.
textualDesc	Varchar	a textual description.
url	Varchar	the url of the primary data provider.
mirror1	Varchar	the url of a data provider mirror.
mirror2	Varchar	the url of a second data provider mirror.
mirror3	Varchar	the url of a third data provider mirror.
repositoryName	Varchar	the optional name of the archive.
adminEmail	Varchar	the e-mail address of the archive responsible.
registeredBy	char(100)	the id of the user who registered the archive.
regEmail	Varchar	the e-mail address of the user who registered the archive.
lastFullHarvest	timestamp	date and time of the last full harvest.
lastIncremHarvest	timestamp	date and time of the last incremental harvest.
timeSpentFull	integer	time spent at last full harvest.
timeSpentIncremal	integer	time spent at last incremental harvest .
fullInterval	integer	interval in hours for the full harvest.
incremInterval	integer	interval in hours for the incremental harvest.
fullContent	Varchar	the original content of the Identify record.

- **archivesAndIncidents**
  Stores relevant incidents for each archive.

archiveId	char(50)	the identifier of the archive.
atTime	timestamp	date and time when the incident happend.
type	Integer	internal type of the incident.
code	Integer	internal warning or error code.
text	Varchar	explanatory text.

- **archivesAndSchemas**
  Stores which metadata schemas belong to which archives.

archiveId	char(50)	the identifier of the archive.
schema	char(50)	the name of the schema.

- **archivesAndLanguages**
  Stores which languages are available in which archives.

archiveId	char(50)	the identifier of the archive.
language	char(50)	the name of the language.

- **archivesAndKeywords**
  Stores keywords describing the archives.

archiveId	char(50)	the identifier of the archive.
keyword	char(50)	the keyword.

- **archivesAndTemporalCoverage**
  Stores which periods of time are covered by which archives.

archiveId	char(50)	the identifier of the archive.
fromDate	date	start date of the period.
toDate	date	end date of the period.

- **records**
  Stores organizational information about each record.

archiveId	char(50)	the identifier of the archive the record comes from.
recordId	char(200)	the identifier of the record itself (inside the archive).
schema	char(50)	the metadata format of the record (one record can be available in several formats).
firstGathered	timestamp	date and time when the record was gathered for the first time.
deleted	timestamp	date and time when (if) the record was reported deleted.
lastGathered	timestamp	date and time when the record was gathered for the last time.
fullContent	Varchar	the full XML content of the record.

- **valuesDcCreator, valuesDcPublisher, ...**
  For each schema and attribute, a table to store the normalized values of the attribute for the individual records.

archiveId	char(50)	the identifier of the archive the record comes from.
recordId	char(200)	the identifier of the record itself (inside the archive).
normalizedValue	according to type	the normalized value of this attribute in this metadata schema for this record.
lengthInTerms	if applicable	the length in terms of this attribute value in this record

- **indexDcCreator, indexDcPublisher, ...**
  For each schema and attribute, a table to store the indexed terms of the attribute for the individual records.

term	Varchar	the indexed term
archiveId	char(50)	the identifier of the archive the record comes from.
recordId	char(200)	the identifier of the record itself (inside the archive).
weight	Real	the weight of this term concerning this attribute and record.
termFreq	Integer	the number of times this term occurs in this attribute in this record (term frequency).

2.4.3 Information retrieval component

In the following paragraphs, we describe the algorithms and concepts used in the *information retrieval component*.

Harvesting

Harvesting records from open archives implies the following steps:

- get a list of record identifiers
- for each identifier, get the available metadata formats
- for each identifier and format, get the whole record

If a new metadata schema is found, then the schema must be analyzed, the attributes and their types stored in the *schemasAndAttributes* table, and the appropriate value and index tables generated (see above).

Indexing

Indexing is done in two steps:

Identify index terms and frequencies

This part can be performed for records from a specific archive, according to a specific condition (e. g. all records from an incremental harvest). It is possible to run this process e. g. for several archives in parallel.

- get the list of schemas that have attributes for indexing
  (there might be schemas where the system does not know how to index the content)
- for each schema, for each attribute a, for each record r of the schema, if the record contains the attribute a:
  - normalize the attribute content and store it in the appropriate values table
  - extract the terms
  - store the length of the attribute content (in number of terms) *l(r,a)* in the appropriate index table
  - for each term$t$, store the number of times it occurs in this attribute a in this record r (*tf(t,r,a)*) in the appropriate index table
- for each record r and term r , calculate the number of times the term occurs in any attribute value of the record (*tf(t,r)*)
- for each record r, calculate its length *l(r)*) in number of terms

Calculate term weights

This part of the indexing process is done for all terms and records in an archive.

The frequencies *tf(t,r,a)*, *tf(t,r)*, and the lengths *l(r,a)*, *l(r)* are calculated for each archive seperately, and only for those records that have changed, as specified in the step above.

Additionally, for calculating the term weights, the following values have to be determined first, for the whole archive:

- for each metadata attribute a to be indexed for the archive, the average attribute value length (*avgl(a)*) in number of terms
- the average record length (*avgl*) in number of terms
- the number of terms in the archive (N)
- for each term t and attribute a, the number of records in the archive in which the term t occurs in the attribute a (*df(t,a)*)
- for each term t, the number of records in the archive in which the term t occurs anywhere in the respective record (*df(t)*)

Then, each term t is weighted as follows:

- With respect to one single metadata attribute a and record r:

$$weight(t,r,a) = (1 - \frac{logdf(t,a)}{logN}) \cdot \frac{tf(t,r,a)}{tf(t,r,a) + 0.5 + 1.5 \cdot \frac{l(r,a)}{avgl(a)}} \tag{2.1}$$

- With respect to one single metadata attribute and archive, but for all records in the archive: This is the average weight of t over all the records in the archive, i. e. the average of *weight(t,r,a)* over all r in the archive.
- With respect to one single record, for all metadata attributes:
  This is the same formula as 2.1, but using frequencies and lengths with respect to the whole record content.

$$weight(t,r) = (1 - \frac{logdf(t)}{logN}) \cdot \frac{tf(t,r)}{tf(t,r) + 0.5 + 1.5 \cdot \frac{l(r)}{avgl}} \tag{2.2}$$

2.4.4 Query language

For the specification of search criteria and for evaluating collection filtering conditions, the Access Service will use the following abstract query syntax:

```
query           = (collectionQuery)*
collectionQuery = (condition* (,archive)*)
condition       = ([weight,] field, fieldCondition+)
fieldCondition  = ([subfield,] predicate, value)
weight          = "+"| "-" | 1..1000
field           = schemaName":"attributeName
```

In the list of collectionQueries, no archive may appear more than once. CollectionQueries are implicitly combined by OR. A weight of "+" means that the condition must be fulfilled, a weight of "-" means that the corresponding conditions must not be fulfilled (this corresponds to the boolean NOT).

For collection filtering conditions, only Dublin Core fields may be specified.

Predicates that will be available:

- lt (less than)
- gt (greater than)
- le (less or equal)
- ge (greater or equal)
- eq (strictly equal)
- $soundex$ (sounding like)
- cw (contains word)

2.5 User interface

The Access Service provides a user interface for archive management. This interface consists of generated HTML pages and is accessible only to those users who are intitled to register archives (*archive administrators*). The rights are managed and checked by the Mediator Service which will call the Access Service's archive management interface only if a user has the appropriate rights.

First of all, the user chooses between two possible actions:

- register a new archive
- edit an archive

2.5.1 Register a new archive

After choosing the action of registering a new archive, the system asks the user for the archive's URL. That means that the corresponding form will contain an input field for the URL, and two buttons, *Submit* and *Clear*. The *Submit* button will submit the URL to the system and bring the user to the next step of the archive registration process, the *Clear* button will clear the input field so that the user can enter another URL. A *Cancel* button is not needed, as the interface is an HTML form which the user can leave at any time desired without further consequences.

Once the user has submitted the URL, the system collects that part of the archive information which is available from the archive itself and presents it to the user. Editing the archive information is described in the following subsection, and the user interfaces for this step are identical for new and existing archives.

After having edited and submitted the archive information, the system presents a success message or a failure report. The HTML page generated for this purpose contains additionally two buttons to enter the archive management process again, namely *register another archive* and *edit an archive*.

2.5.2 Edit an archive

After choosing this action, the user is presented with a list of archives she is entitled to manage. If the user is an *archive administrator*, this is a list of all archives registered by this user. If the user is a CYCLADES*administrator*, this is a list of all archives registered in the system.

The user highlights one archive on the list and then chooses between two actions:

- edit registration information
- unregister

This page of the user interface also contains a *Back* button that takes the user back to the first page of the archive management interface.

Edit registration information

Here, the user can see and edit all the information the system keeps about an archive. In particular, this includes

- text input fields for
  - the archive name
  - the main URL
  - a textual description
- lists of
  - metadata schemas indexed
  - additional URLs (mirrors)
  - languages used in the archive
  - periods of time covered by the archive content
  - keywords describing the archive content
- for each list, buttons to
  - add an entry

  - remove an entry
  - edit an entry

- a *Submit* button
- an *Reset* button
- a *Back* button

The text fields can be edited as required.

Lists are manipulated via the buttons *Add*, *Remove*, and *Edit*. Choosing *Add* or *Edit* results in a new form being generated where the user can specify a new entry for the list, or edit the information about an existing entry. *Remove* opens a confirmation dialog where the user can confirm or cancel the remove action.

The *Submit* button results in the changes being saved and then takes the user back to the list of archives she is entitled to manage (see above).

The *Reset* button reloads the current page with the previous values. All changes are lost.

The *Back* button takes the user back to the list of archives she is entitled to manage, but without saving the changes.

Unregister

When the user chooses this action for one archive in the list of archives, the system opens a confirmation dialog to ask whether the archive should really be unregistered. On confirmation, the user is presented with a success or failure message and a *Back* button that takes her back to the list of archives. On cancel, the user gets back to the list of archives still containing the selected archive.

2.6 Service interaction diagrams

2.7 Service implementation tools

The first prototype of the Access Service will be implemented using the following software:

- Java 2
- Perl 5.6.1 for Linux
- Apache XML-RPC
- Apache Web Server and Tomcat
- Postgres 7.1.2 for Linux

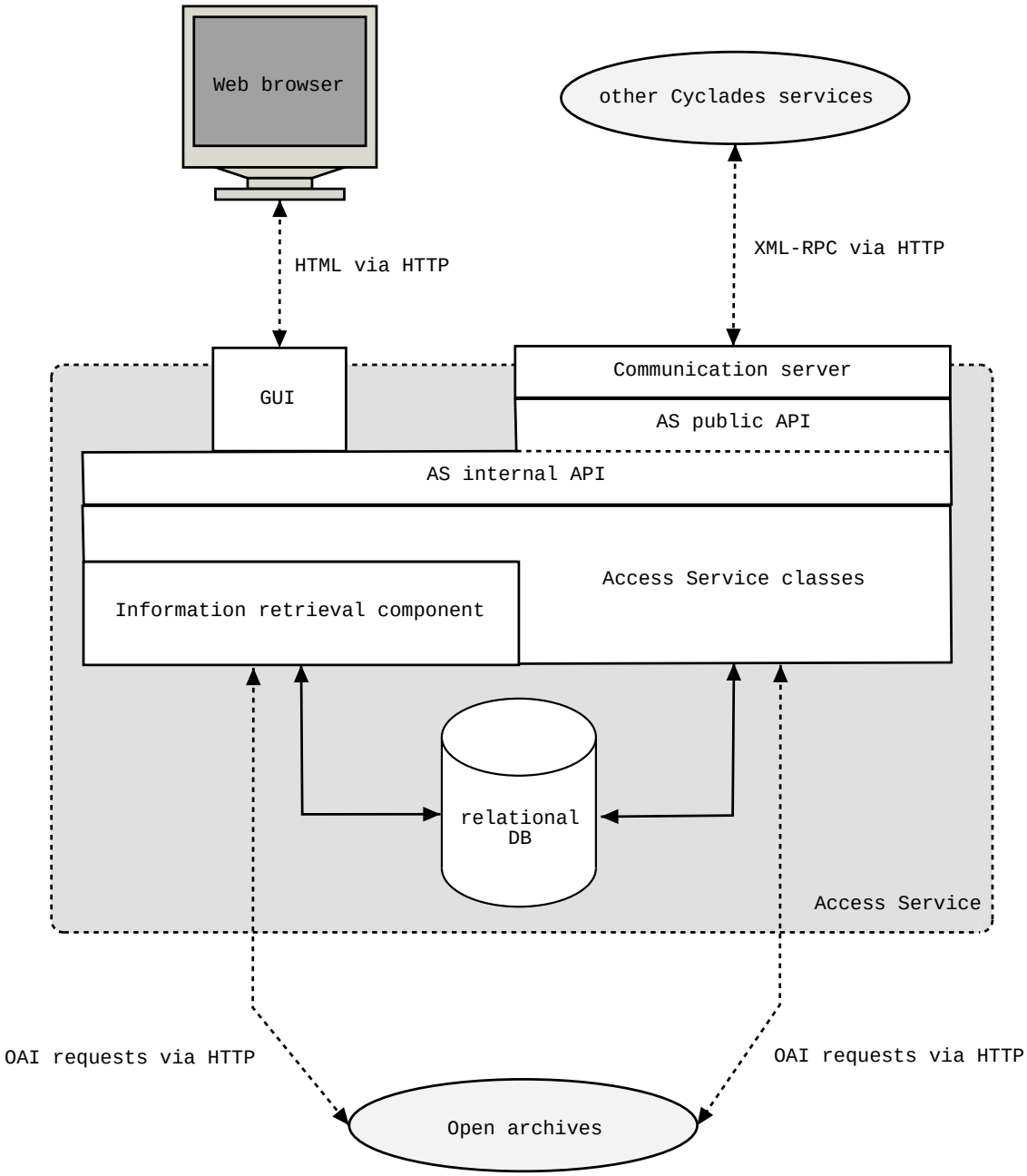

Figure 2.1: Access Service: internal architecture

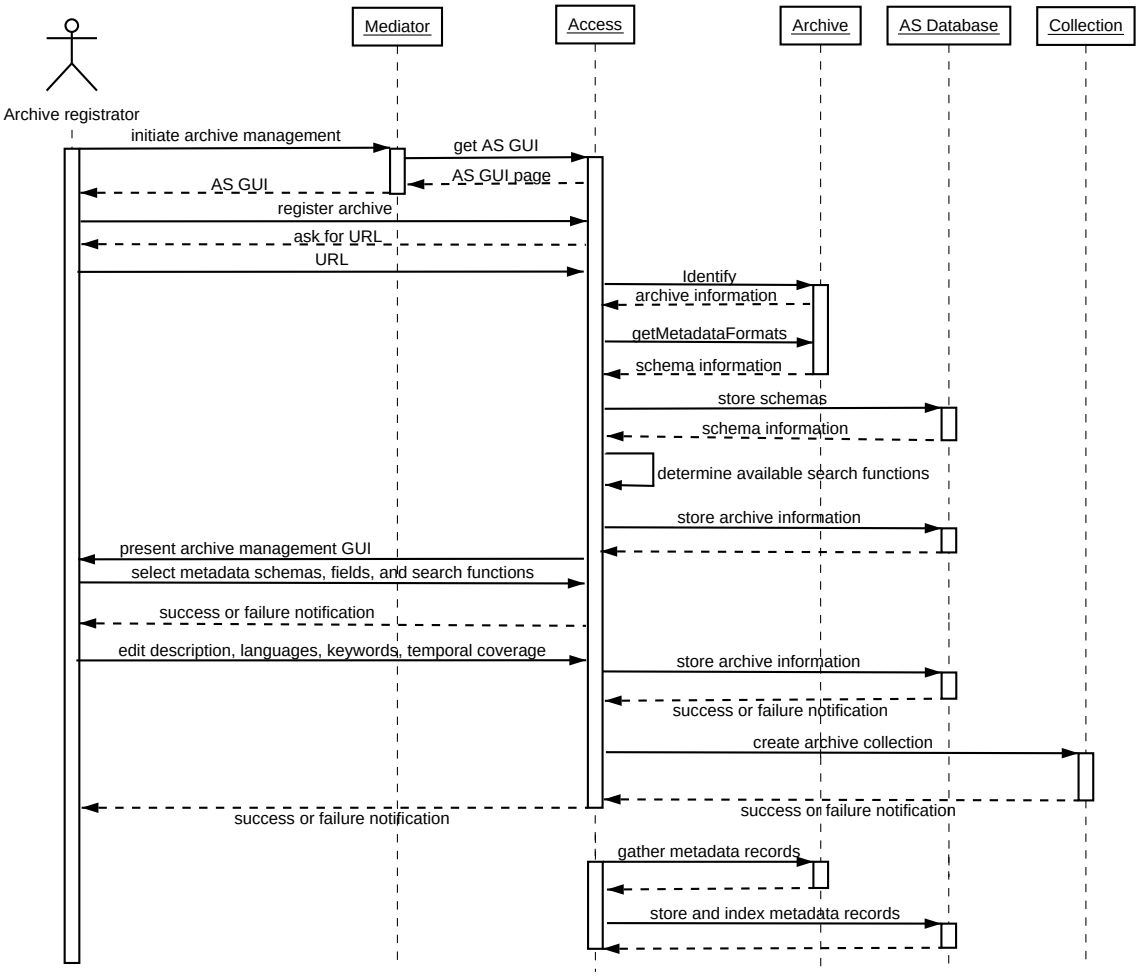

Figure 2.2: Access Service: register archive (interaction diagram)

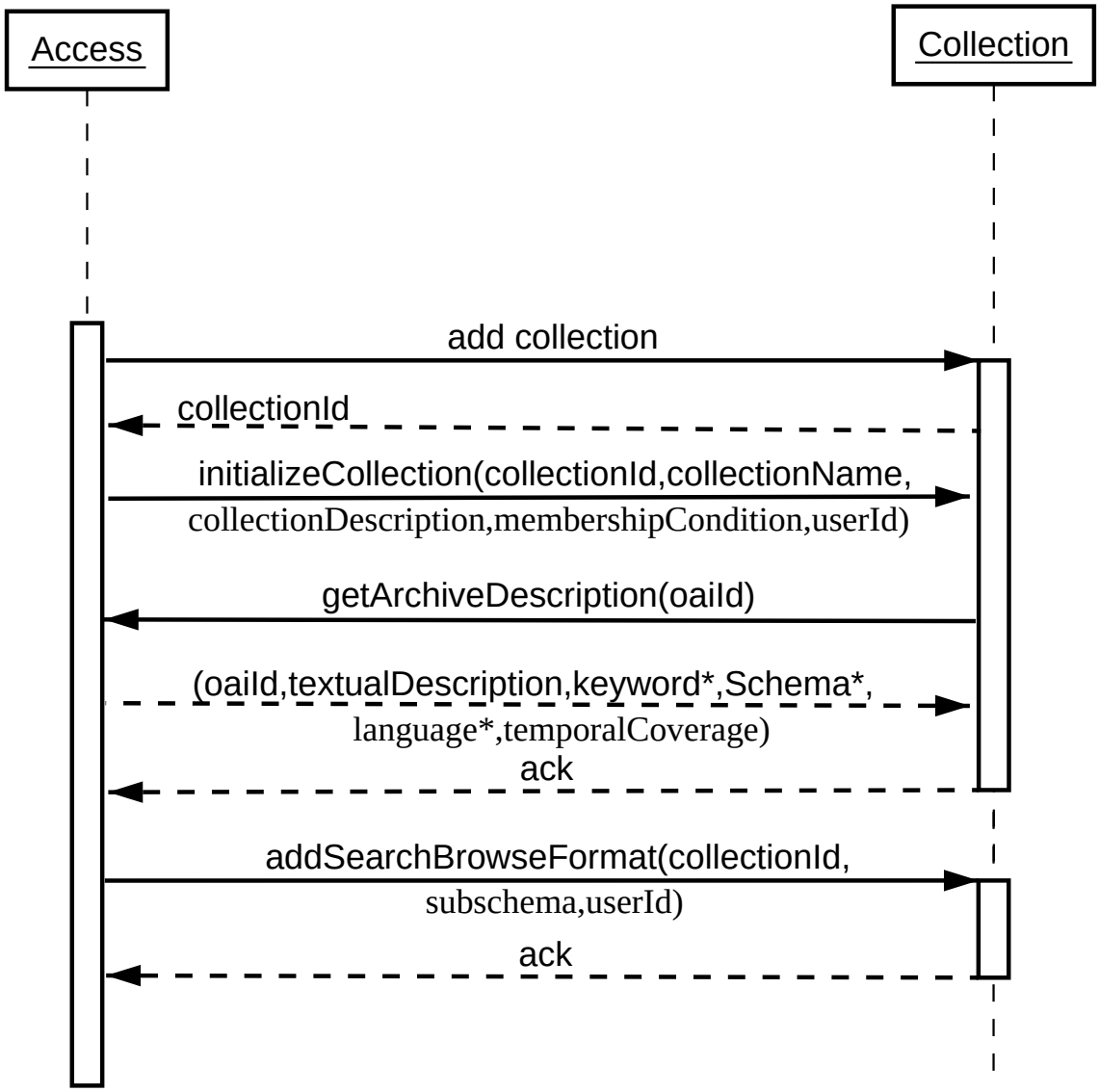

Figure 2.3: Access Service: create archive collection (interaction diagram)

Chapter 3

Collaborative Work Service

3.1 Functionality

The Collaborative Work Service (CWS) stores the private, community and project folders of the Cyclades users. These folders may contain metadata records, queries and, in the case of project folders, also other material. The CWS supports

- folder and contents management,
- collaboration between users by way of folder sharing in communities and projects, including discussion forums and mechanisms fostering mutual awareness, and
- recommendations management.

3.1.1 Folder and contents management

Folders are the persistent repositories for the storage of metadata records and queries. Folders may contain other folders and thus form a folder hierarchy. Cyclades folders come in three flavours: private, community and project folders.

Private folders are for the use of one user only. Community folders are for sharing records and queries with other users and for building up a common folder hierarchy. Community folders may also contain discussion forums where notes may be exchanged in threaded discussions (similar to news groups). All folders of a community are shared by all members of the community.

Project folders are for carrying out shared work connected somehow to metadata records, e. g. producing a review of literature summarized in metadata records, or writing a scientific paper including the research of related literature. Consequently, project folders may contain also documents other than records and queries, and also allow a flexible membership management: project folders may contain subfolders that are only accessible for a subset or superset of the members of the parent folder.

Cyclades folders have a set of attributes intended for the support of Cyclades specific activity: recommendations management, collaboration, and searching and browsing. These attributes are: recommendation preferences, subscription policy preference (community folders only), and associated collections (collections consist of a set of metadata records satisfying a certain criterion; a user has a personal set of preferred collections).

Folder management is concerned with

- creating folders and subfolders,

- moving and copying folders from one location to another, deleting, undeleting and destroying folders,
- editing folder attributes like name, description, associated collections, recommendation preferences, and subscription policy preference (community folders only),
- keeping the personal sets of preferred collections, which are managed in the Collection Service (CS), up to date.

Contents management is concerned with

- saving records and queries from a search and browse session in folders,
- deleting, undeleting and destroying records and queries,
- moving and copying records and queries from one folder to another,
- rating and annotating records and queries,
- management (create, delete, move, copy, rate annotate) of other documents (project folders only).

3.1.2 Collaboration support

Collaboration between users is supported through the possibility of sharing community and project folders along with their contents and folder structure. Additionally, discussion forums may be created within folders to allow informal exchange of notes and arguments. Annotations of specific records and queries also may take the form of discussions among the members of a community or project. In order not to miss shared activity in the CWS, mutual awareness is supported through event icons displayed in the CWS and activity reports that are daily received by email. Also, users may view the list of all existing communities so that they become aware of ongoing community activity. This does not mean that they can look inside communities, only title, description and identity of the community managers are available. To become a member, users may directly join the community if this is allowed by the community's policy, or may contact the managers to be invited to the community.

Collaboration support is concerned with

- inviting or removing members to or from a folder,
- assigning members with the manager role,
- leaving a community or project,
- viewing communities,
- joining a community folder (only for folders open to subscription),
- contacting community managers or other users via email,
- creating discussion forums, adding notes to a discussion forum,
- editing event notification preferences (icons, daily report),
- catching up on events noticed,

and as a basic service

- assigning user identifiers and password management.

3.1.3 Recommendations management

Recommendations in CYCLADES are generated by the Filtering and Recommendation Service (FRS) and then forwarded to the CWS. Recommendations are meant for a specific folder and come in four flavours: the recommendation of records, users, communities and collections. The records are meant to fit into the folder context, the users and communities are recommended as potential partners with similar interests, and the recommended collections should cover material interesting for the folder concerned. Recommendations for a folder are put into a specific recommendations subfolder where the recommendations may be further processed depending on the type of the recommendation. Users may configure their preferences about receiving recommendations on a per folder basis, and may also specify whether they want to be recommended as a user to other users.

Folder profiles, which depend on the records contained in a folder, determine the recommendations received for a folder. Folder profiles are updated by the FRS from time to time. When a user has changed the contents of a folder to a great deal, she may also request an immediate update of the folder profile by the FRS. This is also useful before a search and browse session is started, since then the filtered search mode will work with the updated folder profile.

Recommendation management is concerned with

- editing folder recommendation preferences,
- editing the personal preference for allowing recommendations as user to other users,
- receiving recommendations from the FRS,
- processing recommendations,
- requesting an update of the folder profile.

3.2 Process flow

The functionality of the CWS is accessible through its graphical user interface and through its service method interface (API).

User operation interface The graphical user interface of the CWS in its normal state shows the contents of the current folder as a linear list and has a number of functions that may be selected by the user. The current folder is the current location of the user in the hierarchy of folders that is accessible to her.

The functions of the user interface are invoked by selecting options of pull-down or pop-up menus or by clicking buttons. Function invocation is the starting point of an interaction between user and system. Navigation in the folder hierarchy is an example of a simple interaction: a subfolder entry in the current folder listing works like a button: by hitting such an entry the corresponding subfolder is made the current folder and its contents are shown as a listing of entries. Simple interactions consist only of a one-step process of user and system action:

- **User:** hits a function.
- **System:** executes the function and presents the current folder listing.

The more complex interactions may involve more steps, typically two:

- **User:** hits a function.
- **System:** presents a form for entering additional information.

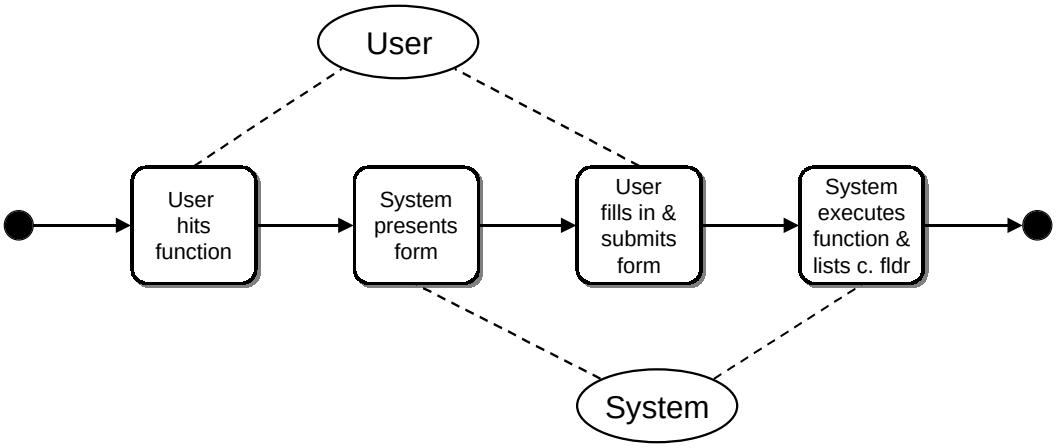

Figure 3.1: User operation activity diagram

- **User:** fills in and submits the form.
- **System:** executes the function and presents the current folder listing, reflecting the changes if any.

In the following, we will use this schema for describing the linear process flow of the CWS user operations.

3.2.1 Create folder

When creating folders we distinguish two cases: the user creates the root of a new folder hierarchy in her home folder, or the user creates a subfolder within in an existing folder hierarchy. In the first case, a number of preferences have to be set which in the second case are inherited from the parent folder.

Create root folder

- **User:** hits function New community/project/private folder in her home folder.
- **System:** presents a form for entering
  - folder name,
  - folder description,
  - collections to be associated to the folder (from the personal set of collections),

  - recommendation preferences (which of the four varieties of recommendations—records, users, communities, collections—are welcome to the folder),
  - subscription policy preference (open to subscriptions from outside or not—community folders only).

- **User:** enters this information and submits the form.
- **System:** creates the new folder and forwards the recommendation preferences to the Filtering and Recommendation Service (FRS). For every associated collection, the new folder is added to this collection's list of folders that have the collection associated—these lists are needed when a collection is removed from the system. In the case of community folders, the new community is added to the list of existing communities. The current folder listing is presented showing the new folder.

Create subfolder

- **User:** hits function New community/project/private folder in some existing community, project, or private folder.
- **System:** presents a form for entering
  - folder name,
  - folder description.
- **User:** enters this information and submits the form.
- **System:** creates the new folder and forwards the recommendation preferences, which are inherited from the parent folder, to the Filtering and Recommendation Service (FRS). For every associated collection which are also inherited from the parent folder, the new folder is added to this collection's list of folders that have the collection associated—these lists are needed when a collection is removed from the system. The current folder listing is presented showing the new folder.

3.2.2 Move and copy

Moving and copying objects within the CWS from one folder to another are generic operations available for folders, records, queries, discussion forums, and any other type of documents. The mechanism works with a container called clipboard that every user has apart from her home folder. More than one object of different type may be moved or copied together.

Moving and copying is a three-step process.

- **User:** selects the entries in the current folder listing to be moved or copied and then hits the function Cut (for move) or Copy (for copy).
- **System:** cuts or copies the selected entries to the clipboard and presents the (changed) folder listing.
- **User:** navigates to the destination folder (this may involve several substeps).
- **System:** presents destination folder.
- **User:** hits the Paste function.
- **System:** moves the selected entries from the clipboard into the current folder and presents the changed folder listing.

Since this operation is generic, it will not be repeated further down for records, queries, discussion forums, or other documents.

3.2.3 Delete, undelete and destroy

Again, deleting, undeleting and destroying objects are generic operations within the CWS available for folders, records, queries, discussion forums, and any other type of documents. The mechanism works with a container called waste that every user has apart from her home folder and clipboard. Several objects of different type may be deleted, undeleted or destroyed together.

Delete

- **User:** selects the objects to be deleted and then hits the Delete function.
- **System:** moves the selected objects to the waste and presents the changed folder listing. Note the following:
  - Deleted objects are still available and may be undeleted again. In their current location, the waste, they provide only limited functionality to the user (essentially Undelete and Destroy).
  - In shared folders, deleting an object deletes it for all members.
  - With some classes, deleting is destroying, i. e. objects of these classes are not moved to the waste when deleted, but are removed from the system (c.f. operation Destroy below). This is the case for recommended users, communities and collections.
  - Deleting a folder deletes all its contents, too.

Undelete

- **User:** hits the Waste function.
- **System:** presents the waste listing, i. e. entries currently contained in the waste.
- **User:** selects the objects to be undeleted and then hits the Undelete function.
- **System:** moves the selected objects to the their original folders and presents the changed waste listing. Note that
  - in shared folders, undeleting an object undeletes it for all members.

Destroy

- **User:** hits the Waste function.
- **System:** presents the waste listing, i. e. entries currently contained in the waste.
- **User:** selects the objects to be destroyed and then hits the Destroy function.
- **System:** removes the selected objects from the system and presents the changed waste listing. Note the following:
  - Destroying a shared folder terminates the membership of the destroyer, but does not destroy the folder, unless the destroyer is the last member.
  - In shared folders, destroying an object other than a subfolder destroys it for all members. The destroyed object is moved to its owner's waste, unless the destroyer is the owner. The reason behind that is to give the owner who originally created the object a final opportunity to dispose of the object.
  - When a community root folder is destroyed by its last member, it is removed from the list of existing communities. This is not already the case when a community root folder is deleted by its last member.
  - Destroying a folder destroys all its contents, too.

3.2.4 Edit folder attributes

Folder attributes that may be edited by the user are

- name,
- description,
- associated collections,
- recommendation preferences, and
- subscription policy preference (community folders only).

Edit name

This operation is generic for all objects in the CWS.

- **User:** hits the function Rename for an entry in the current folder listing.
- **System:** presents a form where the new name may be entered.
- **User:** fills in and submits the form.
- **System:** presents the current folder listing showing the new name.

Edit description

This operation is generic for all objects in the CWS.

- **User:** hits the function Description for an entry in the current folder listing.
- **System:** presents a form where the new description may be entered.
- **User:** fills in and submits the form.
- **System:** presents the current folder listing showing the new description.

Edit associated collections

This operation on a folder is split into two: add collections and remove collections.

Add collections

- **User:** hits the Add collections function for a folder.
- **System:** presents a form showing those collections of the user's personal set of collections that are currently not associated to the folder.
- **User:** selects the collections from this list that she wants additionally to associate to the folder and submits the form.
- **System:** associates the selected collections to the folder. For each collection that is associated, the folder is entered in this collection's list of folders that are associated to it. The current folder listing is presented. Note that
  - the user may check which collections are currently associated to a folder by having a look at the folder's info page (function Info).

Remove collections

- **User:** hits the Remove collections function for a folder.
- **System:** presents a form showing the collections currently associated to the folder.
- **User:** selects the collections from this list that she wants to remove from this list and submits the form.
- **System:** removes the selected collections from the list of collections associated to the folder. For each removed collection, the folder is removed from this collection's list of folders that are associated to it. The current folder listing is presented. Note that
  - the user may check which collections are currently associated to a folder by having a look at the folder's info page (function Info).

Edit recommendation and subscription policy preferences

- **User:** hits the Edit folder preferences function for a folder.
- **System:** presents a form showing which type of recommendations are currently requested for the folder. If the folder is a community root folder, the form also shows the preference for the community's subscription policy (open to subscription from outside or not).
- **User:** sets or resets the preferences for recommendations and subscription policy, the latter if applicable, and submits the form.
- **System:** changes the folder preferences, forwards the changed recommendation preferences to the FRS, and presents the current folder listing. Note that
  - the user may check which preferences are currently set for a folder by having a look at the folder's info page (function Info).

3.2.5 Rate records

- **User:** selects one or more records she wishes to rate and hits the Rate function.
- **System:** presents a form with a rating table: the rows correspond to the records to be rated, the columns to the available rating values, i. e. `very poor`, `poor`, `fair`, `good`, `excellent`. There is an additional column `Not yet rated` which is initially checked if the user has not yet rated that particular record. Otherwise, the option corresponding to the last rating of that particular record by the user is checked.
- **User:** enters ratings by checking the appropriate entries in the rating table and submits the form.
- **System:** stores the ratings and forwards them to the Rating Management Service (RMS). The system presents the current folder listing showing an iconized aggregate rating for every rated record. Note that
  - the user may check which single ratings a record has received by having a look at the record's info page (function Info).

Rating is a generic operation within the CWS. Apart from the forwarding of record ratings to the RMS, the operation is equal for queries, recommendations, or other documents and will not be described again below. Also, mixed types of objects may be rated together in one operation. In this case, only record ratings are forwarded to the RMS.

3.2.6 Annotate records

Annotations of records take the form of threaded discussions (like the discussion forums treated below). The first annotation opens the discussion, subsequent annotations are added to the discussion.

Attach a first annotation to a record

- **User:** hits the Attach note function for a record.
- **System:** presents a form to enter a note as annotation of the record. A note has a type (selectable are `Note`, `Pro`, `Con`, `Important`, `Angry`, `Idea`), a subject line and the free text annotation as contents.
- **User:** fills in and submits the form.
- **System:** stores the annotation and presents the current folder listing. The presence of an annotation is indicated by a respective icon attached to the record. Hitting this icon will show the annotation.

Attach more annotations to a record

- **User:** hits the annotation icon of a record.
- **System:** presents a view of the discussion showing all existing notes organized in threads.
- **User:** chooses between
  - adding a note as a comment to an existing note by hitting its Reply function, or
  - opening a new thread by hitting the Attach note function of the whole discussion (top level function).
- **System:** presents in either case a form to enter a new note.
- **User:** enters a new note and submits the form.
- **System:** stores the new annotation either as a comment to the note chosen or as initial note of a new thread, and presents the new state of the annotation.

Annotating is a generic operation within the CWS. The operation is equal for queries, recommendations and other documents, and will not be described again below.

3.2.7 Invite a new member to a folder

The right to invite new members is restricted to managers. Invitation to communities is only possible in the community root folder and holds for all folders of the community. With project folders, new members may also be invited to subfolders of a project and then have no access to the parent folder of the folder they were invited to.

- **Manager:** hits the Add member function of a folder.
- **System:** presents a form where to select new members from the list of members known to the manager (which are in her address book) and to specify an optional invitation message.
- **Manager:** fills in and submits the form. If some or all of the persons the manager wishes to invite are not listed, she may also proceed to a form (function Add to address book) where she may enter user names and/or email addresses of these persons which are then added to her address book and invited.

- **System:** adds all invitees who are already registered users as members to the folder. For non-registered invitees, i. e. where only a email address has been specified, the email address is forwarded to the Mediator Service for registration and invitation of these persons; an email address object is created and added as placeholder member to the folder. Finally, the system presents the current folder listing.

3.2.8 Remove a member from a folder

The right to remove members is restricted to managers. Removal from communities is only possible in the community root folder and holds for all folders of the community. With project folders, members may also be removed from subfolders of a project and then have no access to the subfolders of this folder.

- **Manager:** hits the members icon attached to the folder in the `your location` field.
- **System:** presents a list of the present members of the folder like in a folder listing. Members shown only with their email address are invited members who have not yet registered (placeholders).
- **Manager:** selects the users she wants to remove and hits the Remove function.
- **System:** removes these members from the folder having the effect that these users have no longer access to the folder and its subfolders. The system presents the remaining members of the current folder.

3.2.9 Assign a member as manager

The right to assign members with the manager role is restricted to managers.

- **Manager:** hits on the members icon attached to the folder in the `your location` field.
- **System:** presents a list of the present members of the folder. Members shown only with their email address are invited members who have not yet registered (placeholders).
- **Manager:** selects the Assign role function of that user to whom she wants to assign the manager role.
- **System:** presents a form showing the possible roles for that user with the current role checked.
- **Manager:** checks the manager role and submits the form.
- **System:** makes the selected user a manager of the folder and all its subfolders, and presents the folder members listing. Note that
  - the manager may check the current assignment of roles and the functions that are accessible to those roles by having a look at this folder's info page (function Info).

3.2.10 Leave a community or project

Leaving a community or project is done by destroying the respective root folder from the user's home folder. With projects, one can also leave parts of the project, i. e. part hierarchies of the project folder hierarchy.

3.2.11 View communities

This operation allows users to become aware of the existing communities.

- **User:** hits the Communities function.
- **System:** presents the listing of the folder `Communities` which has as entries all existing communities which are shown with name and description. Note the following:
  - The `Communities` folder does not belong to the user's folder hierarchy starting at her home folder; to navigate back, the user hits the (iconized) Home function which is always present.
  - The managers of a community are shown on this community's info page (function Info).

3.2.12 Join a community

This is an operation that is only possible for community objects as listed in the `Communities` folder and for community recommendations as listed in the `Recommendations` folder of some folder. In both cases, joining is only possible if the community's policy is open to subscriptions from outside, and if the user is not already member.

- **User:** hits the Join community function of the community she wants to join.
- **System:** makes the user an ordinary member of the community and presents the current folder (`Communities` or `Recommendations`).

3.2.13 Contact community managers

This is an operation that is only possible for community objects as listed in the `Communities` folder and for community recommendations as listed in the `Recommendations` folder of some folder. This operation is only possible for users not being managers of the community.

- **User:** hits the Mail community managers function of the community she wants to contact.
- **System:** presents a form where to enter a message. The addresses of the community managers have already been filled in the `To`-field.
- **User:** fills in and submits the form.
- **System:** sends the message via email to the community managers and presents the current folder (`Communities` or `Recommendations`).

3.2.14 Create discussion forums

- **User:** hits the Add discussion function for adding a discussion forum to the current folder.
- **System:** presents a form where to enter the name of the discussion, and type, subject and contents of the first note of this discussion. This first note is to become the first top level thread of the discussion.
- **User:** fills in and submits the form.
- **System:** creates a new discussion and presents the current folder listing showing the new discussion as a separate entry.

3.2.15 Add notes to a discussion forum

- **User:** hits on the name or icon representing the discussion she wants to add a note to.
- **System:** presents a view of the discussion showing all existing notes organized in threads.
- **User:** chooses between adding a note as a comment to an existing note by hitting its Reply function or opening a new thread by hitting the Attach note function of the whole discussion (top level function).
- **System:** in either case presents a form where to enter a new note.
- **User:** fills in and submits the form.
- **System:** adds the new note to the discussion as a comment to the note chosen or as initial note of a new thread and presents the current state of the discussion.

3.2.16 Edit event notification preferences

Awareness configuration in CYCLADES is on a per object basis. The default awareness configuration is valid for all folders of a user and their contents. This default configuration may be overridden for particular objects concerning the event icons.

Edit default event notification preferences

- **User:** hits the Default events function.
- **System:** presents a table representing the user's current settings of default event notification preferences. The rows of the table correspond to the activation status of the awareness mechanisms and the event types (`read`, `create`, `move`, `change`) and the columns to the awareness mechanisms (`icons`, `daily reports`, `immediate email`).
- **User:** configures the personal default preferences by setting or resetting the entries in the event notification table and submits the form.
- **System:** stores and activates the new preferences and presents the current folder listing.

Edit object event notification preferences

- **User:** hits the Events function of the object she wants to override the default settings for.
- **System:** presents a table representing the user's current settings of default event notification preferences.
- **User:** configures the specific object's preferences by setting or resetting the entries in the event notification table and submits the form.
- **System:** stores and activates the new object preferences and presents current the folder listing.

3.2.17 Catching up on events

- **User:** selects one or more entries (folders, records, queries, discussions, other documents) she wants to catch up on and hits the Catch up function. This means that she acknowledges to have seen the event icons and wants these icons to go away, so that the screen is less cluttered and new event icons become more prominent.

- **System:** presents the current folder listing; the event icons of the entries that the user has selected in the first step disappear.

Note that the catch up action is on a per user basis. Other users' view of the same folder may be completely different with regard to event icons. The appearance also depends on the event notification preferences of a user: some users may have chosen to receive no event icons at all.

3.2.18 Edit personal preferences

Personal preferences have to do with email formats, known editors, user profile, user interface language etc. For CYCLADES, an attribute has been added to the personal preferences that states whether the user allows recommendation of herself to other users.

- **User:** hits the Preferences function.
- **System:** presents a form showing the current setting of the user's personal preferences.
- **User:** sets or resets the option `Allow oneself's recommendation to other users` and submits the form.
- **System:** stores and activates the new settings and presents the current folder listing.

3.2.19 Processing recommendations

Recommendations are received from the FRS and put into the `Recommendations` subfolder of the folder for which the recommendations are meant.

Processing recommendations includes all operations that dispose of received recommendations. The type of the operations possible depends on the type of the recommendations (records, users, communities, collections). Deleting is always possible (and means destroying for user, community and collection recommendations). Moving and copying is only possible for recommended records. Joining and contacting managers is only possible for recommended communities. These operations have been described above in other contexts and are nor repeated here.

For user and collection recommendations, there are two additional operations:

- Invite (a user) and
- Associate (a collection).

Invite to folder

- **User:** hits the Invite function of a user recommendation in the `Recommendations` subfolder of some folder.
- **System:** adds the recommended user as ordinary member to the folder containing the `Recommendations` folder and presents the `Recommendations` folder listing again.

Associate to folder

- **User:** hits the Associate function of a collection recommendation in the `Recommendations` subfolder of some folder.
- **System:** associates the recommended collection to the folder containing the `Recommendations` folder, adds this folder to the collection's list of folders that have it associated, and presents the `Recommendations` folder listing again.

3.2.20 Update folder profile

This operation is useful before searching and browsing when folder contents have been changed a great deal.

- **User:** hits the Update profile function of a folder.
- **System:** forwards this request to the FRS and presents the (unchanged) current folder listing. Note that
  - the user may check when a folder profile update was last requested by taking a look at the folder's history (function History).

Service method interface The service method interface of the CWS is an API that allows other CYCLADES services to interact with the CWS by way of remote procedure calls. Every method has a well-defined signature giving its name, parameters and return value. In the following, we describe the invocation, execution and result of the CWS methods as a three-step interaction:

- **Method call:** shows method name and required parameters.
- **Execution:** describes CWS execution of method.
- **Return value:** describes result of method returned to the calling service.

In the following, we will use this schema for describing the linear process flow of the CWS API method executions.

3.2.21 Create new user

- **Method call:** createUser(name, password, emailAddress, folderId)
- **Execution:** CWS checks name, password, and email address for validity (names must not contain blanks or at-signs (@), passwords must have a certain length, and email addresses must conform to the Internet standard). Then a user object is created along with home folder, clipboard, and waste. If the folder identifier is not an empty string, the CWS checks whether there is a folder with the given identifier and adds the user as a member to this folder.
- **Return value:** User identifier and the URL for requesting display of the folder with the given identifier or of the user's home folder when the given folder identifier is the empty string.

3.2.22 Update password

- **Method call:** updatePassword(name, password)
- **Execution:** CWS checks whether there is a user with the given name and the validity of the new password. The new password is set for the user.
- **Return value:** Void

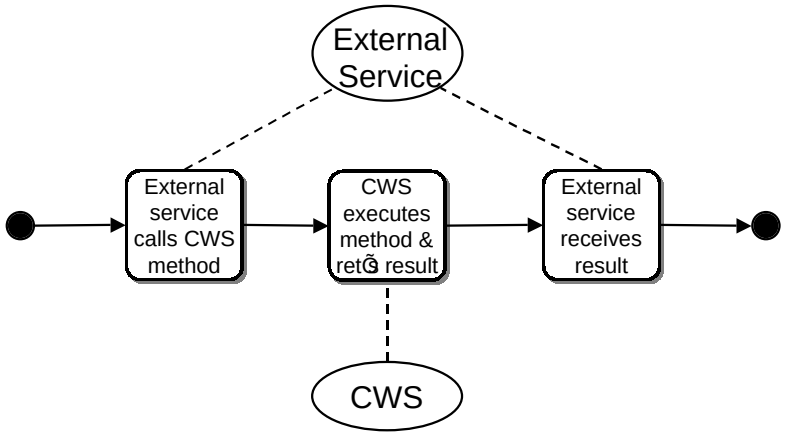

Figure 3.2: API method activity diagram

3.2.23 Get folder information

There are numerous methods for getting information about the folder hierarchies of a user including all folders of a given user and name, description, members, associated collections, records, queries, parent folder and subfolders of a given folder. Since the execution of these methods is rather straightforward, we list these methods with their call and return value.

- **Method call:** getFolders(userId)
  Return value: a list of identifiers of all folders to which the user with the given identifier has access.

- **Method call:** getName(folderId)
  Return value: the name of the folder with the given identifier.

- **Method call:** getDescription(folderId)
  Return value: the description of the folder with the given identifier.

- **Method call:** getMembers(folderId)
  Return value: a list of the user identifiers of all members of the folder with the given identifier.

- **Method call:** getRecords(folderId, timestamp)
  Return value: a list of pairs containing the identifier and classifier label of all records that have been saved, moved or copied into the folder with the given identifier since the time given by the timestamp.

- **Method call:** getQueries(folderId)
  Return value: a list of identifiers of all queries that are contained in the folder with the given identifier.
- **Method call:** getParent(folderId)
  Return value: the identifier of the folder that contains the folder with the given identifier.
- **Method call:** getChildren(folderId)
  Return value: a list of identifiers of all subfolders of the folder with the given identifier.
- **Method call:** getCollections(folderId)
  Return value: a list of identifiers of all collections that are associated to the folder with the given identifier.
- **Method call:** getCommunity(folderId)
  Return value: the identifier of the community to which the folder with the given identifier belongs, or an empty string if the folder does not belong to any community.

3.2.24 Save query

- **Method call:** saveQuery(folderId, userId, query)
- **Execution:** CWS checks whether the user with the given identifier has access to the folder with the given identifier, and then creates a query object corresponding to the query given, which is added as entry to the given folder.
- **Return value:** Void

3.2.25 Save records

- **Method call:** saveResults(folderId, userId, records)
- **Execution:** CWS checks whether the user with the given identifier has access to the folder with the given identifier. For every record in the given records list, a record object is created and added as entry to the given folder.
- **Return value:** Void

3.2.26 Save recommendations

Recommendations come in four flavours: records, users, communities and collections. For every kind of recommendations there is a service method for saving these recommendations. Recommendations are saved in a `Recommendations` subfolder of the folder for which the recommendations are intended. The execution is essentially the same for all kinds of recommendations.

- **Method call:**
  saveRecommendedRecords(folderId, records)
  saveRecommendedUsers(folderId, userIds)
  saveRecommendedCommunities(folderId, communityIds)
  saveRecommendedCollections(folderId, collectionIds)
- **Execution:** CWS checks whether the recommendation preferences of the folder with the given identifier welcome recommendations of the given kind. If this is not the case, false is returned and the execution stops.
  Next, CWS checks whether there is already a `Recommendations` subfolder of the folder with the given identifier. If this is not the case, such a subfolder is created.

For every record in the given records list, a record object is created. If there is already a record entry in the `Recommendations` folder with the same identifier, the new record object replaces the old one; otherwise, the new object is added to the `Recommendations` folder.
For every identifier in the given list of user, community or collection identifiers, it is checked whether there is a user, community, or collection with the identifier given. Then a recommendation object of the requested kind is created. If there is already a recommendation entry in the `Recommendations` folder with the same identifier, the new recommendation object replaces the old one; otherwise, the new object is added to the `Recommendations` folder. After a successful execution, true is returned.

- **Return value:** False, if recommendations of the given kind are not welcome to the folder with the given identifier, true otherwise.

3.2.27 Add or modify collection

This method serves the purpose of notifying the CWS of the creation of a new or the modification of an existing collection.

- **Method call:** addModifyCollection(collectionId, collectionName)
- **Execution:** CWS checks whether it has a collection with the given identifier in its list of collections. If so, its name is changed to the name given in the method call. If not, a new collection object is created in the CWS and added to the list of collections.
- **Return value:** Void

3.2.28 Delete collection

This method serves the purpose of notifying the CWS of the deletion of a collection.

- **Method call:** deleteCollection(collectionId)
- **Execution:** CWS checks whether it has a collection with the given identifier in its list of collections. This collection is then removed from all personal sets of collections and from the lists of associated collections of all folders that have this collection associated. The collection itself is then deleted from the list of collections within the CWS.
- **Return value:** Void

3.2.29 Update personal set of collections

This method serves the purpose of notifying the CWS of the update of a personal set of collections in the CS.

- **Method call:** updatePersonalCollections(userId, collectionIds)
- **Execution:** CWS checks whether there is a user with the given identifier and then replaces this user's personal set of collections by the given list of identifiers. If this list contains identifiers of collections not in the CWS list of collections, the CWS requests the complete list of collections from the Collection Service (CS) and generates the missing collection objects.
- **Return value:** Void

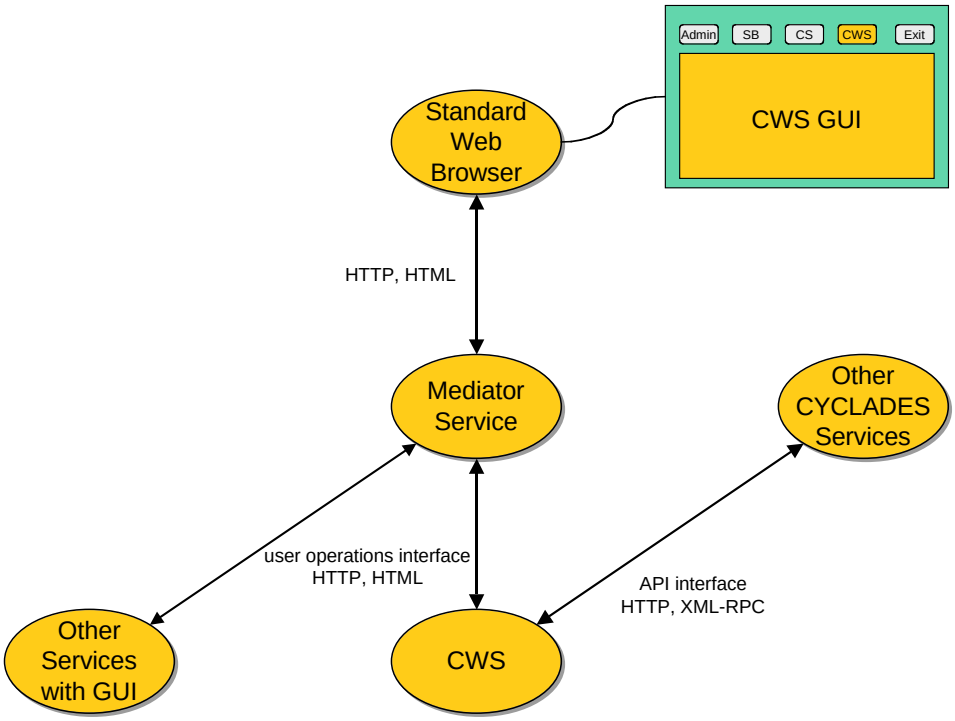

Figure 3.3: Internal architecture of CWS

3.3 Internal architecture

3.3.1 Overview

The CWS provides its functionality via the graphical user interface and via an API to the other Cyclades services (c.f. Figure 3.3). All user interaction via the graphical user interface is mediated by the Mediator Service—that is, both the interaction of the users with the graphical user interface of the CWS and the interaction of the users with the graphical user interfaces of other services. Other services can directly access the CWS functionality via the CWS API. In the case of graphical user interface the communication takes place via HTTP and the format is HTML; in the case of the API the communication takes place via HTTP and the format is XML and XML-RPC.

The CWS consists of a standard Apache Web server that transmits incoming HTTP requests via CGI interfaces to the CWS components that serve on the one hand the user operations and on the other hand the API calls (c.f. Figure 3.4). The CWS is based on the Basic Support for Cooperative Work system (BSCW).

BSCW supports Web-based shared workspaces (called 'folders' in the Cyclades terminology) and provides basic collaboration support that is adapted and extended for Cyclades. The private, project, and community folders will be implemented on top of BSCW shared workspaces. The BSCW kernel provides a modular extension of the World-Wide Web's client-server architecture without requiring modification to Web clients, servers or protocols. The core is a standard Web server extended with the BSCW kernel software, providing both basic shared workspace functionality.

BSCW is extended for implementing the CWS functionality via BSCW's package mechanism (c.f.

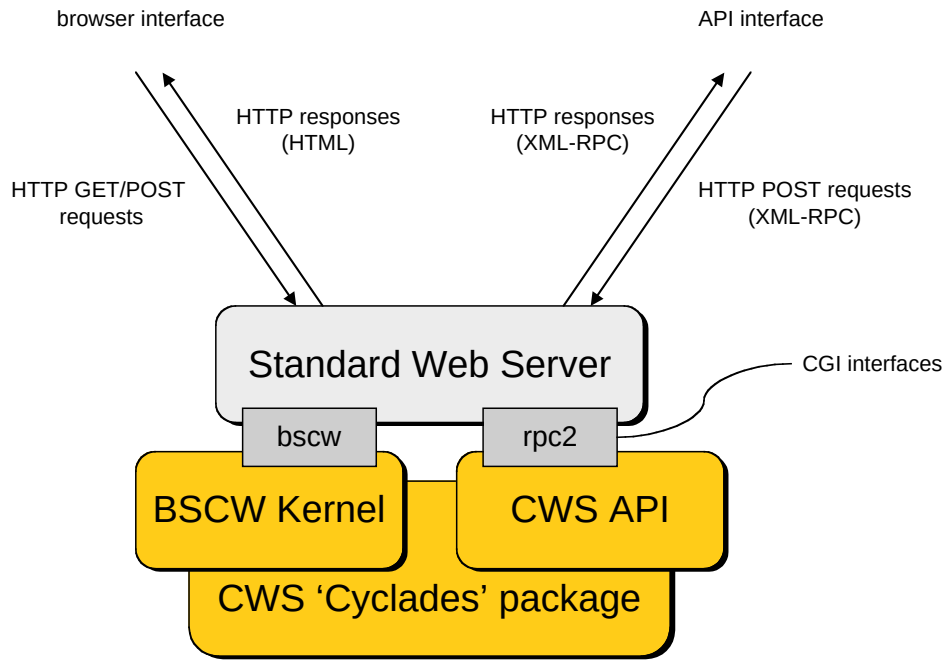

Figure 3.4: CWS components

below). The extensions are contained in the package `Cyclades`. A separate component handles the incoming API calls (translation and dispatch).

At the highest level of abstraction the CWS can be further decomposed into three layers, which deal with user request and API call handling, method and operation handling, and persistent object storage (c.f. Figure 3.5). In the *user request handling layer* the details of the request are formatted as an internal representation called a Request, which is then dispatched to a particular *operation handler*. In the *API call handling layer* the call is translated into internal format, its parameters are checked for correctness, and the call is dispatched to particular *method handler*. Both operation and method handlers implement the functionality requested, such as getting the contents of a community folder, or saving results from a search and browse session. The operation and method handlers interact with the *persistent store* to process the request or call, creating, deleting and modifying objects as necessary, before generating a response. The response is returned to the request or call handling layer for translation into a concrete format suitable for the access method employed. So, for requests via the API the response format is XML-RPC, and for requests via the graphical user interface the format is HTML, which can be displayed in the Web browser.

Each of these three layers provides a well-defined interface, and it is possible to extend the CWS and integrate specialised application services at each level. As well as introducing new request handling components for different methods of access, new operation handlers can be added to provide new functionality or as wrappers around application services, and the persistent store can be accessed to store new kinds of objects without modifying the storage routines themselves.

The BSCW package mechanism The CWS extends the BSCW kernel making use of the BSCW package mechanism. This works as follows. The BSCW kernel may be enhanced by ad-

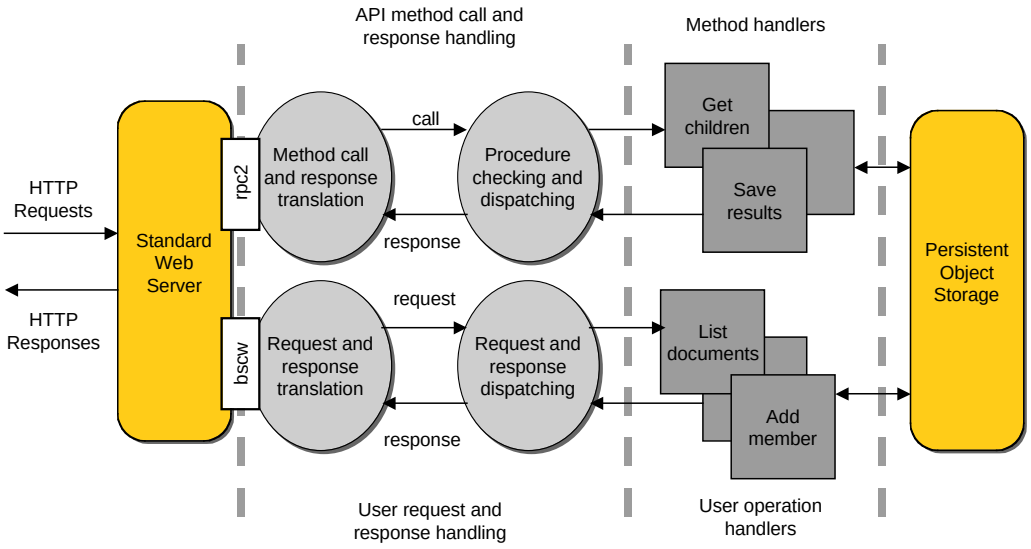

Figure 3.5: CWS server kernel

ditional software packages, which may be added or removed at run-time. The implementation of external packages is organised just like the kernel. It may consist of class definition modules (`cl_*.py`), utility modules (`bs_*.py`), operation handlers (`op_*.py`), interface template files (`*.html, *.txt, *.gif`) and configuration modules (`config*.py`). In order to facilitate the independent development of kernel code and package code, packages are stored in a separate directory, e. g. `packages/Cyclades`. Each package directory itself is structured like the kernel directory, i. e. it contains a subdirectory `src` for the code and a `messages` directory for user interface templates. As Web servers cannot retrieve icons from several icon directories, the icons of a package must be stored in the same directory as the kernel's icons. By calling the function `bs_config.init_packages(some_module)` in a central kernel module, the corresponding package module `some_module` will be imported `after` importing the kernel module `some_module`. This may be used to re-define variables or to add functions and class definitions in a package. This mechanism is particularly useful for the `Action Table`, the global variable containing all actions available.

Currently, the following modules will re-import their counterparts in all packages (if present):

- *cl_action.py* the central module containing all Action objects and the ActionTable,
- *bs_iconconfig.py* user interface icon definitions,
- *bs_config.py* for config.py, the central configuration file,
- *bs_html_ui_config.py* for config_html_ui_config.py, the u/i action configuration file,
- *lg_msgconfig.py* language dependent user interface strings.

3.3.2 User request/response handling

A requirement for CYCLADES is that all user functionality must be accessible from unmodified Web browsers. This requires that all CWS user operations and parameters to these can be composed as requests in standard HTTP, and that all user interface presentations can be composed in standard HTML.

Specifying the request The current standard of the HTTP is supported by all recent Web browsers and is therefore the baseline for specifying requests to the CWS. A request to the kernel is specified using either the HTTP GET request method (when clicking on an HTML anchor or typing a URL into the browser), or the HTTP POST method (when submitting an HTML form). Both GET and POST are treated similarly here for specification purposes. The only difference is that parameters for a GET request are given as part of the URL anchor (separated from the URL information by a '?') while for POST requests they are given as fields in the request data (the MIME types application/x-www-form-urlencoding and multipart/form-data are supported). The format of a typical URL used in GET requests is:

$$\texttt{http} : //\texttt{www.gmd.de/bscw/bscw.cgi/212/619?op} = \texttt{rename\&id} = 532$$

Where `www.gmd.de` is the host address, `bscw` is the script alias, `bscw.cgi` is the CGI request handler, `212/619` is the context information, and `op=rename&id=532` is the operation and parameters. Note that it is not expected that the user will type this URL in a Web browser to access a workspace; rather, this URL is an element of HTTP interacting with the CWS. The CWS provides an HTML user interface which hides the details of URLs behind clickable HTML anchors.

The program to execute is called *CGI request handler*. The role of this script is to format the request details as provided by the server as a Request to be passed to an operation handler. Thus the CGI request handler is the interface between the CWS and the World Wide Web, hiding all Web-specific detail from the other kernel layers.

The *context information* component of the request URL holds object identifiers that establish the context for the request. This context consists of two CWS internal object identifiers which serve different purposes:

- the first one is the previous context, usually an object in the workspace folder hierarchy, which the user can jump back to. This 'placeholder' is a navigation aid, and is useful when the user wants to explore a separate branch of the workspace hierarchy but return to a previous folder easily. For the CWS user interface, this mechanism is required for the clipboard and waste mechanisms.
- the second is the current context, holding the identifier of the folder (or other object) in which the requested operation takes place. For example, for an 'add document' request, this will be the identifier of the folder to which the document should be added.

By storing the context information in the URL in this way it is possible for users to open several browser windows to interact with (different parts of) the shared workspace without requiring details of the context for each browser to be stored by the server. It is also extensible, as new fields can be introduced to store more context information in the URL as necessary.

The final component of the URL specifies the *operation* requested and the *operation parameters*. This information is formatted as a number of name-value pairs in the standard format for a 'query' using the HTTP GET request. For POST requests this information is packaged slightly differently, but is still sent to the server as name-value pairs. The request handler treats the query parameters op and id in a special way: the value of *op* is used to indicate the operation required, while the *id* value(s) identify the object(s) to which the operation should be applied. Other parameters can be supplied if required by the operation (for example, 'new_name' for the operation handler which

renames objects). The parameters can be listed in any order following the '?' query separator. Verification of the parameters is performed by the corresponding operation handler.

For convenience, certain defaults are adopted to simplify the format of the request URL. If no *id* parameter is specified it is assumed that the operation applies to the object identified as the current context. If no *op* parameter is given the default 'get' operation is executed which displays the contents of an object such as a record or folder.

Access to the CWS is controlled by the Mediator Service which has all the information about user names and passwords and handles authentication. The Web server hosting the CWS is configured to accept only requests to the `bscw` script alias from the Mediator Service host.

Building and dispatching the request The Web server receives the URL as part of the HTTP request, and delegates it to the named request handling script, through the script alias mechanism described above. The server copies all the details of the request to the environment variables of the process which the request handling CGI script can then access. The request handling layer must take the request URL (and the input data in the case of POST requests) and construct a Request object for dispatching to the operation handling layer. The generic request handler that a CGI-script calls with a CGI module parameter for parsing the CGI-Environment and generating the reply is shown subsequently:

```
def run(module):
    # Generic Request handler. Call module.parse for setting
    # up the request (e.g. from the CGI-Environment); call the
    # operation handler, and let module.reply generate
    # the reply (e.g. an HTML response)

    # Create a Request object
    request = Request()

    try:
       # Set request attributes
       module.parse(request)

       # Invoke operation handler
       response = request.handle_it()

    # Catch Response and other exceptions
    except Response, error_response:
       response = error_response
    except:
       response = unexpected_response()
    # Generate reply
    module.reply(response)
```

Once a *Request* is constructed and initialised the request handler dispatches the object by a call to the request's *handle_it* method. The operation attribute identifies the operation handler module that should be invoked (c.f. next code fragment). Depending on *request_method* , *handle_it* calls the module's *handle_GET* or *handle_POST* function giving the Request object as parameter. When the operation is finished, the request handler takes the return value of *handle_it* containing the success or error message and passes it on to the reply function of the CGI module. The request handler also catches all exceptions and transforms them into Response objects, which contain debug information in the case of unexpected program errors.

The code fragment below shows the structure of a *Request* object and some of the critical parameters required by operation handlers to service the request. For event logging and access control purposes

it is necessary to know the identity of the user making the request. Hence the User object is recorded in the user attribute. The Request also records the context information, name of the requested operation and any parameters from the request URL.

```
class Request:

    # Instance attributes

    request_method: string
       # HTTP request method, i.e. "GET" or "POST"
    previous_context: Folder
       # First component of context information in URL
    object: Artifact
       # Current context, second component of context
       # information in URL
    operation: string
       # Parameter op in Query
    selected_objects: list of Artifact
       # List of objects identified by id parameters in Query
    single_object: Artifact
       # If specified return object identified by id, otherwise
       # return object. If more than one id parameter is given
       # raise multiple_values Response.
    user: User
       # The (authenticated) user who has issued the request
    language: string
       # The user's preferred language

    # Instance methods

    def handle_it(self): Response
       # Invoke the operation handler in module
       # op_<operation>.

    def want(action, object)
       # Raise no_access Response if action on object
       # is not allowed for user.

    def verify_atom(name): string
       # Return parameter name from Query. Raise
       # missing_parameter Response, if not defined.

    def response(name): Response
       # Return a Response initialized with template name

    def refresh(): Response
       # Commit changes to the persistent store
       # and return a Response which redisplays the current
       # context.
```

Building and formatting the response The Response encapsulates the information to be returned to the requester (c.f. code fragment below). It is instantiated and initialised by an operation handler which provides the *identifier* of the message—a key to some table of parametrised response message templates—and a set of attributes needed for instantiating the message template.

This approach allows the operation handler to specify and return a Response without knowing details of the message format (HTML, plain text, etc.) or the access method (HTTP, email, etc.). The request handling layer thus translates both request and response to hide details of the method of access or protocol used by the requester from the operation handling layer. This arrangement allows extension for additional methods of access with no changes required to the operation handling (or persistent storage) layers of the system kernel.

```
class Response:

    _request: Request
       # The original Request
    _type: string
       # Type of response
    _message: string
       # Response template identifier
    _ecode: integer
       # Error or return code

    # Usually a request handler sets other attributes, which
    # are needed for substitutions in the message template.
```

This scheme also supports different languages in a straightforward manner: for each user there exists a preferences profile that records the language, in which the user wants to interact with the system. This information is available in the Request and is used in the reply function for retrieval of a message template in the correct language. By using the same message naming scheme for message templates in different languages, it is possible to extend the system's support for additional languages without changing the implementation of the Response or the request handler.

3.3.3 User operation handling

The request handling layer invokes operation handlers as described in the previous section. Each operation handler implements an aspect of the basic kernel functionality, such as 'add member' or 'delete document'. As a client processes to the persistent store it is able to retrieve, create, modify, or delete persistent objects.

For example, the code fragment below shows an operation handler that implements the 'rename' operation. The structure is essentially the same for all operation handlers: each must provide at least one of two functions called *handle_GET* and *handle_POST*, which are the entry points to the operation handler from the request handling layer. Both functions are invoked with one parameter, a *Request*, which is a data structure containing the parameters required by the operation. The *handle_GET* function is (normally) assumed to return a *Response* that contains some object data or that asks for more operation parameters, whereas *handle_POST* actually modifies objects. In either case, for a CGI interface incoming HTTP GET or POST requests simply result in invocations of *handle_GET* or *handle_POST* respectively.

```
def handle_GET(request):

    # Get the object to be renamed
    object = request.single_object

    # Check, if the operation is allowed for the
    # requesting user.
    request.want(cl_action.rename, object)
```

```
    # Create a Response selecting the rename template
    response = request.response('rename')

    # Set response attributes needed by the template
    response.name = object.name
    response.op = 'rename'

    return response


def handle_POST(request):

    # Get the object to be renamed
    object = request.single_object

    # Check, if the operation is allowed for the
    # requesting user.
    request.want(cl_action.rename, object)

    # Verify Request parameters
    new_name = request.verify_atom('new_name')

    if object.name != new_name:

       # Change the object
       object.name = new_name

       # Set the appropriate Event
       object.set_event('RenameEvent')

    # Commit changes and redisplay the current context
    return request.refresh()
```

The structure shown in the code fragment above is the same for all operation handlers, though the details will obviously vary depending on the semantics of the operation itself. For some operations, such as 'rename', the Request will include the object to operate on, and some handlers may take a list of objects on which to perform multiple operations (see below). These and other attributes of the Request must be checked for validity—e. g., to ensure that the user provided a folder name when requesting the creation of a new folder and so on. The access rights must also be checked to ensure the user can perform the requested operation on the selected objects. After performing the required operation, a final check is required on committing changed objects back to store due to the optimistic concurrency control strategy adopted. Whether the operation fails or succeeds, the handler must create, initialise and return (or raise) a Response object (exception) containing the appropriate reply.

Handling multiple operations For some operation handlers it is possible that the request specifies a multiple operation; that is, an operation to be performed on a group of objects rather than a single object. In this case the *Request* will contain a list of selected objects to operate on. Should the operation fail for any of these objects for any reason, then the complete operation is said to fail, and an appropriate reply is returned to the requester; no 'rolling back' is required in the store as no object updates are committed until the complete operation has been performed. Treating a multiple operation as atomic ensures consistency and safety; it may be that in some cases a user wants an operation to complete 'as much as possible' and a complete failure requires more work on their part, but the converse where the user wants 'all or nothing' and receives a

partial completion is clearly to be avoided.

3.3.4 The API method call and response handling

Method calls for the CWS API arrive as HTTP POST requests for the `rpc2` CGI script alias; the call itself is encoded in XML according to the XML-RPC protocol. The program that handles these incoming requests is the CGI request handler for this script alias which is a Python script that makes use of F. Lundh's `xmlrpclib`, version 0.99, from Secret Labs AB. The role of this script is to translate the incoming request from its XML encoding into a Python function call and to format the result of this call as a response in XML-RPC.

Each incoming method call invokes the script's `call` procedure with the method name and its parameter tuple as parameters. For example, a method call to `saveRecommendedUser` with a folder idenitfier and a list of and user identifiers as parameters is translated to

`call("saveRecommendedUsers", ("CW_2068", ["CW_52", "CW_191", "CW_126"])).`

The method's two parameters are packaged into an argument-list tuple and are passed to the `call()` method as a single argument. The return value of this function (a Python object) is translated back into a XML-RPC method response and returned to the calling service. When the execution of the `call` function raises an XML-RPC exception this is reported back to the calling service as an XML-RPC fault. Other exceptions are reported back as HTTP error 500 (`CYCLADES Server Error`).

Before the method call is dispatched to the method handler, the method name is checked for validity and the method parameters are checked for number, data type, and null value (optional parameters). This is done recursively for nested parameter structures (`<struct>`s and `<array>`s). Errors raise an XML-RPC exception and are reported back to the calling service via an XML-RPC fault. Method name and parameter checking relies on a static description of the RMS API as (nested) Python objects (lists, tuples and dictionaries depending on the type of the parameters).

The `call()` procedure then turns each method call over to the CWS API central dispatching script's `do_it` procedure with (checked) method name and parameters as arguments.

The following code fragment shows the `call` procedure.

```
def call(method, params):
    from api_def import *

    # method defined for the CWS API

    if method in API.keys():
       import checker

       # check nested parameter structure

       param_defs = API[method][1]
       checker.do_scalar_check(param_defs, params, method)

       # call 'super' function of CWS API server

       func_obj = eval("cws_api.do_it")
       return apply(func_obj, (method, params, ))

    else:
       from xmlrpclib import Fault
```

```
        raise Fault(10002, "No such method: " + method)
```

The result of the `do_it` procedure is the return value of the `call()` procedure which becomes the server's response to the XML-RPC method call.

3.3.5 The API method handling

For each method of the CWS API there is a dispatch function named after the method with a `do_` prepended. The dispatch function for, e. g., `saveRecommendedUsers` is `do_saveRecommendedUsers`. These dispatch functions are also part of the CWS API dispatching script. The dispatch functions call the actual method handlers which are part of the `Cyclades` package. The central `do_it` procedure sets the operating system environment so that subsequent imports are successful and calls the respective dispatch function. Eventual errors are reported back as XML-RPC faults.

The following code fragment shows the `do_it` procedure and the `do_saveRecommendedUsers` dispatch function.

```
import os, sys, time, string
BSCW_DIR = '/servers/bscw-4.0.4/BSCW4/src'
CYC_DIR = '../packages/Cyclades/src'
CGI_DIR = '/servers/apache/cgi-bin'

def do_it(method, params):
    func = eval('do_' + method)

    # set OS environment to CWS package
    os.chdir(BSCW_DIR)
    sys.path[0] = '.'
    sys.path.insert(1, CYC_DIR)

    # call specific dispatch function
    response, errcode, errobj = apply(func, params)

    # switch back to CGI directory
    os.chdir(CGI_DIR)

    if errcode:

       # raise fault
       import errmsg
       msg, xcode = errmsg.msg[errcode]
       from xmlrpclib import Fault
       raise Fault(xcode, msg % (method, errobj))

    return response

    ...

def do_saveRecommendedUsers(folderId, userIds):

    # call method handler
    import ap_saveRecommendedUsers
    return ap_saveRecommendedUsers.handle(folderId, userIds)

    ...
```

The method handlers do the actual work. These handlers, being part of the CWS `Cyclades` package, have access to the BSCW persistent store: they may extract information and also persistently store new information, e. g. user recommendations.

3.4 Data and method specification

The user interface of the CWS and its functionality is implemented by extending BSCW by an additional CWS package `Cyclades`. This package includes a number of classes (`cl_...` files) and of user operation handlers (`op_...` files) described below.

The functionality of the CWS API that may be invoked by other CYCLADES services is described as methods of an abstract class CollaborativeWorkService that are available through the XML-RPC communication protocol. The functionality of this service interface is implemented by API method handlers (`ap_...` files) of the CWS `Cyclades` package. This is also described below.

3.4.1 The classes of the CWS package

The CWS package `Cyclades` needs classes to implement the central CWS concepts users, folders, records and queries. Additionally, new classes are needed for dealing with communities, collections and recommendations. The extension classes build on BSCW core classes contained in `cl_core`, namely Artifact, User, Folder, and Document. Core class User has to be modified slightly for the CWS.

All objects belonging to these classes have unique object identifiers. They are stored persistently in the CWS using the persistent storage facilities of BSCW which allows retrieving objects by their identifiers. All new classes are derived from the core class Artifact and thus inherit (amongst others) the standard attributes identifier, name and description.

Folders

Classes are needed that correspond to CYCLADES private, community and project folders with their respective functionality. An extra class is needed for community root folders because of the fact that community member management is only possible in the root folder of a community. These classes inherit from the BSCW core class Folder and have a similar behaviour with regard to associated collections and recommendations. Therefore we also create an intermediate class CYFolder. In addition, we need a folder class that is to contain the recommendations which CWS receives from the Filtering and Recommendation Service.

The class hierarchy is as follows (class definition file is `cl_cyfolder`):

```
cl_core.Folder
     |
     +-- CYRecommendations
     |
     +-- CYFolder
         |
         +-- CYPrivateFolder
         |
         +-- CYProjectFolder
         |
         +-- CYCommunityFolder
              |
              +-- CYCommunityRoot
```

CYFolder The class CYFolder is a subclass of cl_core.Folder and has the following additional attributes:

- ass_colls: collections associated to a folder (list of CYCollection)
- want_recos: recommendation preferences (integer between 0 and 15, bit encoded for records (0), users (1), collections (2) and communities (3), i. e. a value of 5 signals preference for recommendations of records and collections)
- reco_folder: the folder's recommendations folder (CYRecommendations or None)

Objects of class CYFolder have the following modifications w. r. t. their behaviour:

- When a subfolder of a CYFolder is created, it inherits the attribute values for the associated collections and the recommendation preferences. Subfolders also inherit the members of the parent folder.
- With project folders, membership may be managed down along the hierarchy, adding new and removing old members, which is not the case for private and community folders.
- A CYFolder has at most one recommendations folder (attribute reco_folder).
- There is an additional user operation Update folder profile.

CYPrivateFolder This class corresponds to private folders and is a subclass of CYFolder. It has no additional attributes and methods, but is restricted as follows:

- Only records, queries and other private folders may be added to a private folder.
- No further members may be invited to private folders.

CYCommunityFolder and CYCommunityRoot The class CYCommunityFolder corresponds to community folders and is a subclass of CYFolder. It has two additional attributes

- subscribe: subscription policy preference (0 or 1)
- managers: the managers of the community (list of cl_core.User)

The 'subscribe' attribute indicates whether any registered user may subscribe (invite herself) to this community. This attribute may only be set or changed in the community root folder of class CYCommunityRoot, which is a subclass of CYCommunityFolder. The 'managers' attribute is computed from the access rights of the community members as set by the managers. The original creator of a community is its first manager. The only difference between ordinary community folders and community root folders is that member management of the community is only possible in the root folder. Restrictions concern the following:

- Only records, queries, discussion forums and other community folders may be added to a community folder.
- New members may only be invited in a community root folder.
- Existing members may only be removed in a community root folder.
- Adding and removing members in the community root folder are valid for all the community's subfolders.
- Only community managers have the right for membership management (apart from subscription if the community allows that).

Community root folders represent communities within CWS. Creation and deletion of such folders have to be reflected in the list of all communities in the system (c.f. class CYCommunities below); this concerns the operation handler for creation of community root folders and the standard on_delete method of CYCommunityRoot which has to be overridden to modify the community list accordingly when a community root folder is destroyed.

CYProjectFolder This class corresponds to project folders and is a subclass of CYFolder. It has no additional attributes and methods. The only restriction is that only project folders may be created within a project folder.

CYRecommendations This class represents folders that contain recommendations that are received from the FRS. At most one CYRecommendations folder may be contained in CYFolders. The CYRecommendations folder may contain only recommended records, users, communities and collections, represented by objects of type CYRecord, CYRecoUser, CYRecoComm and CYRecoColl, respectively. The recommendations are meant for the folder containing the recommendations folder.

The CYRecommendations class is a subclass of cl_core.Folder. It has no additional attributes or methods, but the following modifications concerning its behaviour:

- Recommendations folders are not created by a user, but by the system when the first recommendations for this folder arrive from the FRS.
- Recommendations folders have the name `Recommendations` and cannot be renamed.
- Recommendations folders may not be moved or copied, but may be deleted.
- Recommendations folders may only contain recommendations.
- Recommendations are not added to a recommendations folder by a user, but by the system.

Records and queries

Records and queries are implemented as subclasses of the core class Document. The class hierarchy is as follows:

```
cl_core.Artifact
     |
     +-- cl_core.Document
         |
         +-- CYRecord (cl_cyrecord)
         |
         +-- CYQuery (cl_cyquery)
```

Records and queries essentially are XML documents (mime type 'text/xml') which are viewed using a corresponding viewer that is external to the CWS (a primitive XML viewer is included e. g. in the MS Internet Explorer).

CYRecord This class corresponds to the abstract CYCLADES class Record which has attributes id, name, metadata and classifierLabel. The CYRecord class is a subclass of class cl_core.Document with a mime type of 'text/xml', and the body of the document is exactly the XML metadata of the record. Additional attributes take care of the other attributes of the abstract class:

- external_id: record identifier assigned by the Access Service (AS) (string).

- classifier_label: a label indicating the classification of the record by the FRS (string).

Other attributes that are needed internally,

- container_id: the identifier of the folder containing the record (string in CYCLADES object identifier format),
- save_timestamp: the time when the record has been put into its present container, saved from the Search and Browse Service (SBS) or pasted from other folders (seconds since the epoch in UTC),

are computed on the fly from other attributes (history, path) when needed. Restrictions and modifications w. r. t. behaviour are:

- Records are created only by the system on behalf of a user having saved these records in a search and browse session, or as recommendations from the FRS. Records from both sources are represented by CYRecord objects.
- Records are not edited, versioned, or locked, but may be replaced (by the system when records with the same external identifier are saved into, or recommended to, the same folder).
- Records may be moved, copied, deleted, rated and annotated.
- The standard on_rate method is overridden to forward ratings of a record to the RMS.

CYQuery This class corresponds to the abstract CYCLADES class Query which has attributes id, name, queryString, and conditionsAndSourceSchema. The CYRecord class is a subclass of class cl_core.Document with the mime type 'text/xml', and the body of the document is exactly the XML conditions and source schema of the query. The query string of the Query object is put into the description of the CYQuery object. There is one additional attribute:

- external_id: query identifier assigned by the Access Service (AS) (string).

Restrictions w. r. t. behaviour are:

- Queries are created only by the system on behalf of a user having saved these queries in a search and browse session.
- Queries are not edited, versioned, or locked, but may be replaced (by the system when a query with the same identifier is saved into the same folder).

Users

Users are represented by the core class User, user recommendations by the class CYRecoUser. The class hierarchy is as follows.

```
cl_core.Artifact
     |
     +-- cl_core.User
     |
     +-- CYRecoUser (cl_cyreco)
```

cl_core.User This core class is extended by two additional attributes:

- allow_reco: personal preference w. r. t. allowing oneself's recommendation to other users (0 or 1)
- my_collections: set of the user's personal collections (CYMyCollections)

CYRecoUser The CYRecoUser class is a subclass of cl_core.Artifact. It wraps cl_core.User and allows only limited access to information about the user (name, address) and limited operations (send email, invite). Objects of class CYRecoUser are created when a corresponding user has been recommended to a folder. CYRecoUser objects may only appear as entries in CYRecommendations folders. CYRecoUser objects have the following additional attribute:

- orig_user: the user whose recommendation this object represents (cl_core.User).

Restrictions w. r. t. behaviour are:

- Objects of class CYRecoUser are only created when the allow_reco attribute of the corresponding user object is set (value 1).
- Objects of class CYRecoUser may not be moved or copied.
- When objects of class CYRecoUser are deleted they are destroyed at the same time, i. e. may not be undeleted.

Communities

There are three classes for community management

- CYCommunity for objects representing communities in other contexts than folder management.
- CYRecoComm for representing recommended communities.
- CYCommunities for representing the list of all communities in the CWS.

The class hierarchy is as follows:

```
cl_core.Artifact
     |
     +-- CYCommunity (cl_cycommun)
     |
     +-- CYRecoComm (cl_cyreco)
     |
     +-- cl_core.Folder
         |
         +-- CYCommunities (cl_cycommun)
```

CYCommunity This class is a subclass of cl_core.Artifact and has the following additional attribute:

- root: the community root folder which this object represents (CYCommunityFolder)

The values of all other attributes (name, description, subscribe, managers) are derived from the corresponding community root folder. Modifications w. r. t. the behaviour concern:

- Read access to CYCommunity objects is not restricted, i. e. any user may view any CYCommunity object, also non-members of the community. This does not include read access to community contents.
- Objects of class CYCommunity are not manipulated by normal users, but only indirectly by community members.

- Objects of class CYCommunity are created when new community root folders are created, they are destroyed when community root folders are destroyed (not deleted because they could be undeleted again!).

Since CYCommunity objects derive their attribute values from the corresponding community root folders, they do not have to be updated if something changes in the community root folder.

CYCommunity objects allow two additional operations: Join community, if the community is open to subscription, and Mail community managers.

CYRecoComm Objects of this class represent a certain community as recommendation in a recommendations folder. These objects wrap a community object of class CYCommunity, i. e. derive name and description from the original community object. The class has an additional attribute:

- orig_comm: the original community represented by this object (CYCommunity)

Functionality is similar to CYCommunity objects. Note that there may be many different CYRecoComm objects referring to the same CYCommunity object, but that there is only one CYCommunity object for a given community. Restrictions w. r. t. the behaviour are:

- Objects of class CYRecoComm may not be moved or copied.
- When objects of class CYRecoComm are deleted they are destroyed at the same time, i. e. may not be undeleted.

CYCommunities This class is a subclass of cl_core.Folder and contains the currently existing CYCommunity objects as normal entries. There is exactly one object of this class in the system. It is needed when users want to view communities. The CYCommunities object is an attribute `cy_communities` of the pseudo user `Cyclades` who is also playing the role of creator of recommendations, and hence not part of nay user's folder hierarchy.

The CYCommunities object is updated whenever a community root folder is created or destroyed (c.f. above).

Collections

There are three classes for the management of collections within CWS:

- CYCollection for objects representing collections in folder management.
- CYRecoColl for representing recommended collections.
- CYMyCollections for representing the set of personal collections which is defined and modified within the Collection Service (CS).

The class hierarchy is as follows:

```
cl_core.Artifact
     |
     +-- CYCollection (cl_cycollec)
     |
     +-- CYRecoColl (cl_cyreco)
     |
     +-- cl_core.Folder
         |
         +-- CYMyCollections (cl_cycollec)
```

CYCollection This class is a subclass of cl_core.Artifact and has the additional attributes:

- external_id: the collection identifier assigned by the CS (string)
- flst: the folders that have this collection associated (list of CYFolder)

Objects of this class are created or destroyed when the CWS is notified by the CS of a new collection or the deletion of an existing collection. Additionally, the CWS carries out a weekly update on these objects when the CS has not sent any modifications in between.

CYCollection objects keep track of their association to CYCLADES folders via the `flst` attribute. They have specific methods append_content, remove_content and on_delete to keep this relation up to date when collections are associated to folders, when such an association is removed, or when collections are deleted (due to a respective notification of the CS, or detected during a weekly update).

There is one restriction w. r. t. behaviour:

- Collection objects are never manipulated by normal users, but only created and deleted by the system itself.

CYRecoColl Objects of this class represent a certain collection as recommendation in a recommendations folder or as belonging to the personal set of collections. CYRecoColl objects wrap a CYCollection object, i. e. derive values for name and description attributes from the original collection object. The class has an additional attribute:

- orig_coll: the original collection represented by this object (CYCollection)

Note that there may be many different CYRecoColl objects referring to the same CYCollection object, but that there is only one CYCollection object for a given collection.

Restrictions w. r. t. behaviour are:

- Objects of class CYRecoColl may not be moved or copied.
- When objects of class CYRecoColl are deleted they are destroyed at the same time, i. e. may not be undeleted.

CYMyCollections This class is a subclass of cl_core.Folder and contains the current set of personal collections for a given user. It contains only wrapper objects of class CYRecoColl and has an additional attribute

- user: the user whose personal collection set is represented by this object (cl_core.User).

CYMyCollections objects are used when users view their personal set of collections, create folders or associate new collections to folders or disassociate collections from folders. Through an additional attribute `my_collections` of class cl_core.User the personal collection sets are attached to a user.

CYMyCollections objects are updated when the CWS is notified about the creation or deletion of collections and during the weekly update.

3.4.2 The user operation handlers of the CWS package

For every user operation we have listed above, there is a respective handler that takes care of the user input coded as a Request object and generates HTML output as a response for the user coded as a Response object. A user's 'hit' of a function at the user interface is transmitted to the CWS

as a HTTP GET request, a user's submittal of a form in a two-step interaction as a HTTP POST request. Consequently, the operation handler for a two-step interaction has a handle_GET and a handle_POST procedure; for one-step interaction, there is no handle_POST procedure.

In the following, the name of the file that contains the respective handler of the CWS package is listed for most user operations. All file names for operation handlers start with `op_` followed by the operation name. For those operations that are generic and hence are handlers of the BSCW core (like Get, Info, Cut, Copy, Delete, Edit name etc.) the handler file names are not given.

- create community root folder: `op_addcycommunroot`
- create other folders
  - community: `op_addcycommunityf`
  - project: `op_addcyprojectf`
  - private: `op_addcyprivatef`
- edit associated collections
  - add associated collection: `op_addcycollection`
  - remove associated collection: `op_remcycollection`
- view my collections: `op_showmycycollections`
- edit recommendation and subscription policy preferences
  - community root folders: `op_efolder_prefs`
  - other folders: `op_folder_prefs`
- view communities: `op_showcycommunities`
- join a community
  - in the `Communities` folder: `op_join_community`
  - in a `Recommendations` folder: `op_join_reco_comm`
- contact community managers
  - in the `Communities` folder: `op_mail_managers`
  - in a `Recommendations` folder: `op_mail_reco_comm_mgr`
- invite recommended user to folder: `op_invite_reco_user`
- contact recommended user: `op_mail_reco_user`
- associate recommended collection to folder: `op_assoc_reco_coll`
- update folder profile: `op_update_profile`

For the following operations, BSCW core handlers have to be modified to reflect the modified or extended functionality for the CWS.

- edit personal preferences: `op_chprefs`
- invite to folder: `op_addmb`

3.4.3 Abstract CWS classes, API methods and thier handlers

The abstract CWS classes that define the external view of the CWS by other Cyclades services concern the CWS itself and its methods plus the classes that are used as parameters in method signatures: records and queries.

Query

- id
  Description: this is the unique identifier of the query.
- name
  Description: this is a string containing the name of the query. This attribute is optional, i. e. is only present if the user has given the query a name.
- queryString
  Description: this is a string containing the actual query.
- conditionsAndSourceSchema
  Description: this is a string containing the query conditions and the source schema for the query coded in XML.

Record

- id
  Description: this is the unique identifier of the record.
- name
  Description: this is a string containing the name of the record, e. g. the title of the document referenced by the record.
- metadata
  Description: this is a string containing the metadata of the record coded in XML.
- classifierLabel
  Description: this is a string containing a label indicating the classification of the record by the Filtering and Recommendation Service. This attribute is optional, i. e. only records which have been originally recommended by this service have a value for this attribute.

CollaborativeWorkService

Below we list the methods of the abstract class CollaborativeWorkService which constitute the API of the CWS. These methods may be called from other services using the Cyclades inter-service communication protocol XML-RPC. For every method we also list the name of the file in the CWS package that contains the respective method handler as a procedure.

- id
  Description: this is the unique identifier of the service.
- (userId, homeFolderId) createUser(name, password, emailAddress)
  Description: this method is invoked in order to create a new user within the CWS with the given name, password and email address.
  Input: name: a valid user name (unique, longer than 2 characters, no blanks or at-signs (@)).
  password: a password.
  emailAddress: an email address conforming to RFC 822.

Output: userId: the identifier of the newly created user.
homeFolderId: the identifier of the user's home folder.
File: `ap_usermgmt`

- void updatePasswd(name, password)
  Description: this method is invoked in order to set a new password for the user with the given name.
  Input: name: a user name of an existing user.
  password: a password.
  File: `ap_usermgmt`

- folderId* getFolders(userId)
  Description: this method may be invoked in order to get a list of identifiers of folders to which a specific user has access.
  Input: userId: a user identifier.
  Output: a list of folder identifiers.
  File: `ap_folderInfo`

- name getName(folderId)
  Description: this method may be invoked in order to get the name of a folder.
  Input: folderId: a folder identifier.
  Output: a string containing the folder name.
  File: `ap_folderInfo`

- description getDescription(folderId)
  Description: this method may be invoked in order to get the description of a folder.
  Input: folderId: a folder identifier.
  Output: a string containing the folder description.
  File: `ap_folderInfo`

- userId* getMembers(folderId)
  Description: this method may be invoked in order to get the identifiers of the users who have access to a folder.
  Input: folderId: a folder identifier.
  Output: a list of user identifiers.
  File: `ap_folderInfo`

- (recordId, classifierLabel)* getRecords(folderId, timestamp)
  Description: this method may be invoked in order to get the identifiers and classifier labels of the records that have been saved into, or moved to, a folder since a certain time.
  Input: folderId: a folder identifier.
  timestamp: a point in time in UTC.
  Output: a list of pairs containing a record identifier and a classifier label; a missing classifier label is signaled by an empty string.
  File: `ap_folderInfo`

- queries getQueries(folderId)
  Description: this method may be invoked in order to get the queries that are contained in a folder.
  Input: folderId: a folder identifier.
  Output: a list of objects of class Query.
  File: `ap_folderInfo`

- folderId getParent(folderId)
  Description: this method may be invoked in order to get the identifier of the folder that contains a given folder.
  Input: folderId: a folder identifier.

Output: a folder identifier or an empty string, if the given folder is not contained in another folder.
File: `ap_folderInfo`

- folderId* getChildren(folderId)
  Description: this method may be invoked in order to get the identifiers of the folders that are contained in a given folder.
  Input: folderId: a folder identifier.
  Output: a list of folder identifiers.
  File: `ap_folderInfo`

- collectionId* getCollections(folderId)
  Description: this method may be invoked in order to get the identifiers of the collections that are associated to a folder.
  Input: folderId: a folder identifier.
  Output: a list of collection identifiers.
  File: `ap_folderInfo`

- communityId getCommunity(folderId)
  Description: this method may be invoked in order to get the identifier of the community to which a folder belongs.
  Input: folderId: a folder identifier.
  Output: a community identifier or an empty string if the folder does not belong to a community. File: `ap_folderInfo`

- void saveQuery(folderId, userId, query)
  Description: this method may be invoked in order to save a query in a folder on behalf of a user.
  Input: folderId: a folder identifier.
  userId: a user identifier.
  query: a query.
  File: `ap_saveFromSB`

- void saveResults(folderId, userId, records)
  Description: this method may be invoked in order to save a list of records in a folder on behalf of a user.
  Input: folderId: a folder identifier.
  userId: a user identifier.
  records: a list of records.
  File: `ap_saveFromSB`

- boolean saveRecommendedRecords(folderId, records)
  Description: this method may be invoked in order to save a list of recommended records for a folder.
  Input: folderId: a folder identifier.
  records: a list of records.
  Output: false if record recommendations for this folder are not welcome, true otherwise.
  File: `ap_saveRecommendedRecords`

- boolean saveRecommendedUsers(folderId, userIds)
  Description: this method may be invoked in order to store a list of user recommendations for a folder.
  Input: folderId: a folder identifier.
  userIds: a list of user identifiers.
  Output: false if user recommendations for this folder are not welcome, true otherwise.
  File: `ap_saveRecommendedUsers`

- boolean saveRecommendedCommunities(folderId, communityIds)
  Description: this method may be invoked in order to store a list of community recommendations for a folder.
  Input: folderId: a folder identifier.
  communityIds: a list of community identifiers.
  Output: false if community recommendations for this folder are not welcome, true otherwise.
  File: `ap_saveRecommendedCommunities`

- boolean saveRecommendedCollections(folderId, collectionIds)
  Description: this method may be invoked in order to store a list of collection recommendations for a folder.
  Input: folderId: a folder identifier.
  collectionIds: a list of collection identifiers.
  Output: false if collection recommendations for this folder are not welcome, true otherwise.
  File: `ap_saveRecommendedCollections`

- void addModifyCollection(collectionId, collectionName)
  Description: this method may be invoked in order to notify of the creation of a new, or the modification of an existing, collection.
  Input: collectionId: a collection identifier.
  collectionName: the collection name.
  File: `ap_addModifyCollection`

- void deleteCollection(collectionId)
  Description: this method may be invoked in order to notify of the deletion of a collection.
  Input: collectionId: a collection identifier.
  File: `ap_deleteCollection`

- void updatePersonalCollections(userId, collectionIds)
  Description: this method may be invoked in order to notify of the update of a user's personal set of collections.
  Input: userId: a user identifier.
  collectionIds: a list of collection identifiers.
  File: `ap_updatePersonalCollections`

3.5 User interface

In context of the CWS, artifacts are considered as files collected in a shared folder. A folder may contain other folders, which may contain other files or folders. A set of files and/or folders define a dedicated folder such as a private, a project, or a community folder. Artifacts are either private to a particular user (in private folders) or shared among the members of a folder (i. e. among the users belonging to a project or community represented by the particular folder). A registered user of Cyclades who is a member of a folder in that sense might add, move, delete, read or edit artifacts within that folder.

The folder listing

The CWS user interface presents the contents of a folder as the main part of page with a header containing pull-down menus, buttons and action shortcuts. As an example figure 3.6 shows parts of the user interface of the CWS with an open File menu for creating new objects in the folder.

Several classes of artifacts can be created and shared in a folder: various kinds of Folder, Discussion, Note. In project folders, additionally the following artifacts may be created and shared: Document, URL, Search. The Record and Query artifacts are not created via user operations, but transferred from the Search and Browse Service; they may also be shared in folders.

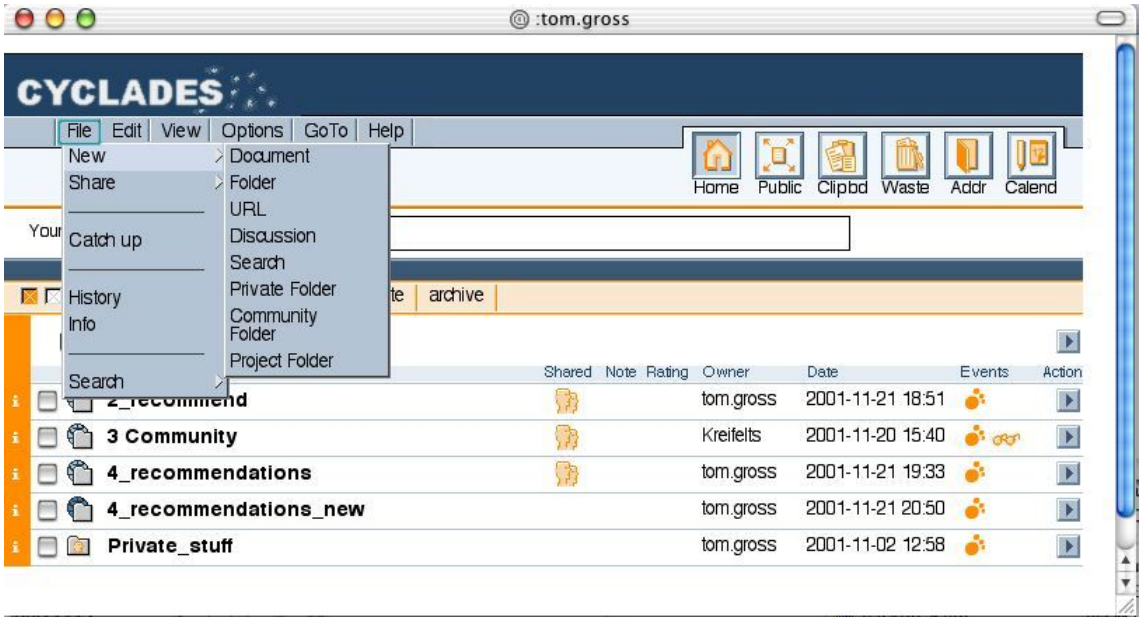

Figure 3.6: CWS user interface mockup

Each artifact is represented by an entry consisting of an information button, a checkbox (that a user has to "tick" to select the artifact for some action), the name of the artifact, some icons, additional data and a pop-up action menu button.

The icon immediately left of the artifact name denotes the artifact type. If the artifact is a document, the icon may represent its MIME type, if it is a search, the icon informs about the search area and the search engine employed, if it is a note, the icon may represent a pragmatic aspect picked by the note's author.

Most prominent in each entry is the name of the artifact. Obviously, a user should choose names describing the content or purpose of each individual artifact. To the right of the artifact name the CWS user interface displays:

- Zero, one or more of the following icons:
  - *Shared Icon* indicates that a folder is shared;
  - *Lock Icon* indicates that someone has set a lock for this artifact;
  - *Note Icon* indicates that a note has been added to the artifact;
  - *Rate Icon* indicates that the document has been rated by one or more community members;
- the user name of its owner;
- date and time of the most recent modification;
- zero, one or more icons indicating that some of the following events have occurred:
  - *New Icon* indicates a new artifact;
  - *Change Icon* indicates changes to the artifact;
  - *Read Icon* indicates that someone has read the artifact;
  - *Modification Icon* indicates recent modifications in a sub-artifact;
- a pop-up menu button showing the actions involving the artifact.

Note that different actions are possible for different types of artifacts. Depending on the individual access rights, the number and type of actions permitted may vary.

Most icons in an artifact entry are "clickable", i. e. one gets more information on a community, a lock, a note, a rating etc. when one clicks on it.

The instant access navigation buttons

The upper right hand corner of the interface provides an icon bar showing buttons applicable to instantly access certain artifacts.

- *Home Folder Icon* represents a user's home folder.
- *Clipboard Icon* recpresents a user's clipboard, which serves as an intermediate store.
- *Waste Icon* represents a user's waste, which helps to prevent unauthorized or unintentional deletion of artifacts: In the CWS, an artifact can be irrevocably destroyed only from the Waste of its owner.
- *Address Book Icon* represents a user's address book, to be used primarily to invite new members to the user's folder.

The menu bar

At the upper left hand corner of the interface there is a menu bar with pull-down menus (and action shortcut icons for the most frequent actions).

New artifacts are added to the current folder by selecting one of the *File* menu options:

File → *New* plus a sub-option from the list *Discussion, Private Folder, Community Folder, Project Folder* in order to create an artifact of the specified type directly on the CWS server. For project folders, also documents, URLs, and searches may be created. Examples are:

- Select *File* → *New* → *Private Folder* in order to create a private folder within the current private or home folder.
- Select *File* → *New* → *Community Folder* in order to create a community folder within the current community or home folder.
- Select *File* → *New* → *Project Folder* in order to create a project folder within the current project or home folder.
- Select *File* → *New* → *Document* to upload a file from the user's local computer system to the current project folder.

If one wants to create a new private, community or project folder, the CWS user interface asks for *name, description, collections* to be associated to that folder; finally the user is asked to specify whether recommendations are requested for *records, users, collections, or communities*, (i. e. tick the appropriate box). For communities one has to also specify in the creation dialogue whether the community is open to be joined by external users.

When a new document is identified to be uploaded to the CWS server, one is asked via an additional dialogue to specify the document's local URL, name, description, MIME type and encoding.

If one wants to rate the new artifact, one may choose one of the following options: *no* (for no rating available), *very poor, poor, fair, good, or excellent*, by ticking the respective radio button.

The *Edit* menu is used to transfer existing artifacts to/from from the clipboard by the following procedure. Select the *Paste* option in the Edit menu to add artifacts that arrived at the Clipboard

from somewhere in the user's folders as a result of the most recent *Copy* or *Cut* action to the current Folder.

Via the *Options* menu, the user may edit *Preferences,* personal user *Details,* user *Communication* specifics, user *Password,* or user *Default Events.*

To navigate among the particular entities one can also use the *GoTo* menu.

- Select *GoTo* → *Home* to get to the user's home folder.
- Select *GoTo* → *Clipboard* to get to the user's clipboard.
- Select *GoTo* → *Waste* to get to the user's waste.
- Select *GoTo* → *Address Book* to get to the user's address book.
- Select *GoTo* → *Communities* to get to the current list of communities.
- Select *GoTo* → *User Info* to receive the specifics of the user information stored in the system.

The multi-action buttons

Preceding the list of artifacts, the CWS user interface provides a number of buttons for actions to be applied to several selected artifacts.

In general, an artifact is selected by marking the checkbox to the left of its name. The *Select all* and *Select none* buttons are shortcuts for selecting or de-selecting all artifacts within the current folder.

Hitting, e. g., the *copy* button invokes copying, and hitting the *cut* button triggers transfer of the selected artifacts to the user's clipboard; hitting the *delete* button invokes transfer of the selected artifacts to the waste. Certain buttons such as *send* and *rate* trigger actions that can be applied only to artifacts of specific types.

Note that artifacts transferred from a folder to the clipboard or to the waste are no longer visible to the members of the folder.

The single action pop-up menus

At the right end of the entry containing the artifact name in the folder listing, the CWS user interface provides for every entry an action pop-up menu for operations to be applied only to that particular artifact. Here one gets all the actions that are applicable to the artifact, including generic ones like *Open, Catch up, History, Info*, or actions that depend on the nature of the artifact, e. g. *Rate, Attach note* for records or *Add Collections, Remove Collections* for CYCLADES folders.

Here in the single action menu the actions more specific to CWS are to be found that do not appear in the menu bar or the multi-actions:

- *Add Collections* to associate additional collections to a CYCLADES folder,
- *Remove Collections* to remove collections from the set of collections associated to a CYCLADES folder,
- *Edit Folder Prefs* to modify the preferences for the recommendations sought for and/or joining rights,
- *Update Folder Profile* to request an immediate update of the folder profile in the Filtering and Recommendation Service,
- *Join Community* to join the selected community,

- *Mail Community Mgrs* to send an electronic letter to the manager(s) of a particular community (e. g. to get permission to join a community).

The configuration of the Action menu depends on the type of artifact—for example, different actions are appropriate for a URL artifact, a folder or a document.

What actions are applicable to which artifacts? Access right management in the CWS is based on roles, whereby a role defines the set of artifacts and actions a user may apply for a specific task. As a consequence, the CWS will not display the single action menu entries for actions that a user may not perform on the specific artifact w. r. t. to the role assigned to the particular user. On the other hand, one may invite, for example, new community members assigning roles to them. Moreover, roles define access profiles which can be attached to any artifact in the CWS. A set of pre-defined roles serves as a starting point: *manager*, *owner* (originally the creator; also accountable for the disk space used), *member* and *restricted member* (read-only access). Role assignments are inherited along the folder hierarchy and can be modified at any time.

Context Sensitive Help

The context sensitive help feature provides assistance to navigate the system. In a number of application environments one may click the *Question Mark* button to get an explanation of the action to be launched.

3.6 Service interaction diagrams

Service interaction diagrams describe the interaction between a user or external service with the CWS in executing one of its functions or methods.

3.6.1 Diagrams for user operations

In the following, service interaction diagrams are given for all user operations that involve a call of the CWS to other services. The diagrams for user operations that do not involve such external calls are very simple and easily derivable from the information on operations and operation handlers presented above.

For the interaction of the user with the CWS, we give the name of the operation invoked by the user (`op=...` part of the HTTP GET request) plus the parameters filled in by the user, if any. Context information like the id of the current folder, the ids of the objects concerned (if different from the current folder) and the id of the user carrying out the operation is not represented in the diagrams, but of course encoded in the request. The HTML output returned to the user as response is described informally.

Create folders

The creation of folders is described for the case of community root folders. When the user actually creates a folder and sets the recommendation preferences, these are forwarded to the FRS. The creation of all other types of folders involves the FRS in exactly the same way as in the creation of community root folders, even if the preferences are inherited from the parent folder and not directly set by the user. The respective diagrams are very similar—only the names of the operations are different—and hence not given.

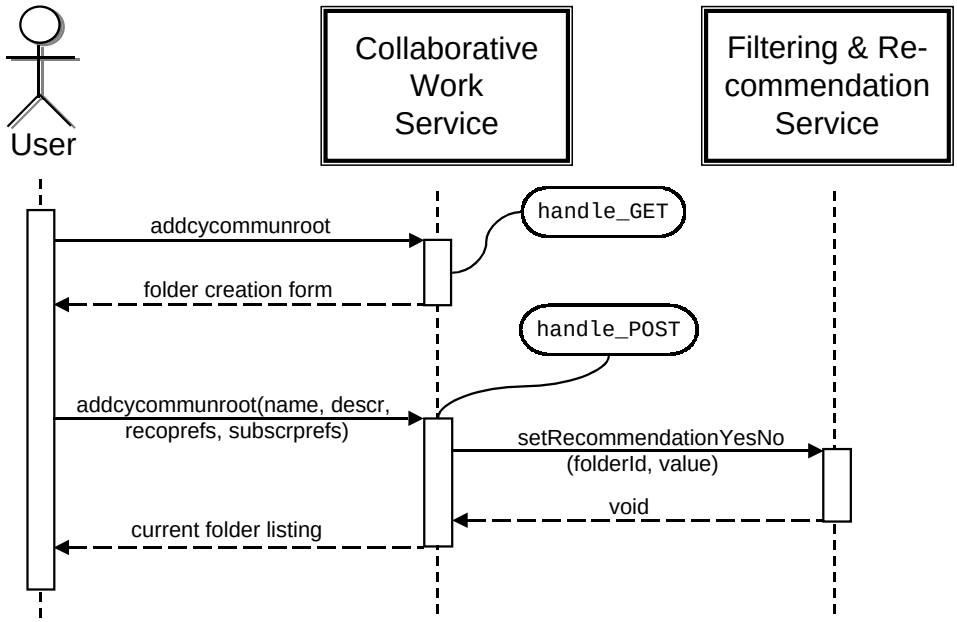

Figure 3.7: Create a community root folder

Edit recommendation preferences

When a user edits the recommendation preferences of a folder, changed preferences are forwarded to the FRS (which of course is not done if the user has left the preferences unchanged). There are two different operations for editing recommendation preferences: one for community root folders (allowing also the editing of the subscription policy preference) and one for all other cases. We represent the latter in the diagram, the diagram for the other case is identical apart from the operation name.

Rate records

The ratings of records are forwarded to the Rating Management Service (RMS). Rating is done using a core operation handler `op_rate` which is not part of the `Cyclades` package. The forwarding to the RMS is done by the `on_rate` method of the CYRecord class which is called whenever an Artifact is rated. This internal detail is not represented in the respective diagram.

Invite a recommended user

Only inviting users which are not already registered involves the Mediator Service (MS). This case is treated in the respective diagram. The later phase of the complete process when the invitee actually registers and becomes a member of the folder to which she has been invited, is not given in the diagram because this is a user operation of the MS.

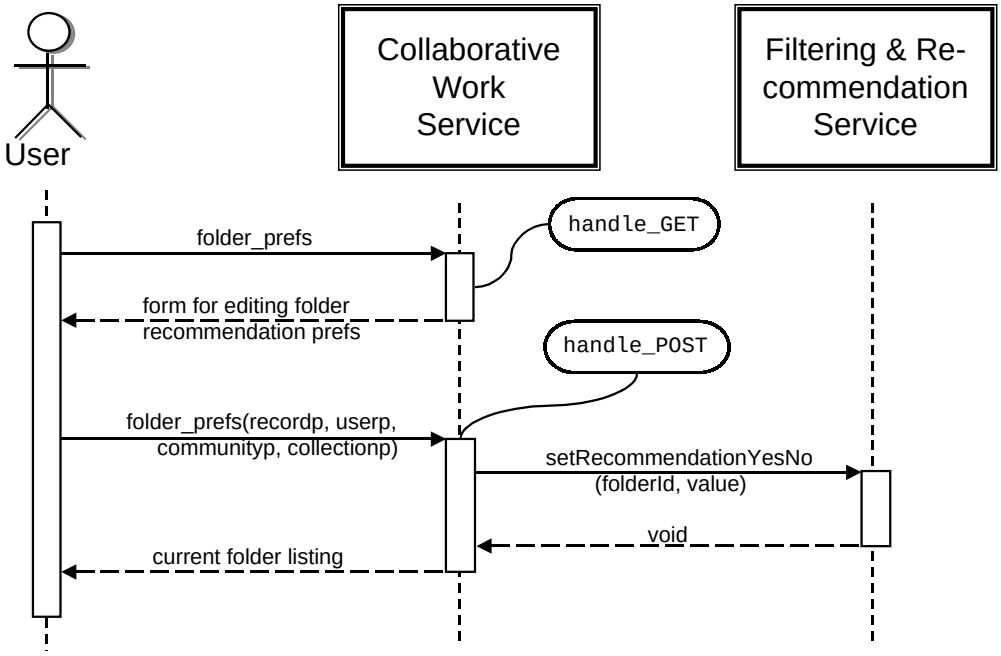

Figure 3.8: Edit recommendation and subscription policy preferences

Update folder profile

Requesting the update of a folder profile is a one-step user operation. The request is simply forwarded to the FRS.

3.6.2 Diagrams for API invocations

Invocation of a CWS API method never involves a call to other services. Consequently, the respective service interaction diagrams are very simple and equal in structure. As an example, the service interaction diagram for creating a new user is given.

3.7 Service implementation tools

In order to install and run the CWS you need the following software:

- Python, Version 2.0
  http://www.python.org/
- xmlrpclib, Version 0.99
  http://www.pythonware.com/products/xmlrpc/index.htm
- Apache Web server, Version 1.3
  http://www.apache.org/

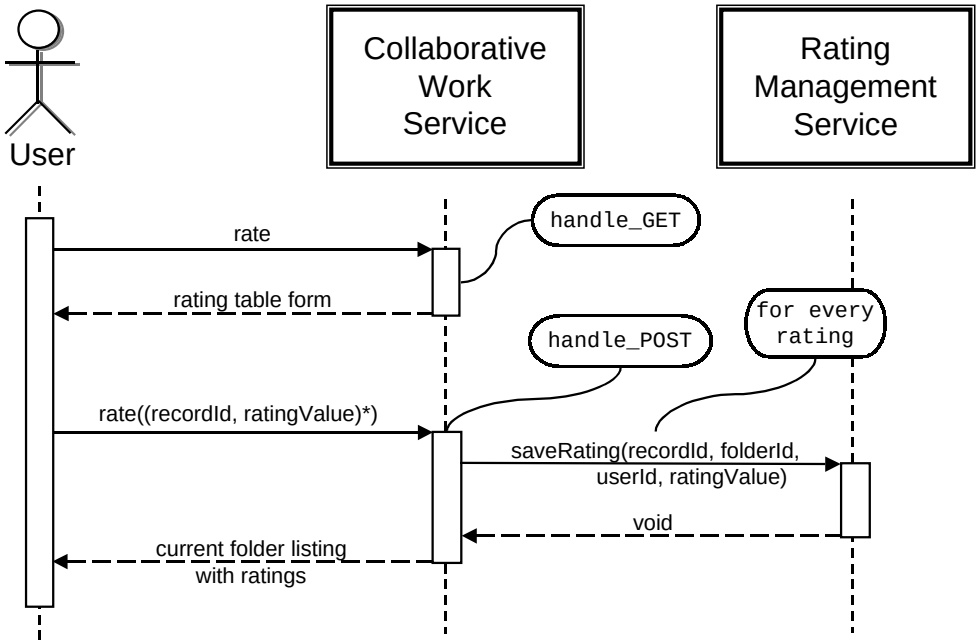

Figure 3.9: Rate records

- BSCW, Version 4
  http://www.orbiteam.de/

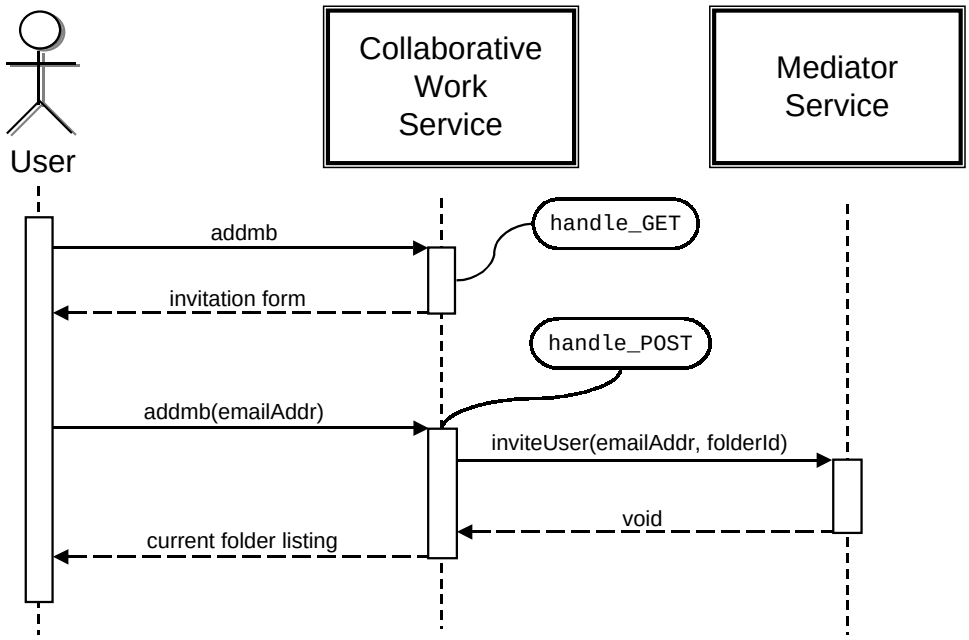

Figure 3.10: Invite a recommended user to a folder

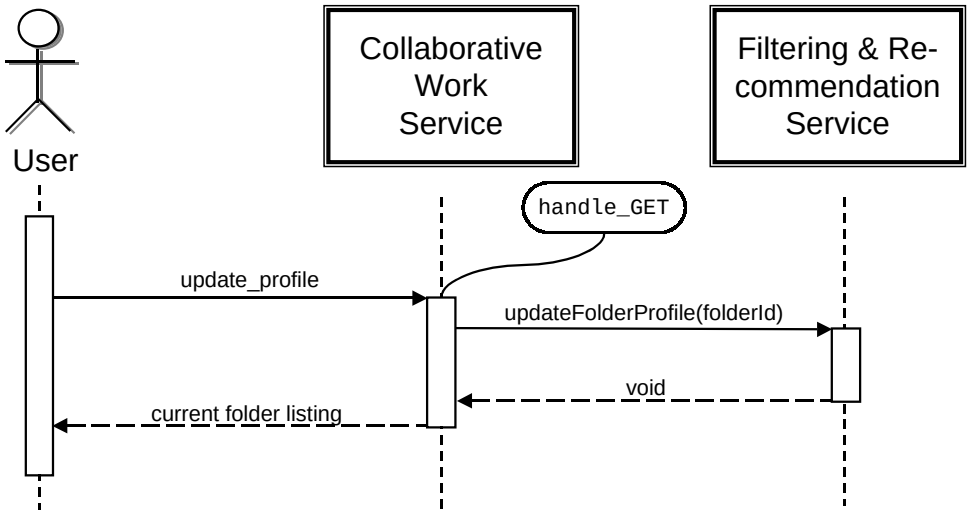

Figure 3.11: Update folder profile

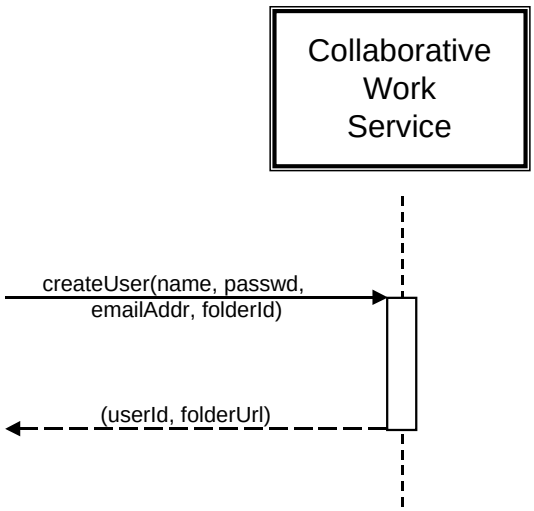

Figure 3.12: Create a new user

Chapter 4

Search and Browse Service

4.1 Functionality

The Search and Browse Service (SBS) offers the CYCLADES user multilevel hypertext searching and browsing, in the form of a user interface to search and browse CYCLADES metadata records, attribute values, and schemas.

4.1.1 Searching and browsing

The first thing to decide for the user is the source for the search. The user can decide to search all CYCLADES metadata records, or only records from a specific virtual collection (for a definition of virtual collection, please refer to the chapter on the Collection Service).

Then, she can formulate a new query or re-use an existing one from the active folder in the folder workspace (for a description of folders, please refer to the chapter on the Collaborative Work Service).

For query formulation, the user can browse the metadata schemas available for the chosen virtual collection, and, where applicable, the values that can be found for specific metadata attributes (e. g. a list of person names). For future re-use, the user can save the query to the folder system.

She can also decide to skip formulating a query explicitly and instead use the current folder as an implicit query by asking for *new records for the active folder* (for this variant, please refer to the chapter on the Filtering and Recommendation Service).

After submitting an explicitly formulated query or the request for new records for the current folder, the user is presented with a ranked list of results. This list can be returned as it is, or it can be filtered using the current folder profile (see the chapter on the FRS for details). From the result list, she can select records to be saved to the current folder in her workspace.

4.2 Process flow

In this section, we describe the search and browse process in more detail. For the personalized variants, please refer to the chapter on the Filtering and Recommendation Service.

4.2.1 Searching and browsing without personalization (SU 8)

One basic assumption in CYCLADES is that the user is always in a current folder. Thus, the Search and Browse Service functionality is called with the id of the current folder.

The search and browse process consists of the following steps:

- **System:** determines the list of collections that are associated to the user's current folder, or (if there are no collections associated to the folder) the user's personal list of collections, and presents the list to the user to browse
- **User:** selects collections to search, or requests to search
  - all collections associated to the folder, or
  - all of her personal collections, or
  - all collections in the system
- **System:** determines the list of queries stored in the user's current folder, and the list of queries the user has submitted during the same search and browse session before (if there are none, this step and the following step are skipped)
- **User:** selects an existing query, or decides to formulate a new one
- **System:** presents the user the query formulation interface, with the appropriate fields filled in if the user selected an existing query in the previous step
- **User:** edits the query
  this comprises
  - adding, changing, or deleting query conditions
  - browsing the metadata schemas available for the selected collections
  - for a specific metadata schema, browsing attribute values (if applicable)
- **User:** saves the query to the current folder, if she wants to re-use it later (this results in the system copying the query to the Collaborative Work Service for persistent storing)
- **User:** submits the query, with or without personalization
- **System:** executes the query and presents a result set of records to the user, ranked by estimated relevance
- **User:** browses the result set, or previous result sets still existing from this search and browse session, possibly selects records to be saved to the current folder
- **System:** if there are any records to be saved, the system passes them to the Collaborative Work Service

4.3 Internal architecture

4.3.1 Overview

Figure 4.1 shows the internal architecture of the Search and Browse Service. It consists of the following layers, respectively components:

- the SBS API
- an implementation layer
- the search and browse GUI
- communication modules

In the following, we describe these individual components.

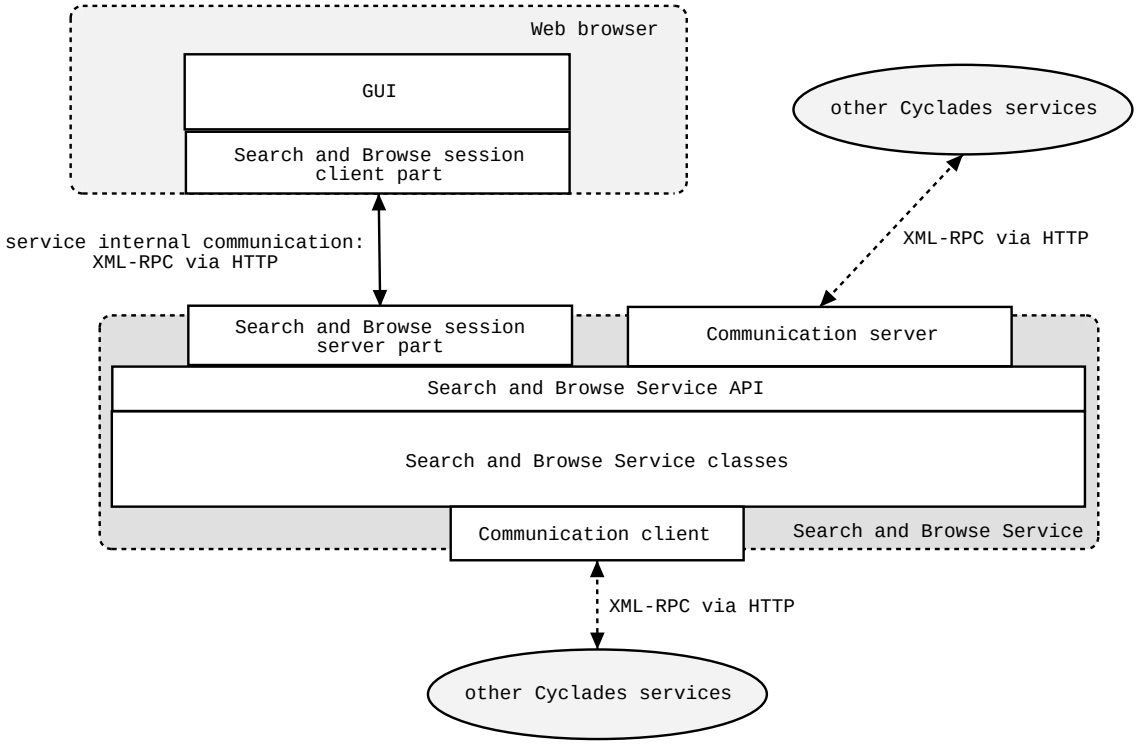

Figure 4.1: Search and Browse Service: internal architecture

4.3.2 Search and Browse Service API

This layer is the API to all Search and Browse Service functions that are accessible to the GUI or to other CYCLADES services. It consists of Java interfaces specifying the available methods. The individual classes (respectively, from the implementation point of view, Java interfaces) are described below in section 4.4.

As the Search and Browse Service works with records, metadata schemas, and archives, all of which are classes primarily by other services and thus described in the corresponding chapters, the Search and Browse Servive will have to use and implement the corresponding Java interfaces for those classes, too.

4.3.3 Search and Browse Service implementation classes

This layer consists of the Java classes implementing the Search and Browse Service's API.

4.3.4 Search and Browse GUI

This component of the Search and Browse Service interacts with the user in a search and browse applet, thus implementing the SBS's main task.

4.3.5 Communication modules

The GUI communicates with the SBS itself via XML-RPC calls for which there is an extra client and server module added between the two parts of the Search and Browse session implementation. The client module translate method calls of SearchAndBrowseSession objects (and other instances

of CYCLADES entity classes, like Archive etc.) into remore procedure calls that are answered at the SBS server by the corresponding server module.

The Search and Browse Service uses methods of the other CYCLADESservices (see next section, 4.6). In the first version of CYCLADES, the communication between the different services is carried out via XML-RPC, thus, the Search and Browse Service's module for external method invocation is implemented as an XML-RPC client running at the SBS (which is different from the client for internal communication running in the Web browser with the GUI applet).

The only request that the SBS accepts from another service is the initialization of a Search and Browse session by the Mediator Service. Thus, the SBS contains a second communication server module for external communication.

As the whole of the communication is encapsuled by special modules, it can later be changed to other protocols as needed.

4.4 Data and method specification

4.4.1 Search and Browse Service

The Search and Browse Service provides the search and browse functionality of the system. This service has no persistent data.

SearchAndBrowseService

This class implements the Search and Browse Service. It maintains a list of search and browse sessions.

public:

- id
  Description: this is the unique ID of the service (string)
- sessions
  Description: a list of SearchBrowseSession objects
- HTML initiateSearch(folderId,userId)
  Description: this method can be called to initiate a search and browse session for the given *folderId* and *userId*, it returns the initial SBS interface page to the caller
  Input. userId: the id of user who started the search
  folderId: the id of the folder the user started the search from
  Output. the first page of the Search and Browse Service GUI

service internal:

- SearchBrowseSession initiateSearch(folderId,userId)
  Description: this method can be called to initiate a search and browse session for the given *folderId* and *userId*
  Input. userId: the id of user who started the search
  folderId: the id of the folder the user started the search from
  Output. a new SearchBrowseSession object
- Record* getNewForFolder(sessionId)
  Description: determines the list of new records that are relevant to the current folder topic,

by calling the Input. sessionId: the id of the SearchBrowseSession object this method refers to
Output. a list of records

- Record* search(sessionId, query)
  Description: returns a list of records that are deemed relevant with respect to the specified query, by calling the search method on the appropriate SearchBrowseSession
  Input. sessionId: the id of the SearchBrowseSession object this method refers to
  query: the query to be evaluated
  Output. a list of records

- Record* filteredSearch(sessionId, query)
  Description: returns a list of records that are deemed relevant with respect to the specified query, filtered with respect to the current folder topic, by calling the filteredSearch method on the appropriate SearchBrowseSession
  Input. sessionId: the id of the SearchBrowseSession object this method refers to
  query: the query to be evaluated
  Output. a list of records

- void saveResults(sessionId, recordId*)
  Description: saves the records specified by recordId* to the user's folder, by calling the saveResults method on the appropriate SearchBrowseSession
  Input. sessionId: the id of the SearchBrowseSession object this method refers to
  list of recordId: the list of record ids of the records that are to be stored to the user's folder

- (Collection*, Schema*) getCollections(sessionId)
  Description: returns the list of collections and the corresponding schemas available in a search and browse session
  Input. sessionId: the id of the SearchBrowseSession object this method refers to
  Output. list of Collection: list of Collection objects
  list of Schema: list of Schema objects

- value* getAttributeValues(sessionId, schemaName, attributeName, maxNo)
  Description: returns the list of values for the specified attribute and schema available in a search and browse session
  Input. sessionId: the id of the SearchBrowseSession object this method refers to
  schemaName: the name of the schema
  attributeName: the name of the attribute
  maxNo: the maximum number of values to be returned
  Output. a list of attribute values, ordered by the default order according to the attribute's type

SearchBrowseSession

One instance of this class corresponds to one user-service interaction sequence, i. e. this is the class that interacts with the user (through a separate user interface).

public:

- id
  Description: the unique ID of the session

- expirationTime
  Description: the timestamp when this object will be deleted, i. e. when the session will expire

- userId
  Description: the ID of the user that initiated the search

- activeFolderId
  Description: the ID of the folder the search was initiated from

- queryHistory
  Description: the list of all queries formulated during this search session (list of strings)

- resultHistory
  Description: the list of result sets obtained during this search session, each result set being a set of Record objects

- relevantRecords
  Description: a list of those records marked as relevant by the user

service internal:

- folderId
  Description: the id of the folder the search was initiated from

- Record* getNewForFolder()
  Description: determines the list of new records that are relevant to the current folder topic
  Output. a list of records

- Record* search(query)
  Description: returns a list of records that are deemed relevant with respect to the specified query
  Input. query: the query to be evaluated
  Output. a list of records

- Record* filteredSearch(query)
  Description: returns a list of records that are deemed relevant with respect to the specified query, filtered with respect to the current folder topic
  Input. query: the query to be evaluated
  Output. a list of records

- void saveResults(recordId*)
  Description: saves the records specified by recordId* to the user's folder
  Input. list of recordId: the list of record ids of the records that are to be stored to the user's folder

- (Collection*,Schema*) getCollections()
  Description: returns the list of collections and the corresponding schemas available in this search and browse session
  Output. list of Collection: list of Collection objects
  list of Schema: list of Schema objects

- value* getAttributeValues(schemaName, attributeName, maxNo)
  Description: returns the list of values for the specified attribute and schema available in this search and browse session
  Input. schemaName: the name of the schema
  attributeName: the name of the attribute
  maxNo: the maximum number of values to be returned
  Output. a list of attribute values, ordered by the default order according to the attribute's type

4.5 User interface

In this section, we describe the search and browse user interface. From the user's perspective, it consists of a set of dialogs for the individual steps of the search and browse process (as described in section 4.1). For more flexibility, we decided to implement this GUI as a Java applet. The user can leave the search and browse environment at any time by selecting one of the other main functionalities (e. g. the collaborative work environment). Then, all search and browse windows that might have been opened by the applet are closed, and the search and browse session is ended.

4.5.1 Query formulation dialog

This is the central dialog of the Search and Browse GUI. It always contains the current query, at the beginning, this is an empty query.

The dialog contains the name of the user's current folder, the list of collections that the query refers to, the name of a metadata schema to be used, a list of query conditions, and the following elements:

- *Select collections* button
  this button opens the *Collection selection dialog* where the user can choose the source collections for the search and browse process
- *Select query* button
  this button opens the *Query selection dialog* where the user can choose a previous query from the same search and browse session, or a query that has been stored in the current folder
- *Select metadata schema* button
  this button opens the *Metadata schema selection dialog* where the user can browse the available metadata schemas
- for every query condition:
  - a drop-down menu to choose a field from the selected metadata schema
  - possibly another drop-down menu to choose a subfield of the selected field
  - a drop-down menu to choose the predicate to be used in this condition (e. g. equality)
  - an input field for a reference value for the predicate, combined with a button to open a *Value browsing dialog* (if applicable)
  - an input field combined with a drop-down menu to specify the weight of the condition (+, -, or a number)
  - *Remove condition* button to remove the respective condition from the query
- *Add condition* button
  this button adds a new query condition
- *Get new records relevant to folder topic* button
  this button ignores the current query (if any) and submits a search request based on the profile of the current folder
- *Submit as is* button
  this button submits the query without personalization
- *Submit personalized* button
  this button submits the query with personalization
- *Clear all* button
  this button resets all conditions and selections to their defaults

Any form of submitting the query, respectively requesting records relevant to the folder topic, results in a *Result list* being opened.

4.5.2 Collection selection dialog

In this dialog, the user can specify which collections the system should use as a source for the following search and browse process.

The dialog contains a list of collections. By default, this is the list of collections associated to the user's current folder. The user can click on a button to switch from this set of collections to her folder-independent personal set of collections. From the list shown in the dialog, the user can highlight one or more collections.

Furthermore, the dialog contains five buttons:

- *Search in selected collections*
  this button is active whenever there is at least one collection highlighted
- *Search in all folder collections*
  this button is active when the dialog shows the list of collections associated to the folder
- *Search in all personal collections*
  this button is active when the dialog shows the list of personal collections
- *Search the whole system*
  this button is always active and results in a search without any collection restriction
- *Cancel*
  this button closes the *Collection selection dialog* without changing the *Query formulation dialog*

If the user had already chosen a query or metadata schema or formulated query conditions in the *Query formulation dialog* before, then the *Collection selection dialog* contains also a warning that switching the collections may result in different metadata schemas being available, i. e. part of the other selections and all the previously formulated query conditions might be dropped.

After clicking one of these buttons, the *Collection selection dialog* is closed, and (except for the *Cancel* button) the selected collections are listed in the *Query formulation dialog*, or the *Query formulation dialog* shows the information that the folder collections, the personal collections, or no collections have been chosen.

4.5.3 Query selection dialog

In this dialog, the user can select an existing query (if there is any).

The dialog contains a list of queries. By default, this is the list of queries that the user has submitted earlier during the same search and browse session, i. e. a query history. The user can click on a button to switch from this set of queries to the set of queries stored in her current folder. From the list shown in the dialog, the user can highlight at most one query. Highlighting a query results in a preview version of the query being shown in the lower part of the dialog window.

Furthermore, the dialog contains the following buttons:

- *Use selected query*
  this button is active whenever there is a query highlighted
- *Cancel*
  this button closes the *Query selection dialog* without changing the *Query formulation dialog*

If the user had already chosen a list of collections or metadata schema or formulated query conditions in the *Query formulation dialog* before, then the *Query selection dialog* contains also a warning that selecting a query may result in different collections being chosen, or in different metadata schemas being available, i. e. part of the other selections and all the previously formulated query conditions might be dropped.

If the *Use selected query button* is clicked, the *Query selection dialog* is closed, and the selected query with its collections, metadata schema, and conditions is shown in the *Query formulation dialog*.

4.5.4 Metadata schema selection dialog

In this dialog, the user can select a metadata schema for the query.

The dialog contains a list of available metadata schemas. From this list, the user can highlight at most one schema. Highlighting a schema results in a second list being shown, containing the schema's attributes. Highlighting an attribute results in information about this attribute being shown, e. g. the type, or a (third) list of subfields.

Furthermore, the dialog contains the following buttons:

- *Use selected schema*
  this button is active whenever there is a schema highlighted
- *Cancel*
  this button closes the *Metadata schema selection dialog* without changing the *Query formulation dialog*

If the user had already chosen a list of collections or formulated query conditions in the *Query formulation dialog* before, then the *Query selection dialog* contains also a warning that selecting a different schema may result in some collections and all the previously formulated query conditions being obsolte.

If the *Use selected schema button* is clicked, the *Query selection dialog* is closed, and the selected schema with the remaining collections (those for which the schema is available) is shown in the *Query formulation dialog*.

4.5.5 Value browsing dialogs

This kind of dialog depends on the type of data to be browsed. A list of person names might for instance first be presented as an overview by first character (A, B, C, ..., Z).

Any *Value browsing dialog*, however, will contain a means to select a value to be used in the current query, and a *Cancel* button to close the dialog without using any value.

4.5.6 Result list

This dialog mainly contains a list of records (the result of a query). The user can choose up to 5 [1] metadata attributes for the records to be shown. (As a query always uses exactly one metadata schema, the list of available attributes is easy to determine.) The user can select one or more records from the list.

The dialog contains the following buttons:

- *Save to current folder*
  this buttons results in the currently selected records being saved to the user's current folder

[1] This number was arbitrarily chosen.

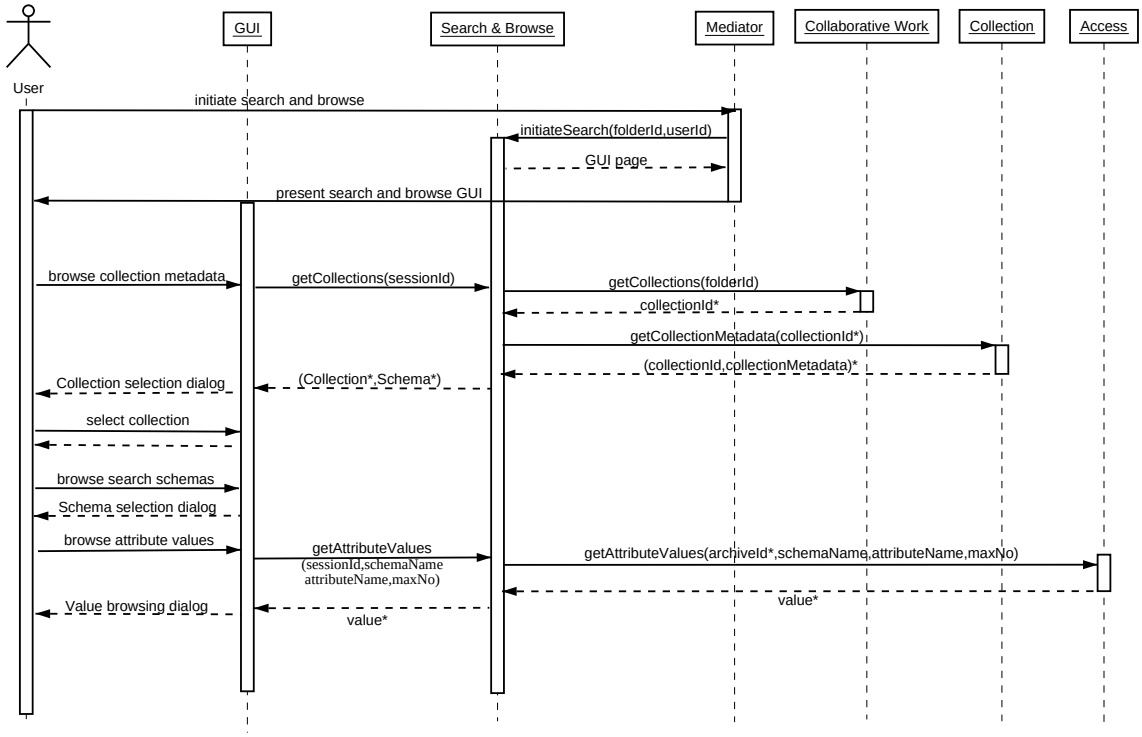

Figure 4.2: Search and Browse Service: select collections and edit query (interaction diagram)

- *Close*
  clicking this button closes the dialog

4.6 Service interaction diagrams

4.7 Service implementation tools

The first prototype of the Search and Browse Service will be implemented using the following software:

- Java 2
- Apache XML-RPC
- Apache Web Server and Tomcat

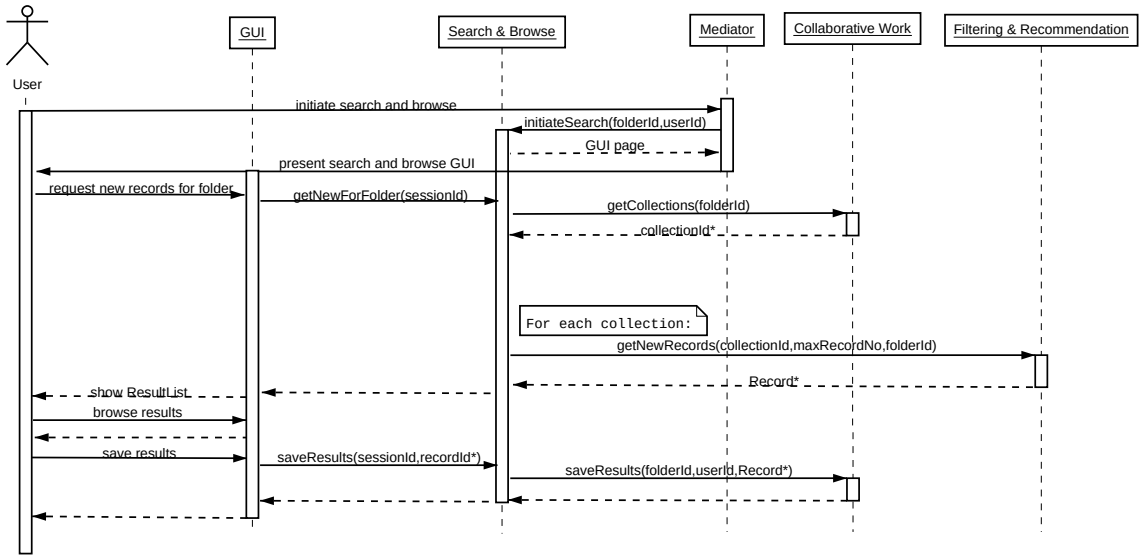

Figure 4.3: Search and Browse Service: get new records relevant to folder topic (interaction diagram)

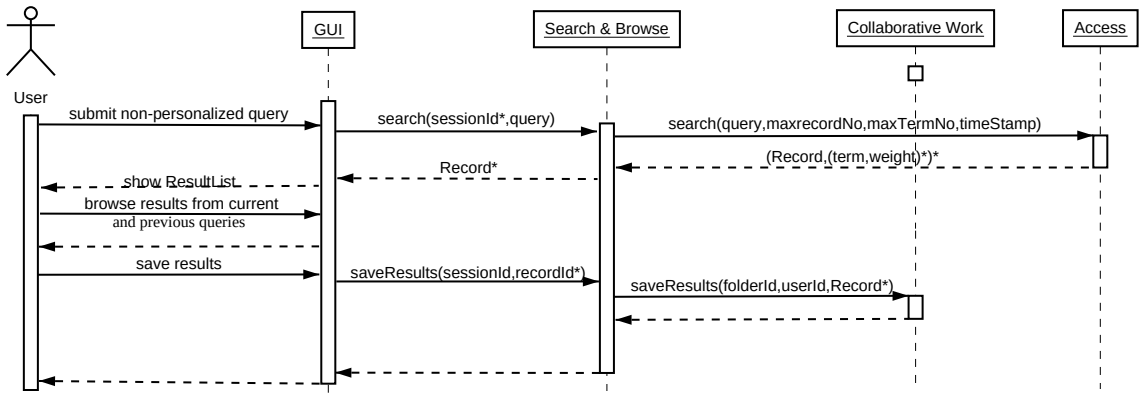

Figure 4.4: Search and Browse Service: submit query without personalization (interaction diagram)

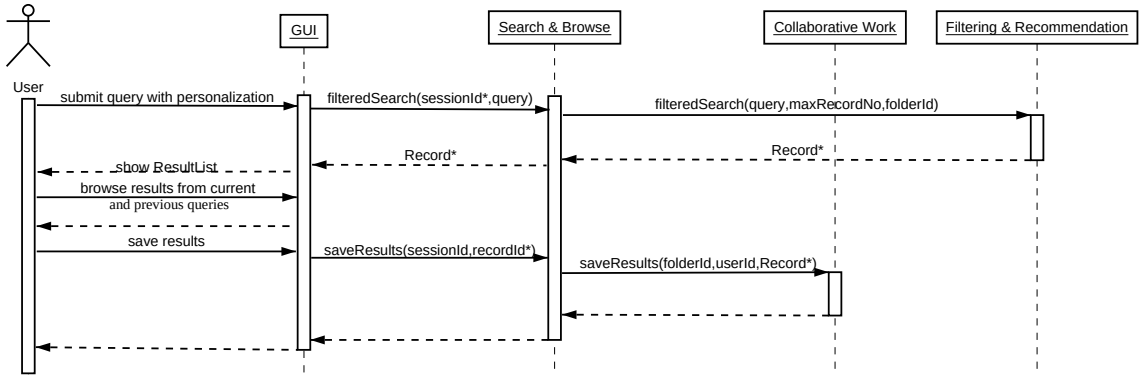

Figure 4.5: Search and Browse Service: submit query with personalization (interaction diagram)

Chapter 5

Filtering and Recommendation Service

5.1 Functionality

The role of the Filtering and Recommendation Service (FRS) is to provide the user with highly flexible and personalized interaction. It is important to notice that this service provides a kind of functionality that is "non-fundamental" to CYCLADES, in the sense that a CYCLADES service of user access to archive records could, in principle, work and be provided without a filtering and recommendation functionality.

Personalization of interaction is here a *content-based* (vs. a form-based) notion. This means that CYCLADES attempts to tailor its behaviour to the user by trying to "understand" the user's interests. This personalization paradigm is thus based on *automatically* generating a "profile" of the user and of her interests (this is done inductively, through a textual and statistical analysis of the documents the user has shown interest in), rather than by asking the user to fill in such a profile. This has obvious advantages, since

- The user is not burdened with the task of filling in a detailed and structured profile of her interests. It is well-known that few users are willing go to the trouble of doing so. Such a profile is instead automatically guessed by the system based on the user's past behaviour.
- Even if the user accepted to fill in such a profile, she would typically not be capable of writing an "effective" profile, i.e. one that achieves effective personalization to her interests. In fact, it is well-known that users do not often display fine-grained self-description capabilities, and it is also well-known (and quite obvious too) that users cannot be expected to know how the profile they are filling in is going to influence the behaviour of the personalization system. Automatically guessing a profile from the user's past behaviour allows instead to obtain fine grained descriptions of the user's interest which are specially tailored to improving document access.

In CYCLADES, personalization is achieved by implementing two basic mechanisms, filtering and recommendation. *Filtering* refers to an activity of personalizing the interaction between user and system based on feedback information provided by the user herself, while *recommendation* refers instead to an activity of personalizing the interaction between user and system based on feedback information provided by other users that the system considers "similar" to this user. A further basic difference between filtering and recommendation is that, while filtering only deals with records (i.e. it consists in estimating whether a given record is or is not interesting to a given user), recommendation deals with records, collections, users, and communities (i.e. it consists in recommending one or more of these to a user if they are deemed interesting to her).

For the Filtering and Recommendation Service, the basic unit of concern is the *folder*, which may be viewed as a thematic repository of records. A folder always has an owner, which may be either an individual user, or a community, or a project. Each folder will typically correspond to one subject (or discipline, or field) the owner is interested in. However, in order to accomplish a truly personalized interaction between user and system, this correspondence is fully idiosyncratic to the owner of the folder; this means that e.g. a folder named `Nuclear Waste Disposal` and owned by user `Sue` will not correspond to any "objective" definition or characterization of what nuclear waste disposal is, but will correspond *to what Sue means by* nuclear waste disposal, i.e. to her personal view of (or interest in) nuclear waste disposal. This user-oriented view of folders is realized by learning the "semantics of folders" from the current contents of the folders themselves.

In the following sections we give detailed use cases for the Filtering and Recommendation Service, along with the algorithms that implement the service. The use cases are:

- Update Folder Profile (see Section 5.1.1), whereby the semantic definition of a folder ("folder profile"), which acts as a filter on what should be retrieved into this folder, is updated by the system.
- Search On-Demand based on Folder Profile (see Section 5.1.2), whereby the folder profile is used as a post-filter on records that are retrieved by (explicit or implicit) user queries;
- Receive Recommendations from the System (see Section 5.1.3), whereby recommendations (of records, collections, users, communities) deemed relevant to a given folder are received into the folder and displayed to the user.

5.1.1 Update Folder Profile (SU 7)

Folder profiles are updated by bringing to bear records that show a shift in the user's interests that are represented by this folder. New records that had originally been retrieved for this folder and have instead been saved by the user in another folder are an indication that the folder profile should be updated to exclude them (i.e. to avoid "requesting" records similar to them in the future). Conversely, new records that have been saved by the user in this folder (and that had possibly been retrieved for another folder) are an indication that the folder profile should be updated to include them (i.e. to "request" also records similar to them in the future).

The modification of a folder profile may be explicitly requested by a user (Update Folder Profile On-Demand – see Section 5.2.1) or may be invoked by the system, typically at regular intervals (**Update Folder Profile at Scheduled Time** – see Section 5.2.2). The former is performed for a single folder profile, while the latter is performed at the same time for all of the folder profiles of a given user.

5.1.2 Search On-Demand based on Folder Profile (SU 9)

This searching modality is based on a folder profile, i.e. a compact representation of the interests of the user relative to this folder. Search On-Demand based on Folder Profile is achieved by using the folder profile as a post-filter on records that are retrieved by a query. The query may be explicit, i.e. issued by the user, or may be implicit, i.e. issued by the system; this latter case occurs when the user, instead of issuing an explicit query, simply requests that all new records that are relevant to the topic corresponding to the folder profile are retrieved.

5.1.3 Receive Recommendation from the System (SU 10)

Recommendations of entities (i.e. records, collections, users, communities) are issued to users based on other users' (implicit or explicit) ratings of records within their own folders, and on the perceived similarity between the interests of the user, as represented by a given folder, and the interests of

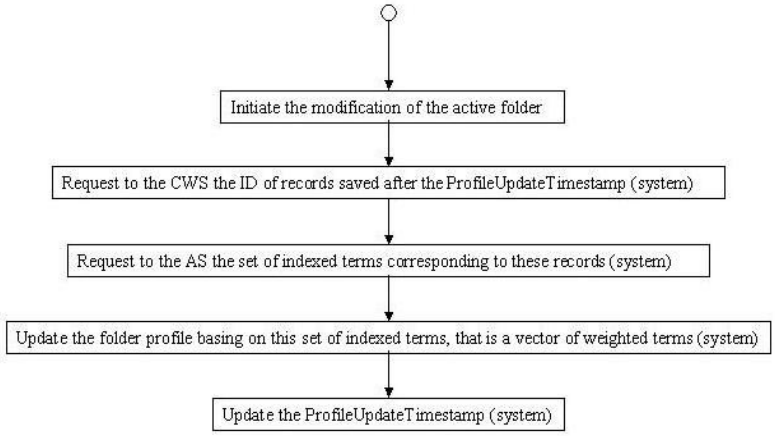

Figure 5.1: Process flow diagram of the Update Folder Profile On-Demand use case.

these other users, as represented by their folders. All recommendations are specific to a given user folder (the *current folder*); this means that the recommendation of a given entity to a user always pertains to a given folder, and has to be understood in the context not of the general interests of the user, but of the specific interests of the user represented by that folder.

The user may choose whether she wants to receive recommendations automatically (*"scheduled recommendations"*), or whether she wants to receive them upon an explicit request (*"on-demand recommendations"*).

5.2 Process flow

The process flow of the Filtering & Recommendation Service is organized around three basic use cases (which are further specialized into more detailed use cases): Update Folder Profile (SU 7), Search On-Demand based on Folder Profile (SU 9), and Receive Recommendation from the System (SU 10).

The Update Folder Profile use case actually consists of two different use cases, Update Folder Profile On-Demand (discussed in Section 5.2.1) and Update Folder Profile at Scheduled Time (discussed in Section 5.2.2).

The Search On-Demand based on Folder Profile use case actually consists of two different use cases, **Search On-Demand based on Folder Profile** (discussed in Section 5.2.2) and **Search On-Demand based on Folder Profile** (discussed in Section 5.2.2).

The Receive Recommendation from the System use case actually consists of four different use cases, depending on the object of the recommendation: records, collections, users, and communities (note that projects cannot be recommended, since participation in them is more restricted than participation in communities). These four use cases are discussed in the next sections from 5.2.4 to 5.2.7.

5.2.1 Update Folder Profile On-Demand (SU 7-1)

The use case consists of the following sequence (see also Figure 5.1):

- The user decides to request the modification of the profile for the current folder. She does so by hitting the Modify current folder profile button.

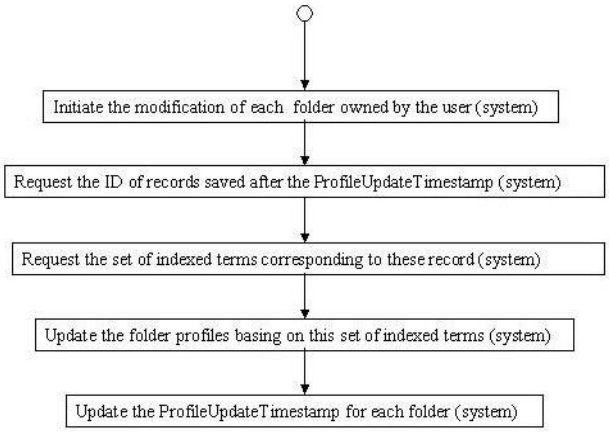

Figure 5.2: Process flow diagram of the Update Folder Profile at Scheduled Time use case.

- The Filtering & Recommendation Service requests to the Collaborative Work Service the set of record IDs that have been saved into the folder after the last time this folder profile has been updated; this time is indicated by the `ProfileUpdateTimestamp`, one for each folder, maintained by the FRS.
- After receiving this set of record IDs, the Filtering & Recommendation Service requests to the Access Service the set of indexed terms corresponding to these record IDs (i.e. a vector of weighted terms for each record).
- After receiving this set of indexed terms, the Filtering & Recommendation Service updates the folder profile and then updates the `ProfileUpdateTimestamp` for the folder profile.

This use case is also described by the interaction diagram of Figure 5.11.

5.2.2 Update Folder Profile at Scheduled Time (SU 7-2)

The use case consists of the following sequence (see also Figure 5.2):

- At the scheduled time, for each folder associated to the user, the Filtering & Recommendation Service requests to the Collaborative Work Service the set of record IDs that have been saved into the folder after the last time this folder profile has been updated; this is indicated by the `ProfileUpdateTimestamp`, one for each folder, maintained by the FRS.
- After receiving this set of record IDs, the Filtering & Recommendation Service requests to the Access Service the set of indexed terms corresponding to these record IDs (i.e. a vector of weighted terms for each record).
- After receiving these sets of indexed terms, the Filtering & Recommendation Service updates the folder profiles and then updates the `ProfileUpdateTimestamp` for each folder profile.

This use case is also described by the interaction diagram of Figure 5.12.

5.2.3 Search On-Demand based on Folder Profile (SU 9)

The use case consists of the following two sequences (see also Figure 5.3 and Figure 5.4):

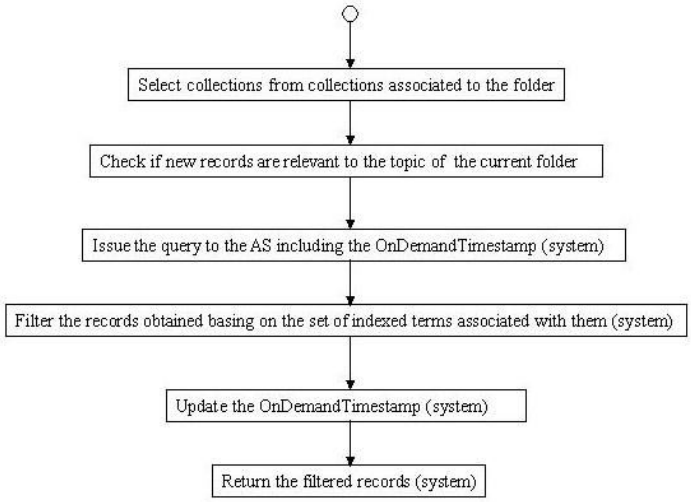

Figure 5.3: Process flow diagram of the Search On-Demand based on Folder Profile use case.

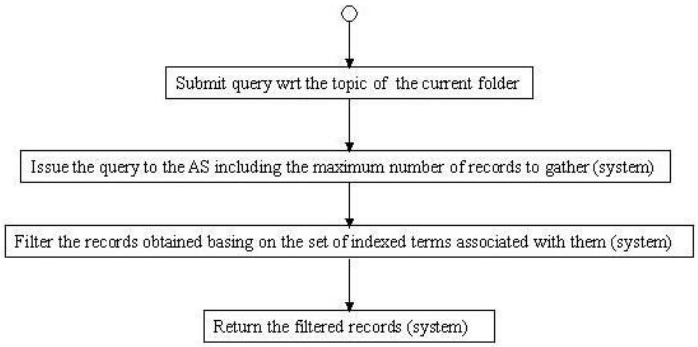

Figure 5.4: Process flow diagram of the Search On-Demand based on Folder Profile use case.

Personalized on-demand searching:

- Select collections (from collections associated to the folder) (as in non-personalized search);
- Instead of explicitly issuing a query, the user may simply ask the system to check whether new records relevant to the topic of the current folder have been gathered for this collection since the user last checked it (*personalized on-demand searching*). In order to do so, she hits the button get new records relevant to folder topic and then hits the enter key. The Search & Browse Service sends the request to the Filtering & Recommendation Service, indicating the maximum number of documents to be returned. The Filtering & Recommendation Service issues the query to the Access Service, including the `OnDemandTimestamp` relative to the current folder (which specifies the time of the last such query issued from this folder);
- the Access Service returns the set of records requested, together with an internal representation for each of them consisting of a set of indexed terms;
- the Filtering & Recommendation Service filters the records received from the Access Service (using the indexed terms), updates the `OnDemandTimestamp` relative to the current folder, returns the filtered records to the Search & Browse Service, and moves to Step 5.2.3;

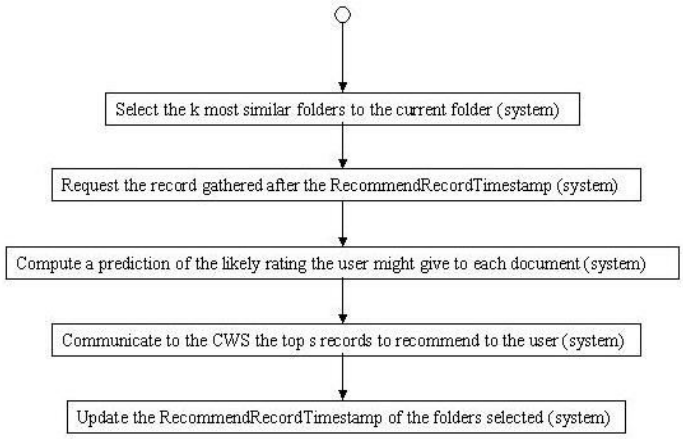

Figure 5.5: Process flow diagram of the Receive Record Recommendation from the System use case.

Personalized ad-hoc searching:

- [Edit query (as in non-personalized search)];
- Instead of plainly submitting the query the user may, if she wishes, submit the query with respect to the folder topic (*personalized ad-hoc searching*). In this case, she formulates their query and then hits the button Submit personalised. The Search & Browse Service sends this query to the Filtering & Recommendation Service, indicating the maximum number of documents to be returned. The Filtering & Recommendation Service issues the query to the Access Service, which returns the set of records requested, together with an internal representation for each of them consisting of a set of indexed terms;
- the Filtering & Recommendation Service filters the records received from the Access Service (using the indexed terms), returns the filtered records to the Search & Browse Service, and moves to Step 5.2.3;
- [Look at results (from history/folder) (as in non-personalized search)].

This use case is also described by the interaction diagram of Figure 5.13.

5.2.4 Receive Record Recommendation from the System (SU 10-1)

The use case consists of the following sequence (see also Figure 5.5):

- The Filtering & Recommendation Service selects the k most similar folders to the current folder. This selection activity uses both folder profiles, which are persistently stored by the Filtering & Recommendation System, and "folder rating profiles" (i.e. normalized vectors of the ratings given by a user within the folder), which are also persistently stored by the Filtering & Recommendation System.
- For each of the k folders selected in the previous step, the Filtering & Recommendation Service requests to the Collaborative Work Service the records contained in the folder that have been retrieved after `RecommendRecordTimestamp`, a timestamp maintained by the Filtering & Recommendation Service that records the last date in which this folder was selected for recommending records to the current folder (`RecommendRecordTimestamp` is thus an attribute of a pair of folders, and not of a single folder).

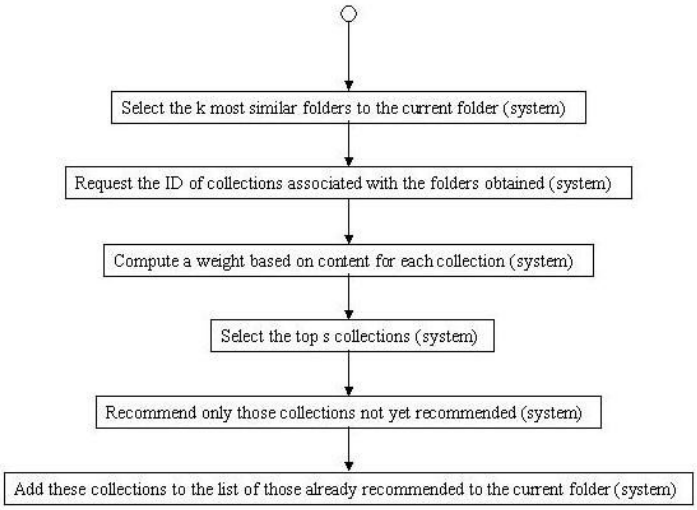

Figure 5.6: Process flow diagram of the Receive Collection Recommendation from the System use case.

- For each of the records returned, the Filtering & Recommendation Service computes a prediction of the likely rating that the user might give to the document. This computation is based on the record content and on a comparison between the rating patterns of the user and those of the users who actually rated the record positively.
- The Filtering & Recommendation Service selects the s "top" documents (i.e. the ones with the highest predicted rating), where s is determined according to some threshold policy, communicates these recommendations to the Collaborative Work Service, and updates the `RecommendRecordTimestamp` of the k selected folders with respect to the current folder.

This use case is also described by the interaction diagram of Figure 5.14.

5.2.5 Receive Collection Recommendation from the System (SU 10-2)

The use case consists of the following sequence (see also Figure 5.6):

- The Filtering & Recommendation Service selects the k most similar folders to the current folder, in the way specified for the content similarity for SU 10-1 (see 5.4), and requests to the Collaborative Work Service the IDs of the collections that are associated to these folders.
- Among the collections returned by the Collaborative Work Service, the Filtering & Recommendation Service selects the s (fixed) "top" collections according to a quality criterion that prefers those collections that are referred to by the highest amount of selected folders.
- Among the s pre-selected collections, the Filtering & Recommendation Service selects the $s' \leq s$ collections that have not yet been recommended to the current folder, checking against a list of collections already recommended to the current folder which is maintained by the Filtering & Recommendation Service.
- The Filtering & Recommendation Service communicates the recommendations for these s' collections to the Collaborative Work Service, and adds these s' collections to the list of collections already recommended to the current folder.

This use case is also described by the interaction diagram of Figure 5.15.

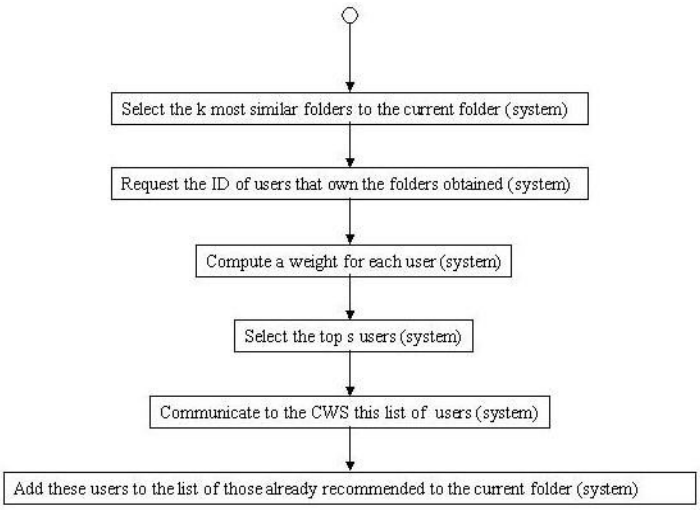

Figure 5.7: Process flow diagram of the Receive User Recommendation from the System use case.

5.2.6 Receive User Recommendation from the System (SU 10-3)

The use case consists of the following sequence (see also Figure 5.7):

- The Filtering & Recommendation Service selects the k most similar folders to the current folder, in the way specified for the content similarity for SU 10-1 (see 5.4), and requests to the Collaborative Work Service the IDs of the owners of these folders.
- Among the users returned by the Collaborative Work Service, the Filtering & Recommendation Service selects the s (fixed) "top" users according to a quality criterion that prefers the users who own the highest amount of selected folders.
- Among the s pre-selected users, the Filtering & Recommendation Service selects the $s' \leq s$ users that have not yet been recommended to the current folder, checking against a list of users already recommended to the current folder which is maintained by the Filtering & Recommendation Service.
- The Filtering & Recommendation Service communicates the recommendations for these s' users to the Collaborative Work Service.
- The Collaborative Work Service selects from these pre-selected s' users the $s'' \leq s$ users who have declared their willingness to be recommended to other users, and recommends them to the current folder. In the meantime, the Filtering & Recommendation Service adds these s' users to the list of users already recommended to the current folder.

This use case is also described by the interaction diagram of Figure 5.16.

5.2.7 Receive Community Recommendation from the System (SU 10-4)

The use case consists of the following sequence (see also Figure 5.8):

- The Filtering & Recommendation Service selects the k most similar community folders (i.e. folders owned by a community) to the current folder, in the way specified for the content similarity for SU 10-1 (see 5.4), and requests to the Collaborative Work Service the IDs of the communities that are associated to these folders.

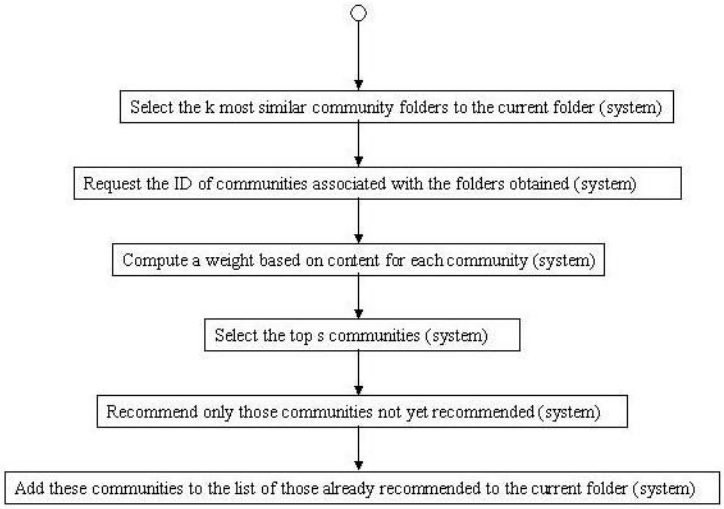

Figure 5.8: Process flow diagram of the Receive Community Recommendation from the System use case.

- Among the communities returned by the Collaborative Work Service, the Filtering & Recommendation Service selects the s (fixed) "top" communities according to a quality criterion that prefers the communities who own the highest amount of selected folders.
- Among the s pre-selected collections, the Filtering & Recommendation Service selects the $s' \leq s$ communities that have not yet been recommended to the current folder, checking against a list of communities already recommended to the current folder which is maintained by the Filtering & Recommendation Service.
- The Filtering & Recommendation Service communicates the recommendations for these s' communities to the Collaborative Work Service, and adds these s' communities to the list of communities already recommended to the current folder.

This use case is also described by the interaction diagram of Figure 5.17.

5.3 Internal architecture

The Filtering & Recommendation Service provides functionality via API to the other CYCLADES services. All the interaction of the users on the graphical user interfaces of other services are mediated by the Mediator Service. The internal components of the Filtering & Recommendation Service are shown in Figure 5.9:

- the FolderDB is the folder profile database where both folder profiles and folder profile ratings are stored;
- the Filter module performs the filtered search within personalization;
- the Recommender is the module that performs the record, collection, user or community recommendation to a specific folder, if requested;
- the Profile Management module updates the folder profile performing the UpdateFolderProfile method, called by the user (**Update Folder Profile On-Demand**) or activated by the system at a certain time (Update Folder Profile at Scheduled Time);

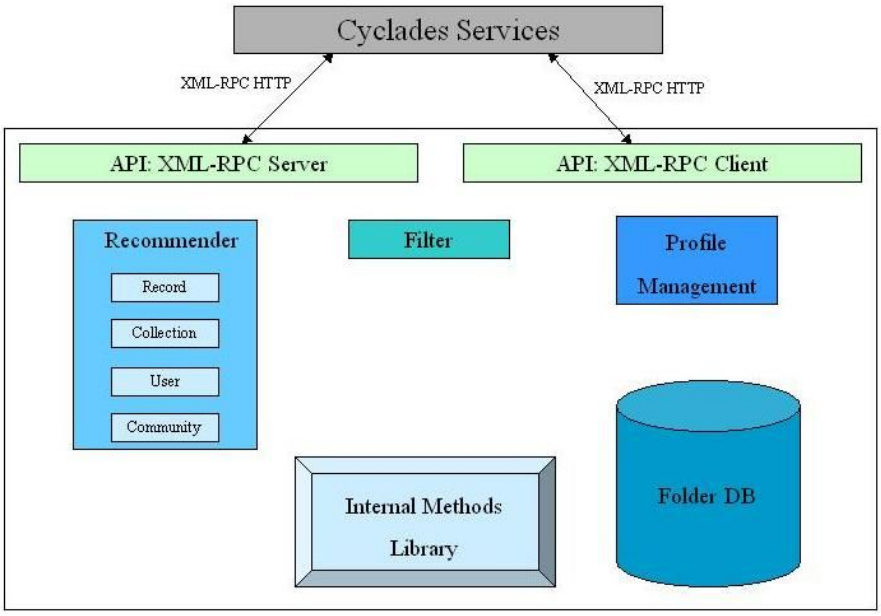

Figure 5.9: Internal architecture of the Filtering & Recommendation Service.

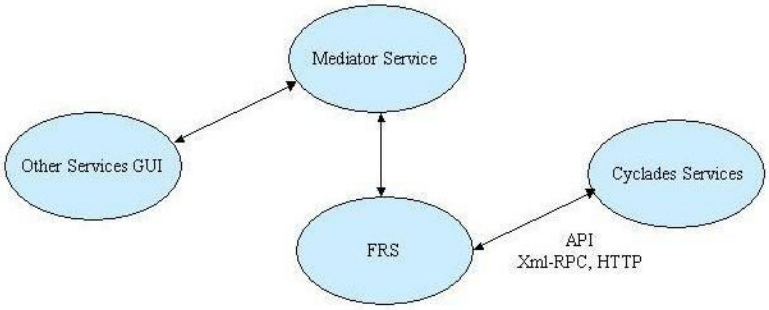

Figure 5.10: Interactions between the Filtering & Recommendation Service and the other CYCLADES services.

- the Internal Methods Library module contains all methods used/performed internally by the Filtering & Recommendation Service;
- the Client and Server module contain all Filtering & Recommendation Service's API, that allows other services to access the functionalities of the Filtering & Recommendation Service and viceversa.

Other services can directly access the Filtering & Recommendation Service functionality via the Filtering & Recommendation Service's API. The API communication takes place via HTTP and the format is XML and XML-RPC (see Figure 5.10).

5.4 Data and method specification

5.4.1 The Filtering & Recommendation Service classes

The Filtering & Recommendation Service has a class `FilteringRecommendationService` and a class `FolderProfile`. The persistent data of this service are the folder profiles.

The methods listed below, which constitute the API of the Filtering & Recommendation Service, may be called from other services using the CYCLADES inter-service communication protocol XML-RPC.

FilteringRecommendationService

- id
  Description: this is the unique ID of the class

- folderProfileList
  Description: this is the list of all folder profiles known to the Filtering and Recommendation Service

- Record* filteredSearch(query,maxRecordNo,folderId)
  Description: this method may be invoked in order to filter records, retrieved according to a query, with respect to the profile learnt from the folder.
  Input. query: the query according to the syntax specified by the access service
  maxRecordNo: maximal number of records to be retrieved
  folderId: the folder ID with respect to which filtering should be performed
  Output. list of records

- Record* getNewRecords(collectionId,maxRecordNo,folderId)
  Description: this method may be invoked in order to get new records, retrieved with respect to the profile learnt from the folder.
  Input. collectionID: a collection ID
  maxRecordNo: maximal number of records to be retrieved
  folderId: the folder ID with respect to which filtering should be performed
  Output. list of records

- void updateFolderProfile(folderId)
  Description: this method may be invoked in order to update the folder profile, i.e. to learn the folder profile.
  Input. folderId: the folder ID for which to learn the profile

- void setRecommendationYesNo(folderId,value)
  Description: this method may be invoked in order to activate/disactivate the production of recommendations with respect to a folder
  Input. folderId: the folder ID
  value: an integer.
  The value contains the bit encoded recommendation preferences of the folder:
  - a zero value stands for no recommendations;
  - bit 0 stands for record recommendations,
  - bit 1 stands for user recommendations,
  - bit 2 stands for collection recommendations,
  - bit 3 stands for community recommendations.

FolderProfile

- id
  Description: this is the unique ID of the class

- profile
  Description: this is the folder profile computed taking into account folder content and user ratings

- recommendedRecordsTimeStamp
  Description: this is a hash table, that for a give folder maintains a timestamp which is the most recent time where a folder F has been considered among the top k similar folders to this current one. The purpose is to avoid duplicated record recommendations

- recommendedCollections
  Description: list of recommended collections for avoiding duplicated recommendation

- recommendedUsers
  Description: list of recommended users for avoiding duplicated recommendation

- recommendedCommunities
  Description: list of recommended Communities for avoiding duplicated recommendation

- onDemandTimeStampList
  Description: this is a list of pairs (owner, onDemandTimeStamp). We consider a list, since in the case of a community folder more than one user may have access to this folder.

- profileUpdateTimeStamp
  Description: time and date of last profile update, related to the content of the folder.

- ratingUpdateTimeStamp
  Description: time and date of last rating update, related to the ratings of the folder.

5.4.2 The internal Filtering & Recommendation Service methods

In the following we describe the methods used internally in the Filtering & Recommendation Service, just those already individualized.

- void updateProfileTimestamp(folderId)
  Description: this method is performed in order to update the Timestamp, both in case of on-demand and scheduled folder profile update.
  Input. folderId: the folder ID

- void updateRatingTimestamp(folderId)
  Description: this method is invoked in order to update the RatingTimestamp, in case of scheduled folder profile update.
  Input. folderId: the folder ID

- void updateOnDemandTimestamp(folderId)
  Description: this method is invoked in order to update the OnDemandTimestamp with respect to the folder profile.
  Input. folderId: the folderId

- void Filter(folderId,(record,(term,weight)*)*)
  Description: this method is invoked in order to filter new records, with respect to the folderId.
  Input. folderId: the folderId
  (record,(term,weight)*)*: a list of records and the pairs (term, weight) associated

- (folderId, simValue)* selectSimilarFolders(folderId)
  Description: this method will be performed in order to select the k most similar folders to the current folder, given as input, in case of recommendations.
  Input. folderId: the folderId
  Output. list of pairs (folderId,simValue)

- (Id, predictValue)* computePrediction(folderId, recommValue)
  Description: this method is invoked in order to compute a prediction of the likely rating the user might give to the object to recommend (records, users, collections, communities).
  Input. folderId: the folderId
  recommValue: the object to evaluate (record, user, community, collection)
  Output. list of pairs (Id, predictValue)

- Id* selectTop(Id, predictValue)*
  Description: this method is performed in order to select only the top s objects to recommend.
  Input. List of recordId: the Id of the object (record, user, community, collection)
  predictValue: the correspondent prediction value
  Output. list of object identifiers

- void updateRecommRecordsTimestamp(folderId)
  Description: this method is invoked in order to update the hash table recommendedRecordsTimestamp with respect to the last record recommendation.
  Input. folderId: the folderId

- void updateRecommendedCollections(folderId,collectionId*)
  Description: this method is invoked in order to update the recommendedCollections list with respect to the last collections recommendation.

- void updateRecommendedUsers(folderId,userId*)
  Description: this method is invoked in order to update the recommendedUsers list with respect to the last users recommendation.

- void updateRecommendedCommunities(folderId,communityId*)
  Description: this method is invoked in order to update the RecommendedCommunities list with respect to the last communities recommendation.

5.4.3 The Filtering & Recommendation Service algorithms

Before giving the detailed description of the algorithms, we fix some notation. By t_k, d_j, and f_i we will denote a term, a document, and a folder, respectively. Terms are usually identified either with the words, or with the stems of words, occurring in documents. A document d_j is represented as a vector of *weights* $d_j = \langle w_{1j}, \ldots, w_{mj} \rangle$, where $0 \leq w_{kj} \leq 1$ corresponds to the "importance value" that term t_k has in document d_j, and m is the total number of terms occurring in at least one "training document" (i.e. a document which the user has saved in some folder).

The folder profile for folder f_i is computed as the *centroid* of the documents belonging to f_i; this means that the profile of f_i may be seen as a document itself (i.e. the mean, or prototypical, document of f_i). The weights of the folder profile for f_i are then computed as

$$w_{ki} = \frac{1}{|\{d_j \in f_i\}|} \cdot \sum_{\{d_j \in f_i\}} w_{kj} \tag{5.1}$$

The centroid of a folder f_i will be denoted by $\breve{f}_i$.

We define the *content similarity* of two documents d_1 and d_2, written $CSim(d_1, d_2)$, as the *cosine* of the angle that separates the vectors representing d_1 and d_2, i.e.

	d_1	...	d_j	...	d_n
f_1	r_{11}	...	r_{1j}	...	r_{1n}
f_2	r_{21}	...	r_{2j}	...	r_{2n}
...	...	...	...	...	...
f_i	r_{i1}	...	r_{ij}	...	r_{in}
...	...	...	...	...	...
f_m	r_{m1}	...	r_{mj}	...	r_{mn}

Table 5.1: The folder-document rating matrix.

$$CSim(d_1, d_2) = \frac{\sum_{k=1}^{m} w_{k1} \cdot w_{k2}}{\sqrt{\sum_{k=1}^{m} w_{k1}^2} \cdot \sqrt{\sum_{k=1}^{m} w_{k2}^2}} \tag{5.2}$$

This formula also allows us to determine the content similarity of a document d_j and a folder profile $\breve{f}_i$, since this latter is, mathematically speaking, also a document.

Given a folder f_i, a document $d_j \in f_i$ and an user u_k, by $0 \leq r_{ijk} \leq 1$ we denote the *rating* given by user u_k to document d_j relative to folder f_i. We average out the ratings given by users relative to the same document-folder pair, by defining r_{ij} as $r_{ij} = \frac{1}{U_{ij}} \cdot \sum_{k=1}^{U_{ij}} r_{ijk}$, where U_{ij} is the number of users for which the rating r_{ijk} is defined. As a consequence, we may represent the ratings as a 2-dimensional matrix, where the rows represent folders and the columns represent documents, as shown in Table 5.1.

We define the *rating similarity* of two folders f_1 and f_2 with respect to a set of documents $\overline{D}$, written $RSim_{\overline{D}}(f_1, f_2)$, as the *Pearson correlation coefficient* of the ratings given in the folders f_1 and f_2 for the documents $d \in D$, i.e.

$$RSim_{\overline{D}}(f_1, f_2) = \frac{\sum_{d_j \in D}(r_{1j} - \overline{r}_1) \cdot (r_{2j} - \overline{r}_2)}{\sigma_1 \cdot \sigma_2} \tag{5.3}$$

where $\overline{r}_i$ is the mean of the ratings $r_{i1}, \ldots, r_{in}$, and σ_i is their standard deviation (see also Table 5.1).

In what follows, the *similarity* $Sim(f_1, f_2)$ between two generic folders f_1 and f_2 will be determined as a linear combination between their content similarity and their rating similarity, i.e.

$$Sim(f_1, f_2) = \alpha^C \cdot CSim(\breve{f}_1, \breve{f}_2) + \alpha^R \cdot RSim_{\overline{D}}(f_1, f_2) \tag{5.4}$$

The linear combination parameters α^C and α^R are to be determined experimentally, according to a policy yet to be defined. From now on, we will simply speak of the similarity of two folders to actually mean the similarity of their centroids.

5.4.4 Detailed algorithm specification of Receive Record Recommendation from the System (SU 10-1)

In order to recommend records to the current folder, a pool of records candidate for recommendation is first assembled. Then, for each of these records, a *recommendation score* is computed, representing the likelihood that the user will deem this record relevant to the current folder. Only the top scoring records are actually recommended.

1. **Select the k most similar folders.**

   The Filtering & Recommendation Service selects the set $MS_k(f_c)$ of the k most similar folders to the current folder f_c, by computing the similarity $Sim(f_c, f_j)$ between the current

folder f_c and each other folder f_j, and then selecting the k folders with the highest similarity value. How to determine the optimal value of k (i.e. whether to use a fixed or variable value of k, and how to determine this value) will be left to experimentation.

2. **Compute the prediction of the document rating.**

   The Filtering & Recommendation Service uses the set $MS_k(f_c)$, selected in the previous step, to compute a prediction of the likely rating that the user (i.e. the owner of the current folder) might give to a document. For each folder in $MS_k(f_c)$, the Filtering & Recommendation Service requests to the Collaborative Work Service the records contained in the folder that have been gathered after the time specified in `RecommendRecordTimestamp`. The Collaborative Work Service returns to the Filtering & Recommendation Service the list of these records. This is the *pool of documents*, P_D, that the Filtering & Recommendation Service has to rate.

   The recommendation score for d_j wrt f_c is computed as a linear combination of a *content-based recommendation score* and a *ratings-based recommendation score*.

   The content similarity among document d_j and the profile $\breve{f}_c$ of the current folder f_c is (see 5.2):

$$p_{cj}^C = CSim(\breve{f}_c, d_j) \tag{5.5}$$

   while the prediction for the document $d_j \in P_D$ with respect to the current folder f_c based on ratings is (see 5.3):

$$p_{cj}^R = \overline{r}_c + \frac{\sum_{f_u \in MS_k(f_c)} (r_{uj} - \overline{r}_u) \cdot RSim_{\overline{D}}(f_c, f_u)}{\sum_{f_u \in MS_k(f_c)} \cdot RSim_{\overline{D}}(f_c, f_u)} \tag{5.6}$$

   where $\overline{r}_c$ is the mean of the ratings in the current folder and $\overline{r}_u, f_u \in MS_k(f_c)$, is the mean of the ratings in its similar folder, i.e. the mean of $\{r_{iu} : f_i \in MS_k(f_c), 1 \leq i \leq m\}$. Note that if $d_j \notin f_u$, then r_{uj} is not defined and the correspondent term in the sum must be omitted.

   For each record $d_j \in P_D$ the *recommendation score* p_{cj} is computed as a linear combination of the prediction based on content and the prediction based on ratings:

$$p_{cj} = \beta^C \cdot p_{cj}^C + \beta^R \cdot p_{cj}^R \tag{5.7}$$

3. **Select the records to recommend**

   In the previous step, the Filtering & Recommendation Service obtained a ranked list based on the final prediction p_{cj} computed for each document $d_j \in P_D$. The System selects from this list of records the top s[1] to recommend to the current folder.

5.4.5 Detailed algorithm specification of Receive Collection Recommendation from the System (SU 10-2)

1. **Select the k most similar folders.**

   The Filtering & Recommendation Service computes the $Sim(f_c, f_j)$ between the current folder f_c and each other folder f_j with respect to the similarity function specified in 5.4, and the result is a ranked list of similar folders to the current folder, from which the System selects the top k folders, $MS_k(f_c)$.

   For each of these k folders in $MS_k(f_c)$, the Filtering & Recommendation Service requests to the Collaborative Work Service the IDs of the collections that are associated to it. The Collaborative Work Service sends to the Filtering & Recommendation Service the list of

[1] s is fixed. Another possible way to proceed is based on a threshold: in this case only those records whose prediction value is greater than the threshold value would be recommended to the current folder.

collections IDs, a *pool of collections*, P_C, that the Filtering & Recommendation Service has to rate.

2. **Compute the prediction of the collection rating.**

   The Filtering & Recommendation Service has to compute a prediction of the likely rating that the user, the owner of the current folder, might give to each collection selected in the previous step. For each $f_j \in MS_k(f_c)$ and for each $\xi_i \in P_C$, let is $f_j(\xi_i) = 1$ if the collection ξ_i is associated to the folder f_j, 0 otherwise; $h_i^{\xi} = |\{f_j : f_j \in MS_k(f_c), f_j(\xi_i) = 1\}|$, i.e. the number of folders associated to ξ_i (*hits*).

   For each collection $\xi_i \in P_C$ the final prediction p_{ci} is computed as follow:

$$p_{ci} = h_i^{\xi} \cdot \sum_{f_j \in MS_k(f_c)} Sim(f_c, f_j) \tag{5.8}$$

3. **Select the collections to recommend.**

   In the previous step, the Filtering & Recommendation Service obtained a ranked list based on the final prediction p_{ci} computed for each collection $\xi_i \in P_C$. The System selects from this list of collections the top s to recommend, and recommends those ones not yet recommended, checking against the list of collections already recommended to the current folder, maintained by the Filtering & Recommendation Service.

5.4.6 Detailed algorithm specification of Receive User Recommendation from the System (SU 10-3)

1. **Select the k most similar folders.**

   The Filtering & Recommendation Service computes the $Sim(f_c, f_j)$ between the current folder f_c and each other folder f_j with respect to the similarity function based on content specified in 5.4, and the result is a ranked list of similar folders to the current folder, from which the System selects the top k folders, $MS_k(f_c)$. For each of these k folders in $MS_k(f_c)$, the Filtering & Recommendation Service requests to the Collaborative Work Service the IDs of the owners of these folders. The Collaborative Work Service sends to the Filtering & Recommendation Service the list of users, a *pool of users*, P_U, that the Filtering & Recommendation Service has to rate.

2. **Compute the prediction of the user rating.**

   The Filtering & Recommendation Service computes a prediction of the likely rating that the user, the owner of the current folder, might give to each user selected in the previous step. For each $f_j \in MS_k(f_c)$ and for each $u_i \in P_U$, let is $f_j(u_i) = 1$ if the folder $f_j \in u_i$, 0 otherwise; $h_i^u = |\{f_j : f_j \in MS_k(f_c), f_j(u_i) = 1\}|$, i.e. the number of folders belonging to the selected user (*hits*).

   For each user $u_i \in P_U$ the final prediction p_{ci} is computed as follow:

$$p_{ci} = h_i^{u} \cdot \sum_{f_j \in MS_k(f_c)} Sim(f_c, f_j) \tag{5.9}$$

3. **Select the users to recommend.**

   In the previous step, the Filtering & Recommendation Service obtained a ranked list based on the final prediction p_{ci} computed for each user $u_i \in P_U$. The System selects from this list of users the top s to recommend, and recommends those ones not yet recommended, checking against the list of users already recommended to the current folder, maintained by the Filtering & Recommendation Service.

5.4.7 Detailed algorithm specification of Receive Community Recommendation from the System (SU 10-4)

1. **Select the k most similar folders.**

   The Filtering & Recommendation Service computes the $Sim(f_c, f_j)$ between the current folder f_c and each other community folder f_j, with respect to the similarity function specified in 5.4, and the result is a ranked list of similar folders to the current folder. The System selects the top k folders in this list, $MS_k(f_c)$.

   For each of these k folders in $MS_k(f_c)$, the Filtering & Recommendation Service requests to the Collaborative Work Service the community, which the folder belongs to. The Collaborative Work Service sends to the Filtering & Recommendation Service the list of communities, a *pool of communities*, $P_{\overline{C}}$, that the Filtering & Recommendation Service has to rate.

2. **Compute the prediction of the community rating.**

   The Filtering & Recommendation Service computes a prediction of the likely rating that the user, the owner of the current folder, might give to each community selected in the previous step. For each $f_j \in MS_k(f_c)$ and for each $\gamma_i \in P_{\overline{C}}$, let is $f_j(\gamma_i) = 1$ if the folder $f_j \in$ community γ_i, 0 otherwise; $h_i^{\gamma} = |\{f_j : f_j \in MS_k(f_c), f_j(\gamma_i) = 1\}|$, i.e. the number of folders $\in MS_k(f_c)$ associated to the selected community (*hits*).

   For each community $\gamma_i \in P_{\overline{C}}$ the final prediction p_{ci} is computed as follow:

   $$p_{ci} = h_i^{\gamma} \cdot \sum_{f_j \in MS_k(f_c)} Sim(f_c, f_j) \tag{5.10}$$

3. **Select the community to recommend.**

   In the previous step, the Filtering & Recommendation Service obtained a ranked list based on the final prediction p_{tc} computed for each community $\gamma_i \in P_{\overline{C}}$. The System selects from this list of communities the top s to recommend, and recommends those ones not yet recommended, checking against the list of communities already recommended to the current folder, maintained by the Filtering & Recommendation Service.

5.4.8 FRS database schema

Several tables are maintained within the FRS. These are listed below.

folder: used to store folder data

Field	Type	Key	Description
folderID	varchar	PRI	folderID of f_i
userID	varchar		userID of owner of f_i
type	varchar	PRI	type of f_i: 0 private, 1 community, 2 project
recommendYesNo	int	PRI	value stored by the setRecommendationYesNo method
recommendedRecordsTimeStamp	date		
profileUpdateTimeStamp	date		
ratingUpdateTimeStamp	date		

folderProfile: used to represent folder profiles

Field	Type	Key	Description
term	varchar	PRI	index term t_k
folderID	varchar	PRI	folderID of f_i
weight	real		weight w_{ki}

folderAuxC: used to store folder collections

Field	Type	Key	Description
folderID	varchar	PRI	folderID of f_i
collectionID	varchar		collectionID collection associated to f_i

folderAuxRI: used to store recommended Items

Field	Type	Key	Description
folderID	varchar	PRI	folderID of f_i
recommendedItem	varchar		recommended Item
type	varchar		type of item, 0 record, 1 user, 2 collection , 3 community

folderAuxOnDemand:

Field	Type	Key	Description
folderID	varchar	PRI	folderID of f_i
userID	varchar		userID of owner of f_i
onDemandTimeStamp	date		

ratings: used to represent ratings

Field	Type	Key	Description
recordID	varchar	PRI	recordID of d_j
folderID	varchar	PRI	folderID of f_i
userID	varchar		userID of u_k
rating	int		rating r_{ijk}

ratingsAux: auxiliary table to manage ratings

Field	Type	Key	Description
recordID	varchar	PRI	recordID of d_j
folderID	varchar	PRI	folderID of f_i
sum	int		value of $\sum_{k=1}^{U_{ij}} r_{ijk}$
numRatings	int		value of U_{ij}

Note that $r_{ij} = \frac{ratingsAux.sum}{ratingsAux.numRatings}$

ratingsProfile: the rating matrix

Field	Type	Key	Description
recordID	varchar	PRI	recordID of d_j
folderID	varchar	PRI	folderID of f_i
rating	real		rating r_{ij}

ratingsProfileAux: auxiliary table to compute prediction

Field	Type	Key	Description
folderID	varchar	PRI	folderID of f_i
sum	int		value of $\sum_{j=1}^{n} r_{ij}$
sumSquare	int		value of $\sum_{j=1}^{n} r_{ij}^2$
numRatings	int		number of ratings for folder f_i

Note that

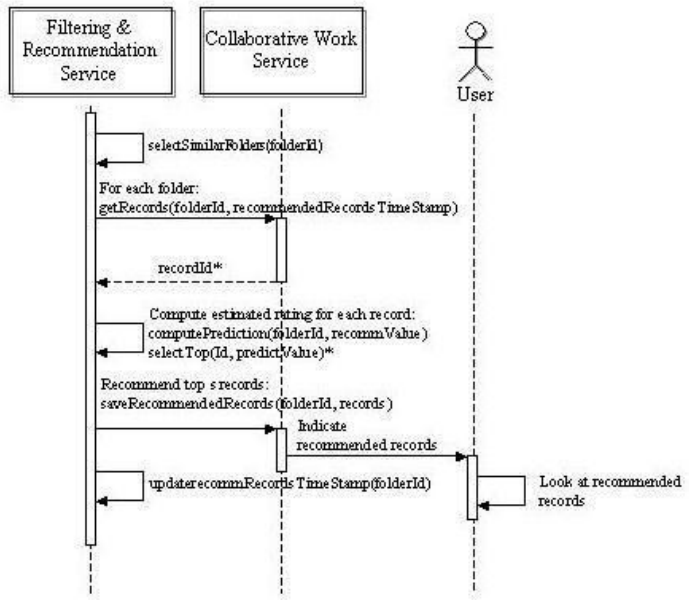

Figure 5.11: Interaction diagram for the Update Folder Profile On-Demand (SU 7-1) use case

$$\overline{r}_i = \frac{ratingsProfileAux.sum}{ratingsProfileAux.numRatings}$$

$$\sigma_i = \sqrt{\frac{ratingsProfileAux.sumSquare}{ratingsProfileAux.numRatings} - \left(\frac{ratingsProfileAux.sum}{ratingsProfileAux.numRatings}\right)^2}$$

5.5 User interface

The Filtering & Recommendation Service has no user interface of its own.

5.6 Service interaction diagrams

5.7 Service implementation tools

The first prototype of the Filtering and Recommendation Service will be based on the following development environment and base technology:

- Java 2 Platform SE v1.3.1
  `http://java.sun.com`
- BerkleyDB
  `http://www.sleepycat.com/`
- Apache XML-RPC v1.0
  `http://xml.apache.org/xmlrpc`
- Apache Web Server 1.3.x + Tomcat 4.0.1
  `http://www.apache.org`

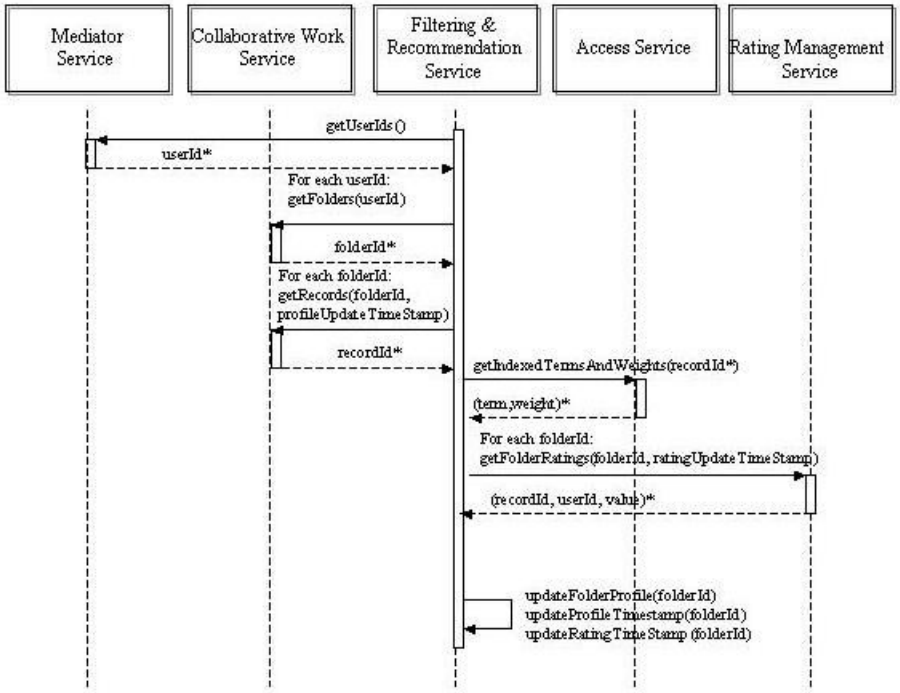

Figure 5.12: Interaction diagram for the Update Folder Profile at Scheduled Time (SU 7-2) use case.

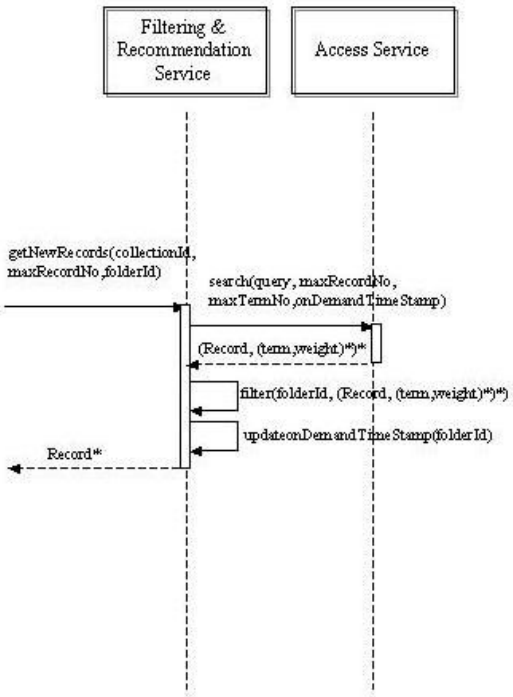

Figure 5.13: Interaction diagram for the Search On-Demand based on Folder Profile (SU 9) use case.

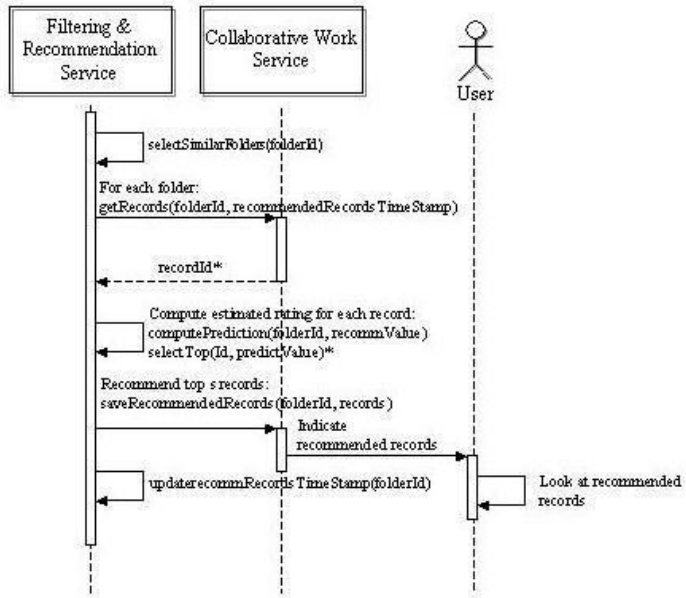

Figure 5.14: Interaction diagram for the Receive Record Recommendation from the System (SU 10-1) use case.

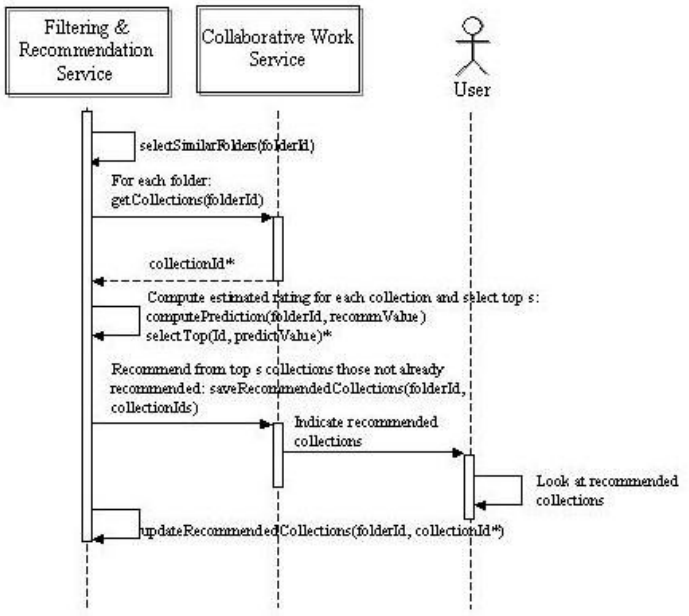

Figure 5.15: Interaction diagram for the Receive Collection Recommendation from the System (SU 10-2) use case.

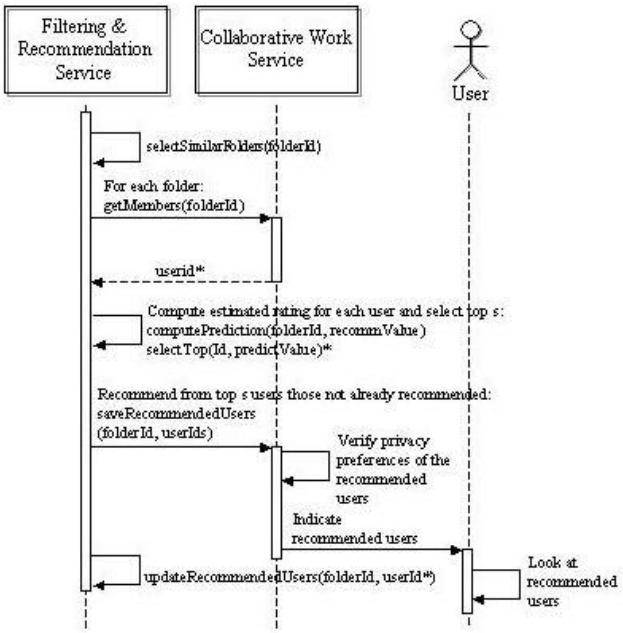

Figure 5.16: Interaction diagram for the Receive User Recommendation from the System (SU 10-3) use case.

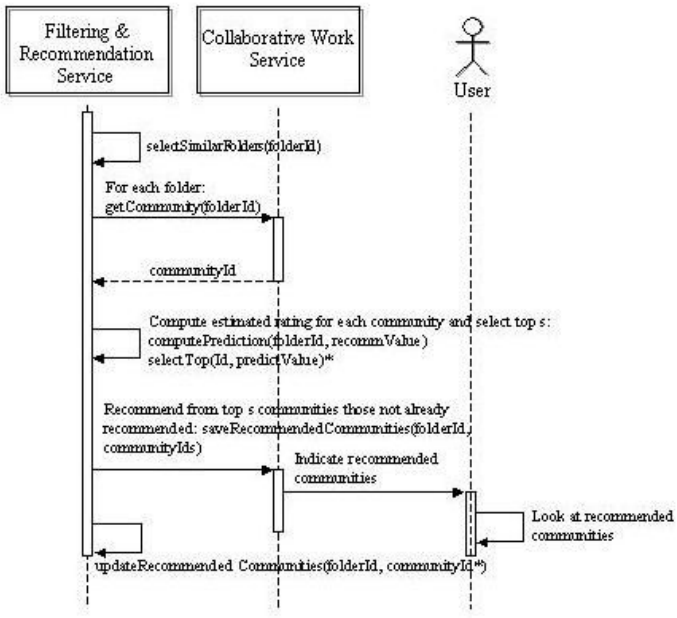

Figure 5.17: Interaction diagram for the Receive Community Recommendation from the System (SU 10-4) use case.

Chapter 6

Collection Service

6.1 Functionalities

6.1.1 Create, Delete and Edit a collection

The Collection Service (CS) accepts requests for the creation, removal and editing of collections and for the dissemination of information about them. It also maintains information about the users authorised to manage the collections, i. e. the collection administrator and the collections owners, and about the underlying information space. A collection create request must specify the identifier of the user who issues the request, the name and textual description of the new collection, and the collection membership condition. A user can create a collection only if she is a collection administrator. If the create request is accepted she becomes the owner of the collection. The owner of a collection can edit a collection textual description and remove the collection.

The membership condition is a condition on the Membership Application Profile (see Section 6.4.5). This Profile contains some of the Dublin Core metadata fields, a field that takes as value metadata descriptions formats and a field that specifies archive identifiers. All the documents that satisfy the membership condition up to an established threshold belong to the collections.

The create collection accepts also a short form for specifying the membership condition. This form consists of the name of an existing collection and a refinement condition. This specification corresponds to the membership condition that is obtained by joining together the conjunction of the membership condition of the existing collection and the refinement condition.

A collection is created only if there exists no collection with the same name, the membership condition is correct and the specified user has the required rights, otherwise an error message is returned.

As a default, the system always maintains a collection named *Cyclades* which includes all the documents harvested by Cyclades and a collection for each of the harvested archives. These last collections have the same name of the corresponding archives and are created when a new archive is registered.

A new collection is always associated with an empty set of search and browse formats. The *Cyclades* collection is associated with the Dublin Core search and browse format.

A delete collection request specifies the name of the collection to be removed and the identifier of the user. The CS processes this request only if the given name corresponds to an existing collection and if the user is either the collection owner or the collection administrator. An error message is returned back if any of these conditions is violated. A deleted collection is removed by any folder that refers to it.

An edit collection request specifies the name of the collection to be edited and a textual description.

The CS processes this request by replacing the existing description with the input one.

6.1.2 Add and remove a search/browse format

For each existing collection the collection owner can define which format can be used for searching and browsing on that collection. documents in the collection. Regarding the search, the format determines the query condition fields, the schema of the records returned as result and how they are displayed. As far as the browse, it specifies the browsable fields, and, again, the schema of the records returned as result and how they are displayed. The system allows to associate more than one search/browse format with the same collection. For example, simple and refined search formats may be associated with a collection in order to support different search granularity.

Search and browse formats can be added to any existing collection, with the only exception of the *Cyclades* collection. Only the owner of the collection and the collection administrator have the rights of associating new search and browse formats.

The adding process starts by browsing the list of schemas supported for that collection and then selecting one of them. In so doing the creator defines which will be the schema of the records returned by the search/browse. Then she chooses the subset of the attributes of selected schema that will characterize the new search/browse format. This choice defines the subschema that will be available in the formulation of the query and that will be displayed as result of a search/browse operation. If the specified subschema is legal and the person who requires the addition of a format has the rights to do it, the new search/browse format becomes available in all the folders that had selected the corresponding collection.

A search and browse format on a collection can be removed by sending a request that specifies the name of the collection and the name of the format to be removed. The request is satisfied only if it refers to an existing collection and an existing format and if the requester is either the owner of the collection or the collection administrator. As a result of the processing of this request, the corresponding search and browse operations become not anymore accessible in any of the folders that refer to it.

6.1.3 Browse collections

The CS maintains the list of active collections and for each of them a set of descriptive metadata. Upon receiving a list collection request, it returns a structured list of collection identifiers and, for each of them, its printable name and its textual description. Specific information about a collection can be obtained by selecting a collection identifier and then invoking a get collection metadata.

6.1.4 Disseminate collection metadata

The CS generates a set of collection metadata as a result of a successful collection creation. This set contains the information that the Cyclades services need to support a collection-based structured view of the information space. In particular, it maintains the list of the archives that disseminate descriptive records about the documents in the collection and the *collection filtering condition*. This condition allows to select, among the records harvested by the associated archives, those belonging to the collection.

6.2 Process flow

6.2.1 Becoming a collection administrator

The user requests to become a collections administrator. The Mediator Service passes the request (with user identifier and e-mail) to the Collection Service that initiates the following process:

- If the user is a collections administrator, it shows a message like *"WARNING: the user is already a collections administrator"*;
- If not, it displays an appropriate input-form where the user specifies the kind of collections that he/she wants/thinks to create.
- The user fills in the form and submits the request to the Collection Service administrator.
- The Collection Service administrator checks such data and decides whether grant or deny the collections administrator rights to the user. Then it notifies such decision via e-mail to the user. If the Collection Service administrator has granted the rights then the Collection Service must store such information.

6.2.2 Create collections

A collection is created as a side effect of the registration of a new archive (Case 1) or to satisfy the need of a community (Case 2). Different activity diagrams apply to the two cases.

Case 1

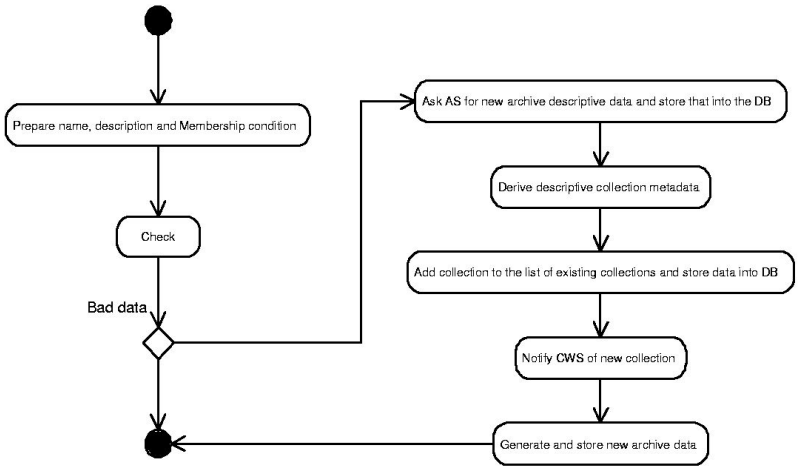

Figure 6.1: Create a collection - Case 1 - activity diagram

The activity diagram in Figure 6.1 shows the steps involved in the creation of a collection as a side effect of the registration of a new archive.

The Access Service prepares the collection name, collection description and Membership Condition of the new archive, then it requests to create a collection to the Collection Service which initiates the following process:

- It checks the received parameters. If they aren't *correct* it raises an exception;
- Otherwise it asks the Access Service to provide a description of the archive that corresponds to the new collection and stores such information into the database for future uses.

- Then the Collection Service generates the collection specific metadata, creates a collection, and registers it as a new member of the set of existing collections.
- The Collection Service then notifies the identifier and the name of the new collection to the Collaborative Work Service.
- In the end, the Collection Service requests to the Access Service additional information about the content of the archive. Then it transforms and stores this information for its internal purposes.

Case 2

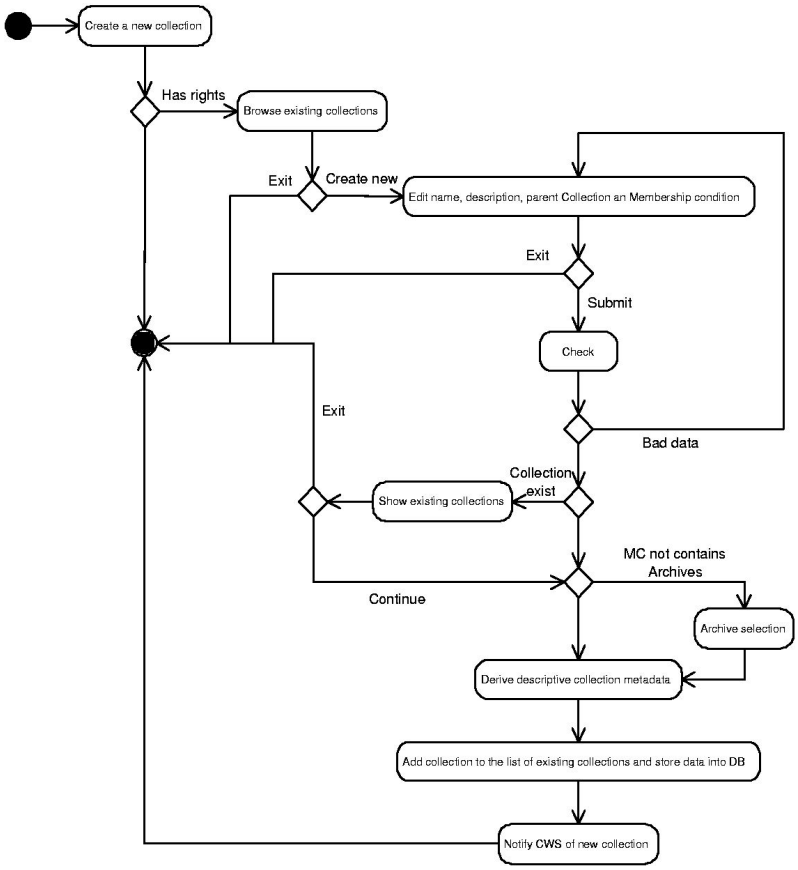

Figure 6.2: Create a collection - Case 2 - activity diagram

The activity diagram in Figure 6.2 shows the steps involved in the creation of a collection in order to satisfy the need of a community.

The user requests to create a new collection. The Mediator Service passes this request to the Collection Service that initiates the following process:

- It checks if the user is a collections administrator. If not, it shows a message like *"WARNING: this operation is not allowed. The user isn't a collections administrator"*. Otherwise it browses all existing collections.
- If the user wants to create a new collection then it *click the button "Create new"*. The Collection Service generates an identifier for the new collection and displays to the user an appropriate input-form for submitting the requested parameters: collection name, collection description, parent collection and Membership Condition.

- The user fills in the form and submits the initialization request to the Collection Service.
- The Collection Service checks the received parameters. If they aren't *correct*, it sends an error message to the user and re-displays the input-form.
- If the collection defined is already present in the set of Collection Service's collections (with or without different name and/or description) the system notifies the user of this trouble. The user may decide to abort or to continue anyway.
- Then it generates the collection specific metadata, creates a collection, and registers it as a new member of the set of existing collections.

  In the collection's metadata we have to generate the Filtering Condition that logically contains two elements: a set of archives and a condition on them. If the Membership Condition does not specify the set of archives involved, the Collection Service must then select an appropriate set of archives where to find the collection's element.
- The Collection Service then notifies the identifier and the name of the new collection to the Collaborative Work Service.

6.2.3 Delete collections

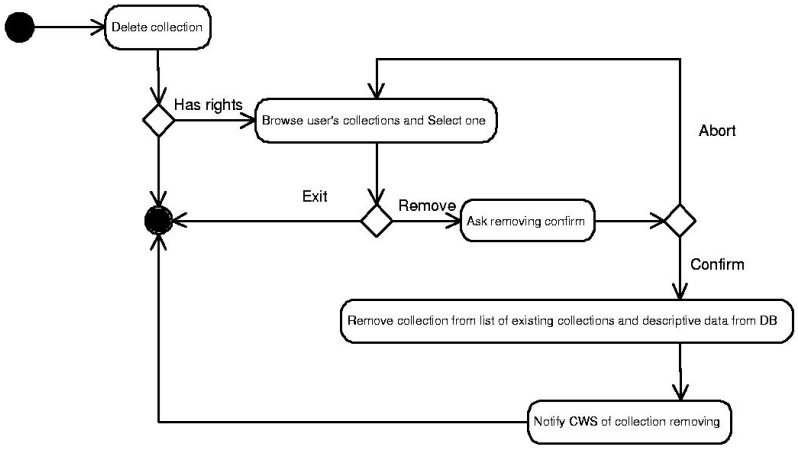

Figure 6.3: Delete a collection - activity diagram

The activity diagram in Figure 6.3 shows the steps involved in the deletion of a collection.

The user requests to delete a collection. The Mediator Service passes this request to the Collection Service which initiates the following process:

- It checks if the user is a collections administrator. If not, it shows a message like *"WARNING: this operation is not allowed. The user isn't a collections administrator"*. Else, it displays the user's collections list.
- The user selects one collection of those listed and *click the button "Remove"*.
- The Collection Service asks the user's confirmation. If the user confirms then the specified collection is also removed from the list of existing collections and DB.
- The Collection Service then notifies the Collaborative Work Service about the removal of the collection.

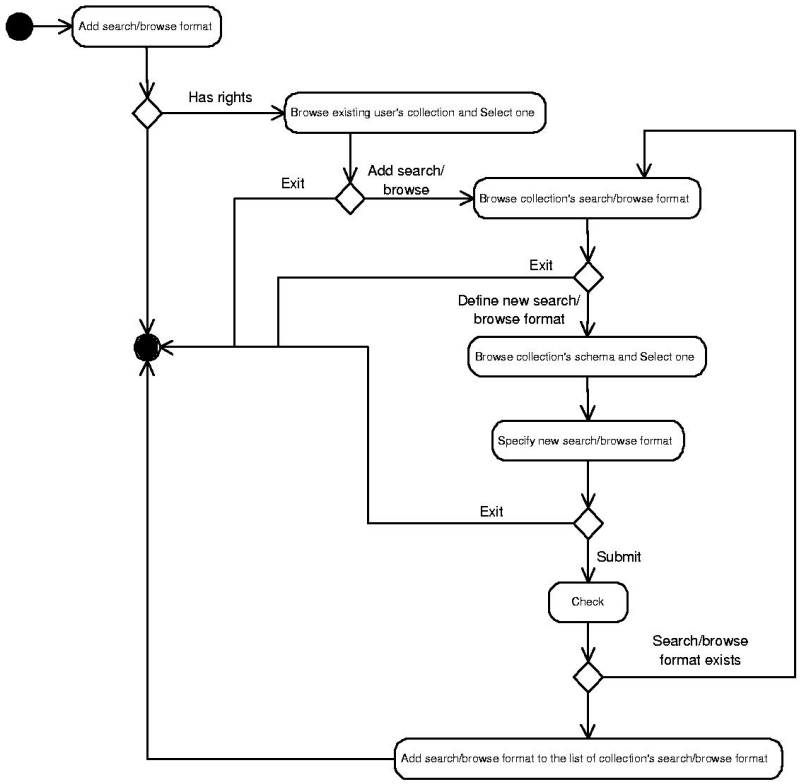

Figure 6.4: Add search/browse format - activity diagram

6.2.4 Add search/browse format

The activity diagram in Figure 6.4 shows the steps involved in the action of adding a search/browse format to a collection's search/browse formats.

The user requests to edit collection related information. The Mediator Service passes this request to the Collection Service which displays a user interface where user *click the button "Add search/-browse format"*. Then the Collection Service initiates the following process:

- It checks if the user is a collections administrator. If not, it shows a message like *"WARNING: this operation is not allowed. The user isn't a collections administrator"*. Else, it displays the user's collections list.
- The user selects one collection of those listed and *click the button "Add search/browse format"*.
- The Collection Service displays the list of search/browse formats available in the selected collection. If the user thinks that no search/browse format meets his/her needs then he/she has to *click the button "Define new search/browse format"*.
- The Collection Service displays the list of schemas available in the collection selected. The user selects one of them as reference schema.
- The Collection Service displays to the user an appropriate input-form for submitting the search/browse format. The user fills in the form and submits data.
- The Collection Service checks the received parameters. If they aren't *correct*, it sends an error message to the user and re-displays the list of search/browse formats available.

- Then the Collection Service creates the search/browse format and registers it as a new member of the list of collection's search/browse formats.

  search/browse format for the collection.

6.2.5 Remove search/browse format

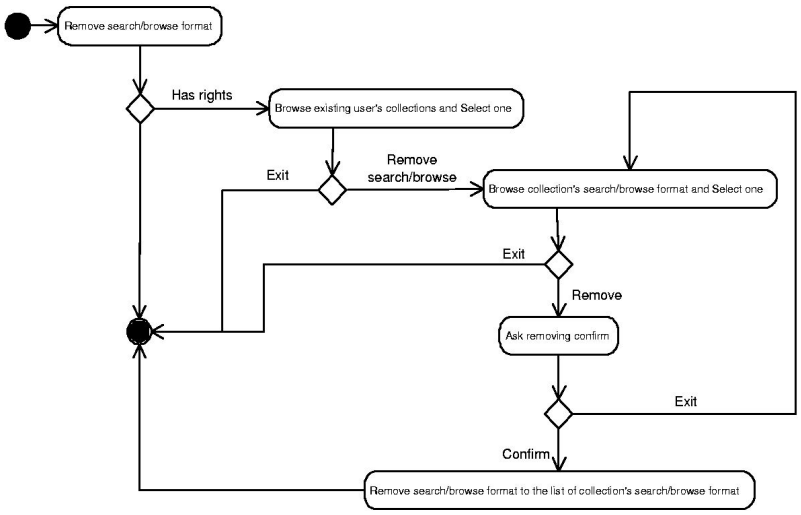

Figure 6.5: Remove search/browse format - activity diagram

The activity diagram in Figure 6.5 shows the steps involved in the action of removing a search/-browse format from collection's search/browse formats.

The user requests to edit collection related information. The Mediator Service passes this request to the Collection Service which displays a user interface where the user *click the button "Remove search/browse format"*. Then the Collection Service initiates the following process:

- It checks if the user is a collections administrator. If not, it shows a message like *"WARNING: this operation is not allowed. The user isn't a collections administrator"*. Else it displays the list of user's collections metadata. The metadata contains the set of search/browse formats available for searching and browsing on the specified collection.
- The user selects one of them and *click the button "Remove search/browse format"*.
- The Collection Service displays the list of search/browse formats. The user selects one of them and *click the button "Remove"*.
- The Collection Service asks the user's confirmation. If the user confirms then the specified search/browse format is also removed from the list of existing search/browse formats of collection and DB.

  the search/browse format.

6.2.6 Edit collection descriptive metadata

The activity diagram in Figure 6.6 shows the steps involved in the editing of a collection's descriptive metadata.

The user requests to edit collection related information. The Mediator Service passes this request to the Collection Service that displays a user interface where the user *click the button "Edit collection's metadata"*. Then the Collection Service initiates the following process:

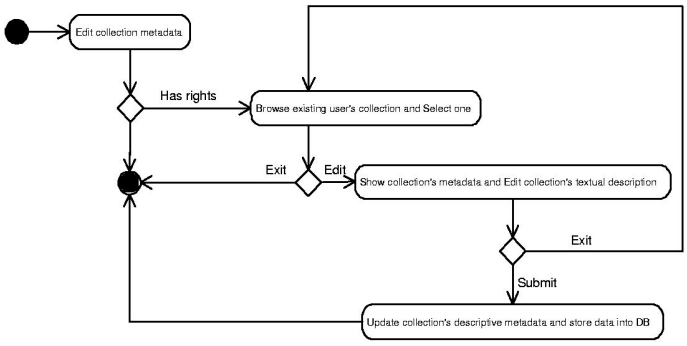

Figure 6.6: Edit a collection metadata - activity diagram

- It checks if the user is a collections administrator. If not, it shows a message like *"WARNING: this operation is not allowed. The user isn't a collections administrator"*. Else it displays the list of user's collections metadata.
- The user selects one collection and *click the button "Edit"*.
- The Collection Service displays to the user an appropriate input-form where the collection's textual description is the only information that the user can edit. The user fills in the form and submits data.
- Then it updates the collection specific metadata the Collaborative Work Service of the applied modification.

6.2.7 Select a user's collection set

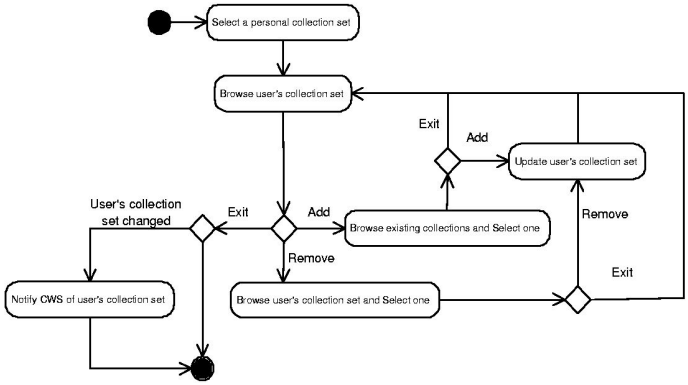

Figure 6.7: Select user's collection set - activity diagram

The activity diagram in Figure 6.7 shows the steps involved in the selection of a user's collection set.

The user requests to edit collection related information. The Mediator Service passes this request to the Collection Service which displays a user interface where the user *click the button "Select personal collection set"*. Then the Collection Service initiates the following process:

- It displays the list of user's collection set, if it has been selected.
- If the user wants to add a collection, then he/she *click the button "Add"*. The Collection Service displays the list of existing collections among which to choose the collections. The

user selects one of them and *click the button "Add"*, then the Collection Service updates the list of user's collections set adding the selected collectionto them.

- If the user wants to remove a collection he/she *click the button "Remove"*. The Collection Service displays the list of user's collections set among which to choose the collection to remove. The user selects one of them and *click the button "Remove"*, then the Collection Service update the list of user's collection set removing the selected collection.

- If the user wants to exit, he/she *click the button "Exit"*. If the user's collection set is changed, the Collection Service notifies the Collaborative Work Service of the new user's collection set.

6.3 Internal architecture

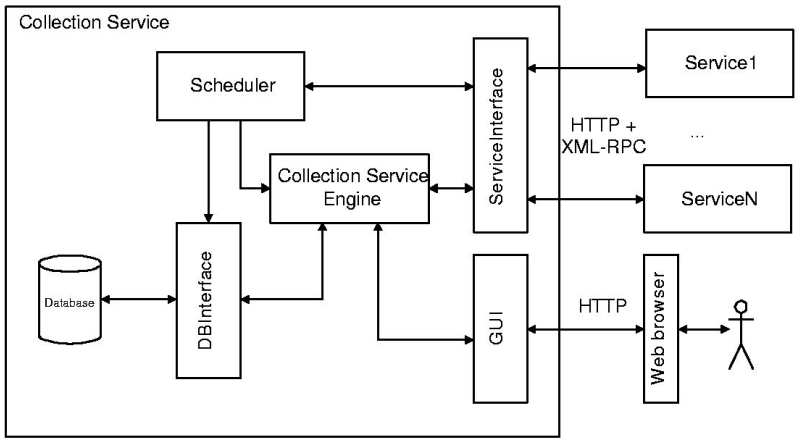

Figure 6.8: Collection Service internal architecture

Figure 6.8 shows the major components of the Collection Service, and how information flow between them. We can note that the Collection Service provides functionality via the GUI (Graphical User Interface) and via an API (ServiceInterface) to the other Cyclades services. The communication between the users and the Collection Service's GUI take place via HTTP using HTML; the communication between a Cyclades service and the Collection Service's API takes place via HTTP using XML-RPC protocol.

Collection Service Engine is the core of the Collection Service, it implements all the functionalities of the service.

ServiceInterface is the API that the Collection Service offers to other services and that other services "must offers" to the Collection Service Engine. "Must offers" means that the Collection Service Engine needs this information to execute his tasks. Such information may be partially offered from different external services, so that this component acquires the partial information supplied from single services, merges them and builds also the information needed. This component is strictly bound to other services' interface and also to the communication protocol used in Cyclades system.

This allows to have a Collection Service Engine independent of changes in:

- *protocol use*: if there are changes in the use of protocol, like the encoding of input and output parameters of a Cyclades service method in a protocol datatype, we have to update only this component;

- *protocol used*: if the communication protocol changes, we have to update only this component;
- *service's interface*: if the name, or the signature, of a methodsupplied from an external service changes, we have to update only this component.

GUI is the component that provides the graphical user interface for the Collection Service.

Scheduler is the component that periodically updates the information about the underlying information space stored in the database. The Collection Service Engine uses the data stored in the database to perform its tasks, so we want that such data will be more similar to real data stored in archives.

DBInterface is the component that provides a database interface to the other service's component.

Database contains all information needed by the Collection Service's activity. Moreover, it is a backup unity of all information about collections, like collections created, user rights and so on.

6.4 Data and method specification

6.4.1 Abstract `CollectionService` class

Fields specification

id the unique identifier of the class.

collections the set of registered collections.

Methods specification

Invoking the Collection Service's method you can throw such default exception:

10000	Generic exception	In all cases where an undefined exception exist.
10001	Wrong parameter number	If you invoke a method with a wrong parameter number.
10002	Wrong parameter type	If you invoke a method with a wrong parameter type.
10003	No such method	If the method invoked isn't defined.

- collectionId addCollection()
  Description: this method creates a new collection identifier which can be assigned to a collection which will be soon created.
  Output: collectionId integer the identifier of the new collection that will be created.

- collectionId initializeCollection(collectionId, collectionName, collectionDescription, membershipCondition, userId)
  Description: this method creates a collection, which parent collection is the *Cyclades* collection, if the membership condition is legal.

Input:	collectionId	string	the identifier of the new collection.
	collectionName	string	the printable name of the collection.
	collectionDescription	string	textual description of the collection.
	membershipCondition	string	the condition to be verified by all the members of the collection coded in XML.
	userId	string	the identifier of the user which sends the request.
Output:	collectionId	string	the identifier of the collection that has been initialized.

- collectionId initializeCollection(collectionId, collectionName, collectionDescription, membershipCondition, userId, parentCollection)
  Description: this method creates a collection whose parent collection is parentCollection, if the membership condition is legal.

Input:			
	collectionId	string	the identifier of the new collection.
	collectionName	string	the printable name of the collection.
	collectionDescription	string	textual description of the collection.
	membershipCondition	string	the condition to be verified by all the members of the collection coded in XML.
	userId	string	the identifier of the user which sends the request.
	parentCollection	string	the identifier of the parent collection in the collection hierarchy.

Output:			
	collectionId	string	the identifier of the collection that has been initialized.

- void deleteCollection(collectionId, userId)
  Description: this method removes a collection from the set of existing collections if: a) the user is authorized to do it and b) the specified collection exists.

Input:			
	collectionId	string	the identifier of the new collection.
	userId	string	the identifier of the user which sends the request.

- void addSearchBrowseFormat(collectionId, subschema, userId)
  Description: this method adds a new search/browse format if: a) the user is authorized to do it, b) the subschema is legal.

Input:			
	collectionId	string	the identifier of the collection.
	subschema	string	the specification of the subschema (coded in XML) used for querying, browsing and displaying results.
	userId	string	the identifier of the user who sends the request.

- void removeSearchBrowseFormat(collectionId, subschemaName, userId)
  Description: this method removes a search/browse format if: a) the user is authorized to do it, b) the name of the subschema identifies an existing format.

Input:			
	collectionId	string	the identifier of the collection from which the search/-browse format has to be removed.
	subschemaName	string	the name of the search/browse format to remove.
	userId	string	the identifier of the user who sends the request.

- (collectionId, collectionName, collectionDescription, parentCollection)* listCollections(userId)
  Description: this method returns the list of existing collections whose owner is userId.

Input:			
	userId	string	the identifier of the user who sends the request

  Output: a list of (collectionId, collectionName, collectionDescription, parentCollection) where:

collectionId	string	the identifier of the collection.
collectionName	string	the name of the collection.
collectionDescription	string	the description of the collection.
parentCollection	string	the identifier of the parent collection.

- (collectionId, collectionName, collectionDescription, parentCollection)* listCollections()
  Description: this method returns the list of existing collections.
  Output: a list of (collectionId, collectionName, collectionDescription, parentCollection) where:

collectionId	string	the identifier of the collection.
collectionName	string	the name of the collection.
collectionDescription	string	the description of the collection.
parentCollection	string	the identifier of the parent collection.

- (collectionId, collectionMetadata)* getCollectionMetadata(collectionIds*)
  Description: for each specified collection identifier, this method returns the corresponding descriptive metadata.

Input: collectionIds string* a list of collection identifiers.
Output: A list of pairs (collectionId,collectionMetadata) where:
collectionId string the identifier of the collection.
collectionMetadata string the collection metadata coded in XML.

- void editCollection(collectionMetadata)
  Description: Update collection metadata description.
  Input: collectionMetadata string new collection metadata coded in XML.

6.4.2 Abstract `Collection` class

Fields specification

id : string - The unique Identifier of the class.

name : string - A printable name of the collection.

description : string - A textual description of the collection.

membershipCondition : string - The condition which characterizes the member of the collection.

parentCollection : string - As the collection may have been structured using a hierarchy, this is the identifier of the parent collection in the hierarchy.

archives : string* - A list of the archives which stores the elements of the collection.

filteringCondition : string - The condition which filters the members of the collection.

schemas : string* - A set of reference to Schema objects associated with the collection.

searchBrowseFormats : string - The set of reference to Subschema objects that specifies the search/browse formats available on the collection.

ownerId : string - The identifier of the person who created the collection.

6.4.3 Abstract `Subschema` class

Fields specification

name : string - The unique name of the subschema

url : string - The URL of a DTD or namespace for the subschema

attributes : (string,string)* - A list (name, datatype) of attributes which characterize the subschema.

6.4.4 Database schema

Figure 6.9 shows the schema of the database used to store the information needed by the Collection Service.

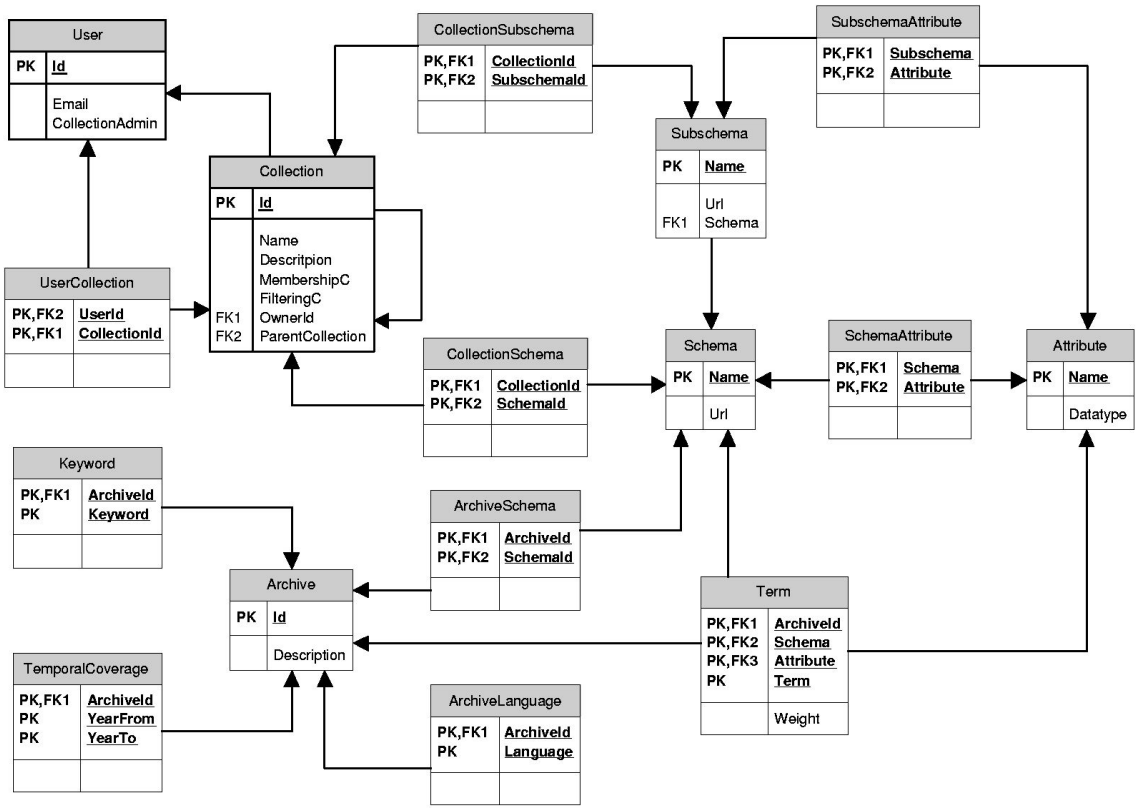

Figure 6.9: Collection Service's database schema

- **Collection**
  Stores information about a created collection.

Id	Varchar	the unique identifier of the collection.
Name	Varchar	the printable name of the collection.
Description	Varchar	a textual description or the collection.
MembershipC	Varchar	the pathname of the ASCII file which stores the Membership Condition
FilteringC	Varchar	the pathname of the ASCII file which stores the Filtering Condition
OwnerId	Varchar	the identifier of the user who created the collection.
ParentCollection	Varchar	the identifier of the parent collection.

- **User**
  Stores information about users.

Id	Varchar	the unique identifier of the user.
Email	Varchar	the e-mail address of the user.
CollectionAdmin	Boolean	*true* if the user has Collection Administrator's rights.

- **UserCollection**
  Stores information about user's selection of personal collection.

UserId	Varchar	the user's identifier.
CollectionId	Varchar	the collection's identifier.

- **Schema**
  Stores information about a metadata schema.

Name	Varchar	the unique name of the schema.
Url	Varchar	the URL of a DTD or namespace for the schema.

- **CollectionSchema**
  Stores information about collection's schema.

CollectionId	Varchar	the collection's identifier.
SchemaId	Varchar	the schema name.

- **Subschema**
  Stores information about a search&browse format.

Name	Varchar	the unique name of the search&browse format.
Url	Varchar	the URL of a DTD or namespace for the subschema.
Schema	Varchar	the schema's name from which the subschema is derived.

- **CollectionSubschema**
  Stores information about the collection's search/browse format.

CollectionId	Varchar	the collection's identifier.
SubschemaId	Varchar	the subschema name.

- **Attribute**
  Stores information about the attribute which characterizes a schema or a subschema.

Name	Varchar	the attribute name.
Datatype	Varchar	the attribute datatype.

- **SchemaAttribute**
  Stores information about the schema's attribute.

SchemaId	Varchar	the schema name.
Attribute	Varchar	the attribute name.

- **SubschemaAttribute**
  Stores information about the subschema's attribute.

SubschemaId	Varchar	the subschema name.
Attribute	Varchar	the attribute name.

- **Archive**
  Stores information about a single open archive.

Id	Varchar	the unique archive's identifier.
Description	Varchar	a textual description of the archive, entered by the archive creator.

- **Keyword**
  Stores information about archive's keywords. Archive's keywords describe the archive content.

ArchiveId	Varchar	the unique archive's identifier.
Keyword	Varchar	a keyword.

- **TemporalCoverage**
  Stores information about the archive's temporal coverage. Archive's temporal coverage is a list of periods, each period is a pair (year, year).

ArchiveId	Varchar	the unique archive's identifier.
YearFrom	Integer	the beginning year of a temporal coverage period.
YearTo	Integer	the end year of a temporal coverage period.

- **ArchiveSchema**
  Stores information about the archive's schema.

ArchiveId	Varchar	the unique archive's identifier.
SchemaId	Varchar	the schema name.

- **ArchiveLanguage**
  Stores information about the archive's language.

ArchiveId	Varchar	the unique archive's identifier.
SchemaId	Varchar	the language name.

- **Term**
  Stores information about the attribute's value stored in an archive.

ArchiveId	Varchar	the unique archive's identifier.
Schema	Varchar	the schema name.
Attribute	Varchar	the attribute name.
Term	Varchar	the attribute value.
Weight	Real	the *weight* of such Term in the archive.

6.4.5 The Membership Application Profile: fields and their definition

An application profile is defined as a metadata schema which consists of data elements drawn from one or more namespaces, combined together by implementors, and optimized for a particular local application. The idea of application profiles grew out of UKOLN's work on the DESIRE project and it is currently one of the most accepted methodological solution for the definition of new metadata schemas. Application profiles allow the implementor to declare how they are using standard schemas.

Element name	Element definition
`dc:title`	A name given to the resource.
`dc:creator`	An entity primarily responsible for making the content of the resource.
`dc:subject`	The topic of the content of the resource.
`dc:description`	An account of the content of the resource.
`dc:publisher`	An entity responsible for making the resource available.
`dc:contributor`	An entity responsible for making contributions to the content of the resource.
`dc:date`	A date associated with an event in the life cycle of the resource.
`dc:format`	The physical or digital manifestation of the resource.
`dc:type`	The nature or genre of the content of the resource.
`dc:language`	A language of the intellectual content of the resource.
`dc:coverage`	The extent or scope of the content of the resource.
`dc:rights`	Information about rights held in and over the resource.
`map:metadataFormat`	The name of a metadata format.
`map:archive`	An archive identifier.

Table 6.1: Membership Application Profile: fields and their definition

In Table 6.1 the elements of the Collection Service Membership Application Profile and a brief description of them are reported. This Profile combines elements drawn from the Dublin Core namespace (`dc`) with elements of a CSMAP namespace (`map`). Figure 6.10 shows the XML-schema for Collection Service Membership Application Profile.

6.5 User interface

When the user enters the Cyclades URL in the Web-browser, the Cyclades start page where the user can log in (for already registered users) or register as new user is shown.

After the login, a screen with 5 buttons (ADMIN, SB, COLL, CWS, EXIT) and a frame/window containing the display of the active service is shown. The Mediator Service will call the services via HTTP cgi calls and will display the HTML code returned by the service in the frame/window. The pressing of the COLL button starts the Collection Service user interface (UI). Functionality offered by such UI are:

- Browse collection set

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
     xmlns:dc="http://purl.org/dc/elements/1.1/"
     xmlns:map="http://..."
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     elementFormDefault="qualified" attributeFormDefault="unqualified">
         <xs:element name="map" type="map:mapType" />
         <xs:complexType name="mapType">
             <xs:choice>
                 <xs:element name="dc:title" type="xs:string"/>
                 <xs:element name="dc:creator" type="xs:string"/>
                 <xs:element name="dc:subject" type="xs:string"/>
                 <xs:element name="dc:description" type="xs:string"/>
                 <xs:element name="dc:publisher" type="xs:string"/>
                 <xs:element name="dc:contributor" type="xs:string"/>
                 <xs:element name="dc:date" type="xs:string"/>
                 <xs:element name="dc:type" type="xs:string"/>
                 <xs:element name="dc:format" type="xs:string"/>
                 <xs:element name="dc:language" type="xs:string"/>
                 <xs:element name="dc:coverage" type="xs:string"/>
                 <xs:element name="dc:rights" type="xs:string"/>
                 <xs:element name="map:metadataFormat" type="xs:string"/>
                 <xs:element name="map:archive" type="xs:string"/>
             </xs:choice>
         </xs:complexType>
</xs:schema>
```

Figure 6.10: Membership Application Profile XML-schema

- Create a new collection
- Delete a collection
- Manage personal collection set
- Add a search/browse format
- Remove search/browse format
- Update collection

The Collection Service UI (Figure 6.11) contains 4 buttons (CREATE, DELETE, MANAGE, BASKET) and a frame/window containing a list of collection. Such list is shown as a tree which reflects the hierarchy of collection and it will be the list of all collections defined.

The CREATE button allows to create a new collection. If a collection is selected, hitting on it in the frame/window, before the user hits the CREATE button, the new collection will become the child of that collection in the hierarchy, otherwise it will become a child of the default root collection *Cyclades*[1]. A child collection is defined as a "refinement" of the parent collection, such means that it is specified as the name of the parent collection and a refinement condition. This specification define a collection whose Membership Condition is obtained by making the conjunction of the Membership Condition of the parent collection and the refinement condition.

The DELETE button allows to remove a collection from the list of existing collections. The user selects a collection in the frame/window, hitting on it, and if the selected collection is one of those

[1]It includes all the documents harvested by CYCLADES.

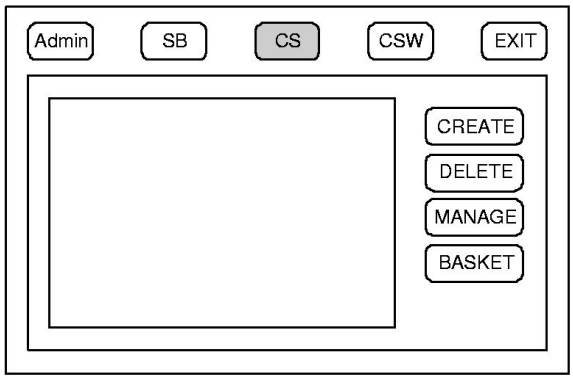

Figure 6.11: Collection Service User Interface

created by the user (*a*) the DELETE button is enabled and (*b*) the user hitting on it removes the collection.

The MANAGE button opens a sub menu which allows users to add or remove a search/browse format from the list of those associated to a collection or to edit the collection descriptive metadata. All that menu options are enabled if the user has selected a collection hitting on it, in the frame/window. When the user hits the *Add search/browse format* button or the *Remove search/-browse format* button the UI prepare and shows a form to do it. When the user hits the *Edit collection descriptive metadata* button the UI prepares and shows a form to do it[2].

The BASKET button allows to manage the personal collection set of the user. So, when the user hits that button the UI shows in the frame/window the list of all collections each one with a check box. The check box is checked if the collection is in the personal collection set of the user, otherwise it remains unchecked.

Moreover, the Collection Service must provide the form which allows a user to make a request to become a collection administrator. This form is reachable hitting the ADMIN button in the general CYCLADES user interface and then hitting the link *Administration of collections*.

6.6 Service interaction diagrams

Service interaction diagrams describe the interactions between the Collection Service and the user or another CYCLADES service in executing one of Collection Service use case.

6.6.1 Becoming a collection administrator

Figure 6.12 shows the service interaction diagram of this operation.

When a user makes a request to become a collections administrator, the Mediator Service passes the request to the Collection Service which prepares a form that the user fills in with descriptive data of the kind of collection he wants to create. The Collection Service administrator using such data decides to grant or deny the collections administrator rights and notifies the user of the decision via e-mail.

6.6.2 Create collections

As described in Section 6.2.2, a collection may be created in two different cases. Different Service Interaction diagrams apply to the two cases.

[2]Note that a user may update the only collection description.

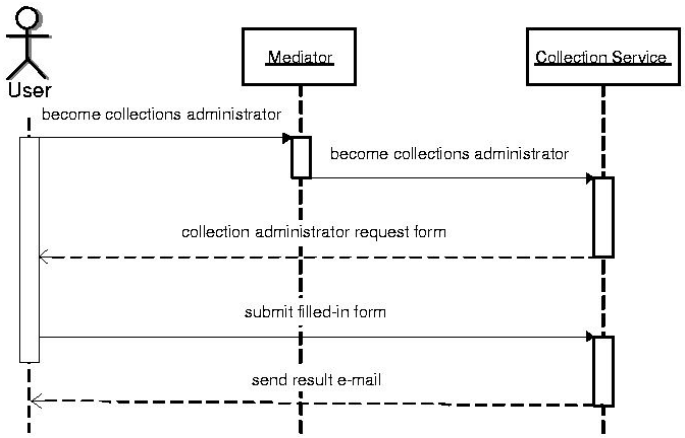

Figure 6.12: Become a collection administrator - interaction diagram

Case 1

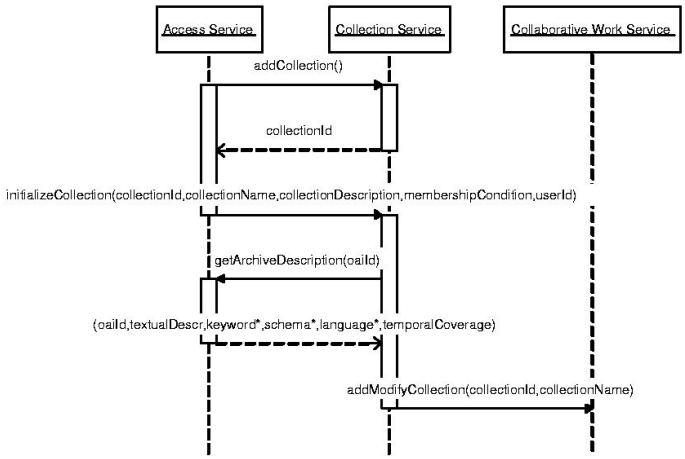

Figure 6.13: Create a collection - Case 1 - interaction diagram

Figure 6.13 shows the service interaction diagram of this operation.

Each time a new archive is registered, the Access Service must create a collection that maintains all the documents of the new archive. It invokes the method `addCollection()` to obtain a valid collection identifier. Then it prepares a name, a description and a Membership Condition that describes the collection and invokes the method `initializeCollection`. At this point the Collection Service creates the new collection and notifies the creation to the Collaborative Work Service invoking the method `addModifyCollection`.

Case 2

Figure 6.14 shows the service interaction diagram of this operation.

When a collection administrator (the only user enabled to do it) wants to create a new collection to satisfy the need of a community, he/she requests to the Mediator Service to show the Collection Service UI. Then, interacting with UI, the user chooses to create a new collection. The Collection Service UI prepares a form that the user fills in with the data describing the collection that he wants to create. So, the user submit the filled in form that invokes the method `initializeCollection`.

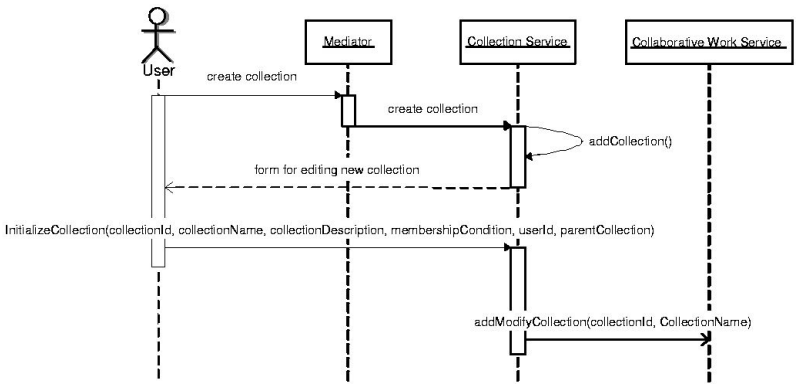

Figure 6.14: Create a collection - Case 2 - interaction diagram

The Collection Service then creates the new collection and notifies the creation to the Collaborative Work Service invoking the method `addModifyCollection`.

6.6.3 Delete collections

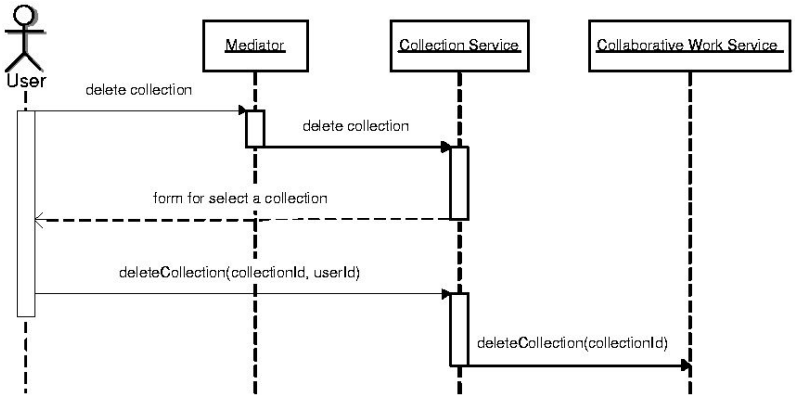

Figure 6.15: Delete a collection - interaction diagram

Figure 6.15 shows service interaction diagram of this operation.

When the collection creator (the only user enabled to do it) wants to remove a collection he/she requests to the Mediator Service to show the Collection Service UI. Then, interacting with UI, the user chooses a collection and hits the button *Remove*. Hitting such button it invokes the method `deleteCollection`. The Collection Service then removes the collection from the list of existing collections and notifies to the Collaborative Work Service that the collection has been removed invoking the method `deleteCollection`.

6.6.4 Add search/browse format

Figure 6.16 shows the service interaction diagram of this operation.

When the collection creator (the only user enabled to do it) wants to add a new search/browse format to a collection he/she requests to the Mediator Service to show the Collection Service UI. Then, interacting with UI, the user choose a collection and hits the button *Manage*, then the button *Add search/browse format*. Then the UI shows a form where the user may specify the

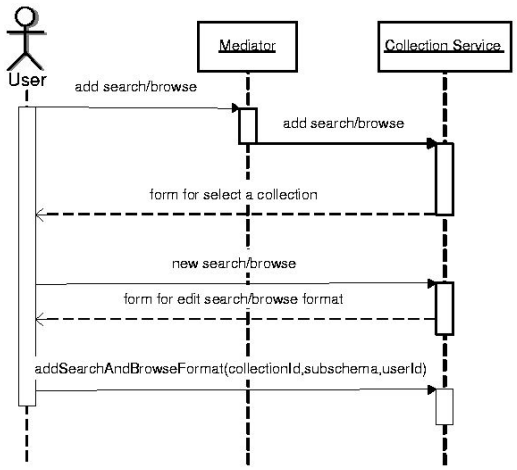

Figure 6.16: Add search/browse format - interaction diagram

new search/browse format[3]. When user submit such filled in form he/she invokes the method `addSearchBrowseFormat`. The Collection Service then adds the search/browse format to the list of collection search/browse format.

6.6.5 Remove search/browse format

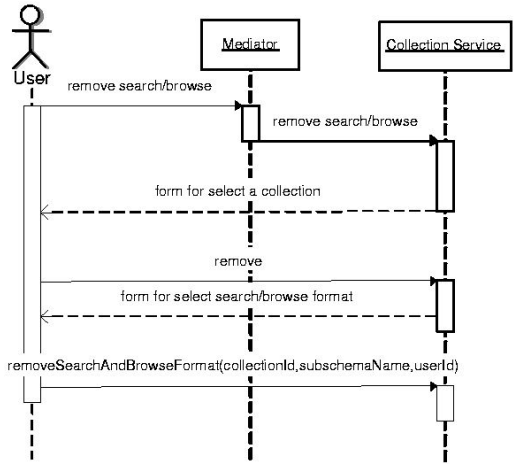

Figure 6.17: Remove search/browse format - interaction diagram

Figure 6.17 shows the service interaction diagram of this operation.

When the collection creator (the only user enabled to do it) wants to remove a search/browse format from that of a collection he/she requests to the Mediator Service to show the Collection Service UI. Then, interacting with UI, the user chooses a collection and hits the button *Manage*, then the button *Remove search/browse format*. At this point the UI shows a form where the user may select one of the search/browse format and hit the button *Remove*. Hitting such button he/she invokes the method `removeSearchBrowseFormat`. The Collection Service then removes the search/browse format from the list of collection search/browse format.

[3]A search/browse format specify the format used for searching and browsing on a collection.

6.6.6 Edit collection descriptive metadata

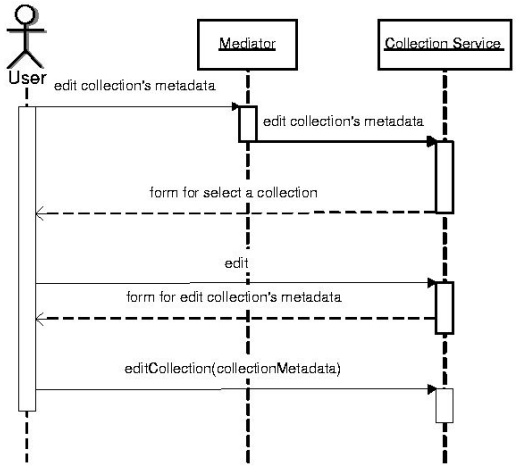

Figure 6.18: Edit collection's metadata - interaction diagram

Figure 6.18 shows the service interaction diagram of this operation.

When the collection creator (the only user enabled to do it) wants to edit collection description, he/she requests to the Mediator Service to show the Collection Service UI. Then, interacting with UI, the user chooses a collection and hits the button *Manage*, then the button *Edit collection's metadata*. At this point the UI shows a form where the user may edit the collection textual description and hit the button *Submit*. Hitting such button, he/she invokes the method `editCollection`. The Collection Service then updates the collection description.

6.6.7 Select a user's collection set

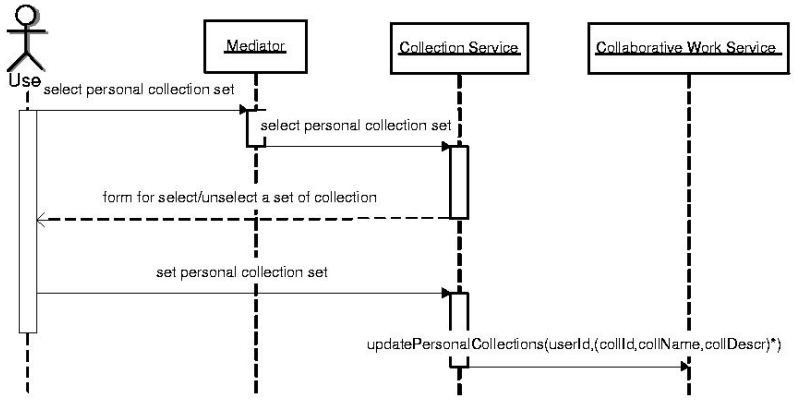

Figure 6.19: Select user's collection set - interaction diagram

Figure 6.19 shows the service interaction diagram of this operation.

When a user wants to update the personal collection set, he/she requests to the Mediator Service to show the Collection Service UI. Then, interacting with UI, the user may hit the button *Basket* and then add or remove a collection from his personal set checking on it. When the user hits the button *Set*, the Collection Service considers the set of collections checked as the user's personal collection set and notifies the Collaborative Work Service invoking the method `updatePersonalCollections`.

6.7 Service implementation tools

Development environment and base technology used:

- Java 2 Platform SE v1.3.1
  `http://java.sun.com`
- Apache XML-RPC v1.0
  `http://xml.apache.org/xmlrpc`
- Apache Web Server 1.3.x + Tomcat 4.0.1
  `http://www.apache.org`
- MySQL 3.23 + mm JDBC driver v1.2c
  `http://www.mysql.com`

Chapter 7

Mediator Service

7.1 Functionality

The mediator is the service of the Cyclades System, which integrates and enables the various services of the system to communicate with each other. It is the service of the CYCLADES to which a service can refer to, in order to get information about other services it wants to communicate with.

The Mediator is also the service of the CYCLADES, that partly handles the registration of the users in CYCLADES and fully handles the loggin of the users to the system. In the case that a new user registers in CYCLADES, it interacts with the Collaborative Work Service so as to obtain that user's id (which is created by the Collaborative Work Service) in the System and some other information for that user. In the case that a user wants to log into CYCLADES, the Mediator does all the appropriate checkings. Also the Mediator, in interaction with the Collaborative Work Service, is involved in the cases of user's invitation.

Finally, the Mediator Service is the service of CYCLADES to which a service can refer to in order to "join" the System. Services are able to "register" in the System by contacting the Mediator and executing, following the communication protocol, the appropriate method of the Mediator Service and specifying the needed parameter.

Thus, the functionalities of the Mediator Service are:

- Inter-System Communication
- User Registration and Login
- Service Registration

7.2 Process flow

7.2.1 Inner-System Communication

Whenever a service (lets say A) needs to communicate with another service (lets say B), it will refer to the Mediator, according to communication protocol. Then the Mediator will send back a list of all the (available) services of kind B to A. Service A will then be able to communicate with each of the services of kind B, using the information contained in that list.

In the Cyclades System each service is described through a class. Hence, the list that the mediator will send is a list of such classes, each one describing a service. The idea of sending a list of classes and not the class itself is considered so as to cover the case of having more than one service

performing the same task (e.g. more than one Access Services). Thus, the system is open to accept more services performing the same task with services that already exist in the system (we could have for example more than one collection services).

7.2.2 User Registration and Login

Registration

The user is performing the process of registering in the system by filling and submitting a username, a password an her e-mail address to the Mediator Service.

- The Mediator Service receives the username, password and e-mail address of the user
- The Mediator Service sends these to the Collaborative Work Service, so as the user to be created in CYCLADES
- The Collaborative Work Service, after creation, sends back that user's id and home folder id to the Mediator
- the user is presented with her home folder

Logging into system

The user is performing the process of logging into the system by filling and submitting her username and password to the Mediator Service.

- The Mediator Service receives the username and password of the user
- The Mediator Service checks for matching between that particular username and password
- On error Mediator Service notifies the user
- Otherwise it sends that user's id to the Collaborative Work Service, which then presents the user with her home folder

7.2.3 Service Registration

Services are able to "register" in the System by contacting the Mediator and executing, following the communication protocol, the appropriate method of the Mediator Service and specifying the needed parameter.

This methods returns a serviceId, which from that point on will be the id of that service in the System. It is desirable for new services to make somehow public their own services-methods (maybe through an on-line page, with probably a short description for each of them), so that others can use them. Each new service that wants to use the basic services of CYCLADES must follow the communication protocol.

- The new serice executes the appropriate method (addService)
- The Mediator Service adds that service to the services Data Base and generates a unique service id, which returns to the service for this service
- The new service receives the service id

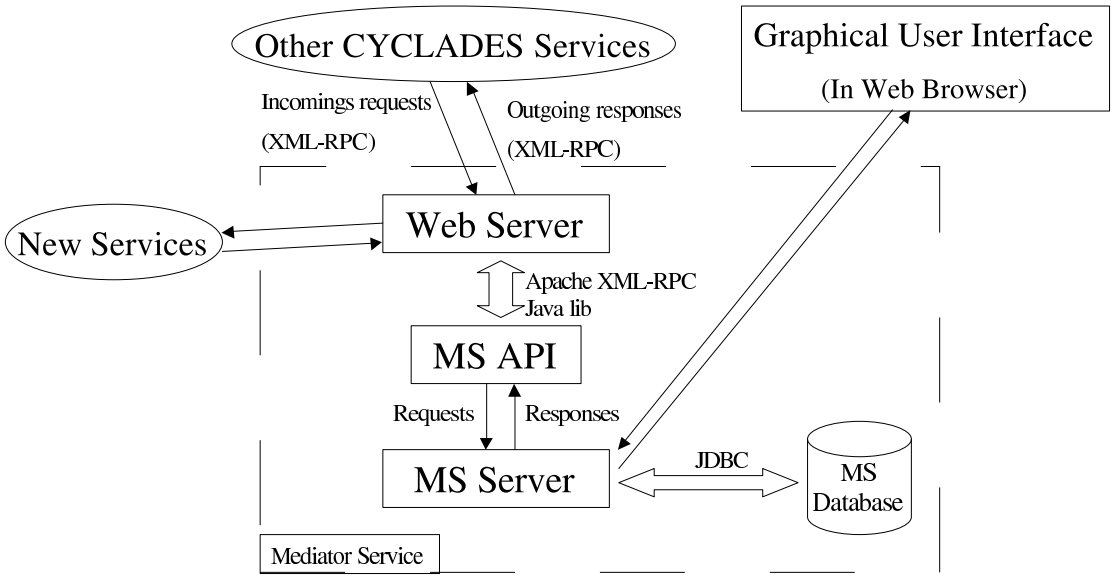

Figure 7.1: Internal architecture of MS

7.3 Internal architecture

7.3.1 Overview

The MS implements a service API which other Cyclades services may use to invoke MS functionality via the XML-RPC protocol over HTTP. Also MS implemetns a Graphical User Inreface, via which users are able access Cyclades. The MS consists of the following four components:

- a standard Web server (Apache) for handling HTTP requests and responses,
- MS API with a method call and response handler that deals with the XML-RPC protocol,
- the MS server, and
- a relational database (Orecle8i) for storing and extracting MS data (users and services).

All MS components are implemented in Java.

7.3.2 MS API

XML-RPC call and response translation

XML-RPC calls that arrive via HTTP are translated into Java objects by a Java library provided by Apache Org. Parameter checking and returned values are handled both by this library. When

the execution of a method/function raises an XML-RPC exception this is reported back to the calling service as an XML-RPC fault. Other exceptions are reported back as HTTP error 500 (`CYCLADES Server Error`).

7.3.3 MS server

The execution of all the functions is being performed by MS server. MS server interacts with the MS Database, using "JDBC Technology", in order to store or to extract MS data. After exexcution, the result (or an exception) is returned back to the MS method call and response handler component.

7.4 Data and method specification

The Mediator Service (MS) has a class MediatorService and a class Service. The persistent data of this service are the various services in the System and the users that have registered the System.

7.4.1 MediatorService Class

- id
  Description: a string specifying the unique ID of the MS service.

- (servId, version, address, quality)* getService(type)
  Description: this method is used in order to get a list of services of particular type.

Input:	type:	string	a service type.
Output:			a list of tuples that describe a service (the ID, the version number, the address and the quality of a service).

- description getServiceDescription(servId)
  Description: this method is used in order to get the description of a service.

Input:	servId:	string	a service ID.
Output:	description	string	the (short) description of the particular service.

- errorLog getErrorLog(servId)
  Description: this method is used in order to get the error log file of a service.

Input:	servId:	string	a service ID.
Output:	errorLog	string	the error log file of the particular service.

- void reportError(servId, errorLogs)
  Description: this method is used in order to report an error(s) for a service.

Input:	servId:	string	a service ID.
	errorLogs:	strings*	a list of error logs, to be added to the already existing error log file of the service.

- serviceId addService(version, address, type, description)
  Description: this method is used in order to add a service to the system.

Input:	version:	double	the version of the service.
	address:	string	the machine address.
	type:	string	the type of the service.
	description:	string	a short description of the service.
Output:	serviceId	string	a service ID.

- void deleteService(servId)
  Description: this method is used in order to delete/remove a service from the system.
  Input: servId: string a service ID.

- void updateService(servId, version, address, description)
  Description: this method is used in order to update the information (version, machine address, description) of a service.
  Input: servId: string a service ID.
  version: double the (new) version of the service.
  address: string the (new) machine address.
  description: string a (new) short description of the service.

- void resetErrorLog(servId)
  Description: this method is used in order to reset the error log file of a service.
  Input: servId: string a service ID.

- void setCollectionAccessRight(userId, collectionId, ON/OFF)
  Description: this method is used in order to set a user's access rights for a collection.
  Input: userId: string a user ID.
  collectionId: string a collection ID.
  ON/OFF: boolean a boolean indicating whether to enable or disable the access right.

- void setArchiveAccessRight(userId, archiveId, ON/OFF)
  Description: this method is used in order to set a user's access rights for an archive.
  Input: userId: string a user ID.
  archiveId: string an archive ID.
  ON/OFF: boolean a boolean indicating whether to enable or disable the access right.

- void addUser(userId, userName, password)
  Description: this method is used in order to add a, newly registered, user to the system.
  Input: userId: string a user ID.
  userName: string a user name.
  password: string the password for that user.

- UserId* getUserIds()
  Description: this methods is used in order to obtain the ids of all users.
  Output: UserIds a list containing the IDs of all users.

- void inviteUser(mailAddr, folderId)
  Description: this method is used in order for a registered user of CYCLADES to be able to invite another unregister user to the system.
  Input: mailAddr: string e-mail address of the invitee.
  folderId: string the folder into which the invitee is invited.

7.4.2 Service Class

- servId
  Description: string this is the unique ID of the service.

- version
  Description: double this is the version of the service.

- address
  Description: string this is the machine address of the service, so as others can communicate with it.

- quality
  Description: double a number between 0-1, indicating the quality of the service.
- type
  Description: string this is the type of the service.
- description
  Description: string this is the (short) textual description of the service.
- errorLog
  Description: string this is for descriping-holding the various errors that occur.

7.4.3 Database Schema

The schema of the database that is used to store information needed by the MS is described bellow. Particularly, the tables which have been created and their atributtes are beeing described.

- **Services**
  Stores information about the services in Cyclades.

servId	Varchar	the unique ID of the service.
version	Double	the version of the service.
address	Varchar	the machine address of the service, so as others can communicate with it.
quality	Double	a number between 0-1, indicating the quality of the service.
type	Varchar	the type of the service.
description	Varchar	the (short) description of the service.
errorLog	Varchar	for descriping-holding the various errors that occur.

- **Users**
  Stores information (which is needed to MS) about the users in Cyclades.

UserId	Varchar	the unique ID of the user.
UserName	Varchar	the username of the user in Cyclades.
Password	Varchar	the password of the user.
MailAddress	Varchar	the e-mail address of the user.
HomeFolder	Integer	the ID of the home folder of the user in Cyclades

- **Invited Users**
  Stores information about the users which have been invited to a folder in Cyclades.

MailAddress	Varchar	the e-mail of the invited user.
FolderId	Integer	the ID of the folder where the user has been invited.

- **ArchiveAccess**
  Stores information about the access privileges a users might has regarding an archive.

UserId	Varchar	this is the unique ID of the user.
ArchiveId	Varchar	this is the ID of the archive.

- **CollectionAccess**
  Stores information about the access privileges a users might has regarding a collection.

UserId	Varchar	the unique ID of the user.
CollectionId	Integer	the ID of the collection in Cyclades.

7.5 User interface

As mentioned above, the MS implements the Graphical User Interface of Cyclades. Through this a user is able to register and login into Cyclades. Also she is able to access the Cyclades services (and their GUIs, when provided) and thus navigate through them.

After succesfull login user is presented with an interface where both her home folder and buttons for accessing CYCLADES services are shown. A user can either navigate through her folders or choose to activate a service. If she chooses to activate another service, then she is presented with that service's user interface.

The design of the global GUI for the CYCLADES is an ongoing activity.

7.6 Service interaction diagrams

The MS service functional interacts with CWS in case of user registration (cf. Figure 7.2) and user invitation (cf. Figure 7.3). Finaly, login interaction diagram is shown in Figure 7.4.

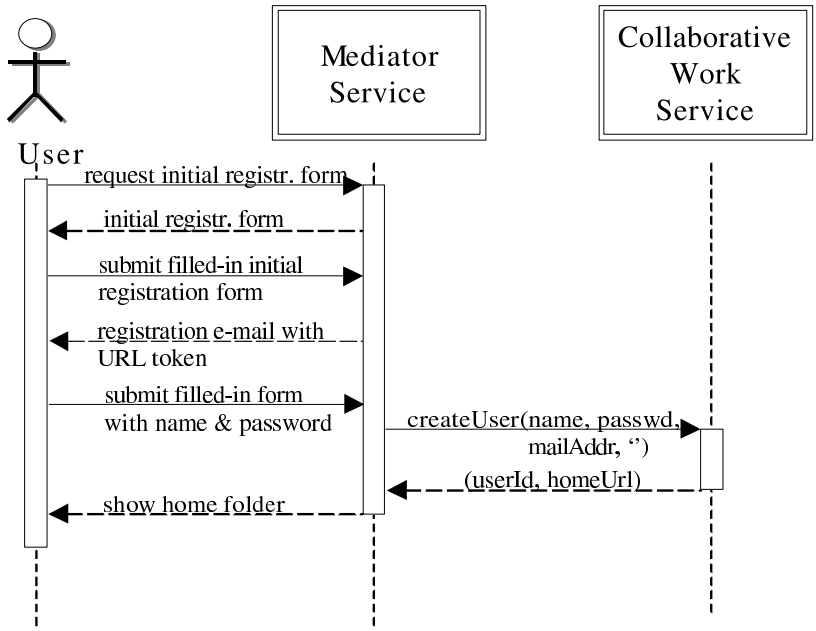

Figure 7.2: Registration interaction diagram

7.7 Service implementation tools

The first prototype of the Mediator Service will be based on the following development environment and base technology:

- Java 2 Platform SE v1.3.1
  `http://java.sun.com`
- Apache XML-RPC v1.0
  `http://xml.apache.org/xmlrpc`

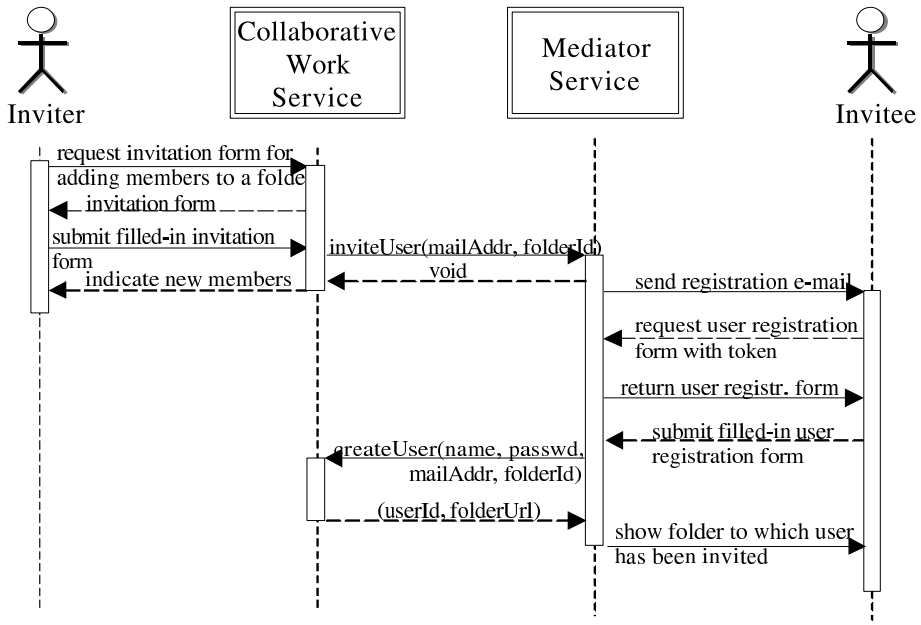

Figure 7.3: Invitation interaction diagram

- Apache Web Server 1.3.x + Tomcat 4.0.1
  `http://www.apache.org`
- Oracle8i Database Family
  `http://www.oracle.com/oracle8i`

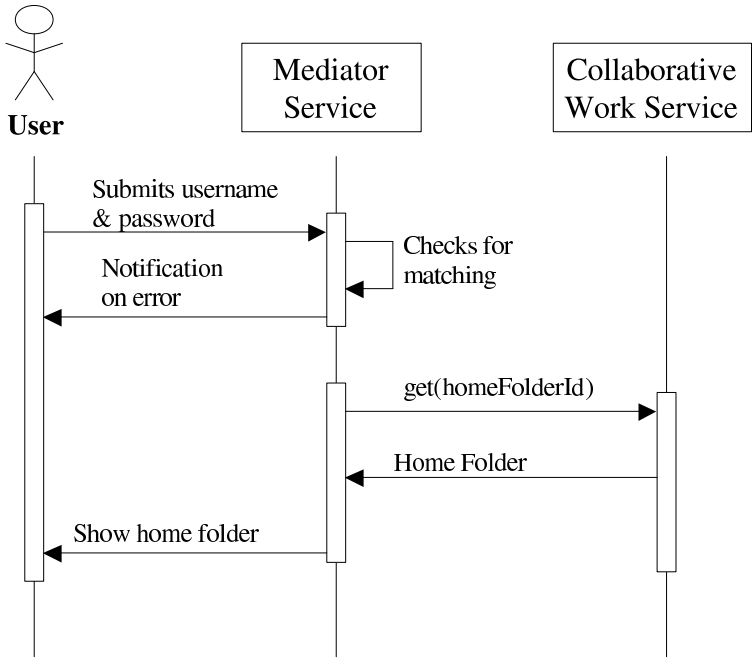

Figure 7.4: Login interaction diagram

Chapter 8

Rating Management Service

8.1 Functionality

The Rating Management Service (RMS) stores record ratings that take place within the Collaborative Work Service (CWS) and makes them available to other services. The RMS stores the ratings in form of a table where every row corresponds to a rating with entries for the record that has been rated, the user that made the rating, the date and time when the rating took place, the folder in whose context the rating was made, and finally the rating value itself.

The RMS makes its ratings available as 'projections' to users, records and folders, i. e. it produces upon request lists of

- all ratings that a particular *user* has made,
- all ratings that a particular *record* has received,
- all ratings that have taken place in a particular *folder*.

All these ratings lists contain only ratings that happened *after* a point in time that is specified in such a service request.

8.2 Process flow

The RMS offers no direct user operations, but only a service method interface to other Cyclades services. This API may be used by other services to interact with the RMS by way of remote procedure calls. Every method has a well defined signature giving its name, parameters and return value. In the following, we describe the invocation, execution and result of the RMS methods as a three-step interaction:

- **Method call:** shows method name and required parameters.
- **Execution:** describes RMS execution of method.
- **Return value:** describes result of method returned to the calling service.

We will use this schema for describing the linear process flow of the CWS API method executions.

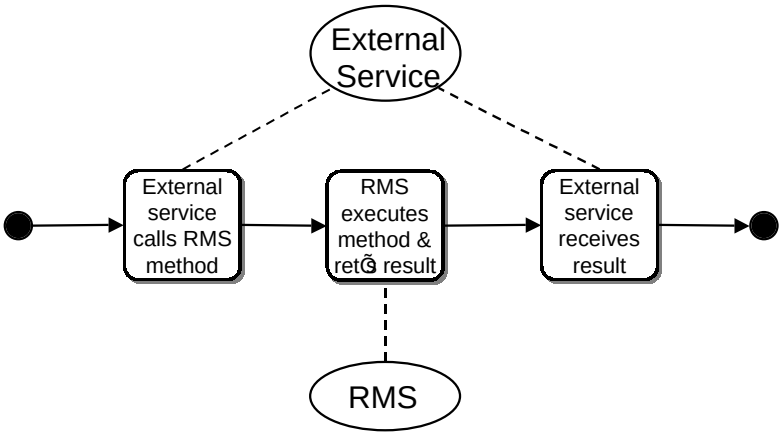

Figure 8.1: API method activity diagram

8.2.1 Save a rating

- **Method call:** saveRating(recordId, folderId, userId, ratingValue)
- **Execution:** The RMS generates a timestamp and stores the given rating into its rating table. An older rating of the same record by the same user within the same folder is overwritten by the new rating.
- **Return value:** Void.

8.2.2 Get all ratings of a user

- **Method call:** getUserRatings(userId, timestamp)
- **Execution:** The RMS extracts all ratings from its table that the given user has made since the time given.
- **Return value:** A list of triples containing the record identifier, the folder identifier and the rating value of the extracted ratings.

8.2.3 Get all ratings of a record

- **Method call:** getRecordRatings(recordId, timestamp)

- **Execution:** The RMS extracts all ratings from its table that the given record has received since the time given.
- **Return value:** A list of triples containing the folder identifier, the user identifier and the rating value of the extracted ratings.

8.2.4 Get all ratings within a folder

- **Method call:** getFolderRatings(folderId, timestamp)
- **Execution:** The RMS extracts all ratings from its table that have taken place within the given folder since the time given.
- **Return value:** A list of triples containing the record identifier, the user identifier and the rating value of the extracted ratings.

8.3 Internal architecture

8.3.1 Overview

The RMS implements a service API which other CYCLADES services may use to invoke RMS functionality via the XML-RPC protocol over HTTP. The RMS consists of four components:

- a standard Apache Web server for handling HTTP requests and responses,
- a MySQL relational database for storing and extracting the ratings,
- the RMS method call and response handling that deals with the XML-RPC protocol, parameter checking and method dispatching, and
- the RMS server that contains the specific method handlers.

The specific RMS components are implemented in Python.

When one of the other CYCLADES services such as the FRS sends a request in the form of a HTTP request in XML-RPC via the CGI interface of RMS Web server (c.f. Figure 8.2), the central script of the RMS method call and response handling component is executed which translates the XML-RPC into an internal procedure call, checks the method name and parameters, and dispatches the call to the RMS server. The RMS server connects to the RMS ratings database and calls the corresponding method handler to actually execute the procedure call (e. g. save a rating or get all ratings of a user) making use of the database. It produces the requested result which is then routed to the requesting service the same way back.

8.3.2 RMS API handling

XML-RPC call and response translation

XML-RPC calls that arrive via HTTP POST requests are translated into Python objects by the CGI request handler, a Python script that uses F. Lundh's `xmlrpclib`, version 0.99, from Secret Labs AB. Each incoming method call invokes the script's `call` procedure with the method name and its parameter tuple as parameters. For example, a method call to `saveRating` with record, folder and user identifiers and the rating value as parameters is translated to

$$\texttt{call}(\text{"}\texttt{saveRating}\text{"}, (\text{"}\texttt{AC\_http}://..../\texttt{038967}\text{"}, \text{"}\texttt{CW\_1200}\text{"}, \text{"}\texttt{CW\_3456}\text{"}, 4)).$$

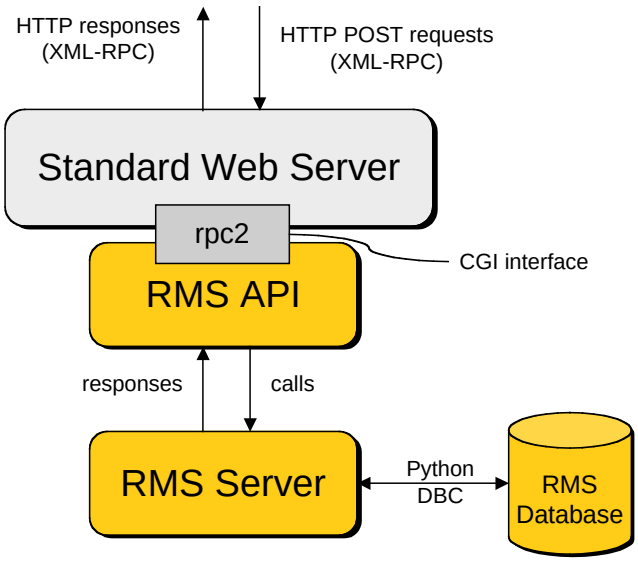

Figure 8.2: Internal architecture of RMS

The method's four parameters are packaged into an argument-list tuple and are passed to the `call()` method as a single argument. The return value of this function (a Python object) is translated back into a XML-RPC method response and returned to the calling service. When the execution of the `call` function raises an XML-RPC exception this is reported back to the calling service as an XML-RPC fault. Other exceptions are reported back as HTTP error 500 (`CYCLADES Server Error`).

Method name and parameter checking

Before the method call is dispatched to the RMS server, the method name is checked for validity and the method parameters are checked for number, data type, and null value (optional parameters). This is done recursively for nested parameter structures (`<struct>`s and `<array>`s). Errors raise an XML-RPC exception and are reported back to the calling service via an XML-RPC fault. Method name and parameter checking relies on a static description of the RMS API as (nested) Python objects (lists, tuples and dictionaries depending on the type of the parameters).

Method call dispatching

The `call()` procedure then dispatches each method call to the RMS server using that server's central `do_it` procedure with (checked) method name and parameters as arguments.

The following code fragment shows the `call` procedure.

```
def call(method, params):
```

```
    from api_def import *

    # method defined for RMS API?

    if method in API.keys():
       import checker

       # check nested parameter structure

       param_defs = API[method][1]
       checker.do_scalar_check(param_defs, params, method)

       # call 'super' function of RMS server

       func_obj = eval("rms_api.do_it")
       return apply(func_obj, (method, params, ))

    else:
       from xmlrpclib import Fault
       raise Fault(10002, "No such method: " + method)
```

8.3.3 RMS server

The RMS server contains the central `do_it` procedure and the method handlers that are named after the methods they handle with a `do_` prepended:

- `do_saveRating`
- `do_getUserRatings`
- `do_getRecordRatings`
- `do_getFolderRatings`

The central procedure connects to the RMS ratings database using `MySQLdb`, a Python DBC package for MySQL. It then calls the corresponding method handler with the original parameters plus the database connection and a database cursor needed for using the ratings database. The result (or an exception) is handed back to the RMS method call and response handling component.

The following code fragment shows the `do_it` procedure.

```
def do_it(method, params):
    import MySQLdb

    # connect to database

    try:
       db = MySQLdb.connect(host=HOST, user=USER, passwd=PASS, db=DB)
       cursor = db.cursor()
    except MySQLdb.Error:
       from xmlrpclib import Fault
       raise Fault(18001, '%s: %s' % (sys.exc_type, sys.exc_value))

    # call method handler
```

```
    func = eval('do_' + method)
    p_list = list(params)
    p_list.append(db)
    p_list.append(cursor)
    response = apply(func, p_list)

    # close database connection

    try:
       db.close()
    except:
       pass

    return response
```

As an example, the signature of the method handler for saving a rating is shown:

```
def do_saveRating(recid, fldrid, usrid, val, db, cursor):
```

8.4 Data and method specification

8.4.1 The RMS API

The RMS implements the methods `saveRating` for storing a rating in the RMS database, `getUserRatings` for retrieving all ratings of a specific user, `getRecordRatings` for retrieving all ratings of a specific record, and `getFolderRatings` for retrieving all ratings made in the context of a specific folder. These methods are methods of the abstract class RatingManagementService, which constitute the API of the RMS. These methods may be called from other services using the inter-service communication protocol XML-RPC.

- void saveRating(recordId, folderId, userId, ratingValue)
  Description: this method may be invoked in order to store a rating in the RMS rating table. The rating is specified by the identifier of the record that has been rated, the identifier of the folder that contained the record when it was rated, the identifier of the user that rated, and the rating value that was assigned to the record by the user. The rating timestamp is generated within the RMS.
  Input: recordId: a record identifier.
  folderId: a folder identifier.
  userId: a user identifier.
  ratingValue: an integer representing the rating value.

- (recordId, userId, value)* getFolderRatings(folderId, timestamp)
  Description: this method may be invoked in order to get all ratings that the records contained in a given folder received since a given point in time. The ratings are specified by the identifier of the record that was rated, the identifier of the user who rated, and the rating value assigned to the record by the user.
  Input: folderId: a folder identifier.
  timestamp: a point in time coded as an ISO 8601 date/time in UTC.
  Output: a list of triples containing a record identifier, a user identifier and a rating value each.

- (folderId, userId, value)* getRecordRatings(recordId, timestamp)
  Description: this method may be invoked in order to get all ratings that a given record

received since a given point in time, regardless of the folders in which the record was contained when rated (note that the same record may be contained in several different folders even at the same time). The ratings are specified by the identifier of the folder that contained the record when it was rated, the identifier of the user who rated, and the rating value assigned to the record by the user.
Input: recordId: a record identifier.
timestamp: a point in time coded as an ISO 8601 date/time in UTC.
Output: a list of triples containing a folder identifier, a user identifier and a rating value each.

- (recordId, folderId, value)* getUserRatings(userId, timestamp)
Description: this method may be invoked in order to get all ratings that a given user has assigned to arbitrary records since a given point in time. The ratings are specified by the identifier of the record that was rated, the identifier of the folder which contained the record when it was rated, and the rating value assigned to the record by the user.
Input: userId: a user identifier.
timestamp: a point in time coded as an ISO 8601 date/time in UTC.
Output: a list of triples containing a record identifier, a folder identifier and a rating value each.

8.4.2 The RMS Ratings Database Schema

The RMS Ratings database has one relation with the following schema:

- **ratings**
Stores rating information (which record, where, who, how and when)

rec_id	varchar(200)	the identifier of the record that has been rated
folder_id	varchar(16)	the identifier of the folder where the record has been rated
user_id	varchar(16)	the identifier of the user who has rated
value	int(2)	the rating value
stamp	datetime	the time when the record was rated

Ratings are identified by the record that has been rated, the user that has rated, and the folder that contained the record when it was rated. There is no rating identifier.

8.5 User interface

As mentioned above, the RMS has no user interface of its own. The CWS provides the GUI for the RMS. In the CWS users can select some records in the current folder, hit the Rate function, and select one of five pre-defined rating values (`very poor`, `poor`, `fair`, `good`, `excellent`), the rating is then shown on the GUI. Furthermore, the ratings are stored in the RMS database via the saveRating method of the RMS. Figure 8.3 gives an idea of the dialog the CWS provides for rating records.

8.6 Service interaction diagrams

The RMS service has two types of service interactions: get-requests where other services query the RMS for information; and put-requests where other services want to store information in the RMS. Subsequently we will present one example of a get-request: the `getUserRatings` request (c.f. Figure 8.4).

Further, we include an example for a put-request: the `saveRating` request (c.f. Figure 8.5).

Figure 8.3: Rate record user interface of CWS

8.7 Service implementation tools

In order to install and run the RMS you need the following software:

- Python, Version 2.0
  http://www.python.org/
- xmlrpclib, Version 0.99
  http://www.pythonware.com/products/xmlrpc/index.htm
- Apache Web server, Version 1.3
  http://www.apache.org/
- MySQL, Version 3.23.39
  http://www.mysql.com/
- MySQLdb, Version 0.91
  http://sourceforge.net/projects/mysql-python/

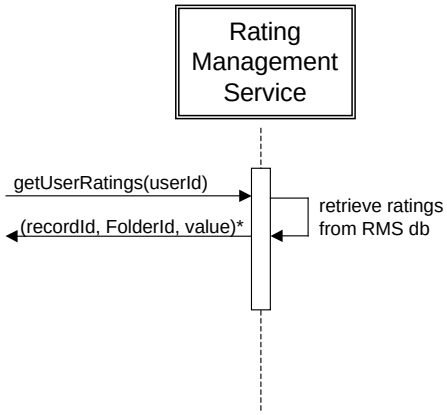

Figure 8.4: getUserRatings service interaction diagram

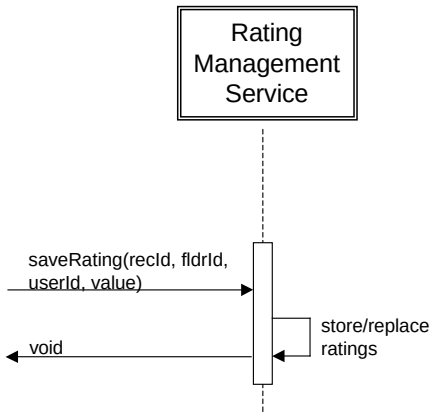

Figure 8.5: saveRating service interaction diagram

Chapter 9

Communication protocol

The communication protocol chosen for Cyclades is XML-RPC. In the following, the XML-RPC protocol is introduced, and it is compared with other available technology. Finally, the parameter encoding for Cyclades service interaction via the XML-RPC protocol is specified in some detail.

9.1 XML-RPC

XML-RPC is a simple protocol for implementing cross-platform, distributed applications. As its name suggests, communication between the distributed applications is done via remote procedure calls. The XML-RPC protocol is based on Internet standards: method calls and responses are transmitted using HTTP, and the bodies of the calls and responses are encoded in XML.

There are implementations available for numerous platforms (Unix, Windows and MacOS) and programming languages (C, C++, Java, Perl, PHP, Python, Tcl and others). Many of the implementations are open source and free to use.

The following example shows a simple XML-RPC method call (without the HTTP header).

```
<methodCall>
  <methodName>sample.sumAndDifference</methodName>
  <params>
    <param><value><int>5</int></value></param>
    <param><value><int>3</int></value></param>
  </params>
</methodCall>
```

Method calls consist of the method name and zero or more parameters as shown. A method response contains a single parameter or a fault consisting of a fault code and a fault string which are encoded in XML as

```
<methodResponse>
  <params>
    <param>
      <array>
       <data>
        <value><int>8</int></value></param>
        <value><int>2</int></value></param>
       </data>
      </array>
```

```
    </param>
  </params>
</methodResponse>
```

and, for the fault case,

```
<methodResponse>
  <fault>
    <value>
      <struct>
       <member>
        <name>faultCode</name>
        <value><int>10000</int></value>
       </member>
       <member>
        <name>faultString</name>
        <value><string>No such method</string></value>
       </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

XML-RPC supports the following six scalar data types:

- `<int>`: a 32 bit signed integer
- `<boolean>`: either 1 (true) or 0 (false)
- `<string>`: a string containing any character except `<` and `&` which are encoded as `&lt;` and `&amp;`
- `<double>`: a double-precision signed floating point number
- `<dateTime.iso8601>`: date and time encoded in the compact version of the ISO 8601 standard
- `<base64>`: base64 encoded binary data

and two compound data types:

- `<struct>`: a collection of `<member>`s consisting of a `<name>` and a `<value>`. Names are strings and values may be of any XML-RPC data type. Nesting of structs and arrays is possible.
- `<array>`: a one-dimensional array of values encoded as a single `<data>` element with any number of `<value>`s. The values may be of any type including structs and arrays. Mixing of types in one array is possible.

A complete specification of the protocol including the HTTP header requirements is to be found at `http://www.xmlrpc.com/spec`. XML-RPC is a trade mark of UserLand Software.

9.2 Comparison with other protocols

In this section, XML-RPC is compared with other generic XML-based protocols. The comparison is rather brief and is kept at a high-level due to: (a) the large number of XML-based protocols available (b) the low-level intricacies of the individual proposals and (c) the domain-specificity of several of the proposals. The list of XML-based protocols is constantly expanding. The following are considered as generic XML protocols (in addition to XML-RPC):

- SOAP (http://www.w3.org/TR/SOAP/)
- Open WDDX (http://www.openwddx.org/)
- XMI (http://www.alphaworks.ibm.com/tech/xmiframework)
- Jabber (http://www.jabber.org/)
- ebXML (http://www.ebxml.org/)
- BizTalk (http://www.biztalk.org/)
- Wf-XML (http://www.oasis-open.org/cover/WFXML10a-Alpha.html).

Other XML-based protocols are specifically related to Web services:

- WSDL (http://www.w3.org/TR/wsdl)
- WIDL (http://www.w3.org/TR/NOTE-widl)
- UDDI (http://www.uddi.org/specification.html)

Other domain-specific protocols include:

- P3P (http://www.w3.org/P3P/)
- CCPP (http://www.w3.org/TR/NOTE-CCPP/)
- E-Speak (http://www.e-speak.net/).

A detailed comparison of all these protocols is beyond the scope of this deliverable. In the remainder of this section we briefly present SOAP and perform a comparison with XML-RPC for the purposes of implementing the functionality of Cyclades. SOAP is the closest competitor to XML-RPC, whereas other protocols such as WSDL and UDDI are complementary to the goals of both XML-RPC and SOAP and can be used for higher-level service descriptions.

9.2.1 SOAP (http://www.w3.org/TR/SOAP/)

According to its specification document, "SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses."

SOAP (Simple Object Access Protocol) is a Remote Procedure Call (RPC) protocol that uses standard protocols for transport—either HTTP for synchronous calls or SMTP for asynchronous ones. It is language and platform independent and uses XML for the encoding of transmitted data. It defines two types of messages, namely Request and Response. A SOAP request consists of the three parts mentioned above. A response is returned as an XML document within a standard HTTP reply. The XSI/XSD tagging scheme can be used optionally to denote the type of result.

SOAP extends the functionality of XML-RPC by making extensive use of namespacing and attribute specification tags in all parts of messages. One limitation of XML-RPC that is addressed in the SOAP proposal, is that of unnamed complex structures (structs and arrays). Arrays and structs can be named in SOAP, whereas in XML-RPC they are anonymous groupings of elements of mixed data types and anonymous sets of name-value pairs. Furthermore, SOAP allows users to define enumerated types and their own simple or complex data types.

9.2.2 Evaluation and Summary

XML-RPC, although quite humble in the goals it sets, is a simple and effective way for requesting and exchanging information using HTTP at the transport level. The design choice of XML-RPC was to be as simple as possible and still permit complex data structures to be exchanged and processed. We regard its simplicity, cleanliness and its short learning curve as its main advantages over SOAP which is quite complex and by no means a "lightweight" protocol. Furthermore, XML-RPC has been around for quite a while and is stable and open to community feedback. It should be noted that XML-RPC is not endorsed by any companies or organizations. In contrast, the main driving forces behind the development of SOAP are Microsoft and IBM. It has been submitted for endorsement to the World-Wide Web Consortium but its status remains that of a technical note and is not considered as a candidate recommendation. Although SOAP extends the functionality of XML-RPC by permitting the customization of the different message portions, we believe that the overhead it creates for heterogeneous systems to process SOAP messages is far greater than the benefits (at least for the needs of Cyclades). In addition, the specification of SOAP is not stable and may change at any time.

9.3 Parameter encoding

Since the XML-RPC protocol is to be used for the communication between the Cyclades services, the input and output parameters of the Cyclades service methods as specified in this document have to be encoded as XML-RPC data types with a commonly agreed upon encoding.

¿From the XML-RPC definition it is clear that the input parameter list of a method is a `<params>` construct consisting of zero or more single `<param>`s each having a `<value>` of a certain data type, so that

```
<methodCall>
  <methodName>listCollections</methodName>
  <params></params>
</methodCall>
```

and

```
<methodCall>
  <methodName>saveRating</methodName>
  <params>
    <param><value><string>CW_1001</string></value></param>
    <param><value><double>2.55</double></value></param>
  </params>
</methodCall>
```

are both legal method calls. Consequently, no specific construct is needed for encoding the 'tuple' of input parameters, even if it is empty.

This is not true for the output parameters. The definition states that the output of a method in the non-fault case is a `<params>` construct consisting of exactly one `<param>` having a `<value>` of a certain data type. That means that

```
<methodResponse>
  <params></params>
</methodResponse>
```

and

```
<methodResponse>
  <params>
    <param><value><string>CW_23456</string></value></param>
    <param><value><double>2.55</double></value></param>
  </params>
</methodResponse>
```

are *not* legal method responses. Consequently, an empty output value ('void') and a tuple of output parameters need a specific encoding.

A review of the signatures of the CYCLADES service methods show that there are the following kinds of parameters:

- *booleans*
- *integers,* e. g. fault codes
- *floating point numbers,* e. g. term weights
- *strings,* e. g. object names and descriptions
- *timestamps*
- *void* representing the empty return value
- *object identifiers,* e. g. identifiers of users, folders, records etc.
- *objects of a certain class,* e. g. a record or a query
- *tuples,* e. g. a pair of object identifier and string
- *lists* of object identifiers, objects or tuples

There is currently no use of binary data in CYCLADES method signatures.

We treat the encoding of the different kinds of parameters in the order they appear in the above list.

9.3.1 Booleans, integers and floating-point numbers

Parameters of these types are encoded in the corresponding XML-RPC scalar types `<boolean>`, `<int>` and `<double>`. Note that boolean *true* is encoded as

`<boolean>1</boolean>`

and boolean *false* is encoded as

`<boolean>0</boolean>`.

9.3.2 Strings

CYCLADES strings are meant to be Unicode strings. If no other encoding is given in the XML header of the XML-RPC message, the standard XML encoding utf-8 is used.

Note that for 8-bit strings entered into the system at the browser interface, it will be impossible to determine the correct interpretation since the interpretation is a browser setting not accessible to a service with a user interface. For instance, for the CWS a latin-1 interpretation is used as default.

9.3.3 Timestamps

Timestamps are encoded in the `<dateTime.iso8601>` data type which uses the compact version of the ISO 8601 standard, e. g.

`<dateTime.iso8601>20011209T22:10:01</dateTime.iso8601>`.

All timestamps in CYCLADES are in Universal Time Coordinated (UTC).

9.3.4 Void

Since there is no 'natural' way to encode void as a method response, an empty string will be used to encode a void method response, i. e.

```
<methodResponse>
  <params>
    <param><value><string></string></value></param>
  </params>
</methodResponse>
```

A confusion with an empty string as return value is not possible since a method is defined to either return void or a string.

9.3.5 Object identifiers

Object identifiers in CYCLADES are printable ASCII strings (Unicode ordinal numbers 32–127) with a length greater than 3. The first two characters denote the CYCLADES service which generated the identifier, the third character is a separator, an underscore ('_'), and the rest of the object identifier is the service internal object identifier.

Service denotations are:

- ME: Mediator Service
- CO: Collection Service
- FR: Filtering and Recommendation Service
- AC: Access Service
- SB: Search and Browse Service
- CW: Collaborative Work Service
- RM: Rating Management Service

Since object identifiers are strings, they are encoded like other strings, e. g.

`<string>CW_1001</string>`.

Note that utf-8 encoding of ASCII strings leaves them unchanged.

9.3.6 Objects of a certain Cyclades class

Examples of such objects are records and queries. They are encoded as `<struct>`s where the `<member>`s correspond to the class attributes. A record with attributes id, name and metadata is consequently encoded as

```
<struct>
  <member>
    <name>id</name>
    <value>
      <string>AC_http://arXiv.org/abs/alg-geom/9712032</string>
    </value>
  </member>
  <member>
    <name>name</name>
    <value>
      <string>A morphism of intersection homology</string>
    </value>
  </member>
  <member>
    <name>metadata</name>
    <value>
      <string>&lt;?xml version="1.0"?>&lt;record>...&lt;/record></string>
    </value>
  </member>
</struct>
```

Since nesting of `<struct>`s and `<array>`s is possible, attribute values may also be lists, tuples or other objects.

Class attributes may have no value assigned in a particular instance, e. g. the classifier label attribute of a record may be undefined as may be the query string of a query. Depending on the programming language used, such attributes may be simply missing or may have a default value assigned, usually a nil value (`Null` in Java, `None` in Python). There is no specific nil value in XML-RPC, and encoding nil as an empty string (as was done for 'void') would give rise to confusion. Therefore the `<member>`s corresponding to missing or undefined attributes are simply left out in the encoding of the object.

9.3.7 Tuples and lists

Tuples and lists are encoded as `<array>`s. Since `<array>`s may contain values of different types, and since nesting of a `<array>`s and `<struct>`s is possible, no problems arise with tuples containing objects, lists of objects or other tuples. As an example, the encoding of a list of pairs of terms and weights is given.

```
<array>
  <data>
    <value>
      <array>
       <data>
        <value><string>early</string></value>
        <value><double>0.2341</double></value>
       </data>
      </array>
```

```
    </value>
    <value>
      <array>
       <data>
        <value><string>roman</string></value>
        <value><double>0.5325</double></value>
       </data>
      </array>
    </value>
  </data>
</array>
```

9.3.8 Fault codes

Faults are reported in XML-RPC as a distinct type of method response: a struct having as members a fault code and a fault string. An example was given in the introduction.

In CYCLADES, there are general fault codes and strings to be used by all services in a unified way. Additionally, there are specific fault codes and strings that every service may define and use. CYCLADES fault codes start at 10000, the assignment to the services is as follows:

- 10,000 - 10,999: CYCLADES general fault codes
- 11,000 - 11,999: Access Service fault codes
- 12,000 - 12,999: Search and Browse Service fault codes
- 13,000 - 13,999: Filtering and Recommendation Service fault codes
- 14,000 - 14,999: Collection Service fault codes
- 15,000 - 15,999: Collaborative Work Service fault codes
- 16,000 - 16,999: Rating Management Service fault codes
- 17,000 - 17,999: Mediator Service fault codes

The exact values of fault codes and fault strings are to be determined during the course of the implementation.

9.3.9 Mapping of XML-RPC data types to Java and Python

In CYCLADES, two programming languages are used: Java and Python. For Java, the Helma XML-RPC implementation of H. Wallnöfer (http://xmlrpc.helma.org/) or its adaptation as Apache XML-RPC (http://www.apache.org/xmlrpc) is used. The XML-RPC data types map to the following Java data types:

- `<int>` maps to *java.lang.Integer*
- `<boolean>` maps to *java.lang.Boolean*
- `<string>` maps to *java.lang.String*
- `<double>` maps to *java.lang.Double*
- `<dateTime.iso8601>` maps to *java.util.Date*
- `<struct>` maps to *java.util.Hashtable*

- `<array>` maps to *java.util.Vector*
- `<base64>` maps to *byte[ ]*

For Python, F. Lundh's xmlrpclib (http://www.pythonware.com/products/xmlrpc) from Secret Labs/PythonWare is used which has three extra classes to cater for some XML-RPC data types not readily available in Python. The mapping of data types is as follows:

- `<int>` maps to *int*;
- `<boolean>` maps to *xmlrpclib.Boolean*;
- `<string>` maps to *string* or *unicode*;
- `<double>` maps to *float*;
- `<dateTime.iso8601>` maps to *xmlrpclib.DateTime*;
- `<struct>` maps to *dictionary*;
- `<array>` maps to *list* or *tuple*;
- `<base64>` maps to *xmlrpclib.Binary*.

Appendix A

Terminology

We include here a glossary of technical terms used in this document and in the project. Words occurring in the term definitions and typeset in sans serif denote other terms defined in the glossary.

- *annotated document*
  An annotated document is an archive document that has been annotated (with comments, critiques, pointers to other documents, etc.) by one or more CYCLADES users (typically belonging to the same community or project).

- *archive*
  An archive is a set of records, each of which describes a document by means of metadata. An archive is uniquely identified by a string called the *archive identifier*.

- *archive document*
  See document.

- *collection*
  A collection is a set of records, defined by a set of archives and an optional *filtering query* (i.e. a selection criterion) which uses only attributes of the Dublin Core metadata standard. The members of the collection are all and only the records from the given archives which satisfy the query. A collection may be described by different metadata schemas, and different search services may be associated to each schema. From the user's point of view, a collection is a set of documents associated with specific formats of search and browse operations. In this perspective, a collection is described (i) by a set of criteria specifying which documents belong to the collection (*membership condition*) and (ii) by the format of the search and browse operations.

- *community*
  A community is a set of users sharing a common (scientific or professional) background or view of the world. Within CYCLADES, communities are characterized by a shared interest in documents and records. This common interest manifests itself as a hierarchy of community folders where records of common interest are stored and possibly annotated.

- *community folder*
  A community folder is a folder owned by a community. It can be accessed and possibly manipulated by several users who are members of the community.

- *document* (also called *archive document*)
  An archive document is a resource described by one or more metadata records that are contained in, and retrieved from, archives. Note that an archive need not contain actual documents; depending on the organisation hosting the archive, it might as well contain only

metadata records. An archive document may be a text (e.g. a paper, a report, a journal article), or may be expressed in any other kind of media.

- *external document*
  An external document is an arbitrary external file that can be stored in a project folder.

- *folder*
  A folder is a repository of metadata records (and, in the case of project folders, of documents too). A folder can be a private folder, or a project folder, or a community folder. Collections can be associated to folders; if this is the case, the scope of a query formulated by a user is, by default, the collection associated to the folder. Queries can be stored in folders, allowing the user to resubmit or edit them at any later time. Thus, a folder can also be seen as an "environment" corresponding to a certain topic of interest to its owner.

- *home folder*
  Each user has a special folder called the home folder, which constitutes the root of the folder hierarchy that is visible to this user.

- *metadata record*
  See record.

- *metadata schema*
  A metadata schema is a set of attribute definitions, each consisting of the attribute name and the attribute type.

- *open archive*
  An open archive is an archive that complies with the Open Archive Initiative standard.

- *private folder*
  A private folder is a folder owned by a single user. This folder can only be accessed by its owner, and is invisible to other users.

- *project*
  A project is a group of scholars working together, presumably at a common task. They might not only want to share records, but also documents, and might want to modify them, by working in a common workspace (i.e. in a common folder system).

- *project folder*
  A project folder is a folder owned by a project, i.e. accessible to all and only the members of a project. Only project folders can also contain documents, since private folders and community folders can contain only records.

- *record* (also called *metadata record*)
  Records are the entities contained in an archive. Each such record consists of a set of attributes and values describing a document, according to a specific metadata schema. One document can be described by several metadata records using different schemas. Note that the document described by the record might not be physically available from the archive that contains the record. However, this is seldom the case, and in general we assume that the document is indeed physically available from the same archive that contains a metadata record pointing to it.

- *rating*
  A rating is a relevance value assigned to a record by a user. The user may rate a record explicitly by assigning it a certain value, or implicitly by selecting the record from a query result list and storing it in a folder. In the latter case, the system assumes the record to be relevant, and automatically assigns to it a positive rating.

- *subfolder*
  A subfolder is a folder that is contained in another folder, called the *parent folder*. Each folder can have at most one parent folder. If a subfolder is copied to another folder, then the folder and its entire contents are actually copied (i.e. it is not the case that only an alias is created), thus creating a new, distinct folder.

- *term*
  A term is the minimal unit of meaning within a document. A possible approach is to make the set of terms contained in a document coincide with the set of *words* occurring in it, excluding "stop words". Another possible approach is to use, instead of words, *word stems*, i.e. the morphological roots of words. These different approaches will be experimentally compared in CYCLADES.

- *term weight*
  A term weight is a numeric value that denotes the importance of a term in a document. In CYCLADES we use non-binary term weights, i.e. numeric values belonging to the [0,1] interval.

- *user*
  A user is a person registered in the CYCLADES environment. A user has an identity, an e-mail address, a password, and a home folder. Users can have further attributes, such as affiliation, postal address, telephone number.