Hannes Ebner

Collaborilla

An enhancement to the Conzilla concept browser for enabling collaboration

July 21, 2006

M.Sc. Thesis at the Department of Computer and Systems Sciences
Corresponds to 20 weeks of full-time work

Royal Institute of Technology (KTH), Stockholm, Sweden

Contact information

Collaborilla Project

Internet: http://collaborilla.sourceforge.net

Author

Hannes Ebner
E-mail: hebner@kth.se

Supervisors

Matthias Palmér & Ambjörn Naeve
Knowledge Management Research Group
School of Computer Science and Communication
Royal Institute of Technology

Hercules Dalianis
Department of Computer and Systems Sciences
Stockholm University / Royal Institute of Technology

Department of Computer and Systems Sciences

Stockholm University /
Royal Institute of Technology
Forum 100
164 40 Kista
Sweden

Internet: http://www.dsv.su.se
Telephone: +46 8 16 20 00
Fax: +46 8 703 90 25

Abstract

The research field Knowledge Management (KM) is about improving methods to structure and filter information. A concept browser makes it possible to navigate through complex information structures. Conzilla is such a concept browser. It is designed to present knowledge, to set *concepts* into relations to each other, and to make browsing through the resulting *context-maps* possible. Conzilla allows information and content being tied to concepts and concept-relations.

The collaboration facilities in Conzilla are limited. Basic elements such as a lookup mechanism and lifecycle information for information structures are missing. Before knowledge can be contributed, it is necessary to make sure that dependencies are fulfilled and the history of an edited object is obtained. This thesis is about providing these missing parts.

To be able to load a container, the information about the location of a component has to be held by a central registry. To resolve eventually existing dependencies, it is also necessary to register the components and its references. This thesis provides a design which eliminates the existing restrictions. The aim is to allow real collaboration through a remote services infrastructure, realized with *Collaborilla*. The theoretical background is discussed as well as a practical solution, including a prototype of a remote collaboration service.

Contents

1 Introduction 1
1.1 Background 1
1.1.1 Concept Browsing 1
1.1.2 Terminology 2
1.2 Problem definition 2
1.2.1 Presentation of Information 2
1.2.2 Technical Aspects 3
1.3 Hypothesis 3
1.4 Expected results 4
1.5 Purpose 4
1.6 Method 5
1.7 How to read this document 5

2 State of the Art 6
2.1 Conzilla 6
2.1.1 Identifying components 6
2.1.2 Resolving 6
2.1.3 Referring 7
2.1.4 Collaboration 7
2.2 Technologies 8
2.2.1 Resolving and Referring 8
2.2.2 Information Directory 10
2.2.3 Data Storage 11

3 Elements of Collaboration 13
3.1 Collaborational Processes 13
3.1.1 Containers in Pairs 13
3.1.2 Loading context-maps 13
3.1.3 Publishing context-maps 14
3.2 Information to be Published 14
3.2.1 Storage Independency 14
3.2.2 Types of Information 14
3.2.3 Identifiers 14
3.2.4 Locations 15
3.2.5 RDF-Information 15
3.3 Versioning of Information 16

3.4 Publishing Data at Remote Locations 16
3.5 Appropriate Technologies 17
3.5.1 Information Directory 17
3.5.2 Services 18

4 Realization 19
4.1 Technologies to be used 19
4.2 Building an Information Directory 19
4.2.1 Tree Structure 19
4.2.2 Data Modeling 20
4.2.3 Versioning 22
4.3 Components of Collaborilla 22
4.3.1 Overview 22
4.3.2 Service 23
4.3.3 Client Interface 24
4.3.4 Protocol 24
4.3.5 Distribution and Documentation 25
4.4 Integration into Conzilla 25
4.4.1 Interfaces 25
4.4.2 Graphical User Interface 26

5 Limitations and Potentials 27
5.1 Limitations 27
5.1.1 Integration into Conzilla 27
5.1.2 Remote Files and Versioning 27
5.1.3 Rights Management 27
5.2 Server Protocol Enhancements 28
5.2.1 Stateless Protocol 28
5.2.2 Locking 28
5.2.3 Transactions 29
5.2.4 Implementation as Web Service 29

6 Conclusions 30
6.1 Overview 30
6.2 What has been done? 30
6.3 What remains to be done? 31

References 32

Abbreviations 34

Information Directory 36
A.1 OpenLDAP Software Suite 36
A.2 Object Classes and Attributes 36

Collaborilla 39
B.1 Interface 39
B.2 Protocol 45

List of Figures

1.1 Editing a context-map in Conzilla 2

2.1 Example resolver table 7

4.1 Collaborilla generic tree structure 20
4.2 Example URI mapped into an LDAP directory 21
4.3 Attributes of the custom object class 21
4.4 Revisions of a Collaborilla entry 22
4.5 Overview of the Collaborilla structure 23
4.6 Components of a Collaborilla deployment 24
4.7 Changes to the Conzilla class structure 25

B.1 Status codes of the Collaborilla protocol 47

1

Introduction

1.1 Background

1.1.1 Concept Browsing

The research field Knowledge Management (KM) is about improving methods to structure and filter information. A concept browser makes it possible to navigate through such a complex information structure. See [18] for a scientific paper describing how a concept browser is a new tool for improving Knowledge Management.

Conzilla is such a concept browser. It is being developed at the Knowledge Management Research (KMR) Group at the Royal Institute of Technology (KTH). The application is Open Source, everybody can download[1] and use it for free. As a concept browser, Conzilla is designed to present knowledge, to set *concepts* into a relation to each other and to make browsing through the resulting *context-maps* possible. Conzilla allows information and content being tied to specific concepts and concept-relations. The standard toolset includes the *Unified Modeling Language* (UML) [5, 8] and supports the creation of class, activity, use-case, and process diagrams. See the screenshot in figure 1.1 to get an idea of how information is presented by Conzilla.

The official website of Conzilla is an excellent resource to get more information on how to create context-maps with Conzilla.

The result of a recent redesign was Conzilla in Version 2.0. A context-map is not a single file anymore, it became an extendable entity. This allows the authoring and lifecycle of a context-map to be a collaborative process. Furthermore, to make collaboration useful it is important to know who modified which information at which time etc. This information is called provenance information. Today, elementary parts, such as lookup mechanisms and provenance information are missing for efficient collaboration. This thesis is about providing those missing parts.

[1] URL: `http://www.conzilla.org`

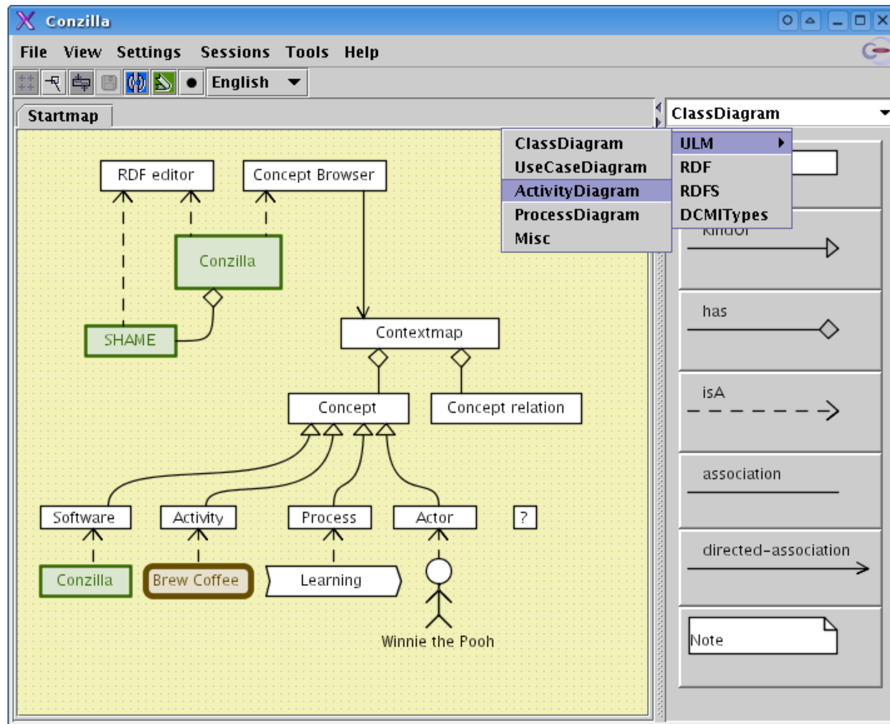


Fig. 1.1. Editing a context-map in Conzilla.

1.1.2 Terminology

According to [18, ch. 2], the following terms will be used in this thesis:

Concept .... Representation of some thing
Context.......... Graph containing concepts and concept-relations
Context-map..... Graphical representation of a context
Content.......... Information linked to a concept or a concept-relation
Component ...... Concept, concept-relation, context, context-map, or content

In the following it is referred to resources whenever components or containers are discussed.

1.2 Problem definition

1.2.1 Presentation of Information

An issue to be solved is the presentation of collaborational information to the user. It is complementary to the technical way of looking at a problem. Is has to be discussed which parts of the decision-making require user-interaction.

The basic questions related to the presentation of information are:

- How is the provenance information of a component presented?

- How should be dealt with conflicting information, for example different titles for a concept?
- At which point of the collaboration processes should the user be presented with information?
- Will the user be able to make a decision based upon the to be presented information under all circumstances?

1.2.2 Technical Aspects

Contrary to loading a web page identified by a *Uniform Resource Locator* (URL), the information that makes up a component is *not* contained in a single file uniquely identified by a URL. The information that makes up the component may be located in multiple containers[2] around the internet and new information may be expressed with time, independently of eventually existing previous information. To avoid broken references to containers, the identifiers of containers are not given as URLs directly. Instead, a lookup procedure is resolving identifiers into locations. Since the information of a single component is allowed to be spread out, it is important to keep track of who modified the information and when it was contributed.

The implementation of such collaboration capabilities raises several questions:

- How are components published?
- How is a specific component discovered?
- Which set of containers is relevant for a specific component?
- How is a container retrieved without breaking references?

1.3 Hypothesis

Prerequisites currently fulfilled by Conzilla:

- Components are identified via globally unique persistent identifiers.
- The information defining a component is held in one or several containers.

To be able to load a container, the information about its location has to be held by a central entity, comparable to a card index. The client just knows about the container's identifier, the location of it can be requested from this entity. I propose to introduce a remote *resolving service* and a client implementation in Conzilla to realize this approach.

In order to get the references right, it is necessary to register the components and its dependencies at another entity. The dependencies are part of

[2] Typically files

the information about a component, so it makes sense to manage the provenance information here as well. Provenance information has to contain the "contribution role", for example *original author* or *contributor*. It also has to hold the date of contribution and information about who the contributor was. Further information should be allowed but not forced. To present the provenance information to the user, the Conzilla user interface has to be adapted accordingly.

Comparable to a remote resolving service, a remote *referring service* should be introduced. Perhaps it is possible to hold all information at the same place and to bring both service together.

Hence, the containers needed for loading a component would be found through a two step procedure:

- Identifiers of relevant containers are found through the remote referring service, triggered by the identifier of the component.
- Containers' identifiers are resolved into locations via the remote resolving service.

1.4 Expected results

The expected outcome of this thesis is:

- A coherent design of how collaboration is done and perceived; a technical description as well as how it is experienced from a user perspective.
- A practical solution to find relevant containers of a specific component, including provenance information: a remote referring service.
- A practical solution find out where to a container is located, given its identifier: a remote resolving service.
- Improved visibility of the collaborational aspect in Conzilla; the presentation of provenance information.

1.5 Purpose

The collaboration possibilities in Conzilla should allow:

- Referencing context-maps via hyperlinks.
- Reusing concepts and concept-relations others have created.
- Extending context-maps that others have created.
- Adding content to others concepts, concept-relations and context-maps.
- Adding comments and further information on others concepts, concept-relations and context-maps.

For all these references, reuses, extensions, and additions it should be possible to find out who did them, what the purpose was, when the modification was done etc. Hence, the collaboration possibilities have to be made available in a way that preserves and allows inspection of provenance information.

1.6 Method

The research includes an analysis of the current abilities of Conzilla regarding handling of *Uniform Resource Identifiers* (URIs) and a possible integration of a remote referring and a remote resolving service to enable real collaboration. Therefore is it necessary to analyze possible storage solutions like WebDAV, see [9], or LDAP, which is specified in [26], with its free implementation OpenLDAP[3].

It is necessary to know about the data structure before thinking of storing the data somewhere. Which technology allows storage with minimal or no modifications to the structure? It is also reasonable to take a look at established resolving mechanisms such as DNS (see [15, 16]). Perhaps it is possible to use similar methods or techniques for the referring and resolving within Conzilla.

Research of the current state of the art within this area will be performed. If techniques exist which are suitable for this problem, they will be applied or adapted. Otherwise solutions will be developed. All implementations will be done in the programming language Java using an Open Source License. Any server solution will be deployed on Linux. An incremental development process with early prototyping is required.

1.7 How to read this document

In the first chapter the background information is given to introduce the reader into the topic and to create a picture on what this thesis is about. The next chapter "State of the Art" discusses the current state of Conzilla. Collaboration-related technologies are mentioned as well. In "Elements of Collaboration" the necessities for collaboration are evaluated. The following chapter "Realization" describes the actual implementation and realization of the previously discussed elements. "Limitations and Potentials" gives an overview of restrictions and possible enhancements, which leads into the last chapter "Conclusions", where the results of this thesis are being discussed. The technical realization is described in more detail in the appendices.

[3] URL: `http://www.openldap.org`

2

State of the Art

2.1 Conzilla

2.1.1 Identifying components

A component is either a concept, a concept-relation or a context-map.

Conzilla uses *Uniform Resource Identifiers* (URI) instead of Uniform Resource Locations (URL) for retrieving files. The idea behind is that it is acceptable to tell the user to choose a globally unique identifier, but it would be inconvenient to decide about a permanent location of a file upon creation, which would also increase the probability of having broken links within context-maps. The resulting approach of resolving a URI into a URL is more complicated but allows a much more flexible handling of data. If a file is moved, the references[1] do not have to be updated. Instead, the entry in the resolver is modified.

See [22, 21] and the *Conzilla tutorial*[2] for more detailed information.

2.1.2 Resolving

It is possible to specify a URL for each URI, which is not very efficient. To solve this issue, a local one-to-many resolver-table is used in addition. If the URI cannot be resolved directly, the resolver tries to resolve the URI one level above, see also the example in figure 2.1. This is performed until the URI can be resolved. The URL is concatenated then with the significant part of the to-be-resolved URI.

The only URI-scheme which is supported right now is URN:PATH[3], see [13] for details. To make the resolver-table work properly, all containers have to have a URN:PATH identifier.

[1] Many and hard to find references make it hard to update this information
[2] Linked on `http://www.conzilla.org/doc/`
[3] *Uniform Resource Name* (URN)

URI	Location
/org/conzilla	http://www.conzilla.org
/org/conzilla/local	file:/home/he/.conzilla2/local
/org/conzilla/people	http://people.conzilla.org

Fig. 2.1. An example of a resolver table.

The example in figure 2.1 holds several entries. If Conzilla tries to resolve the URI /org/conzilla/people/hannes/info.rdf, it looks for the best correlation in the local resolver-table. In this case, this is /org/conzilla/people. After combining the matched URI and the URL, the resulting URL is http://people.conzilla.org/hannes/info.rdf. The example URI above in URN:PATH notation would be urn:path:/org/conzilla/people/hannes/info.rdf.

2.1.3 Referring

In Conzilla it is possible to navigate between context-maps in two ways: by using *contextual-neighborhoods* and *hyperlinks*.

2.1.3.1 Contextual-neighborhoods

The contextual-neighborhood of a concept is the aggregation of all context-maps which refer to this concept. It is implicitly created and cannot be modified directly, it is defined through the usage of concepts by different context-maps. The list of context-maps in a contextual-neighborhood also depends on the loaded data within Conzilla. If Conzilla did not load a context-map which uses a specific concept, it will not show up in the contextual-neighborhood.

2.1.3.2 Hyperlinks

Hyperlinks can be created and modified by the user, they are under explicit control. Basically they work into the opposite direction of contextual-neighborhoods. Hyperlinks are used by concepts to refer to context-maps, whereas contextual-neighborhoods describe the usage of concepts within context-maps.

2.1.4 Collaboration

2.1.4.1 Data storage

Conzilla stores the necessary information in RDF[4] in two different containers: the *presentation container* and the *information container*. The presentation

[4] Resource Description Framework, see http://www.w3.org/RDF/

container keeps information about the graphical representation of concepts and concept-relations. The descriptive information like author, title, description, etc is stored in the information container. The containers may be located in the same file.

For the purpose of collaborating around context-maps it is important to be able to split components and spread the parts into several files. This allows for having a fine-grained control over context-maps. People can edit maps without touching the original information, the risk of destroying someone else's information is minimized, and all contributors have full control over the information they want to publish.

The separation into files for information and presentation, as well as a URI for concepts and layouts accordingly makes it possible to use separate tools for publishing information and presenting existing information.

2.1.4.2 Sessions

In Conzilla, context-maps are edited within *sessions*. Every session has its own *namespace*, which is used to generate globally unique identifiers for the components. Sessions are important for collaboration, as they guarantee a unique URI for each created component, provided that the namespace is chosen wisely. A session also holds information about which containers are used for presentation and information.

2.2 Technologies

2.2.1 Resolving and Referring

2.2.1.1 Domain Name System

The DNS (defined in [16]) and its extending *Resource Records*[5] offer a variety of additional fields which allow more than a simple *Hostname-to-IP* translation.

The DNS RR for specifying the location of services (DNS SRV) was defined in RFC 2782 [11]. The SRV RR allows the specification of servers depending on the service and the domain. Several different servers can be used in the service infrastructure of a single domain. The SRV-record works as a pointer to those servers. For more information on this RR see the RFC.

The *DNS-Based Service Discovery* (DNS-SD) has been discussed in an Internet-Draft, see [4]. DNS-SD offers (among other fields) a TXT record which is destined for optional data, it could be used for the information we need. A DNS message has a limited size of 512 bytes (defined in [16, section

[5] DNS RR, see `http://www.dns.net/dnsrd/rr.html`

2.3.4.]), which makes it too small for storing long paths or other information. This size restriction applies to UDP[6] connections only.

In Conzilla it is necessary to resolve a whole URI (described in section 2.1.2) and not just a hostname, the DNS does not seem to be applicable in general.

2.2.1.2 Persistant Identifiers

Persistant Identifiers (PI) are used to replace URLs and to create stable references where needed. With PI the time and effort to maintain a directory of links is reduced. It is commonly used for digital publications (e.g. in library databases), which get a worldwide unique identifier. Using PIs reliable references to documents are possible.

A PI consists of hierarchical elements, like *namespaces* (Namespace ID, NID) and *subnamespaces* (Subnamespace ID, SNID). Currently established PI systems are the *Handle system*, the Digital Object Identifier (DOI), Persistant URL (PURL), and Uniform Resource Names (URN).

Handle System

The Handle system[7], specified in RFC 3650 [25], was developed to be able to assign and manage PIs to digital resources on the Internet. The information associated to a handle (including its location) can be modified without changing the handle itself. The administration is decentralized, each handle may be administered by a different authority.
The structure of a handle is simple:

```
<Handle Naming Authority> "/" <Handle Local Name>
```

Each handle consists of a prefix (Handle Naming Authority) and a suffix (Handle Local Name). The prefix is a numerical code, referring to the institution. The suffix may be any string value.

Digital Object Identifier

The DOI system[8] is based on identifying and exchanging resources of intellectual property. At the same time DOI provides a technical and organizational framework, which allows the administration of resources and the linkage of authors and service providers.

---

[6] UDP is the standard protocol for DNS queries; the possibility to fall back on TCP may exist

[7] URL: `http://www.handle.net`

[8] URL: `http://www.doi.org`

The DOI system consists of 3 components: metadata, a DOI as persistant identifier, and the technical implementation of the Handle system. The structure of a DOI has been standardized (ANSI/NISO Standard Z39.84), see also the Handle system.

PURL

PURLs[9] are not persistant identifiers, but they can be transfered into existing standards like URN. From a technical point of view, PURL uses a *redirect* command of HTTP to resolve a PURL into a URL.

Well-known examples are Internet services like *tinyURL*[10], which are primarily designed to make a long URL short.

Uniform Resource Name

The URN system is designed to keep the complexity of the deployment as low as possible. Therefore the URN standard [24, 14] specifies how already existing namespaces (like URLs), numbering schemata or protocols (like HTTP) can be easily transfered or integrated into the URN schema.

A URN is composed of hierarchical elements, such as a NID, a SNID, and a Namespace Identifier Specific String (NISS). The following example shows a generic URN:

```
<URN>:<NID>:<SNID>-<NISS>
```

Examples in the wild are URN:NBN (National Bibliography Number) [12], an internationally administered namespace for national libraries, or URN:PATH [13], which is also used by Conzilla.

2.2.2 Information Directory

2.2.2.1 Requirements

To store the necessary information for the referring and resolving services a database backend is needed. Data can be stored using any available technologies, the question is which method offers the greatest efficiency and flexibility. An optimal solution would be to store information without bigger modifications, the transformation of available information to stored information should be as straight-forward as possible.

To get a picture of which technologies we can fall back on, the basic principles of common database systems have to be discussed. It would be also

[9] URL: `http://purl.oclc.org`
[10] URL: `http://www.tinyurl.com`

possible to store data in a plain filesystem, but this would ignore the availability of specialized systems (which provide greater flexibility) within this area.

Which storage technology is to be used will be discussed in the next chapter.

2.2.2.2 Database Management Systems

A *(Relational) Database Management System* (DBMS or RDBMS) is a commonly used technology to hold a large amount of data. A single database usually consists of several tables, which are associated to each other using key fields. This approach avoids having redundant information in the system, if the basic principles of database normalization are followed[11].

Depending on the structure of the database and its data, queries can be complex and thus make the design of the solution complicated. Versioning of data is not natively supported.

2.2.2.3 Lightweight Directory Access Protocol

Lightweight Directory Access Protocol (LDAP) is another widely deployed database technology. LDAP is an information directory, the tree is built out of the directory entries. Each entry consists of one or more attributes which hold the information. Each entry can be access through a unique identifier, the *Distinguished Name* (DN). The DN describes the exact position within the directory. [10] LDAP is optimized for read-access, which makes it a good solution for information directories with predominant read operations.

The design of LDAP allows the creation of entries as children of tree-nodes (resulting in subtrees), making it possible to use a previously created directory structure. Custom data types can be created if the native schemata of the LDAP distribution are not sufficient. Versioning of entries is not natively supported.

2.2.3 Data Storage

2.2.3.1 FTP

In addition to storing in a local filesystem, Conzilla also supports remote storage through FTP. This makes it possible to "collaborate" to a certain degree, it allows other contributors to open a file and make changes. Since it cannot be guaranteed that the file is opened by one contributor exclusively, information may get lost if changes are overridden by accident. Apart from that it is not

[11] See also "Codd's rules", `http://en.wikipedia.org/wiki/Codd%27s_12_rules`

possible to do a rollback to an earlier version if it turns out that the changes were not desired. This is problematic in a multi-user environment.

A benefit of FTP is that it is a common situation to have an FTP service running in parallel to an HTTP service. The usage of this protocol is straightforward, it is well supported by client libraries. The resolver-table in Conzilla can hold FTP or HTTP locations, allowing read-only access by using HTTP. However, due to lack of native versioning of files, FTP is of limited use in collaborative environments and limits the possibilities for designing collaborative software.

RFC 959 [23] is a good resource for more information on FTP.

2.2.3.2 WebDAV

An alternative to FTP is WebDAV, an extension to HTTP. It extends the read-only protocol with read-write capabilities. As the name of this standard[12] [9] says, WebDAV was designed to support authors and contributors in their collaborative work.

To avoid concurrent modifications, the protocol supports the locking of resources. This is one of the basic requirements for making distributed authoring possible. In order to access an earlier revision of a resource, WebDAV can be setup on top of a *Subversion*[13] (SVN) repository. When one or more files are stored at a WebDAV location, a new revision is created in the SVN repository, making older revision still available. The contributors know what has been changed by making use of *commit messages*[14].

WebDAV support can be enabled directly in a webserver[15], so there is no need to activate an additional service. The integration into the webserver reduces deployment and firewall issues, and has the advantage of being able to use its authentication and rights management (which is probably more mature than an own implementation from scratch).

[12] "Web-based Distributed Authoring and Versioning", specified in RFC 2518
[13] Open Source version control system, often described as the successor to CVS
[14] A message which is given at the time a resource is modified in a repository
[15] E.g. by using `mod_dav` in Apache

3

Elements of Collaboration

3.1 Collaborational Processes

3.1.1 Containers in Pairs

Conzilla and Collaborilla know about two different kinds of references: *original* and *relevant* containers. If a container is referred to as original, a depending resource cannot be loaded without it, since it contains essential information. A relevant container is not really a dependency, it is more an optional extension to a resource, it may be loaded or not. The user should have the possibility to decide about the usage of relevant containers.

Before a container can be fully loaded, its dependencies have to be taken into consideration. A presentation container is always dependent on an information container, but an information container can be loaded without loading the presentation container as well. To handle this problem and avoid loosing information in the information directory, a presentation container always has to refer to an information container as an original container. This way containers are always published in pairs.

3.1.2 Loading context-maps

Loading a context-map requires several steps and requests from the Collaborilla service:

1. A context-map is to be loaded.
2. Original and relevant URIs for the context-map are requested. These are the direct dependencies of the context-map.
3. The original URIs of the context-map's dependencies are requested. This enables the pairing of containers as described before.
4. The URIs are resolved into locations.
5. Conzilla is now able to load all containers and to show the context-map.

3.1.3 Publishing context-maps

Context-maps are published by pushing required information into the information directory using the Collaborilla service. The following elements have to be published:

1. Containers are uploaded to public storage space, e.g. FTP, WebDAV, etc.
2. Locations of the containers.
3. Dependencies of the context-map and the containers.
4. Provenance information (RDF-data) of the resources. (At the beginning of the publishing process the contributor is presented with an input dialog to provide such information.)

3.2 Information to be Published

3.2.1 Storage Independency

The information which is discussed in this chapter should be seen as independent from the storage technology. It should be possible to replace the storage backend at a later stage without having to change the design of the system too much.

3.2.2 Types of Information

Two different kinds of data are to be published: *formal* data and *informal* data.

Formal data will contain URI to URI and URI to URL mappings, and it will be used for the resolving of persistant identifiers into real locations. Another part of the the formal data will be relations between resources. A component can only be loaded if the containers in which it is included are loaded as well.

The informal data will consist of more complex information. We need information about the document itself, authors, contributors, a history of changes. Since Conzilla is based on RDF, why not use it also for this type of information? RDF can be stored in a database as well as in a flat-file, so the decision about the storage backend can be taken at a later point.

3.2.3 Identifiers

Conzilla uses globally unique identifiers, so it makes sense to use the same identifiers within Collaborilla to make information about a component available. The URIs in Conzilla can be specified in different notations. The URI is

written either like a URL [1] or in URN:PATH [13] notation. Since a URI can be given in arbitrary notation, it is necessary to keep the design of Collaborilla independent from it. This issue will be discussed in section 4.2.1.

As mentioned above, for the *Referring Service* it is important to store the dependencies of a component. For example, if `URI1` depends on[2] `URI2` and `URI3`, it is necessary to store this piece of information somehow. With such information available, Conzilla is able to request information about the component identified by `URI1`.

3.2.4 Locations

Resolving a URI into a URL (which is part of the *Resolving Service*) requires formal information about the location of the requested component to be stored. If such information is not available, it is perhaps possible to construct it indirectly using a parent identifier as described in section 2.1.2. One data-field per entry is enough to hold the location information.

The difference with respect to the current situation will be that Conzilla does not have to rely on a local resolver-table anymore. The remote resolver-table will be built out of published files and their locations, which will be made publicly available. The local resolver table can be kept anyway, it does not become unnecessary because of this. It can be used in “offline” mode, when there is no network connection or if public locations should be overruled.

3.2.5 RDF-Information

The RDF-information of a component represents the informal data. It holds information such as author, contributor, comments, etc.; enough information to follow a components history. Localization is supported, the information can be given in several languages. The included information depends on the implementation in Conzilla respectively the *SHAME*-library [3], which is used by Conzilla.

The storage of the RDF-data could happen in 2 ways. Either by not touching it and storing it as a whole (more thoughts on this in section 3.3), or by splitting it into its elements and storing it natively. This depends very much on the storage backend. For LDAP it is possible to follow the approach of generating RDF-models out of an LDAP directory [6] using OWL[4] (see [1] for the reference) ontologies.

---

[1] See [3] for the specification; the definition of the generic syntax of a URI [2] is interesting as well as it updates the definition of a URL

[2] This stands for: it cannot be loaded without those components being loaded as well

[3] URL: `http://kmr.nada.kth.se/shame/`

[4] A Web Ontology Language

To keep a high level of abstraction (see 3.2.1 why this is desirable), the handling of RDF-models should not be too specialized for a certain storage backend.

3.3 Versioning of Information

To be able to follow the history of a component, it is necessary to have deprecated information available. It should be easy to look at or revert to an earlier version of a specific resource. The reasons for reverting may be different, it could be unintentional mistakes as well as malicious manipulations. With a proper versioning it is also possible to follow for example the contributors and their changes on a timeline. The question *"Who modified which part, for which reason, and when?"* can be answered with the help of such a component-history. Read-only access to this history is desirable, as an accurate status of the deprecated information cannot be guaranteed otherwise.

To avoid unexpected side-effects, the history should contain full snapshots instead of differentials between two consecutive revisions. This makes it also possible to change the structure of the stored information without having to break backwards compatibility at a later point of development. Apart from this, saving differentials would not work with RDF-information as it would be necessary to decode this information instead of saving it as a "blob"[5]. Building a revision out of a congregation of differentials would impact the server performance badly (the more revisions the worse).

3.4 Publishing Data at Remote Locations

It does not make sense to publish the location of a file (see section 3.2.4) if the file itself cannot be accessed publicly. This would be the case if the file is kept at the author's computer only. So before the location of a file is published, the file has to be uploaded to a common storage (which can be accessed by the target group without problems) as well.

Conzilla already supports storing of files at remote locations. Right now this is restricted to the FTP [23], which does not allow versioning at all. Versioning is important, as discussed in section 3.3. It seems to make sense to enhance Conzilla with a protocol which natively allows the integration of a version control system. Right now this can be realized with WebDAV [9], see section 2.2.3.2.

[5] The RDF-information is not parsed and saved as it is

3.5 Appropriate Technologies

3.5.1 Information Directory

In section 2.2.2 possible backends for the information directory are discussed. Before a decision can be made, it is necessary to know about the most important criteria which are being discussed in this section.

How is the information identified?

As mentioned in section 2.1.1, Conzilla makes use of unique URIs to identify containers and components. It makes sense to use the same values as sort of "primary keys" to store and retrieve information. A different approach would require an additional layer, to translate between the URIs of Conzilla and an eventually introduced new naming schema.

Which type of data is to be stored?

It is necessary to take care of formal and informal data, see 3.2 at the beginning of this chapter. There is no binary data; the RDF-data is natively processed in the *Extensible Markup Language* (XML), so this and other required fields (e.g. URLs) can be stored as *String* values.

How important is the data structure?

The structure of the information is flat and non-hierarchical. The information for a container or a component is defined on a per-object basis, so it is just relevant for the object itself. The only connection between several entries is eventually existing dependencies as mentioned in section 3.2.3. The structure of the information is independent from the storage structure, which is discussed later.

Is versioning problematic?

According to section 3.3 versioning of information is necessary. However, it is not supported by the described database systems; a custom solution has to be developed for our system, which increases complexity but does not seem to be a problem.

Conclusion

Using URIs to identify entries is possible using a relational database as well as with an LDAP directory. The same applies to the data types and to the versioning, for these items the backend does not matter.

The data structure makes the difference. URIs as identifiers allow us to build a tree, similar to directories in a filesystem. LDAP (see section 2.2.2.3)

supports the creation of such a tree. What it looks like is described in the next chapter. In addition to this, the optimization of LDAP for read operations makes it a good choice for the backend of the information directory. There will be write operations as well (for example when information is published), but the use case of fetching information will occur much more often[6].

3.5.2 Services

3.5.2.1 Resolving and Referring

How are the remote resolving and referring services made available? Is it convenient to have two (independent) services or does it make sense to provide one service which covers both functionalities? Are the services different enough, e.g. is it possible to use them separately?

Both services are based on the same information directory, as well as the implementation of the custom versioning. It is possible to use them separately, but in practice the resolving service will always be utilized in connection with the referring service, this is part of the publishing process. Two services would also mean: two running service applications, two implemented protocols to communicate with the services, and two implementations in the client applications. Too much overhead for services for which the functionality is not separated enough.

The logical consequence is to implement resolving and referring in the same service, the convergence is sufficient.

3.5.2.2 Access

A protocol for communication between the client and the service has to be defined, the next chapter contains details on this. Conzilla will access the service through a generic interface, so the underlying technology[7] can be easily exchanged.

[6] Comparable to web pages: published once, read many times

[7] Conventional Client/Server communication, Service-Oriented Architecture (SOA), etc.

4

Realization

4.1 Technologies to be used

As a result of the discussion in the previous chapter the technologies for the implementation have been chosen.

The used programming language is Java. This choice makes the integration into Conzilla easier and is independent of the system architecture. The storage solution for the information directory is OpenLDAP, an Open Source LDAP implementation. It has also been decided to define a common protocol for both the referring and the resolving services.

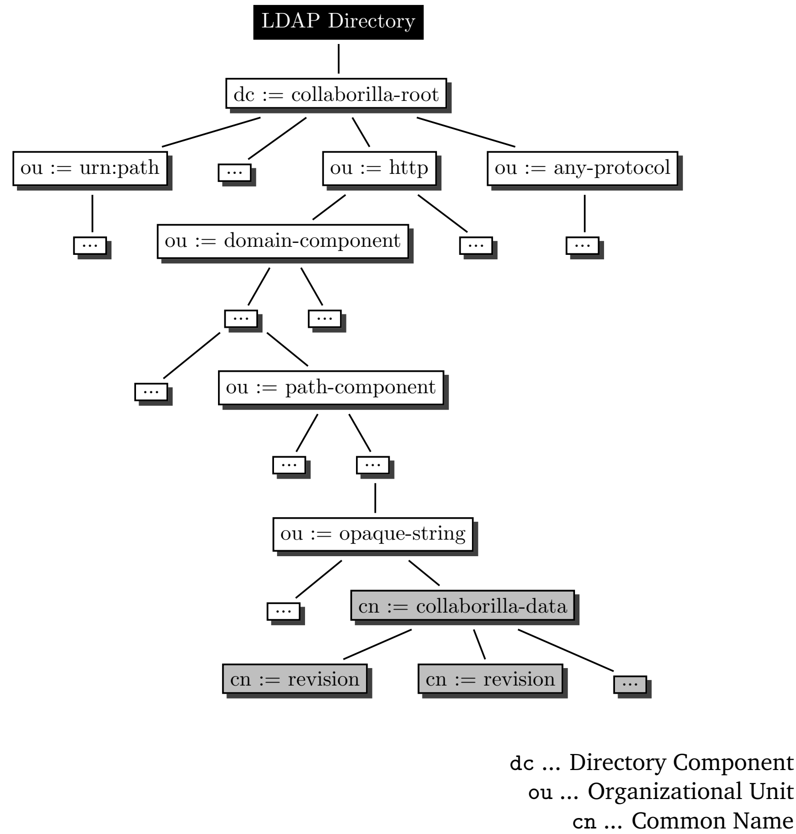
4.2 Building an Information Directory

4.2.1 Tree Structure

LDAP provides various object classes for creating entries, with *MUST* and *MAY* attributes. MUST attributes have to exist upon creation of an entry, whereas MAY attributes contain optional information and may be omitted. [10]

To build an LDAP tree structure which reflects the identifiers, it is necessary to tokenize the URIs used within Conzilla. Every single component of the URI has a corresponding entry. A URI-component can be a protocol, a part of a domain (top level and subdomains), a part of a path or an opaque string (e.g. filename). It is not possible to build a tree without splitting the URI, the result would be a flat structure.

Collaborilla uses the object class *Organizational Unit* (OU) to build the tree. The only required attribute is *ou*, whose value is set to the name of the URI-component. To store the meta-information (see 3.2.2 for the types of information), Collaborilla uses its own custom object class, which is described in the following section. In Figure 4.1 a generic illustration of the information directory is shown.



`dc` ... Directory Component
`ou` ... Organizational Unit
`cn` ... Common Name

Fig. 4.1. The Collaborilla generic tree structure.

Example

Given the URI `http://kmr.nada.kth.se/people/hannes`, the corresponding LDAP distinguished name (DN) is `ou=hannes,ou=people,ou=kmr,ou=nada,` `ou=kth,ou=se,ou=http`. There is also a directory component (DC) in the DN; it is omitted here because it depends on the configuration of the LDAP server installation. The domain- and path-components are not restricted to one entry. The tree-path to an entry has as much nodes as a URI has components.

The Collaborilla data is stored in a "meta-data only" node, the DN for this example is `cn=collaborilla-data,ou=hannes,ou=people,ou=kmr,ou=` `nada,ou=kth,ou=se,ou=http`. See figure 4.2 for a graphical representation of this example.

How versioning works is discussed in section 4.2.3.

4.2.2 Data Modeling

It is possible to store all required values with attributes of standard object classes. This brings several difficulties, as there are: not all suitable attributes

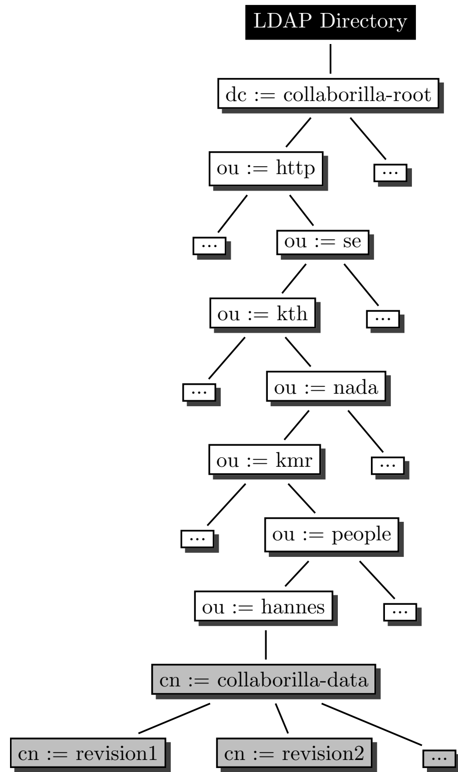


Fig. 4.2. An example URI mapped into an LDAP directory (resulting in a branch) with Collaborilla information nodes (gray).

are contained in one single object class. It is necessary to create an entry based on several object classes, just a few attributes of those classes would be used. This is not always possible as some object classes require attributes, which will not be used at all by Collaborilla. Dummy values would be required. To avoid this and to be able to use custom sizes for attribute (e.g. long String values for the RDF-data) the best and most elegant solution is to introduce a new object class with customized attributes.

Attribute	Description
`cn`	Identifier, number of revision)
`collabLocation`	URL, used for resolving
`collabUriOriginal`	URI, refers to original containers
`collabUriOther`	URI, refers to relevant containers
`collabContextRdfInfo`	RDF-data for a context
`collabContainerRdfInfo`	RDF-data for a container

Fig. 4.3. The attributes of the custom object class of Collaborilla.

The attributes of the Collaborilla object class are shown in figure 4.3. Some attributes (*cn* and *description*) are inherited from the standard object

class *top*. Explicit timestamp attributes are missing in the Collaborilla object class. The timestamps with the date and time of creation and last modification are internal LDAP attributes which are part of every entry.

All attributes except *cn* are optional. E.g. if a location is assigned to an identifier, the *collabLocation* attribute is used, no other attribute is necessary. All attributes for holding locations or identifiers are not restricted in the amount of values. This allows keeping several alternative locations (perhaps with different protocols) for one URI. Multiple values are also necessary for the URI attributes, a component can have more than one dependency.

4.2.3 Versioning

The most recent information is held in a *collaborilla-data* node. If values are modified, e.g. because of changed dependencies or a new location, the current information is copied into a new child-node (named after the number of the revision) and the parent node is modified with the new information. See figure 4.4 for an example with several revisions. The *collaborilla-data* node is a revision itself, but is not identified through a revision number.

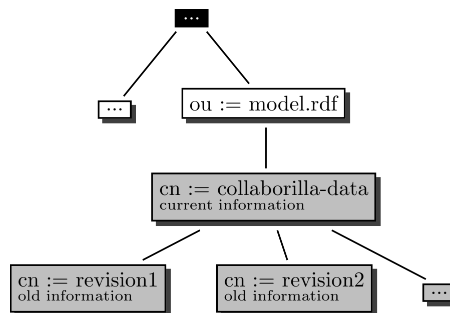


Fig. 4.4. A Collaborilla entry with current and outdated information in several revisions.

The structure (object class, attributes) of a revision is the same as a node with current information. This makes it easy to access older versions of information; the data of old entries should not be modified in order to keep an authentic history of an entry.

4.3 Components of Collaborilla

4.3.1 Overview

Collaborilla consists of two parts with different functionality: the client side and the server side. All classes are implemented in Java, so it is possible to

integrate Collaborilla into Conzilla without writing a separate client implementation.

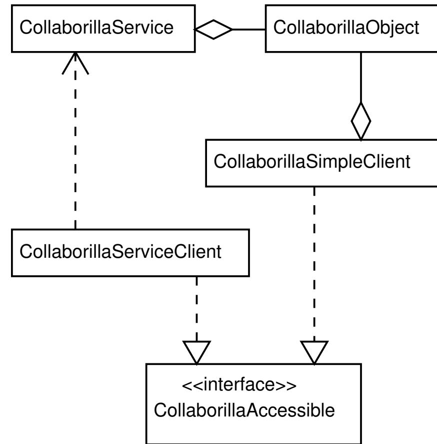


Fig. 4.5. An overview of the structure of Collaborilla.

In figure 4.5 the most significant parts of the architecture are shown, including how they work together logically. The classes *CollaborillaServiceClient* and *CollaborillaSimpleClient* are implementations of the interface *Collaborilla-Accessible*. They will be used from within Conzilla. The class *CollaborillaObject* is an abstraction of the *LDAP for Java* (JLDAP) library and contains the main intelligence behind Collaborilla. It implements e.g. building of an information tree and versioning.

The class CollaborillaService includes the implementation of the discussed services and makes uses of the previously mentioned classes. Its `main()` method starts a multi-threaded server and listens on a configurable TCP-port for incoming connections.

4.3.2 Service

A Collaborilla installation requires a configured and running LDAP service[1] to which the service can connect to store information in.

The service listens on a configurable TCP port for connections. It is multi-threaded and "speaks" a clear-text protocol, which is described in the sections 4.3.4 and B.2.

[1] Even possible on the same server

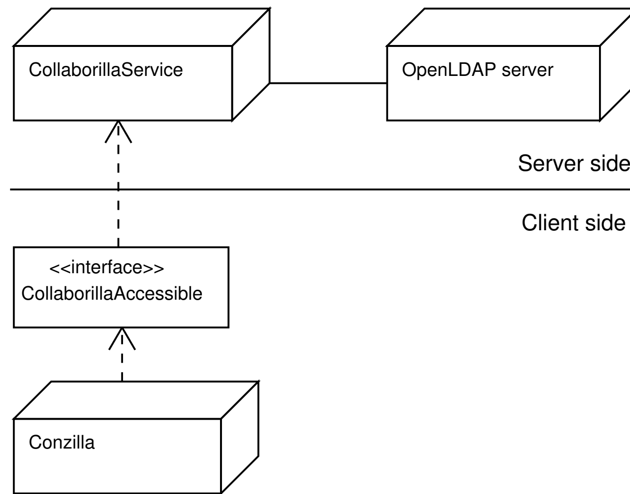


Fig. 4.6. The components of a Collaborilla deployment.

4.3.3 Client Interface

The client classes implement an interface (see section B.1 for details), which makes it possible to switch the client class in Conzilla without changes in the source code. Right now two clients are implemented. The class *Collaborilla-ServiceClient* connects to the Collaborilla service and is the client which is supposed to be used in a productive environment. The second implementation *CollaborillaSimpleClient* does not require an up and running Collaborilla service as it connects directly to the LDAP directory. This simple implementation is a proof of concept and not dedicated to a stable environment.

Error handling is done through an own exception class *Collaborilla-Exception*, it includes (and wraps) also eventually occurring errors in the LDAP directory.

4.3.4 Protocol

The protocol is held in clear-text, the service could also be accessed through a common terminal client. The commands can be grouped in read-only, modifying and temporary modifying commands.

Read-only commands request already existing information, whereas modifying commands add, remove or change information. A temporary modifying command is used to set the session between the client and the server into a different context. An example is to set the number of the requested revision. The retrieved information will be different afterwards.[2]

More details on the protocol can be found in the appendix, section B.2.

[2] Although the same URI is being accessed and nothing changes in the directory itself

4.3.5 Distribution and Documentation

The Collaborilla project is available as a Sourceforge project[3]. The source code including *Ant*[4] build scripts can be checked out of a Subversion repository. Collaborilla is distributed under the *GNU General Public License Version 2*[5] (GPL).

The source code is thoroughly documented, a documentation of all classes and methods can be generated with *Javadoc*[6].

4.4 Integration into Conzilla

4.4.1 Interfaces

Before Collaborilla can be integrated properly, changes to the Conzilla interface structure have to be done. In order to connect to the Collaborilla service, the client interface *CollaborillaAccessible* has to be integrated into Conzilla. In figure 4.7 the class and interface structure with the most important components is shown. The gray items have to be adapted or newly created.

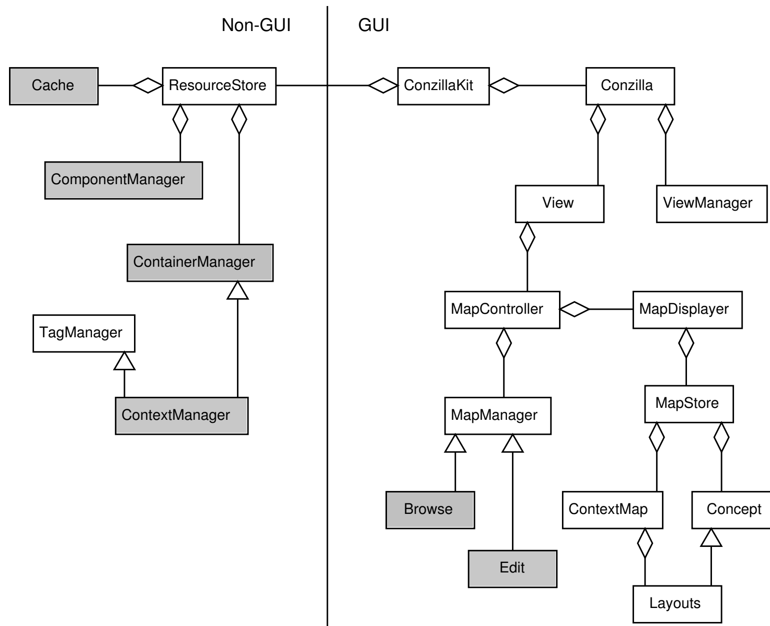


Fig. 4.7. Changes to the Conzilla class structure.

[3] URL: `http://collaborilla.sf.net`
[4] URL: `http://ant.apache.org`
[5] URL: `http://www.gnu.org/copyleft/gpl.html`
[6] URL: `http://java.sun.com/j2se/javadoc/`

The functionality of the *ContainerManager* will be extended by a new interface called *RemoteContainerManager*. It provides all necessary methods to retrieve and publish public information. The *ComponentManager* will be extended by a *RemoteComponentManager* in the same way. In addition to methods for publishing and unpublishing, there will be methods to request the locations, associated container information (dependencies), RDF-information, as well as the dates of creation and last modification.

The *ContextManager* and the *EditMapManager* have to be modified to make use of the newly available functionality. It is also useful to implement a caching mechanism to decrease the number of requests to the Collaborilla service.

4.4.2 Graphical User Interface

To be able to maintain collaboration data the Graphical User Interface (GUI) of Conzilla has to be adapted. Two different user interfaces are necessary: one for manipulating the remote resolver table, and a second for providing provenance information.

The already existing possibility of maintaining a local resolver table does not have to be removed. It can be kept for using Conzilla in "offline mode"[7] or to override the remote resolver table by having a higher priority for locally specified locations.

Remote Resolver Table

The User Interface (UI) for editing the remote resolver table can be similar to the already existing one, see 2.1.2. It should include the possibility to publish locations to the server, as well as functionality to unpublish or modify already published locations.

Provenance Information

When publishing e.g. a context-map, it is important to announce who the author or contributor was, and to give some background information. The *SHAME* library[8] provides editors and query interfaces for RDF metadata and is used by Conzilla. SHAME can be used to request the needed information from the publisher, see also section 3.2.5.

[7] There is no "offline mode" in the current version of Conzilla
[8] Standardized Hyper Adaptable Metadata Editor, `http://shame.sf.net`

5

Limitations and Potentials

5.1 Limitations

5.1.1 Integration into Conzilla

The integration into Conzilla as it is discussed in section 4.4 has not been carried out yet. One of the outcomes of this thesis is a prototype of the Collaborilla service.

5.1.2 Remote Files and Versioning

Conzilla's only supported remote storage protocol is FTP, which does not support integration into a *Revision Control System* (RCS). In order to maintain the history of a file, it is necessary to enhance Conzilla with the WebDAV protocol, see also section 2.2.3.2. WebDAV supports integration with the modern RCS Subversion.

5.1.3 Rights Management

Rights management has not been implemented, nor has it been discussed. This would exceed the scope of this thesis. The current implementation of the Collaborilla service allows anybody to change anything. The problem of malicious manipulation of published information is existant, but the versioning of data in the information directory lessens this design weakness.

Write-access to published files is more difficult as it requires authenticated admission e.g. to the FTP service where the file has been published at. With WebDAV it is possible to use the webserver's authentication mechanisms as well.

5.2 Server Protocol Enhancements

5.2.1 Stateless Protocol

The Collaborilla protocol is stateful. After submitting a URI all following commands operate on the entry of the same URI until a new URI is sent to the server. If data is requested for example with a *GET* command, the server expects a *URI* command to be sent before. This makes at least 2 commands necessary for simple read operations, which can be avoided by using stateless read-only commands. Similar to *HTTP* all necessary data can be sent in a single command to the server. An approach like the Representational State Transfer (ReST, see [7]) would be appropriate for doing this.

A generic description of single-command read operations:

```
<OPERATION> <URI> <REVISION>
```

To retrieve the locations for the URI `http://conzilla.org/concepts` from revision *2* it is now necessary to send 3 commands:

```
URI http://conzilla.org/concepts
SET REVISION 2
GET URL
```

With a stateless protocol enhancement it is enough to send one single command:

```
GETURL http://conzilla.org/concepts 2
```

5.2.2 Locking

It can lead to inconsistent data in the directory if two or more clients access the same entry with a modifying command at the same time. To avoid this, an entry should be accessed in a mutual exclusive way, similar to synchronized thread-programming. To achieve this a URI could be locked automatically for the time it is accessed. Alternatively a locking command could be sent by a client manually before a modification is done. This requires the client to unlock an entry properly before closing the connection, which can lead to problems if a connection is closed unexpectedly. Considering this, a server-internal automatic locking mechanism might be better.

Locking can be realized in different ways: locking by setting an attribute of the LDAP entry or by doing it inside the server without touching the LDAP entry. If an attribute is set it has to be analyzed how locking impacts the server performance under high load, as it requires a locking and an unlocking operation on the LDAP directory. If it is done within the server the performance will not be affected noticeably. However, this can cause problems if the directory is not just accessed by the server, but also by for example a Web Service.

5.2.3 Transactions

If several modifying commands are sent to the server consecutively and an error occurs while executing one of them, the client application has to rollback to an earlier revision manually. This produces overhead in the client application and can be avoided by introducing transactional updating of data as we know it from Relational Database Management Systems (RDMS). An implementation of the *ACID*[1] *model*, as it is known from database theory, would be reasonable.

Transactions also solve the problem with concurrent write operations, as described in the section before. Before sending a batch of commands, a transaction is started automatically by the command to create a new revision. After the commands are sent, the client completes the query by sending *COMMIT* or *ROLLBACK*, depending on the intention. If all commands are successful, the server returns an *OK*. If one or more commands fail, or the client disconnects without committing, the service performs an automatic rollback, without making any client interaction necessary.

5.2.4 Implementation as Web Service

Instead of using our own protocol and the *CollaborillaService* we could implement a Web Service using the SOAP[2], called *CollaborillaWebService* for example. This would result in a stateless protocol similar to the approach described in section 5.2.1, the difference is the technique for transmitting the data.

An advantage of Web Services in the context of reachability is the used port number. It is possible to integrate it into an already existing HTTP-server, so it can be accessed through the standard HTTP or HTTPS ports[3]. Lots of firewalls are configured very restrictively, so it can happen that the access to services which use non-standard ports (like CollaborillaService) are blocked. CollaborillaService could also be configured to listen on port 80, but a captious packet-filter would detect that this is not HTTP and block it.

[1] Atomicity, Consistency, Isolation, Durability
[2] Simple Object Access Protocol, specifications at `http://www.w3.org/TR/soap/`
[3] TCP port 80 respectively port 443

6

Conclusions

6.1 Overview

The purpose of this thesis was to extend Conzilla with collaboration facilities. The task was to place the cornerstones for optimal collective performance as it is required in modern working groups. This document provides solutions for the questions which have been raised in the introductive chapter.

The main goal was to provide a coherent design of an infrastructure to enable collaborative work, which lead to a prototype to show that a practical solution can be based on that design. This prototype contains remote referring and remote resolving functionalities, which have been implemented in the Collaborilla service.

6.2 What has been done?

By using the Collaborilla service it is possible to find the relevant containers of a specific Conzilla component to fulfill eventually existing dependencies. The possibility to resolve an identifier into a real location makes it possible to load containers no matter where they are stored. A central register like the Collaborilla service helps to avoid redundant information and supports efficient reuse of already existing components. The meta-data (provenance information) is based on the contribution role.

The questions from the first chapter can be answered now:

- Information about the location of a component is held by the Collaborilla service and can be requested by Conzilla, which just knows the component's identifier.
- Components and their dependencies are registered at the Collaborilla service and make it possible to entirely load a context-map, including the resources it is dependent on.

- The “contribution role” of an author is described by the provenance information, which is held by the Collaborilla service. The provenance information allows to find out who contributed which information, for which purpose, as well as the time and date of the contribution.

6.3 What remains to be done?

The integration of Collaborilla into Conzilla requires changes to the Conzilla class structure and has not been carried out during this thesis. To be specific, the ability to publish and unpublish context-maps and their meta-data, as well as the utilization of the remote resolver table is missing. Creation and modification of provenance information and the remote resolver table is depending on user interfaces which have to be created.

Semantic collaboration in the *Human Semantic Web* [19] requires strategies to realize *conceptual calibration* as discussed in [17]. The evolution of Conzilla and Collaborilla makes it necessary to think about supporting discourse management as well, as this is a cornerstone of serious collaboration which should not be underrated.

With such objective targets, Collaborilla will contribute well to e-learning platforms[1] and give collaborative modeling a new perspective.

[1] See [20] for a presentation of an infrastructure, an architecture, and adequate frameworks and tools

References

1. S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. *OWL Web Ontology Language*. Recommendation, World Wide Web Consortium (W3C), February 2004.
   URL: `http://www.w3.org/TR/owl-ref/`.
2. T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. Request for Comments 3986, Internet Engineering Task Force, January 2005.
   URL: `http://www.ietf.org/rfc/rfc3986.txt`.
3. T. Berners-Lee, L. Masinter, and M. McCahill. *Uniform Resource Locators (URL)*. Request for Comments 1738, Internet Engineering Task Force, December 1994.
   URL: `http://www.ietf.org/rfc/rfc1738.txt`.
4. S. Cheshire and M. Krochmal. *DNS-Based Service Discovery*. Internet-Draft, Internet Engineering Task Force, June 2005. `draft-cheshire-dnsext-dns-sd-03.txt`.
5. Committee JTC 1/SC 7. *Unified Modeling Language (UML) Version 1.4.2*. Standard ISO/IEC 19501:2005, International Organization for Standardization, April 2005.
6. S. Dietzold. *Generating RDF Models from LDAP Directories*. Crete, Greece, 2005. Proceedings of the SFSW 05 Workshop on Scripting for the Semantic Web.
7. R. T. Fielding. Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine, 2000.
   URL: `http://www.ics.uci.edu/~fielding/`.
8. M. Fowler. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley, 2004.
9. Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. *HTTP Extensions for Distributed Authoring – WEBDAV*. Request for Comments 2518, Internet Engineering Task Force, February 1999.
   URL: `http://www.ietf.org/rfc/rfc2518.txt`.
10. B. Greenblatt. Internet Directories: How to build and manage applications for LDAP, DNS, and other directories. Prentice Hall, 2001.
11. A. Gulbrandsen, P. Vixie, and L. Esibov. *A DNS RR for specifying the location of services (DNS SRV)*. Request for Comments 2782, Internet Engineering Task Force, February 2000.
    URL: `http://www.ietf.org/rfc/rfc2782.txt`.
12. J. Hakala. *Using National Bibliography Numbers as Uniform Resource Names*. Request for Comments 3188, Internet Engineering Task Force, October 2001.
    URL: `http://www.ietf.org/rfc/rfc3188.txt`.
13. D. LaLiberte and M. Shapiro. *The Path URN Specification*. Internet-draft, Internet Engineering Task Force, September 1995.
    URL: `http://www.hypernews.org/~liberte/www/path.html`.

14. R. Moats. *URN Syntax*. Request for Comments 2141, Internet Engineering Task Force, May 1997.
URL: http://www.ietf.org/rfc/rfc2141.txt.
15. P. Mockapetris. *Domain Names - Concepts and Facilities*. Request for Comments 1034, Internet Engineering Task Force, November 1987.
URL: http://www.ietf.org/rfc/rfc1034.txt.
16. P. Mockapetris. *Domain Names - Implementation and Specification*. Request for Comments 1035, Internet Engineering Task Force, November 1987.
URL: http://www.ietf.org/rfc/rfc1035.txt.
17. A. Naeve. *The Garden of Knowledge as a Knowledge Manifold - A Conceptual Framework for Computer Supported Subjective Education*. Technical report, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 1997.
URL: http://kmr.nada.kth.se/papers/KnowledgeManifolds/cid_17.pdf.
18. A. Naeve. *The Concept Browser - a new form of Knowledge Management Tool*. Lund, Sweden, October 2001. Proceedings of the 2nd European Web-based Learning Environments Conference (WBLE 2001).
19. A. Naeve. *The Human Semantic Web - Shifting from Knowledge Push to Knowledge Pull*. *International Journal on Semantic Web & Information Systems*, 1(3):1–30, July-September 2005.
URL: http://kmr.nada.kth.se/papers/SemanticWeb/HSW.pdf.
20. A. Naeve, M. Nilsson, M. Palmér, and F. Paulsson. *Contributions to a public e-learning platform: infrastructure; architecture; frameworks; tools*. *International Journal of Learning Technology*, 1(3):352–381, 2005.
URL: http://kmr.nada.kth.se/papers/SemanticWeb/Contrib-to-PeLP.pdf.
21. M. Nilsson. *The Conzilla Design - The Definitive Reference*. Technical report, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 2000.
URL: http://conzilla.org/doc/conzilla-design/conzilla-design.html.
22. M. Nilsson and M. Palmér. *Conzilla - Towards a concept browser*. Master's thesis, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 1999.
URL: http://kmr.nada.kth.se/papers/ConceptualBrowsing/cid_53.pdf.
23. J. Postel and J. Reynolds. *File Transfer Protocol (FTP)*. Request for Comments 959, Internet Engineering Task Force, October 1985.
URL: http://www.ietf.org/rfc/rfc959.txt.
24. K. Sollins and L. Masinter. *Functional Requirements for Uniform Resource Names*. Request for Comments 1737, Internet Engineering Task Force, December 1994.
URL: http://www.ietf.org/rfc/rfc1737.txt.
25. S. Sun, L. Lannom, and B. Boesch. *Handle System Overview*. Request for Comments 3650, Internet Engineering Task Force, November 2003.
URL: http://www.ietf.org/rfc/rfc3650.txt.
26. M. Wahl, T. Howes, and S. Kille. *Lightweight Directory Access Protocol (v3)*. Request for Comments 2251, Internet Engineering Task Force, December 1997.
URL: http://www.ietf.org/rfc/rfc2251.txt.

Abbreviations

ACID .... **A**tomicity, **C**onsistency, **I**solation, **D**urability
CN .... **C**ommon **N**ame
CVS .... **C**oncurrent **V**ersions **S**ystem
DBMS .... **D**ata**b**ase **M**anagement **S**ystem
DC .... **D**irectory **C**omponent
DNS RR .... **DNS R**esource **R**ecords
DNS .... **D**omain **N**ame **S**ystem
DN .... **D**istinguished **N**ame
DOI .... **D**igital **O**bject **I**dentifier
FTP .... **F**ile **T**ransfer **P**rotocol
GPL .... GNU **G**eneral **P**ublic **L**icense
GUI .... **G**raphical **U**ser **I**nterface
HTTPS .... **H**ypertext **T**ransfer **P**rotocol over **S**ecure Socket Layer
HTTP .... **H**ypertext **T**ransfer **P**rotocol
ID .... **Id**entifier
IETF .... **I**nternet **E**ngineering **T**ask **F**orce
JLDAP .... **J**ava **LDAP** Class Libraries
KMR .... **K**nowledge **M**anagement **R**esearch **G**roup
KTH .... **K**ungliga **T**ekniska **H**ögskolan, english: Royal Institute of Technology, Stockholm
LDAP .... **L**ightweight **D**irectory **A**ccess **P**rotocol
LDIF .... **L**DAP **D**ata **I**nterchange **F**ormat
NID .... **N**amespace **ID**
NISS .... **N**amespace **I**dentifier **S**pecific **S**tring
OID .... **O**bject **I**dentifier
OU .... **O**rganizational **U**nit
OWL .... **W**eb **O**ntology **L**anguage
PI .... **P**ersistant **I**dentifier
PURL .... **P**ersistant **URL**
RCS .... **R**evision **C**ontrol **S**ystem
RDBMS .... **R**elational **D**ata**b**ase **M**anagement **S**ystem
RDF .... **R**esource **D**escription **F**ramework
ReST .... **Re**presentational **S**tate **T**ransfer
RFC .... **R**equest **f**or **C**omments
SHAME .... **S**tandardized **H**yper **A**daptable **M**etadata **E**ditor
SNID .... **S**ub**n**amespace **ID**

SOAP . . . . **S**imple **O**bject **A**ccess **P**rotocol
SOA . . . . **S**ervice-**O**riented **A**rchitecture
SVN . . . . **S**ub**v**ersio**n**
TCP . . . . **T**ransmission **C**ontrol **P**rotocol
UI . . . . **U**ser **I**nterface
UML . . . . **U**nified **M**odeling **L**anguage
URI . . . . **U**niform **R**esource **I**dentifier
URL . . . . **U**niform **R**esource **L**ocator
URN . . . . **U**niform **R**esource **N**ame
W3C . . . . **W**orld **W**ide **W**eb **C**onsortium
WebDAV . . . . **Web**-based **D**istributed **A**uthoring and **V**ersioning
XML . . . . E**x**tensible **M**arkup **L**anguage

A

Information Directory

A.1 OpenLDAP Software Suite

Collaborilla uses the OpenLDAP[1] software suite. OpenLDAP supports standard LDAP and includes the stand-alone LDAP daemon *slapd*, the stand-alone LDAP update replication daemon *slurpd* and client libraries implementing the LDAP protocol.

Collaborilla makes use of JLDAP, an associated OpenLDAP project which provides LDAP access from within Java applications and was contributed by Novell.

A.2 Object Classes and Attributes

Object Identifiers

In LDAP OIDs[2] (Object Identifiers) are used to uniquely identify the components of an LDAP directory, such as object classes, attributes, syntaxes, matching rules, just to mention the most important ones.

The used space for the OIDs is temporary, experimental, OpenLDAP specific, and should not be propagated publicly.

```
objectIdentifier CollaborillaLDAPRoot 1.3.6.1.4.1.4203.666
objectIdentifier CollaborillaLDAPAttrType CollaborillaLDAPRoot:1
objectIdentifier CollaborillaLDAPObjClass CollaborillaLDAPRoot:3
```

Custom Attributes

If the object class makes use of attributes which are not included in the LDAP server distribution, they have to be defined in the schema-file before the object class is specified itself.

[1] URL: `http://www.openldap.org`

[2] See `http://www.alvestrand.no/objectid/` for a public directory and more information on OIDs

```
attributetype ( CollaborillaLDAPAttrType:201
        NAME 'collaborillaUriOriginal'
        DESC ''
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{4096}
        )

attributetype ( CollaborillaLDAPAttrType:202
        NAME 'collaborillaUriOther'
        DESC ''
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{4096}
        )

attributetype ( CollaborillaLDAPAttrType:203
        NAME 'collaborillaContextRdfInfo'
        DESC ''
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{65536}
        SINGLE-VALUE
        )

attributetype ( CollaborillaLDAPAttrType:204
        NAME 'collaborillaLocation'
        DESC ''
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{4096}
        )

attributetype ( CollaborillaLDAPAttrType:205
        NAME 'collaborillaContainerRdfInfo'
        DESC ''
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{65536}
        SINGLE-VALUE
        )

attributetype ( CollaborillaLDAPAttrType:206
        NAME 'collaborillaObjectType'
        DESC ''
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{32}
        SINGLE-VALUE
        )

attributetype ( CollaborillaLDAPAttrType:207
        NAME 'collaborillaObjectDeleted'
        DESC ''
```

```
        EQUALITY booleanMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.7
        SINGLE-VALUE
        )
```

Custom Object Class

One MUST-field: *cn* (Common Name). It is used as a unique identifier to differentiate between releases and holds the number of the release. The rest of the fields is optional, as this information may be given or not.

```
objectClass ( CollaborillaLDAPObjClass:200
        NAME 'collaborillaObject'
        DESC ''
        SUP top
        STRUCTURAL
        MUST (  cn )
        MAY (   description $
                collaborillaUriOriginal $
                collaborillaUriOther $
                collaborillaContextRdfInfo $
                collaborillaLocation $
                collaborillaContainerRdfInfo $
                collaborillaObjectType $
                collaborillaObjectDeleted )
        )
```

B

Collaborilla

B.1 Interface

The interface *CollaborillaAccessible* provides the functionality to publish information and to handle published information. The interface is listed below.

```
1  public interface CollaborillaAccessible
2  {
3      /**
4       * Connects to the service.
5       */
6      public abstract void connect()
7          throws CollaborillaException;
8
9      /**
10      * Disconnects from the service.
11      */
12     public abstract void disconnect()
13         throws CollaborillaException;
14
15     /**
16      * Checks whether the connection is up.
17      */
18     public abstract boolean isConnected();
19
20     /**
21      * Sets the URI of the LDAP entry and rebuilds the Base DN.
22      *
23      * @param uri URI
24      * @param create Tells the method to create the object of it does
25      *               not exist yet
26      */
27     public abstract void setIdentifier(String uri, boolean create)
28         throws CollaborillaException;
29
30     /**
31      * Returns the number of the current revision.
32      *
33      * @return Current revision number. If we work with an up-to-date
34      * object (the latest revision) the returned value is 0.
```

```
35          */
36         public abstract int getRevisionNumber()
37             throws CollaborillaException;
38
39         /**
40          * Sets the number of the revision. After setting the revision
41          * the Base DN will be rebuilt and all operations will be performed
42          * at the revision with the number of the parameter.
43          *
44          * @param rev Revision number. Should be 0 to return to the most
45          * recent LDAP entry.
46          */
47         public abstract void setRevisionNumber(int rev)
48             throws CollaborillaException;
49
50         /**
51          * Returns the number of revisions in the LDAP directory.
52          *
53          * @return Number of available revisions
54          * @throws LDAPException
55          */
56         public abstract int getRevisionCount()
57             throws CollaborillaException;
58
59         /**
60          * Returns information of the current revision.
61          *
62          * @return Info of the current revision, currently RDF info. Will
63          * be probably changed in future.
64          * @throws LDAPException
65          */
66         public abstract String getRevisionInfo()
67             throws CollaborillaException;
68
69         /**
70          * Returns information of a current revision.
71          *
72          * @param rev
73          * @return Revision info
74          * @throws LDAPException
75          * @see #getRevisionNumber()
76          */
77         public abstract String getRevisionInfo(int rev)
78             throws CollaborillaException;
79
80         /**
81          * Sets the current revision to the most recent entry and copies all
82          * data into a new revision. Performs a setRevision(0).
83          *
84          * @throws LDAPException
85          */
86         public abstract void createRevision()
87             throws CollaborillaException;
88
```

```
89      /**
90       * Restores a revision and makes it the most recent revision.
91       *
92       * The current entry is copied to a revision, all fields removed and
93       * the fields of the to-be-restored revision are copied to the most
94       * recent entry.
95       *
96       * @param rev Revision which should be restored
97       */
98      public abstract void restoreRevision(int rev)
99          throws CollaborillaException;
100
101     /**
102      * Reads all URLs of the entry and returns a String array. If the
103      * Location attribute of this entry does not exist it will try to
104      * construct Locations by querying the entries of the parent URIs.
105      *
106      * @return Collection of URLs
107      * @throws LDAPException
108      */
109     public abstract Collection getAlignedLocation()
110         throws CollaborillaException;
111
112     /**
113      * Reads all URLs of the entry and returns a collection of Strings.
114      *
115      * @return Collection of URLs
116      * @throws LDAPException
117      */
118     public abstract Collection getLocation()
119         throws CollaborillaException;
120
121     /**
122      * Adds a new URL field to the LDAP entry.
123      *
124      * @param url URL
125      * @throws LDAPException
126      */
127     public abstract void addLocation(String url)
128         throws CollaborillaException;
129
130     /**
131      * Modifies an already existing URL in the LDAP entry.
132      *
133      * @param oldUrl URL to be modified
134      * @param newUrl New URL
135      * @throws LDAPException
136      */
137     public abstract void modifyLocation(String oldUrl, String newUrl)
138         throws CollaborillaException;
139
140     /**
141      * Removes a URL from the LDAP entry.
142      *
```

```
143      * @param url URL to be removed
144      * @throws LDAPException
145      */
146     public abstract void removeLocation(String url)
147         throws CollaborillaException;
148 
149     /**
150      * Reads all URIs of the entry and returns a String array.
151      *
152      * @return Array of URIs
153      * @throws LDAPException
154      */
155     public abstract Collection getUriOriginal()
156         throws CollaborillaException;
157 
158     /**
159      * Adds a new URI field to the LDAP entry.
160      *
161      * @param uri URI
162      * @throws LDAPException
163      */
164     public abstract void addUriOriginal(String uri)
165         throws CollaborillaException;
166 
167     /**
168      * Modifies an already existing URI in the LDAP entry.
169      *
170      * @param oldUri URI to be modified
171      * @param newUri New URI
172      * @throws LDAPException
173      */
174     public abstract void modifyUriOriginal(String oldUri, String newUri)
175         throws CollaborillaException;
176 
177     /**
178      * Removes a URI from the LDAP entry.
179      *
180      * @param uri URI to be removed
181      * @throws LDAPException
182      */
183     public abstract void removeUriOriginal(String uri)
184         throws CollaborillaException;
185 
186     /**
187      * Reads all URIs of the entry and returns a String array.
188      *
189      * @return Array of URIs
190      * @throws LDAPException
191      */
192     public abstract Collection getUriOther()
193         throws CollaborillaException;
194 
195     /**
196      * Adds a new URI field to the LDAP entry.
```

```
197      *
198      * @param uri URI
199      * @throws LDAPException
200      */
201     public abstract void addUriOther(String uri)
202         throws CollaborillaException;
203
204     /**
205      * Modifies an already existing URI in the LDAP entry.
206      *
207      * @param oldUri URI to be modified
208      * @param newUri New URI
209      * @throws LDAPException
210      */
211     public abstract void modifyUriOther(String oldUri, String newUri)
212         throws CollaborillaException;
213
214     /**
215      * Removes a URI from the LDAP entry.
216      *
217      * @param uri URI to be removed
218      * @throws LDAPException
219      */
220     public abstract void removeUriOther(String uri)
221         throws CollaborillaException;
222
223     /**
224      * Returns the RDF info field.
225      *
226      * @return RDF info field
227      * @throws LDAPException
228      */
229     public abstract String getContextRdfInfo()
230         throws CollaborillaException;
231
232     /**
233      * Sets the RDF info field.
234      *
235      * @param rdfInfo RDF info
236      * @throws LDAPException
237      */
238     public abstract void setContextRdfInfo(String rdfInfo)
239         throws CollaborillaException;
240
241     /**
242      * Removes an eventually existing RDF info field.
243      *
244      * @throws LDAPException
245      */
246     public abstract void removeContextRdfInfo()
247         throws CollaborillaException;
248
249     /**
250      * Returns the RDF location info field.
```

```
251      *
252      * @return RDF location info field
253      * @throws LDAPException
254      */
255     public abstract String getContainerRdfInfo()
256         throws CollaborillaException;
257
258     /**
259      * Sets the RDF location info field.
260      *
261      * @param rdfLocationInfo RDF location info
262      * @throws LDAPException
263      */
264     public abstract void setContainerRdfInfo(String rdfLocationInfo)
265         throws CollaborillaException;
266
267     /**
268      * Removes an eventually existing RDF location info field.
269      *
270      * @throws LDAPException
271      */
272     public abstract void removeContainerRdfInfo()
273         throws CollaborillaException;
274
275     /**
276      * Returns the description field of the LDAP entry.
277      *
278      * @return Description
279      * @throws LDAPException
280      */
281     public abstract String getDescription()
282         throws CollaborillaException;
283
284     /**
285      * Sets the description field of the LDAP entry.
286      *
287      * @param desc Description
288      * @throws LDAPException
289      */
290     public abstract void setDescription(String desc)
291         throws CollaborillaException;
292
293     /**
294      * Removes the description field of the LDAP entry.
295      *
296      * @throws LDAPException
297      */
298     public abstract void removeDescription()
299         throws CollaborillaException;
300
301     /**
302      * Returns the entry and its attributes in LDIF format. Can be used to
303      * export an existing entry from the LDAP directory.
304      *
```

```
305      * @return LDIF data
306      * @throws LDAPException
307      */
308     public String getLdif()
309         throws CollaborillaException;
310 
311     /**
312      * Returns the date and time of the creation of the LDAP entry.
313      *
314      * @return Timestamp
315      * @throws CollaborillaException
316      */
317     public Date getTimestampCreated()
318         throws CollaborillaException;
319 
320     /**
321      * Returns the date and time of the last modification of the
322      * LDAP entry.
323      *
324      * @return Timestamp
325      * @throws CollaborillaException
326      */
327     public Date getTimestampModified()
328         throws CollaborillaException;
329 
330 }
```

B.2 Protocol

The protocol of Collaborilla is held in clear-text and stateful. This means that the information has to be set in a context before it can be stored or retrieved. This is done during the initialization.

Initialization

After establishing a TCP connection, a Collaborilla session has to be initialized with a command telling the Collaborilla service the URI of the entry in the information directory.

```
URI <uri>
```

If the URI does not exist, the service will answer with an error. Should the URI be created during the first access, the command is extended with the optional parameter `NEW`:

```
URI NEW <uri>
```

The *URI* command can be sent whenever during a client/server session. After execution the service-thread will operate in the newly set URI-context.

Commands

After setting the URI the main commands can be sent to the server. As there are, grouped by the subject:

Revision Handling

```
GET REVISIONCOUNT
GET REVISION
SET REVISION <rev nr>
GET REVISIONINFO <rev nr>
ADD REVISION
RST REVISION <rev nr>#
```

Locations

```
GET ALIGNEDURL
GET URL
ADD URL <url>
MOD URL <old url> <new url>
DEL URL <url>
```

Identifiers

```
GET URIORIG
ADD URIORIG <uri>
MOD URIORIG <old uri> <new uri>
DEL URIORIG <uri>

GET URIOTHER
ADD URIOTHER <uri>
MOD URIOTHER <old uri> <new uri>
DEL URIOTHER <uri>
```

RDF-information

```
GET CONTEXTRDFINFO
SET CONTEXTRDFINFO <rdf data>
DEL CONTEXTRDFINFO

GET CONTAINERRDFINFO
SET CONTAINERRDFINFO <rdf data>
DEL CONTAINERRDFINFO
```

Timestamps

```
GET TIMESTAMPCREATED
GET TIMESTAMPMODIFIED
```

LDAP Data Interchange Format (LDIF)

```
GET LDIF
```

Status Codes

The CollaCollaborillaocol defines also status codes, which are sent within status messages from the service to the client, after the execution of a command.

The generic structure of a status message:

```
<protocol>/<version> <status code> <message>
```

E.g. after the successful execution of a command the server will answer:

```
COLLAB/1.0 200 OK
```

Some of the status codes are borrowed from HTTP. The currently used list of status codes and their meanings follows.

Status Code	Description
200	OK, execution successful
201	Entry created
400	Bad request
401	Not authorized
403	Forbidden
404	Entry not found
408	Client timeout
500	Server error
501	Internal error
503	Service unavailable
600	Client disconnect
601	No such object
602	No such attribute
603	No such value
604	Modified
605	Server timeout
606	Attribute or value exists
999	Unknown error

Fig. B.1. The status codes of the Collaborilla protocol.