University of Zurich
Department of Informatics

A Distributable Data Management Layer for Semantic Web Applications

submitted by:
Lukas Kern
Müllheim, Switzerland

student number:
01-707-017

written at the:
Department of Informatics,
University of Zurich,
Prof. Abraham Bernstein, Ph.D.

supervised by:
Peter Vorburger

submitted on:
April 3, 2006

Abstract

This thesis introduces a generic data management framework capable of dealing with distributed knowledge represented in Semantic Web languages. Persistent data storage, data querying, retrieval, annotation, versioning and security in terms of authentication and authorization are key features and are thoroughly discussed in regard to traditional software principles such as distribution, openness, robustness and scalability. This work emerged from a process support system project named NExT whose architecture called for a novel data management framework approach.

First, the envisioned framework's specific requirements are determined. In the second part, appropriate overall concepts are elaborated and subsequently a complete system architecture is presented. The thesis closes with the presentation of a reference implementation that can be used by the NExT system. The implementation furthermore reveals the architecture's feasibility as a proof-of-concept prototype.

Zusammenfassung

Diese Diplomarbeit stellt ein generisches Framework zur Verwaltung verteilter Information vor, die in Semantic Web Sprachen beschrieben sind. Persistente Datenhaltung, Abfrage, Retrieval, Annotation, Versions-Management und Sicherheit in Bezug auf Authentifikation und Autorisation gehören zu den Hauptfunktionalitäten und werden in Bezug auf traditionelle Grundsätze der Softwareentwicklung wie Verteilung, Offenheit, Robustheit, und Skalierbarkeit diskutiert. Die vorliegende Arbeit entstand aus einem Projekt zur Entwicklung eines Prozess-Unterstützungs-Systems namens NExT, dessen Architektur einen neuen Ansatz für ein Datenverwaltungs-Framework erforderte.

Zu Beginn werden die spezifischen Anforderungen des vorgesehenen Frameworks bestimmt. Im zweiten Teil werden entsprechende allgemeine Konzepte erarbeitet und eine komplette System-Architektur vorgestellt. Die Diplomarbeit schliesst mit der Präsentation einer Referenzimplementierung, welche direkt von NExT verwendet werden kann. Diese zeigt ferner die Realisierbarkeit der Architektur im Sinne eines Proof-of-Concept Prototypen.

Acknowledgements

I would like to thank sincerely all the people who encouraged and supported me throughout the time of writing this thesis. I especially express my gratitude to my closest friends that provided me with useful inputs and critical suggestions.

Special thanks go to my girlfriend Isabelle who tolerated many late night and weekend hours and who always encouraged me in times when my own confidence and optimism seemed to weaken.

I am deeply indebted to my parents who made studying possible in the first place and who supported me in all my endeavors throughout this entire period.

I would also like to thank both my supervisors Peter Vorburger and Michael Dänzer. They have always been available and took time for discussion. Their help guided me through this work and led me to deeper understanding of the compelling requirements of researcher in the NMR domain. It has been a pleasure to work with them.

Last but not least, I thank Professor Abraham Bernstein for having given me the opportunity of writing the diploma thesis in one of my favorite fields of interest and for the precious time he dedicated whenever I needed it.

Contents

1. Introduction

The methodological route from the era of traditional computer science to the era of information technology (IT) and communication has started long ago. Whether or not this route has yet reached its final destination is subject to endless discussions about philosophical beliefs of what the two expressions fundamentally represent. Rather than loosing ourselves in such discussions, we want to recall the motivations that led to the continuous evolutional process.

Computer science simply deals with data and the way pieces of such data do relate between one another. As a fact, there is probably no one computer system that generically speaking goes beyond this simple illustration. Any architecture that you perfunctory may look at eventually describes an application (or an orchestration thereof) which processes data as its overall functionality. Leaving aside all other factors and taking this simple model for granted, we legitimately might ask ourselves how it comes that IT has been an ongoing field of research and most certainly will remain to do so in the future. Obviously, IT hides some complex problem fields that neither research centers nor the industry has yet solved to the ultimate satisfaction. A lot has been achieved as far as data centric processes are concerned but little effort has yet undertaken in order to solve the very fundamental of IT, which lies in the handling of information rather than data.

Indeed, it is the difference between representing either data or information that pressed for the circumscribed methodological change above. For a long time applications were destined to support the handling of data. The requirements usually were met if an application offered some reading and writing functionality. Eventually the users requested some semi-automated actions that needed to be executed on them. Nowadays, the application's requirements have fundamentally changed. They urge for intelligent behavior and want applications to deal with information rather than data. Machines inherently need to understand their actions and must be aware of their implications. Welcome to the world of semantics.

1.1. Goal of the Thesis

The goal of this thesis is to contribute to the combined field of Web Semantics and Web Services. The thesis in particular is part of the ongoing project NExT [1] that deals with the development of an NMR (nuclear magnetic resonance) domain specific process workflow system. The project currently resides in its infancy and its current focus is set towards finding an appropriate detailed architecture. The concrete contribution of this thesis hence consists of a detailed system architecture of some of the major components and a corresponding reference implementation that serves as a prove-of-concept. In the following, we look at how the project initially emerged and find out what its fundamental visions and specific goals are. Having this background, we finally describe the concrete components that the thesis is expected to elaborate.

1.1.1. Some background – Initiation of a novel project

The project is named after the developing application NExT[1] [1] and was initiated in January 2005. The motivation for the project has arisen when two professors from significantly different but somewhat complementary research fields discussed the potential of a joint project. Mr. Konstantin Pervushin is Professor and head of the BioNMR research group at the Institute of Physical Chemistry from the ETH (Swiss Federal Institute of Technology) in Zurich. He and his group specialize on the atomic-resolution and structure-dynamics investigations of biomolecular systems using Nuclear Magnetic Resonance (NMR) technology. Mr. Abraham Bernstein is Professor and head of the Dynamic and Distributed Information Systems group at the Department of Informatics from the University of Zurich, Switzerland. He and his group dedicate the focus on investigating new means for supporting the dynamic activity that human and computational actors have due to our ever-changing environments.

Professor Pervushin concludes that in the everyday life of an NMR experimenter there are two issues that are of substantial concern. First, experimenters require not only a profound knowledge about the research domain but additionally require consolidated knowledge about the principles of the NMR technology. Having to acquire a good understanding in both areas leads to a trade-off when focusing on the primary research goal is considered fundamental.

Second, spectroscopists use a considerable number of different, specific tools but appropriate IT based support is vastly missing. An experiment consists of a huge conglomerate of single subordinate experiments of which pre- and post-conditions together with the emerging results determine the course of the overall experiment. Hence, such experiments reveal complex dynamic process workflows that though are not controlled by any means.

By resolving above issues, Professor Pervushin is convinced to (i) achieve substantial improvements of the overall process of NMR spectroscopy experiments and (ii) to lay out the path for the research area to mount to a new level.

At a preliminary meeting the conclusion was drawn that a novel system is to be developed which specifically solves the concerning issues. The first step taken towards the agreed goal was to conduct a master thesis in order to asses the feasibility of the envisioned system and to decompose the aiming goal into manageable subordinate problem fields. Hence, Michael Dänzer agreed to take on this duty and worked out a very promising feasibility study. In his diploma thesis, he elaborated upon the natural requirements of the system and worked out a component-oriented architecture. The feasibility was proven by an additional prototype that focused on the critical aspects. With the completion of this master thesis and the revealing prove for the feasibility of NExT the path for the envisioned project definitely was paved.

[1] NExT – NMR EXperiment Toolbox

1.1.2. Focusing aspects

NExT has been proven feasible. Furthermore, the overall requirements are defined and a component- or domain-oriented architecture determines the coarse-grained system architecture [1]. My distinct contribution consists of the elaboration of a fine-grained architecture that reveals a solution for

i a generic data access abstraction layer,
ii a reasoning mechanism over distributed resources,
iii a version management system suitable for data defined in OWL/RDF,
iv an extensible annotation concept destined for modeled NMR projects, and
v mechanisms for distributed authentication and authorization.

We of course will discuss each of these problem domains in detail throughout the course of this thesis. The motivation in the next chapter gives a detailed introduction and raises some of the most obvious deficiencies. For now, we leave it by this simple enumeration.

1.2. Target Readers

This thesis primarily addresses two different categories of readers. On the one hand, system architects and programmers learn about concepts and derived implementations that may be used in some rather novel and state-of-the-art problem domains. On the other hand, programmers that in particular develop on the NExT project get to know how some important components do integrate into the overall architecture and how they be best used in concrete situations.

System architects ideally concentrate on the chapters Requirements and Design Concepts. The former chapter gives them the context in which the elaborating system is being placed and expected to do its job. The latter chapter then engages into discussions about plausible solutions and their immanent pros and cons.

Programmers may want to concentrate on the chapter System Architecture and Implementation. However, in order for them to understand as to why components are designed in one but maybe not the other way, they preliminary ought to get themselves familiar with the requirements which are discussed in chapter 4. The content of these chapters is self-explanatory. If a programmer is only interested in the integration of RDF and OWL he/she can read chapters 6.1 through 6.4.

NExT developers are kindly invited to read all chapters in the provided order. Motivation and Vision give them a solid basis for a clear and intuitive understanding of the subsequently discussed concepts and realizations. Especially the chapter about the vision may be of great importance. Being aware of the different futuristic scenarios helps to open one's mind for new approaches and helps to become sensitive for critical solutions and/or implementations. To theses readers, most valuable will be the chapter about the reference implementation. It depicts how the architecture is used in practice but also reveals where it may fail to meet the expected behavior and/or functionality.

2. Motivation

The initial drive for this thesis came from the analysis of NExT's prototype [1] whose main goal is to show the feasibility of the envisioned process workflow system described in the previous chapter. While this prototype is able to describe the various envisioned functionalities and nicely reveals how these can be implemented into a coherent and extensible system architecture, it however could obviously not focus on the details of the many components. The completion of the prototype hence revealed a considerable list of future work concerning the elaboration of different components and missing conceptual frameworks especially in lower levels concerned with communication, data distribution and the like. Representing a continuation of the initial work by Michael Dänzer, this thesis hence focuses on the elaboration of some major system components. As the major part of the initial prototype is considered evolutionary, it is rational to start improving the fundamental components upon which higher-level functionality can later be built. Furthermore, as NExT is to become a widely distributed system, data storage and reasoning reveal central aspects that should be dealt in some generic fashion in order to make sure that other components do not redundantly need to deal with again. Eventually we determined that in respect to the project's ongoing progress, the next steps should concentrate on the elaboration of following concrete aspects:

- i a generic data access abstraction layer,
- ii a reasoning mechanism over distributed resources,
- iii a version management system suitable for data defined in OWL/RDF,
- iv an extensible annotation concept destined for modeled NMR projects, and
- v mechanisms for distributed authentication and authorization.

The thesis looks at each of the mentioned architectural aspect and elaborates a fine-grained architectural design and a corresponding reference implementation. The latter can be integrated into the existing evolutionary prototype and furthermore serves as a prove-of-concept. In the following a motivation for each of the mentioned aspects is presented in given order. At the beginning, the lifecycle of a typical NMR project is presented since this information is fundamental for the further understanding not only as far as motivational aspects are concerned but as well, as far as topics discussed throughout the course of this thesis will be at stake. The chapter eventually is rounded with an outlook to related work.

2.1. The experiment's lifecycle

Table 1 characterizes the four phases of a typical NMR experiment. For each phase, the primary task is summarized by a headline and described in an overview-like fashion. The keywords give hints to the technical and design-oriented point of view, which shall confront us in the discussion about the concrete requirements shortly hereafter.

Phase 1: Acquisition	**Primary Task:** Acquisition of spectroscopic data. **Description:** (1) Collecting spectra from the various repositories that are maintained by the experimenter themselves, the team, the institution or community-like platforms. (2) Determination of the quality of spectra. The expressiveness and usefulness of the spectra from the previous task are assessed. The assessment for "good" or "bad" spectra is mainly practiced on the basis of aesthetic analyses. Nice looking diagrams tend to represent expressive and useful experimental results. On the other hand, unaesthetic diagrams tend to be useless and are to be discarded. **Keywords:** Data retrieval, diagrams rendering, efficient organization of artifacts **Duration:** approx. 1 month
Phase 2: Assignment of Resonances	**Primary Task:** Assignment of nuclear magnetic resonances in the collected spectra from phase 1. **Description:** Spectra are analyzed in terms of their structure. Thereby, most structures can easily be elaborated when referring to similar projects in the past. Two aspects are crucial in order to efficiently perform this phase: First reading and/or comparing documentation of similar completed experiments give hints as how to assign the resonances in a particular case. Second, experience is shared among experiments in the field and helps understand facing specifities or exceptionalities. The process of calculating the concrete structure is done with the help of numerous tool-like applications. Oftentimes such tools are executed in sequence whereby the output from one tool respectively is used as the input for the next tool. The tool running time ranges from few seconds up to several days. The output similarly ranges in size from few kilobytes up to several gigabytes. **Keywords:** repository querying by similarity measures, piping data through input-output generating tools, knowledge and experience exchange, various time and data load factors **Duration:** 3-4 months
Phase 3: Building consensus model	**Primary Task:** Aggregation of the elaborated knowledge from phase 2. **Description:** The various results from the different calculations are aggregated and unified. Naturally, this process detects eventual inconsistencies that emerged in previous phases. Furthermore, the vast majority of conflicting data is impossible to be detected at any earlier stage. Loop backs to phase 2 or even phase 1 are very common in order to correct the mentioned inconsistencies. **Keywords:** data aggregation, inconsistency detection, incremental loop backs **Duration:** approx. 4 months

Phase 4: Publication	**Primary Task:** Writing of a comprehensive documentation. **Description:** This phase resembles a typical documentation phase of any research experiment. Due to the enormous amount of data and information that were acquired over the period of time, the main difficulty lies in the process of regaining the relevant data and elaborated knowledge. Typically, the process is shaped by tedious searching actions in order to gather all the required artifacts. **Keywords:** searching relevant information, high data volume **Duration:** approx. 2 months

Table 1. Phases of a typical NMR project

Each of the above phases represents a discrete task domain and reveals inferable requirements. In the following, each architectural aspect is related to the above presented lifecycle in order to infer the corresponding concrete requirements.

Notice, that the requirements for phase 2 where subject to the elaboration of the data model and the automatic process execution in [1]. Hence, phase 2 will virtually be ignored in this discussion. Interesting readers in particular are referred to chapter 4.1 (Functional requirements) and chapter 5 (Models) in [1].

2.2. Management of distributed data

The lifecycle of a typical NMR experiment reveals the importance of distribution in terms of both data and services. An NMR experimenter may want to share its data in various forms ranging from a small network that is destined for a particular research group up to high scaling and global accessible system environments. Hence, NExT virtually becomes a system in which the distribution of whatsoever component may be required. In the following, we first address the popularity of the notion of so-called data access layers[2] and reason to what extent NExT may be able to make use of them. Second, we will discuss the potential of the OWL-S API[3] [2] from the Mindswap group [3] that is planned to be used as part of the system architecture.

2.2.1. Persistent, distributed data storage

The ability to store objects persistently and in a convenient and flexible way is an important requirement for the majority of applications. Hence, a lot of research and industrial effort has been put into the development of so-called data access layers that address these requirements and hence try to offer maximal convenient data access functionalities. In fact, there nowadays exists a

[2] A data access layer (DAC) is a software component that provides simplified access to data stored in a persistent storage facility

[3] Java API for programmatic access to manage and execute Web Services described in OWL-S [5].

broad variety of different solutions ranging from abstract, theoretical concepts up to ready-to-use implementations.

However, NExT is assumed to store data in the form of OWL [4] and OWL-S [5] data elements. In this respect, the number of existing data access layers vanishes to a small number. Indeed, at the time of writing only two APIs are worth mentioning. Jena [6] is an open source API that mainly serves as a typical object binding facility for RDF [7] and OWL data. Second, the OWL-S API developed by the Mindswap group is the corresponding facility for data described in OWL-S. While both APIs represent promising tools within each of their focused fields they nevertheless do not support various features that we strongly can expect from a full-fledged data access layer.

This thesis elaborates an architecture that tries to solve the broached inadequacy. It deals with many of the problems that have successfully been solved as far as relational databases are concerned but that remain unsolved when data in RDF, OWL and OWL-S respectively are at stake.

2.2.2. The use of the OWL-S API in a distributed fashion

The system architecture of NExT described in [1] foresees a specific type of data access layer that eventually is grounded on the OWL-S API. It namely introduces a so-called KB Access Layer [1] whose main goal is to provide a broad set of functionalities regarding the management of NExT related data. The architecture elaborates on the specific functionalities that this component is expected to provide to its upper layer clients. Furthermore, it talks about the internal mechanism that eventually is able to perform the actual work. However, because the architecture only focuses on the primary needs and corresponding feasibilities, it ignores many inferable aspects that are as important. For instance, the architecture does not adequately elaborate any specific concerns in terms of distribution but in this respect somewhat blindly relies onto appropriate support by the suggested OWL-S API.

In order to build a fine-grained architecture that behaves in the foreseen manner and keeps feasible when transferred into a widely distributed environment, a profound discussion on the concrete requirements is unavoidable. One goal of this thesis is the elaboration of such an architecture. In the next chapter, we will have a look at the distinct requirements that we are facing in generic distributed environments. Among other topics, we will discuss the different types of transparencies and their relevance in respect to NExT. With this in mind we then will be able to evaluate to what extent the OWL-S API can be directly used by the NExT core and to what extent it may be appropriate to shield it by some sort of an additional component or hierarchical layer.

2.3. The Semantic Web and reasoning over distributed resources

The internet, in some sense, can be regarded as a huge database. The major difference compared to traditional database systems is that the managed data does not conform to a well-defined model that describes the structure and required semantic. Facilities to put machine-understandable data onto the Web are therefore becoming a high priority for many communities [8]. The Web can reach its full potential if it becomes a place where data can be shared and processed by automated tools as well as by people. Tomorrow's programs must be able to share and process data even when these programs have been designed and built very independently. Here is where the Semantic Web comes into play.

2.3.1. Semantic Web and reasoning

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." [9]

The initiators of the Semantic Web (SW) [10] see the emerging technology as an enabler for computers to better work in cooperation with people. As a fact, they are not saying that the technology as such is going to bring any concrete or direct improvements. Being an enabler, the SW makes new business and IT concepts possible but not necessarily a-priori feasible [8]. Let us make an example in order to get this straight:

We hypothesize that the goal (or vision) of the SW has been achieved. All data out in the web is accessible as well as machine-readable. Let us assume that we want to arrange some vacation. There are many travel agencies that can do this job for us. Since the SW has all the data that we need ready and machine-readable, we though want to do the arrangements ourselves. So lets then start and check for flights, airport transfers, hotels, rental cars and so on. Our goal is to get the most preferred arrangement to the lowest possible price. We can search for an online portal that allows us to query the web for the requested services. If such did not exist, we as well just could program such a program ourselves. It turns out that a satisfying portal indeed does not exist. Furthermore, it also turns out that just programming our own tool was a bit of a too enthusiastic idea. The problem namely lies in the way of how we go about and find all relevant data. Just because it is all out there, does not yet mean that we can use it in an efficient way. In this scenario, we finally may as well decide against the primary plan and end up consulting a travel agency.

The example brings us right to the point. The SW obviously cannot function without some kind of sophisticated reasoning support. First, as the internet will serve as one big database with a volume that exceeds the load of traditional databases by multiple orders of magnitude, the effort that is required to locate any piece of data increases dramatically. Simple searching algorithms in this context are no longer feasible. Instead, intelligent and sophisticated query support is vital when managing data. Second, the fact that data is accessible does not necessarily mean that this data is also feasibly available. In the example, we theoretically have all data at hand. However, since it is scattered all over the network and not hosted at one well-defined location (as this is the case with traditional databases), we face the problem of locating the relevant data providers. As long as we do not know what sort of data is hosted at a provider, we need to contact all of them in order for a query to return a complete result. However, this clearly is not practicable. When we talk about design concepts for the envisioned system architecture in chapter 5, we will eventually find out how we can solve these two broached deficiencies.

2.3.2. NExT in the context of the Semantic Web

So, why do we talk about the Semantic Web when the question is about reasoning in the context of NExT? The answer is simple. NExT uses OWL and OWL-S. Hence, NExT's distributed system environment is going to resemble the Semantic Web. When we talk about design concepts (chapter 5) and the fine-grained system architecture (chapter 6) we sometime may come back to this early discussion. Recalling what problems the semantic web is exposed to and how it may deal with may help us to draw our architecture.

2.4. Versioning

Versioning is a widely and everyday used mechanism to organize arbitrary artifacts in terms of their inherent evolution process. Hence, almost every artifact is subject to some sort of a versioning mechanism when it comes to their concrete handling and/or development. While in some situations version indications are obvious, yet in other situations such tend to be hidden but can be revealed if the concrete characteristics and or development process is closely elaborated. While the former situation seems obvious, the latter may need some additional explanation. Let us therefore have a simple example.

> Hidden versioning for example takes place if you need to write a job application, and for the sake of convenience or simplicity, you just grab one of your older CVs that you accidentally still can find on the desktop of your personal machine. As most content of the CVs still is accurate, you may only need to do some minor adjustments in order to reflect the latest circumstances. After having done the adjustments, you save the document by either overwriting the older CV or by creating a new document with a similar name. In either case, the saving process is subject of some hidden, unconscious versioning aspects. If you happened to overwrite the previous CV, you just did some sort of a commit meaning you have committed your latest modifications to the original document. On the other hand, if you created a new document with hopefully a meaningful new name, you automatically created some sort of a new version.

In the context of NExT, there are two main aspects that reveal the necessity of a flexible and scalable versioning mechanism. They both are separately discussed hereafter.

2.4.1. NExT - a multi-user system

NExT by its nature is a multi-user system. Hence, different users may simultaneously work on the same data that is retrieve from arbitrary data repositories. While simultaneous reading processes are generically not of a problem, writing processes ought to be subject to scrutiny. If an experimenter A temporarily retrieves data from a remotely located repository, changes it, and subsequently commits the resulting modifications, there may be another experimenter B who – in the meantime – may have retrieved the same original data. If experimenter B commits its modifications slightly after A, it will blindly overwrite A' previously made modifications. The stereotypical scenario reveals that NExT obviously faces the same problems than do all common database systems as far as guaranteeing data integrity is concerned. Versioning in this regard may represent an adequate and promising alternative to the implementation of a transaction mechanism [11], which is usually done for database systems but which is still subject to ongoing research as far as its application is destined for distributed environments.

2.4.2. Process workflow definitions similar to programming code

Defining an NMR experiment in regard to its corresponding workflow process definition can be compared to the process of writing application program code (source code). In both situations, the actor is concerned with composing heavily interrelated components. In the case of source code, the programmer defines classes that eventually will depend upon one another. Hence, in the case of the definition of an NMR project process workflow as described in [1], the NMR experimenter defines elements such as NMRExperiments, NMRExperimentalSteps or NMRProcesses that inherently tend to be related to one another. Eventually, a complete NMR experiment process workflow evolves to complex multi-layered element structure. An overview of the NExT process model is given in appendix A.1.

While in the field of software-engineering so-called CVS (Concurrent Versions System) [12] tools are used to handle the evolving complexity as modification procedures and simultaneous working are at stake, the NExT system yet misses any such functionality. However, as NExT claims to represent a powerful and full-fledged application system to be used by NMR experimenter by their everyday work, some sort of CVS-like functionality will certainly turn out to become crucial when user acceptance is at stake. Note furthermore that because NMR experiments are described in OWL and probably will be stored in specific data repositories (such as triple stores) the integration of an existing well-proven CVS tool may not be feasible or at least not possible without severe architectural adaptations.

2.5. Annotation and OWL

Generically, annotation is defined as extra information, not being part of the corresponding domain, and associated with some kind of data. In practice, annotating information depends on the domain wherein it is used. For example, annotations for written documents, such as articles or books, typical reveal knowledge about the authors, the publication date, the subject category, and so on. Standards aim at providing interoperable and well-defined solutions. The Dublin Core Metadata Initiative [13] thereby is probably the most widely used framework.

OWL can be regarded as an extension to RDF (Resource Description Framework) that – as its name reveals – aims at providing a framework for describing resources [14]. The difference between RDF and OWL mainly lies in the expressiveness of the language. While RDF focuses on a formalism that allows to generically handling metadata, OWL goes one step further and focuses on a framework, which allows the handling of metadata with the addition of semantics.

2.5.1. Metadata vs. Annotation

We talked about annotation on the one hand and metadata on the other hand. Now, what actually is the difference between these to terms? Both of them seem to deal with "data about data". There indeed is only a small, but not unimportant difference between the two expressions. Metadata is the more generic term. It represents information about a thing – apart from that thing itself. With other words, it defines the characteristics of a thing by using concrete property values. Annotation, on the other hand, is metadata as well but with an additional restriction. In order for metadata to become annotation related information, the defining properties must not be part of the modeled domain.

Let us have an example to get that straight: We assume that we need to build an online reservation platform for a car-rental company. The online service thereby is required to give detailed information about the available cars. As a result, we need a data model with some sort of a car representation. For the sake of simplicity, we model the car with three properties; namely: model, color and number of seats. Since we also want to keep track of data modifications, we add an additional property last-modified. While all four properties define metadata, only the latter property also defines annotation information. The property last modified does not have a meaning within the modeled domain. It solely gives information about the modeled car, thus the data record that is to represent a car in the real world.

2.5.2. Mindswap's annotation support

Mindswap's OWL-S API supports annotation out-of-the-box. The concept is simple and promising. Because OWL supports the description of generic metadata as a key functionality, OWL is as good when it come to the description of generic annotation. The API provides its base model entity with the functionality that allows adding and removing arbitrary annotations. In the underlying OWL/RDF model, annotation data is described by regular properties. The concept offers a very flexible and powerful solution. First, it can be used in combination with any annotation framework. Second, the fact that annotation information is described by regular OWL/RDF properties, the same functional richness as is provided for the latter is applicable. Third, given the previous two points, the solution seems to incorporate maximal extensibility.

However, this nice flexibility does not come without its cost. The concept namely leaves the burden of handling concrete annotation standards up to the client. As a fact, we face the question how we must design the data model in the business logic layer in order to keep maximal flexibility but still provide for a concrete solution. As for now, we leave it by this short introduction. We will come back to this question when we talk about design concepts in chapter

5. Design patterns that allow us to bridge the gap between the business logic layer and the interface given by the OWL-S API in a lower layer will be discussed in chapter 6.

2.6. Authentication and authorization in a distributed environment

Applications can be seen as tools that provide services to people and/or other applications. In this sense, applications and people act as users, which abstractly can be called entities or principals whereas the latter is much more appropriate for this field. In order for the different principals to consume reliable, convenient and foremost individualized services, it is common for these principals to be labeled in a way that allows them to be uniquely identified. All modern systems indeed have the notion of some sort of labeled entities. Hence, all of them do face the problem of how to authenticate an association between a particular label and a particular applicant (thus principal). Note at this point, that an application or a user from the real world may not necessarily be represented by merely a single principal but rather by an arbitrary number thereof. Such in particular is the case, when an application or user acts by multiple roles.

Authorization refers to granting or denying access to specific resources based on the principal's identity [15]. Clearly, authentication and/or authorization are direct complements. Neither of them would separately pursue a real purpose.

Let us have an example that puts this rather abstract notion into some practical context. As a student at the University of Zurich, I own a personal mailbox which is hosted at some server at the campus. In order to check my mail I usually make use of the available webmail application. At the login page, I provide my student number and my personal password and then click "Login" in order to get authenticated and to finally enter my mailbox. The student number (or student ID) represents the unique label, which tells the University who I am. Because the S-number is not protected and thus could theoretically be provided by any one student or even any one person I need to prove my identity. In other words, I have to prove that it is actually me, the legal representative of the given S-number, which requests a login. Here, in this example, a password is used to give that additional prove. Authentication runs behind the scene. Upon successful authentication, webmail uses the principal, whose name is the provided S-number, in order to infer my individual permissions.

2.6.1. Why decentralized authentication is required

Authentication and Authorization (AA) traditionally is used within self-contained environments in which user management is centrally hosted [16]. Teams in companies or institutions, for example, run applications that tend to be accessible within their private network. AA in this case is commonly covered by some kind of central user management. Usually there is a single distinct service that is responsible for verifying credentials and providing user permissions. When we broaden the scope and look at institution-wide networks, we discover the same analogy. Apart from eventual sophisticated mechanisms in order to boost scalability and performance, the concepts behind the AA facilities remain the same. In fact, a lot of times only one single facility handles the verification of user credentials and their permissions. Hence, we deal with a very strong dependency on centralized services [16].

NExT represents a distributed system. Both, client and service components are equally distributed over the network. While clients do not interact with one another, they though must rely on services in order to be able to fulfill their work. Services on the other hand may rely on one another. It is typical that certain services concentrate on basic, atomic-like operations and that yet other services will act as coordinators and use these former ones in order to carry out some higher-level operations. Let us abstract from clients and different services and collectively treat them as any interdependent component. The fact that such components may reside in many different administrative domains implies that they most probably do not belong to the same central security realm. A component authenticates and authorizes itself at its local authentication and authorization service (AAS). Being authenticated this component can contact whatsoever counterpart within the same security realm. Nevertheless, it cannot contact components in any other realm. As a matter of fact, it is obvious, that components have to get the ability to authenticate themselves at remote security realms by some way or another.

At the time of writing, two different authentication and authorization technologies mainly exist that can be used in distributed systems [15]. One solution makes use of digital certificates [15] that are based on private and public keys. The other solution is Kerberos [16]. Note at this point that there are many other concepts that pretend to provide solutions for distributed environments. There though is a high chance that they base on either of the two concepts and thus do not provide any novel technique.

2.6.2. A solution for NExT

From the previous discussion, we have learned that both, digital certificates and Kerberos provide authentication services for distributed systems. Nevertheless, both solutions do not come without compromises. Certificates rely on the existence of a public key infrastructure (PKI) [15] and trusted certificate authorities (CA) [15] whose installation in the first place and the required legitimation is burdensome. Furthermore, different companies and institutions that already rely on such an infrastructure however may not necessarily trust the same CA, which yet reveals another problem, namely the creation of a well-defined trust hierarchy. Kerberos, the second introduced solution, inherently forces all participating parties to use Kerberos for their principal authentication and authorization mechanism. While a lot of companies and institutions indeed may already use Kerberos due to its popularity and its adequate feature support, NExT nevertheless should not ignore companies and institutions that do not do so. Rather than relying on one distinct authentication and authorization technology, NExT should make it possible to use different technologies at different locations within its emerging distributed environment simultaneously. While following this idea seems promising, it however raises yet another aspect. The system in this regard namely needs to find out how it can integrate such fundamental different technologies (which hence rely on different underlying concepts) without the need for significant adaptations. A transparent and plugin-like integration procedure would obviously be highly welcomed. These and plausibly some many more compelling concerns will be discussed and hopefully solved throughout the course of this thesis. For now, we leave with this short and intentionally uncompleted introduction.

2.7. Related Work

In this section, related topics concerning the elaboration of above illustrated problem domains are discussed. Due to the rather diverse set of focusing aspects, we however intentionally do not cover all of them but focus on the more important. Each topic is briefly introduced in terms of principles or underlying concepts and secondly rounded out with a discussion about the concrete relation to this thesis.

2.7.1. JXTA

JXTA [17] is a project that aims at providing a generic communication technology for peer-to-peer systems [18]. The project was initiated by Sun Microsystems in April 2001 when the importance and wide spread of Peer-to-Peer systems steadily increased. Napster and Gnutella, among many others, are concrete systems that evolved during the mentioned time. As an open-source project, the development community has steadily grown. At the time of writing more than 16.000 members have registered and claim to contribute actively to the technology's evolution.

JXTA functions as a virtual *overlay network* [19]. It represents a generic communication framework that does not require any type of specific technology. The overall architecture defines a set of protocols that intends to enable peers to communicate successfully with one another. Interoperability thereby is of central importance and attains the major focus. The architecture claims to enable interconnected peers to be able to (i) easily locate and communicate with each other, (ii) to participate in community-based activities, and (iii) to offer services to one another seamlessly across different platforms and networks [19].

The JXTA architecture

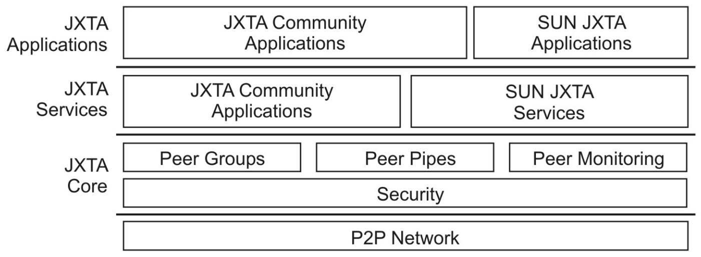


Figure 1. JXTA architecture [19]

The coarse-grained architecture (figure 1) of JXTA can be divided into three hierarchical layers. The basis is defined by the so-called JXTA Core. The layer consists of basic services that deal with (i) the connection of endpoints bound to participating peers, (ii) the management of pipes that are used in order to send messages between communicating peers, (iii) the query functionality which is used to find existing peers and to infer about their providing functionalities as well as their concrete offering interfaces, and (iv) the proliferation of peer's state information in an observer-based fashion. In addition to the described functionalities, it also defines the notion of peer groups that enable the system to subdivide its participating peers by various aspects such as location, closely related functionalities, security boundaries, etc.

The second layer is represented by JXTA services that build upon the illustrated base functionalities. Typical components in this layer are sophisticated search and query services as well as higher-level communication services destined for generic file and data stream transfer. While some services are developed by the community, others are provided by Sun Microsystems and understood as part of the reference implementation.

Finally, the third layer is represented by concrete utility-like applications that directly interact with the subordinate JXTA services in order to provide plug-in-like functionalities to arbitrary systems that depend on peer-to-peer communication. This top layer in this sense functions as a convenient access point for the application's specific components in charge of the communication functionality.

Communication protocols

In order to get an idea of the system in terms of its functioning and in the way peers eventually are enabled to communicate with one another, we briefly look at the defined protocols. Each of them namely reveals a distinct aspect of the architecture. In the following, the six protocols are listed with a short explanation.

- **Peer Discovery Protocol.** JXTA offers its client (an arbitrary system application) a transparent discovery service. Peers, advertised services and pipes can be searched on the basis of a generic request-response protocol.
- **Peer Resolver Protocol** is used in order to translate discovery queries of a particular format into a generic format that is then globally understood so that it can be sent out into the network of peers.
- **Rendezvous Protocol.** JXTA uses an advertisement concept in order to manage communication among peers. The rendezvous protocol in this sense is used to create and maintain the connections between peers which not necessarily are aware of each other's existence and location in advance.
- **Peer Information Protocol** is used by a peer to query other peers about their individual state. Since the format of a concrete query and possible response are not defined, interoperability is achieved in combination with the Peer Resolver Protocol that is able to translate specific formats into appropriate standard formats.
- **Pipe Binding Protocol.** A peer sends and receives messages over advertised pipes. This protocol thus allows a peer to create a pipe and subsequently bind it onto one of its endpoints. Additionally it allows other peers to bind themselves onto existing pipes in order to be able to receive/send messages from/to the pipe's claimed initiator.
- **Endpoint Routing Protocol.** In some situations, a peer is protected behind a firewall and hence cannot directly be contacted by other peers. This protocol allows a so-called End-Point Router to act as an intermediate which is able to send message from and to peers behind a firewall.

Relation to the thesis

As NExT aims to become a widely distributed system with services residing on machines within different administrative domains as well as various system environments, communication becomes a crucial aspect. JXTA in this regard could eventually be used for the underlying technology when communication between remote services is concerned. When we talk about the design concepts (chapter 5) for the elaborating system architecture (chapter 6) we will realize how close some of the JXTA's offered functionalities indeed relate to certain presented lower-layer aspects. While JXTA deals with aspects regarding communication technologies for specific distributed systems, the thesis rather abstracts from such particulars and deals with aspects such as appropriate distribution topologies, or coordination patterns among data storage services.

2.7.2. Edutella

Edutella [20] is a specific JXTA project that plans to extend the JXTA framework with the W3C metadata standard, RDF. Currently the project is hosted at the JXTA platform and is actively maintained by the CID (Center of User-Oriented IT design) at the Royal Institute of Technology, Stockholm, Sweden, in cooperation with the Uppsala Learning Lab at the University of Uppsala, Sweden and the Stanford Infolab, California, USA.

The basics of the Edutella framework

Edutella builds upon the JXTA architecture and defines so-called Edutella services and Edutella peers [21]. The former represent web services that are defined by either WDSL or OWL-S. The latter live on the JXTA Application Layer and use the functionally provided by the former, the Edutalla services. The project plans to elaborate five distinct services, whereas the focus primarily is set onto the first of them. The following table depicts each service with a short description.

Query Service	Providing a standardized query exchange mechanism for RDF metadata stored in distributed RDF repositories.
Replication Service	Providing data persistence, availability and workload balancing while maintaining data integrity and consistency.
Mapping Service	Translating between different metadata vocabularies to enable interoperability between different peers.
Mediation Service	Defining views that join data from different sources and reconcile conflicting information that emerge from the corresponding data aggregation.
Annotation Service	Enabling annotation in a distributed fashion such that it must not necessarily be stored together with their targeting artifacts.

Table 2. Edutella services

The framework uses the notion of RDF data repositories when describing Edutella peers. A repository consists of RDF statements and describes metadata according to referenced RDF schemas. Because the Query Service interface is mandatory, every repository supports a so-called *local* RDF storage layer query language [21]. For example, if an arbitrary repository uses a relational database in order to store the individual RDF statements, the local storage layer query language probably would be SQL. On the other hand, if that repository uses JENA as its persistence framework, the local storage layer query language almost probably would be RDQL.

Mediation services are able to provide a coherent and transparent view onto an arbitrary number of RDF repositories. They in this respect act as transparent proxies onto a set of subordinate repositories. Because every repository may support an individual query language that is not necessarily compatible with others, mediation services can only act as proxies if a common query and query result representation along with corresponding transformation processes are available. As a result, the framework defines a Query Exchange Language (QEL) [21] and a so-called Edutella Common Data Model (ECDM) [21] in order to solve the mentioned deficiency. Both of them are defined in RDF and collectively provide the syntax and the necessary semantics for a standard query interface. The transformation from a local query language into the QEL and vice versa is performed by an Edutella wrapper sitting at the corresponding repository. The ECDM allows transferring a result set from a repository's local data model into a common data model that is understood by all peers in the network. The remaining interfaces are self-explanatory and do not need any further explanation. Moreover, they are considered rather traditional services that do not incorporate any novel concept or otherwise worth-mentioning aspects. The main reason we presented Edutella is their promising concept regarding the QEL and EQDM.

Relation to the thesis

The concept that is used to distribute RDF statements within a distributed system is very similar to the underlying principle that we apply to the elaborating system architecture. While Edutella primarily focuses onto storage of metadata, our system architecture in contradiction uses RDF to store both primary data and corresponding metadata collectively. The problem of how to query distributed data repositories in a transparent and interoperable way however remains the same for both projects. When we present the design concepts for the management of distributed data in chapter 5.1, we will realize that some of the underlying principles are identical, and that in these situations only the context in which they are applied will differ.

2.7.3. OWL-S Matchmaker

The OWL-S Matchmaker project [22] elaborates a solution to the problem of finding web services by the definition of requesting service capabilities in addition to traditional descriptive keyword based filtering methods. The project is lead by the Intelligent Software Agents group [22] at Carnegie Mellon University (CMU), Pittsburgh, Pennsylvania, USA. At the time of writing, the project is at an advanced stage where concrete implementations are integrated into particular system environments and represent the necessary prove of concepts and corresponding feasibility.

Matchmaker agent system

The project's concept is based on the notion of an agent system. An agent thereby can represent (i) a service providing entity, (ii) a service requesting entity, or (iii) even both if its duty is to act as a service composer relying onto other agent's providing services in order to offer more complex and/or higher level services [23]. While regular agents provide and/or consume arbitrary services that can be characterized by their inputs, outputs, preconditions and resulting effects (IOPE) [24], a so-called specific agent called Matchmaker acts as a "yellow pages" of agents with their corresponding capabilities described in terms of services and their corresponding IOPEs. The Matchmaker thus allows agents to find each other by providing a mechanism of registering each agent's capabilities.

The semantic description of services (IOPE) is achieved with the OWL-S service description language. However because this language only defines the way by which semantic information is described but does not provide any direct functionality in order to reason about such data, the project elaborates a specific matching algorithm as well as a corresponding matching engine. The matching algorithm is designed such that it offers following characteristics. It (i) is configurable in terms of the minimal match acceptance degree, (ii) it does not follow a hard true and false matching mechanism, (iii) it allows for automatic dynamic discovery, and (iv) is able to perform an intelligent selection upon the set of found services. The Matching Engine implements the mentioned algorithm and hence can be regarded as a specific OWL-S reasoner that performs sophisticated similarity measurements upon the service's IOPE specifications.

OWL-S/UDDI Matchmaker

The Matchmaker agent system as briefly introduced above is capable of dealing with web services defined by the relatively novel OWL-S service description language. However, because the majority of web services are defined by the de-facto Web Services standard [25] using SOAP [26] and WSDL [26], the project currently does not offer any substantial benefit to businesses running WSDL web services. Developing an OWL-S web service hence is currently not rational because there would be no other service that eventually would make use of it due to compatibility reasons. As a result, the project team decided to enhance the UDDI (Universal Description, Discovery and Integration) [27] service, which acts as the corresponding "yellow pages" for WSDL web services, in order to become compatible with both WSDL and OWL-S.

Figure 2 depicts the overall architecture of the resulting OWL-S/UDDI Matchmaker. The underlying principle constitutes of the fact that the UDDI service is kept unchanged and that the OWL-S/UDDI Matchmaker solely acts as a wrapper to the UDDI service. When the CommunicationModule (CM) receives an OWL-S formatted advertisement, it sends it to the Translator, which constructs a regular UDDI service description and registers with the UDDI. In the second step, the CM creates an advertisement for capability matching and sends it to the OWL-S Matching Engine (ME) that is represented by the previously described Matchmaker agent. Requests follow the opposite direction. The CM sends an OWL-S formatted request to the ME that subsequently performs the capability matching. The result of the matching is a set of capability advertisements with a reference to their corresponding UDDI service description records. After retrieving the latter, the answer set is returned to the requesting client. Services that rely onto WSDL and SOAP do not need to interact with the OWL-S/UDDI Matchmaker but can directly connect to the UDDI service.

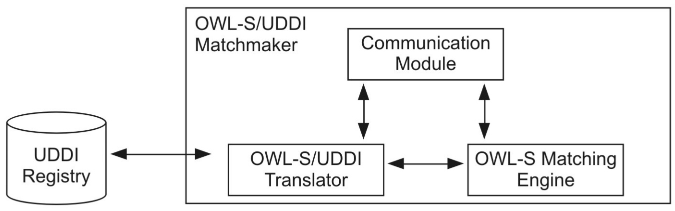


Figure 2. OWL-S/UDDI Matchmaker architecture [23]

Relation to the thesis

NMR experiments are described by OWL-S services. The NRM process model (see appendix A.1) defines all major data entities as direct or indirect subclasses of the OWL-S service class. Furthermore, the system architecture does not focus on a specific reasoning mechanism but supports for the integration of arbitrary third party reasoning facilities. As a result, it may be possible to make use of the Matching Engine in order to provide the functionality for semantic capability reasoning. However, the two systems yet ground on different communication methodologies. While the Matchmaker service relies onto an agent system, the system architecture builds upon traditional client-server interaction. An eventual integration of the Matching Engine may not be as straightforward as it may seem at first glance.

2.7.4. Web of Trust and PGP-compatible protocols

In a distributed system, communication is one of the major concerns. A node in such a system usually does not rely on its own but engages into various communications with its counterparts. For example, a node may act as a typical server and thereby may offer functionalities to nodes acting as clients. Another slightly different communication pattern is represented by a peer-to-peer system. In this case, two arbitrary nodes engage into a bidirectional conversation. While at one point in time node A acts in the role of the client and requests services from a node B, at yet another point time, the roles may be shifted, and hence node B starts acting as a client and vice versa requests services from node A. Depending on the type of conversation, roles may be shifted on a frequent basis resulting in a complete bidirectional communication.

The problem with communication in distributed systems is the mechanism by which nodes can make sure they indeed do communicate with the actor they believe they communicate with. While there are many different authentication techniques available, none of them is yet able to address the introduced problem in distributed systems satisfactorily. Scalability and trustworthiness in insecure and open networks are usually the common pitfalls.

Web of Trust [28] describes a concept that accounts for the establishment of authenticity concerning identification of actors in untrusted environments. Common protocols, which exhibit the notion of a Web of Trust, are PGP (Pretty Good Privacy) [28] and it's various related PGP-compatible co-implementations.

The basic concept

PGP's principle is based on common public key cryptography [15]. However, in contradiction to traditional public key infrastructures (PKI) [16], PGP breaks the hierarchical trust architecture and with this becomes able to follow the notion of a Web of Trust. While in traditional PKIs all participating actors need to trust well-defined and inferable Certificate Authorities (CAs) that are given the exclusive power of issuing certificates, PGP relies on a peer-to-peer based approach and authorizes the actors themselves to sign each other's certificates. In this sense, PGP public key certificates attain their authenticity by receiving as many as possible attests from co-actors such as friends or other related persons that believe to be able to authenticate a particular actor's claimed identity. The more attests a public key certificate obtains, the more likely it is to be trusted by arbitrary co-actors. Let us make an example that illustrates the power of this trivial idea.

> Alice signs Bob's public-key certificate that she knows is authentic. Bob then forwards his signed certificate to Carol who wishes to communicate with Bob privately. Carol, who knows and trusts Alice as an introducer, finds out that Alice is among Bob's certificate signer. As a result, Carol can be confident that Bob's public key is authentic. However, if Carol does not know nor trust any of Bob's signers, including Alice, she would be skeptical about the authenticity of Bob's public key. In this case, Bob needs to find another introducer whom Carol trusts to sign Bob's certificate. (Modified example from [28])

The scenario nicely reveals how trust can proliferate within an emerging network of interrelated actors without the need of a central authority. A Web of Trust hence follows the idea of inherently incorporating trust into the system instead of defining endless trust delegations that generically fail in attesting the final instance and whose trust system is only as trustworthy as are the controlling instances.

Web of Trust RDF ontology

In the recent, the Semantic Web Interest Group [30] has developed an initial draft vocabulary in order to account for public key cryptography concerning RDF represented data. At the time of writing the initial version is published under the XMLNS[4] domain and is given the name WOT (Web of Trust RDF Ontology) [31]. Nevertheless, there is no evidence as to whether or not it has already been used in some corresponding projects. Hence, status of usability as well as completeness in terms of defined classes and properties yet remain uncertain.

Relation to the thesis

Public key cryptography will be of interest when we look at the requirements in terms of authentication. As the envisioned system deals within a widely distributed environment, some sort of distributed authentication will be necessary. Due to the fact that NExT needs to be build in

[4] xmlns.com is an internet domain created for the purpose of simple Web namespace management.

a maximal open and flexible fashion the system architecture will almost certainly need to cope with multiple technologies. Making use of a Web of Trust may be one feasible solution.

The recent publication of the WOT vocabulary reveals the feasibility for cryptography in terms of OWL/RDF defined data. Once the vocabulary's usability will have been proven NExT may discuss its use in regard to the various security aspects. Note that NExT may not only be interested in its authentication capability but as well may discuss possible usage scenarios in terms of data integrity and authenticity.

3. Vision

The history of IT to date has marked an important and continuing trend. We are talking about ubiquity in terms of computing. Ubiquity reveals that mass cost reductions has made it possible to introduce processing power into devices that – only a couple of years ago – would have been uneconomic or even unimaginable. The trend shapes the world of IT significantly. The internet is just one of many examples that can prove this perception. We recall that the internet – as we know it today – did not exist until the mid 1990's. However, today, we cannot think of a world without it. As the trend towards ubiquity in terms of computing continues, the required systems and applications become ever more interconnected. On the one hand, we would like to have the relevant information readily available at all times. Yet on the other hand, we though do not want to care about the management of that information in the first place. In terms of software architecture, this statement reveals that we ought to look out for a novel and powerful abstraction concept that makes it possible to handle information as it were a commodity.

In the following, we present a visionary system that tries to address above broached aspects as far as fully transparent management of information and offering of mass (customized) services (applications) are concerned.

3.1. System architecture

Our visionary system represents itself by a twofold API. On the one hand, it serves as a transparent access layer when terminal-like applications need to access lower level service components. On the other hand, it serves as a coordinator when the latter need to communicate with one another. The system in this sense is said to provide a downstream API for the business logic tier and an upstream API for the mentioned lower-level components.

The overall architecture is simple and is depicted in figure 3. Applications with a user interface are called terminals and are expected to act as typical thin clients. Note that the expression terminal is intentionally borrowed from the old-day mainframe systems. As we soon will see, they indeed have many similarities to the terminals of these days. UAP stands for Unified/Universal Access Provider and represents what we previously called the twofold API. Its purpose is similar to a generic object broker [26]. While the upstream interface is used by the terminals, the downstream interface is used by the so-called self-contained components (SCC). Self-contained components represent services on different abstraction levels and typically interact with one another. While some SCCs concern about fundamental tasks, others rely onto former ones in order to concern about higher-level, thus more complex, services. The basis finally consists of the notion of a global database, here named Universal Database (UDB). Our vision intends to abstract from any concrete concept. The chosen names for the different components as a result try to be as neutral as possible.

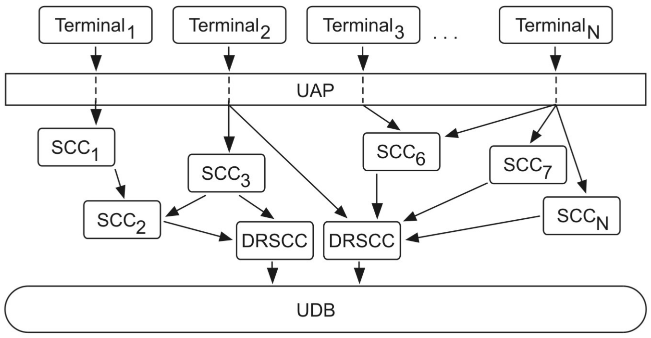


Figure 3. Overall architecture

3.2. Terminals and SCCs

Terminals range from very simple to very complex. While one terminal may represent a traditional read-only directory for people's contact information, another terminal in contradiction may represent a complete enterprise resource programming (ERP) system. While for the former terminals it may be feasible to rely onto one single suitable SCC, the latter terminals almost certainly will require many of them in order to be able to perform their broad set of functionalities. The concept's core idea is that there are as many particular SCCs as will be needed, and that the development of particular terminals abstractly speaking is solely a matter of picking and coordinating the right SCCs.

Obviously, this idea is only rational if building SCCs is less expensive than directly putting their functionalities into the terminals in the first place. The concept assumes that the majority of SCCs is simultaneously used by a high number of different terminals. Hence, a SCC is thus able to incorporate significant economies of scale. An individual SCC provides one or multiple services within a complete, well-defined and self-contained context. If for the delivery of a particular service an SCC is able to make use of other already existing services and thereby can reduce the complexity and/or required computing power, it is highly encouraged to do so. The concept namely envisions that there are no two SCCs that will ever provide two identical services. Note, that for the sake of simplicity, we intentionally abstract from the notion of eventual replication aspects for purposes such as security or performance. If this conceptual idea is strictly applied, the development of SCCs becomes a fairly simple job. The majority of SCCs namely will be able to provide their services by relying on suitable lower-level services and by dealing with the corresponding resulting coordination. Only the very fundamental SCCs eventually will need to deal with some sort of direct information processing.

From the software engineering point of view, the concept does not represent any novel aspect. After all, it does not do anything more than strictly adhering to the common two design principles of information hiding [32] and separation of concerns [32]. However, there yet is probably no single system that follows the two mentioned principles in the envisioned fashion from the lowest up to the highest architectural tier and furthermore does so in a global scope that eventually claims to unite any application out in the world. The following figure depicts the UAP with its underlying SCCs that communicate with one another by providing particular relation dependent interfaces.

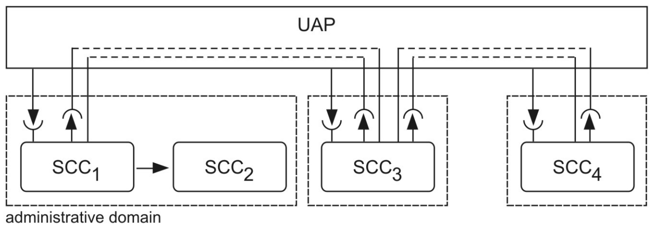


Figure 4. Architecture of the Universal Access Provider

A SCC registers with the UAP in order to announce its existence. It thereby provides a specific advertisement interface that allows the UAP to reason upon its provided services. Note that although the UAP is depicted by one single component, it as well may be distributed and in this sense may rather be seen as a conglomerate of an arbitrary number of identical, benevolent, and interconnected components. When a particular SCC wants to make use of another SCC's provided services, it addresses the UAP and asks it to return a list of appropriate SCCs given concrete functionality criteria. If the returned list is not empty, the particular SCC picks one of the contained interfaces and initiates a connection to the corresponding counterpart. In situations where a bidirectional communication channel is needed, the latter does the same in the opposite direction. As each interface is assumed self-explaining, any SCC is thus able to communicate with any other SCC without the need of any preliminary specific setup.

3.3. Universal Database

Analogous to the UAP, the Universal Database (UDB) represents an abstract view onto a global database. In this sense, the vision does not claim to have one single database that serves for all storage needs but only tries to sensitize for such a way of thinking. We assume that each SCC can connect to its given UDB interface and that this interface is in charge of finding and retrieving whatever data is needed. Note however, that this interface is restricted to handle querying and retrieving aspects. Writing access is managed differently. The concept foresees that the UDB is subdivided into sections (so-called UDB sections) and that each section is controlled by exactly one distinct SCC, which in this case is called a DRSCC (data repository SCC). Any writing aspect such as the creation, the modification or the deletion of particular data entities can only be performed by the corresponding DRSCC. As a result, if an arbitrary SCC has retrieved a particular data entity through its given UDB interface, it afterwards needs to get in contact with the corresponding DRSCC, if for example it wants to update the eventually modified entity. While a UDB section may contain several different types of data entities, all entities of a particular type are assumed to reside in one distinct UDB section which results in a 1:n relation between UDB sections and data entity types.

The clear distinction between global read-only access and single SCC based writing access represents the crucial point of the envisioned concept. If information becomes a commodity and thereby the corresponding data volume increases by several magnitudes, the main problem no longer will be how to modify and/or create data entities, but will be (i) how to find appropriate data entities and (ii) how to make sure that not only some but the entire set of actually matching data entities be found. Having comfortable access to all available data, sophisticated reasoning can be performed in order to account for the mentioned deficiencies. Furthermore, and probably as important, it becomes possible for single data entities to define relations to arbitrary other entities not necessarily of their own type and not necessarily within their own UDB section. From a business point of view, the possibilities that emerge from the definition of arbitrary relations among data entities managed by different DRSCCs (and thus different administrative domains), are almost infinite. Whatsoever information from one company can directly be bound onto whatsoever information from an arbitrary other company while preserving the required restrictions to make sure data integrity is guaranteed.

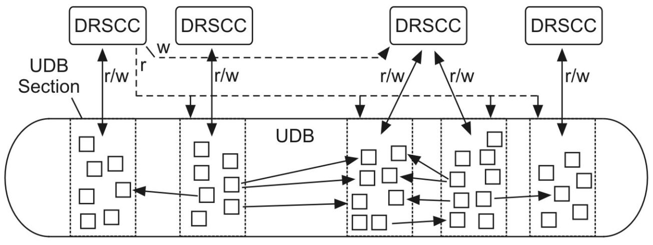


Figure 5. Architecture of the Universal Database

3.4. Conclusion

While the presented visionary system claims to provide a promising solution in order to be able to cope with the ongoing trend of ubiquity and commodity in terms of information, it nevertheless does so by only covering the problem domain on a fairly high abstraction level that ignores a lot of important aspects that eventually may be crucial in order to determine as whether or not such a system can ever become feasible.

Throughout the elaboration of the system architecture outlined in the introduction and motivation chapter, we will come across many technical and design specific aspects from which we nicely abstracted in visionary concept above. In this sense, the presented vision provides a useful broader context into which the elaborating system architecture can be laid when aspects such as transparency, scalability and the like will be at stake.

4. Requirements

The requirements analysis in [1] elaborates the components of a suitable data model and describes how experiments can be modeled in terms of OWL-S services. While this specification was sufficient in order to design the overall system architecture, it reveals various missing aspects when it comes to the definition of the fine-grained architecture. This chapter specifies the requirements for the architectural aspects introduced in the initial motivation. In the first part, we look at the overall requirements in terms of openness and transparency. As NExT represents a widely distributed environment these two aspects are fundamental and need to be discussed outside of any concrete functional context. In the second part, we look at each of the five functionalities: (i) management of distributed data, (ii) querying over distributed data, (iii) versioning, (iv) annotation, and (v) security aspects regarding authentication and authorization.

4.1. Openness

One of the most important requirements of distributed systems is openness [11]. Openness imposes that a system needs to offer services according to standard rules that describe the syntax and semantics. The rules thereby need to be open and hence need to be complete[5] and neutral[6].

In computer networks, a set of standard rules (usually referred to as a protocol) typically governs the format, the content and the meaning of messages that are sent and/or received between end nodes. Due to the inherent way networking services like TCP [15] or UDP [15] are used, it is obvious that they must meet a high level of openness. Protocols are thus specified in a particular way that allows their services to be portable and interoperable. Portability[7] and interoperability[8] are the two major aspects that need to be achieved in order for the system to become complete and neutral [11].

Analogous, services in distributed systems call for the same characteristics: They need to be equally portable and interoperable. Furthermore, if we take into consideration that such systems – in contrast to above presented network services – not only need to communicate with one single client but rather tend to interact with an arbitrary number of counterparts, flexibility is an additional important characteristic. Portability is required because distributed systems need to run in many different environments where each of them has its own specialties. Interoperability is required since components in distributed systems usually are built by different manufacturers but need to co-exist simultaneously. Flexibility, to some extent, derives from the previous ones.

[5] Complete means that everything that is necessary to make an implementation is specified.

[6] A neutral specification does not prescribe what an implementation finally should look like. However it describes what the features are.

[7] Portability characterizes to what extent an application (or more generally a piece of software) which was developed for a distributed system A can be executed on a different system B without any modifications.

[8] Interoperability characterizes the extent by which two implementations of a system or a component from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard.

Since it is common that for a particular component multiple implementations with different characteristics are available, it must be possible to tell the system, which concrete component it actually needs use. In other words, the system must be configurable by various aspects.

4.2. Degree of transparency

Another important characteristic of a distributed system is the fact that services and resources are transparently distributed [11]. A transparent system is able to present itself to users and applications as if it were only residing on a single machine. Table 3 lists the different types of transparencies and provides a short description. In the following, we look at each presented transparency in detail and discuss its relevance in regard to the NExT system. At the end, we conclude which types of transparencies are crucial and therefore need to be provided under any circumstances. We furthermore reveal which remaining transparencies would be nice to have or may be required in the future.

Transparency	**Description**
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a resource is in memory or on disk

Table 3. Different forms of transparency in a distributed system [11]

4.2.1. Access Transparency

In the context of NExT access, transparency reflects the fact, that a persisted object can be serialized in many different ways. NExT is going to use OWL for its serialization, as this is more than obvious when using the OWL-S API. As long as this determination holds true, we can safely ignore this transparency. Nevertheless, if we wanted to be flexible in terms of what concrete persistent storage mechanisms (i.e. triple store [54], object oriented and/or relational database, etc.) will be used this transparency becomes essential.

4.2.2. Location Transparency

Location transparency refers to the fact that users cannot tell whether a resource is physically located in the system. Since NExT uses data entities that are derived from RDF and/or OWL entities, this transparency becomes immanent. Both frameworks (RDF and OWL) assume that resources are being named by unique logical names that do not give any information as to where the resource physically is found. However, if we do not want to stick to RDF and OWL as the underlying persistence service, we still must account for it.

4.2.3. Migration Transparency

Migration transparency may or may not be necessary depending on the particular requirements. We can imagine a scenario where a particular user on her machine wants to submit a resource into a group-wide data container. If migration transparency were not supported, all group users including her subsequently would need to update that resource's unique identification. Because such a mechanism would represent a fairly uncomfortable system, NExT, obviously is expected to offer some adequate migration transparency.

4.2.4. Relocation Transparency

Relocation transparency refers to the fact that the system is able to move a particular data element from one location to another while it is actively used. In regard to the definition of process workflows, this transparency is not necessarily required. An NMR researcher does not directly work onto remotely located data but always works onto copies of them that primarily have been retrieved and put into some kind of a local data container.

However, the transparency is crucial as far as the execution of remote services is concerned. After completion of a process workflow definition for an NMR project, the NMR experimenter uses an execution engine in order to automatically execute the composed structure of processes. While the entire process workflow description is locally available, the referenced services may be scattered throughout the network. At run time, the execution engine reads a particular process specification and connects to the defined service port. If the service port however happens to be located on a remote location, we face the probability that at the same time the remote service is contacted by the execution engine, an NMR researcher may decide to move that service to a different location. As a result, either moving of currently used services need to be restricted or relocation transparency must be provided.

4.2.5. Replication Transparency

In the context of NExT, replication transparency does not represent a necessity per default. The system may nicely run without this feature. However, if performance becomes a problem and adequate solutions need to be considered, caching (a specific type of replication) oftentimes is an effective solution. We realize that if we want to keep our system as flexible and adaptive as possible, we need to make sure that at least replication transparency is foreseen and can be implemented when needed.

4.2.6. Concurrency Transparency

Concurrency transparency represents a crucial aspect for the NExT system. As data elements in repositories can simultaneously be accessed by multiple users, possibly concurrent occurring data modifications need to be addressed adequately. The system in this sense needs to make sure that data consistency and data integrity be preserved and that the necessary tasks are performed transparent to users of the NExT system. After all, the latter need to be able to focus on their primary work and must not be disturbed by such rather fundamental, technical aspects.

4.2.7. Failure Transparency

Failure transparency means that a user does not notice that a resource (that she eventually has never heard of) fails to work properly and that the system subsequently recovers from that failure. In terms of NExT, the majority of failures will probably occur when arbitrary services are being executed that either are in their initial testing phase or just were not specified carefully enough and hence do not account for all possible usage scenarios. Assuming that such failures will occur on a frequent basis, an NMR experimenter cannot work with the system efficiently as long as it throws exception or even may crash whenever such mentioned failure do occur. NExT in this regard ought to account for some sort of failure transparency. However, we intentionally leave the prescribed requirement somewhat fuzzy. Failure transparency is one of the hardest issues when dealing with distributed systems. At the time of writing, this topic is still subject to ongoing research. Satisfying and promising solutions have not yet been found. NExT, in this regard, is expected to come up with some adequate solution but is not required to guarantee complete transparency as such may not be possible.

4.2.8. Persistence Transparency

A major aspect that a system - not only in a distributed environment but also in general – is concerned with, is the transparency of data persistence [11]. Transparency in this context means that a client (i.e. a software component or an application) does not have to care neither about the way persistence is achieved nor about the time such operations need to be executed. In the case of NExT, the management of data plays a major role. Process workflows are specified in OWL-S and subsequently stored in some sort of distributable data repositories. In order for the NExT core to be able to concentrate on its primary tasks, it however must not deal with lower-level functionalities and hence as well should not be concerned with the persistence of its data in the first place. Persistence transparency in this regard is highly expected and should not be missing under any circumstances.

4.2.9. Conclusion

We separately looked at each type of transparency and put them into the context of NExT. Summarizing, we conclude that NExT ideally needs to account for all of them. Access, location, migration, concurrency and persistence transparency are vital and must not be missing at any circumstances. Relocation, replication and failure transparency are nice-to-have. However, our architecture better accounts for these features right away so that they can easily be implemented in the future when some of them might indeed become critical.

The latter argument may need some further explanation: Just because we can neglect some features today, does not necessarily mean that we can neglect them tomorrow. Software – as we all know – is exposed to continuous evolution [25]. As a result, the initial requirements steadily grow and transform the system into an increasingly complex product. Although we cannot prevent it from becoming ever more complex, we - to some extent – can control the degree at which this process takes place. A good architecture always accounts for evolution and tries to make sure that future extensions will nicely fit into the overall architecture [32].

4.3. Management of distributed data

NMR spectroscopists work in teams of various geographical topologies. While one team may consist of members from the same building or even the same floor, another team may consist of members from many different institutions in various countries all over the globe. Hence, physical distribution in terms of infrastructure (i.e. spectrometer), data, computing and actors (services and users) is an inherent characteristic of the field of NMR spectroscopy and therefore needs substantial attention.

Three of the four presented phases of the NMR experiment's lifecycle (see chapter 2.1) call for distribution in terms of data and computing. The first and the second phase request the system to be able to query and retrieve data from different, physically distributed repositories. The fourth phase calls for the same requirements but additionally requests the system to deposit data into remote repositories, which yet reveals a completely different scenario. The fine-grained functionalities can be categorized into three problem fields. In particular, these are (i) the granularity of shareable data, (ii) collaboration topologies, and (iii) the notion of a workspace and data repositories. In the following, we elaborate the concrete requirements for each of these fields.

4.3.1. Granularity of shareable data

NExT is expected to offer sharing capabilities on all five abstraction layers[9] of the data model for NMR experiments. Data items which represent a Case, a ProcessPlan, an Experiment, an ExperimentalStep or a Process need to be shareable on an individual basis. The following two examples depict arbitrary conceivable scenarios in which this requirement becomes legitimate.

Scenario 1: A researcher is working on a yet incomplete Case and wants to share the accomplished part of her work with members of a community. The particular part to be shared can consist of a set of elements that individually refer to different abstraction layers. While she may share an entire Experiment in one situation, yet in another situation, she may as well decide to do so on the level of an ExperimentalStep or a single Process.

Scenario 2: A researcher imports a complete Case from a repository into her private working area. She realizes that except for one particular Process she can use the entire Case as is. The only thing she has to do before executing the project is to substitute that particular Process. After the corresponding substitution and subsequent testing, she

[9] The five abstraction layers refer to the hierarchy by which the process workflow of NMR projects is defined. These in particular are: Case, ProjectPlan, Experiment, ExperimentalStep and Process.

eventually decides to offer the slightly modified Case back to the community. In this particular situation, it obviously does not make sense to publish the entire Case. A way better solution is to solely put the alternative Process into the community owned repository and additionally bring it some relation with the original Case.

4.3.2. Collaboration topologies

The lifecycle of a typical NMR experiment discussed in chapter 2.1 reveals that an experimenter for a significant part of the overall time needs to collaborate with other experimenters of the field. As NExT is to support NMR researchers by their everyday work, it therefore has to offer appropriate collaborative functionalities.

Collaboration not only takes place on the basis of individual researchers but as well on the basis of teams thereof. Several teams, usually from different institutions and/or companies, work on a joined project whereby experiments are elaborated collectively. The correspondingly required collaboration topology is illustrated in the following figure. It is important to notice that there is no central point where communication is controlled. Each team is expected to be able to communicate with its counterparts directly.

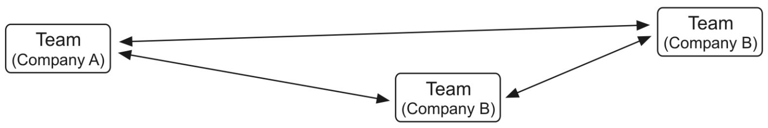


Figure 6. Team collaboration topology

As outlined in the motivational chapter, NExT eventually is expected to be used on a large distributed scale. The scenario in which multiple divisions from different institutions and companies may work on joined projects represents a feasible future constellation. In this regard NExT must be able to cope with an arbitrary number of collaborating teams. Because not only teams but as well complete divisions may collaborate, NExT must also deal with the notion of hierarchical topologies as described in figure 7. A node in this sense can be seen as a simplified abstraction of its underlying topology. With this principle, teams from different companies and institutions are able to communicate with one another but do not necessarily need to be aware of the exact internal structure of their counterparts. Figure 7 for example shows that the team from company B does not exist as is. That company only knows of the teams #1, #2 and #3. However, as such may not be important for researchers from other companies, they can look them as one unit – thus, as the team from company B.

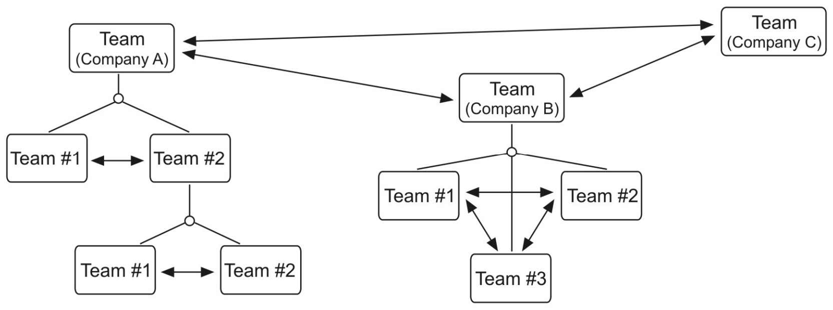


Figure 7. Recursive team collaboration topology

4.3.3. Workspace and data repositories

So far, we talked about the collaboration topologies that NExT needs to account for. Let us also look at the way researchers expect NExT to handle their concrete collaboration needs.

For the sake of simplicity, the requirements in regard to collaboration are reduced to sharing data elements of NMR project process workflows. A messaging services [11] or the functionality of collaborative storage for additional resources such as raw data material resulting from the execution of particular research applications (tools) are out of the scope of this discussion.

NExT needs to manage an arbitrary number of so-called data repositories that run independently from one another and that optionally can be distributed within a local or wide area network (LAN/WAN). In principle, such a data repository can be thought of as a place where data is persistently deposited and later be queried and/or retrieved. The repository is used in a number of different conceivable scenarios. (i) It can run on the user's local machine and serve as a private repository for a particular user. (ii) It can be hosted on a team wide server and be used as a repository for data that is to be shared among team members. (iii) A repository can be setup by multiple collaborating teams or complete divisions and be used as a common platform where data is collectively shared. In the latter scenario, the service may be hosted within the administrative domain of either a participating team or an unbiased third party player. Note that these scenarios are closely related to the introduced collaboration topologies above.

In addition to the notion of repositories, the system must as well provide the notion of a workspace. In phases 1, 2 and 4 a researcher deals with a significant amount of data stored at different locations. In order for the researcher to be efficient, she needs to be able to temporarily aggregate relevant data on the notion of a desktop. Thereby the intention is that data on a desktop is easily worked with and accessing time reduced to a minimum. The only difference between a workspace and a repository in terms of offered functionalities is the aspect of data persistence. While repositories serve as persistent data storage components, a workspace is transient and used for temporary storage during active working time.

4.4. Reasoning over distributed data

Reasoning is the process of finding data elements that meet particular characteristics that cannot necessarily be inferred from analyzing the elements in isolation but usually only can be inferred when looking at the relations among them. The motivation therefore is obvious. Publishing data in the first place would be useless if that data could not be searched for in an appropriate way. As data elements of a process workflow definition do not so much describe themselves but rather engage into distinct relations to other elements, many questions cannot be answered if we only look at their descriptive attributes. Hence, an NMR experimenter when searching for particular data elements such as Processes or ExperimentalSteps oftentimes as well is interested in their relations and indirect inferable characteristics.

The requirements in terms of reasoning can be subdivided into two aspects that can best be circumscribed by the two questions: "Where to query" and "What to query". In the following, we discuss both aspects in detail.

4.4.1. Where to query

"Where to query" reveals the necessity of defining the relevant data upon which a certain query is to be executed. It usually is not appropriate to run a query upon all available data. NExT stores data elements in different data repositories. At some place in the preference settings, a user defines her individual list of such repositories. Once a particular repository is specified in the list, it is understood to be registered, meaning that it is principally available to the NMR experimenter. When a query is executed, a so-called *relevant set* of repositories serves as the basis upon which the query is actually performed. The relevant set logically represents a subset of all currently registered repositories and might be adjusted on a regular basis as different queries for different purposes are executed. NExT in this sense must allow running a particular query not only on a per repository basis but also on a well-defined collection thereof. The result from a query upon multiple repositories must be presented in an appropriate aggregated format. The origin of a data element in the result set furthermore must always be inferable.

4.4.2. What to query

"What to query" reveals the expressiveness of the query mechanism. In phases 1, 2 and 4, NMR experimenters are concerned with tedious searching processes. Most of the time they need to look for artifacts that represent certain specific similarities. Depending on the concrete purpose of the query, different similarities and combinations thereof are imaginable. Table 4 depicts the concrete similarity aspects that need to be offered by the query mechanism. Because versioning and annotation need further specific attention, they later will be discussed in dedicated sub-chapters.

Similarity Aspects	**Description**
Element Type	Searching by the type of a data element such as a Case, a ProcessPlan, an Experiment, an ExperimentalStep or a Process.
Element Identification	Searching by the unique identification of a data element.
Version	Searching in regard to specific versions. I.e. all elements of version 1.5 or below.
Annotation	Searching by matching annotation aspects such as author, date of creation, etc.
Input/Output	Searching by matching input and output parameters in terms of their types and/or particular value ranges for data entities that represent services to be executed by a corresponding execution engine. I.e. All services that, as input, expect a floating point number representing a distance measurement.

Table 4. Similarity aspects

While the first four similarity aspects are straightforwardly understandable, the last aspect may need some further explanation. One of the major goals of NExT is the definition of process workflows for NMR projects. Hence, the majority of data elements that the system needs to deal with are not simple business objects in the sense of a person record, some invoice or shipping protocols as is oftentimes the case when looking at applications' specific data models. In contradiction, NExT is a process workflow system that handles the description of processes that can automatically be executed by a corresponding execution engine. In [1] the definition of an NMR project is specified by a hierarchical structure of elements that all can be regarded as specific types of process definitions. When an experimenter works on a particular process workflow, she primarily deals with the composition of such processes. Because many process definitions not only can be used for one specific project but also can in many situations can be reused for numerous subsequent projects, querying for suitable processes becomes a major task in every NMR project. If we remember the four phases of the lifecycle of a typical NMR project, we notice that the described task is part of the acquisition phase.

In order for an NMR experimenter to be able to find out whether required process definitions already exist, she needs to be able to perform some type of capability matching [11, 33]. The notion of capabilities thereby refers to the functionality of defining searching criteria based on inputs, outputs, preconditions and effects (IOPE) [24]. A solution that is based on a somewhat regular directory service following the notion of categories and/or taxonomies is considered insufficient. The reason thereby is obvious. In the NMR domain, a process does not necessarily need to represent a self-contained and well definable service. It might as well represent a partial task thereof. A lot of processes therefore cannot feasibly be characterized by solely relying on a specified set of descriptive attributes. At least the specification of input and output constraints is required.

4.5. Versioning

A NMR project is described by a process model that incorporates the notion of recursive processes. A Case entity wraps a ProcessPlan entity. The ProcessPlan then acts as a container for entities of type Experiment. Yet, Experiments define the workflow of a NMR project in terms of their sequential tasks. And tasks finally are represented by atomic and composite Processes (see appendix A.1).

Versioning per se is imaginable on every level of the process model structure. As data entities build relation to one another and as well may be located in different data containers, versioning no longer reveals a trivial task. Three different aspects are of major importance. (i) The process model structure represents a workflow, which eventually is executed by an execution engine. Hence, versioning might have an effect on the way a workflow is executed. (ii) As data is shared by multiple users, modifications usually are subject to some sort of versioning mechanism. (iii), People oftentimes are confused about the strict meaning of a version. Sometimes versions relate to the history of modifications; yet in other situations, they reveal slightly different data. How can we know which meaning is intended in what situation? In the following, the requirements in regard to versioning are separately discussed for each broached aspect.

4.5.1. Process plan execution

A ProcessPlan of a particular Case may happen to run multiple times as the execution engine may stop due to errors in the workflow specification. Since the resulting pathway must not necessarily be overwritten by subsequent runs, a Case needs to be assigned a version. For every subsequent execution, the researcher can decide whether the Case (thus the entire workflow description) is to be assigned a new version or whether former history data is to be overwritten. In other words, by creating different versions of a Case the researcher is not forced to overwrite previously acquired history data that consists of the values history[10] and the pathway[11].

In phase 1 and 2, a ProcessPlan is populated with Experiments and their subordinate composite and atomic Processes. When the workflow specification phase is completed, individual Experiments can be conducted by executing the corresponding workflow definition. Oftentimes the initial run will not succeed due to errors in workflow specifications or will not bring upon the expected results due to suboptimal parameter settings and the like. As a result, the researcher needs to modify some part of the ProcessPlan and needs to restart the execution again. Modifying and executing will alternate until the execution finally succeeds and the results are as expected. Since previous execution runs do not necessarily have to be overwritten by subsequent runs, a Case needs to be assigned a version.

[10] The value history keeps track of the process parameters (element attributes) during execution of a NMR project's process workflow. [1]

[11] The pathway keeps track of the sequence of which processes (Experiment, ExperimentalStep, Process) are executed by the execution engine. [1]

4.5.2. Model entity modifications

Entities such as composite and atomic Processes or ExperimentalSteps of previously completed projects may arbitrarily be reused for other projects. If these need to be changed in order to become applicable, they need to be assigned different versions. In this situation, they must never overwrite the original entity nor should they be saved as sole copies and thereby loose the relation to its original. In this regard, every entity of the process model needs to have a version assigned.

4.5.3. Versions and variations

Due to the fact that a particular process model entity can be referenced by an arbitrary number of other entities, the concrete context in which an entity is used may differ significantly. Hence, there is a high chance that different variations of a particular entity may be needed over time. While there may be no difference between a version and a variation of a model entity in terms of data representation, there nevertheless may be a significant difference in terms of its semantics. Figure 8 illustrates the usage of versions and variations.

Versioning accounts for the fact that defining the workflow of an NMR project is a repetitive task of defining process model elements and their concrete attributes. The attributes of a single entity may be changed several times until the final and correct setting is eventually found. In order for the researcher to be able to track these changes, she makes use of some versioning method. In this sense, a version tells a researcher how often and in what order an element has iteratively been changed over time. In other words, it reveals the modification history. In figure 8 different versions are represented by stapled data elements.

The notion of variations is required because a NMR experimenter needs to be able to adapt an existing process model entity to suit her particular situation. In fact, this is one of the major tasks an experimenter is employed with during the phases 1 and 2. In this sense, taking an existing data entity and changing its attributes does not lead to a new version but rather to an alternative solution for the corresponding context. In summary, variations are slightly different data entities that though still represent the same basic workflow element. In figure 8 variations are depicted within dashed rectangular and assigned a variation identification.

In the case of NExT, modeling an NMRProcess with different Groundings can be regarded as a suitable usage scenario. The definition of a concrete NMRProcess except for its Grounding oftentimes can be used by many different research teams. Because their infrastructure in terms of computing tools and/or spectrometers usually differs, each of them needs to modify the corresponding Grounding in order to adapt it to their particular infrastructure. By assigning different variations to different Groundings, the system is able to express the corresponding relation between them.

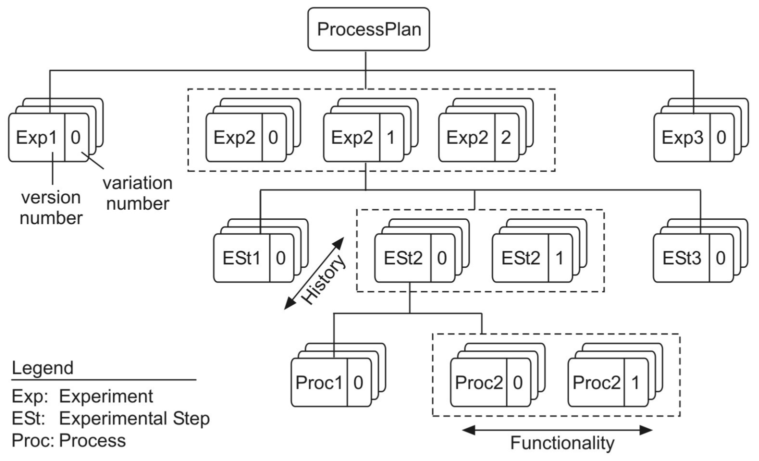


Figure 8. Versioning in the process model

Above discussion neither presents a conceptual solution to the handling of versions and variations nor does it prescribe concrete requirements. However, it reveals the concrete problem situation that NMR researchers do face in the particular aspect and points out what tasks NExT need to provide in order to receive appropriate acceptance. In this sense, NExT has to find a way to efficiently and satisfactory deal with the presented problem domain.

4.6. Annotation

NExT needs to support researchers in the NMR domain by their everyday work. One of the major overall aspects in order to achieve the many goals outlined in [1] is the notion of an adequate and efficient documentation feature. In this context, the system is expected to offer annotation capabilities that

- complies with major standards,
- are extensible in terms of concrete supported attributes,
- include adequate searching and reasoning aspects,
- and are provided by an optional component.

The first aspect implies that a concrete implementation should comply with at least one of the widely used annotation frameworks. Additionally it must go for an architectural design that allows to later replacing the implemented framework by some other standard that by then may significantly be favored over the former one due to improved features or overall acceptance.

The second aspect implies that the set of provided annotation attributes is easy extensible. Most of the standard annotation frameworks such as VCARD [34] or DCMI (Dublin Core Metadata Initiative) [13, 35] are specified by a well-defined set of attributes that can be used to express annotation information. The requirements in terms of NExT go one step further. The system not only needs to offer the assignment of interoperable annotation attributes defined by the chosen standard but as well needs to offer the assignment of generic metadata information (attributes). Figure 9 illustrates the abstract idea and implicitly reveals the concrete requirements specification. Each process model entity can be assigned well-defined annotation attributes of the chosen standard framework. This aspect is illustrated by the composite attributes box to the right of the entity. All other surrounding rectangular boxes represent generic metadata attributes. Note that a particular attribute can be assigned multiple times. In the figure, for example three different hints are attached. Note furthermore that an attached metadata attribute can optionally have assigned the user who for example acts as the creator or last modifier of the annotation or metadata attribute.

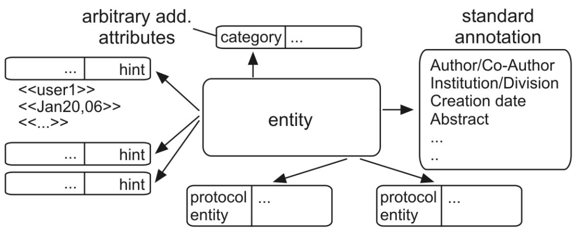


Figure 9. Extensible annotation concept

The third aspect defines that an NMR experimenter not only needs to be able to adequately annotate its data but also needs to be able to reason about all annotation specific attributes. Assuming an experimenter over time has assigned her name to each created data entity, she later may want to be able to start a query in order to find all of her defined data entities.

The forth aspect reveals that support for above discussed annotation is to run optionally. While some installations may need to run with full-fledged support, others may run with limited or zero support.

4.7. Authentication and Authorization

Security aspects concerning authentication and authorization need to be addressed independent from one another. In contradiction to the majority of applications, NExT must not assume that authorization is solely used in combination with authentication or vice versa. The concrete requirements for the two functionalities are separately discussed in the following.

4.7.1. Authentication

The requirements in terms of authentication are depicted in the list below. A detailed discussion on each of the presented points is given hereafter.

- Compatibility for a wide range of standards
- Support for different procedures in different system environments
- Transparent login procedures with least user interaction

- Configurable
- Implemented as an optional feature

Compatibility is required in terms of a wide range of authentication and authorization standards. This requirement is legitimate because NExT is expected to run in various different system environments. On the one hand NExT may be used by a single user that runs all the required system components on one and the same machine. In this situation, security plays a neglectable role. Usually some sort of a Unix-like user and group file is sufficient in order to manage the security policy. On the other hand NExT may as well be used in a widely distributed network which spans over multiple administrative domains. If this is the case, each administrative domain may run its individual authentication and authorization procedure. As a result, NExT may not only need to comply with a small set – let us say one or two – of the most widely used technologies but most likely may need to cope with a broad set thereof.

Allowing different procedures in different environments is closely related to the first point. The main aspect already has been discussed in the previous paragraph concerning the fact that different administrative domains do naturally stick to different authentication and authorization standards. The second aspect is equally important but oftentimes neglected. The notion of different procedures also reveals that authentication and/or authorization processes at different administrative domains may significantly vary among one another. While in terms of authentication one administrative domain may use a single-sign-on service [16] and thus only requests the user to authenticate at startup of the operating system, yet in another domain this may not be the case. Hence, NExT also needs to offer adequate authentication functionality that can be used on an optional basis.

Data repositories introduced in chapter 4.3 offer functionalities to store information in a flexible, distributive and collaborative manner. An NMR experimenter defines an arbitrary number of data repositories that serve as the set of principally available repositories. Every data centric operation such as a query, a retrieval or an update runs in the context of a corresponding repository thereof. As a result, the process of connecting to a particular data repository is likely to occur on a permanent basis. Furthermore, because access to a repository usually is subject to specific access restrictions, the process of connecting to a remote data repository usually is not possible without appropriate authentication. In order for the user to be able to concentrate on her primary work, NExT needs to provide a solution that handles the necessary authentication procedures in a maximal transparent fashion.

Authentication aspects that arise from the previous points are expected to be configurable. It is needless to say that recompilations of particular system components will not be accepted. The main reason therefore is that such a procedure would heavily reduce overall maintainability [32].

Authentication aspects need to be implemented as independent as possible. The system has to be deployable even if certain or all of the discussed features above are not implemented. Depending on the concrete system environment or the concrete usage scenario, authentication functionality may not necessarily be required. Furthermore, it is highly desirable to be able to implement the envisioned system architecture with sole focus on the main features. If authentication were not considered optional, such however would not be possible.

4.7.2. Authorization

In terms of authorization NExT needs to account for the fact that different data needs to be assigned different access permissions. In this sense, the system is expected to support some sort of standard access control. While the choice for a concrete standard is up to the implementation, at least following concrete characteristics are expected and must definitely not be missing.

- The system works with the following distinct access rights: READ gives a user the right to consume a particular piece of data, WRITE gives her the permission to modify, hence DELETE gives the right to delete some particular data.
- The definition of access rights can be assigned either on the basis of data elements or entire data repositories. Obviously, an implementation that optionally is able to account for both features would be welcomed.
- Authorization must be supported by the concept of principals [36]. A user in this sense can be assigned multiple principals meaning that she has multiple identities.

5. Design Concepts

Design concepts represent the building block of every thoroughly elaborated system architecture. They describe the pursued underlying structural ideas, applied metaphors [37] and analogies to other well-known areas in or off the corresponding field [32, 37]. This chapter principally leans on the structure of the previous one and elaborates on concrete concepts for each of the presented requirements. Because generic software aspects such as transparency, openness [11], scalability [11] or substitutability [32] are major elements that significantly shape a concept's intentions and visions, sometimes not only the raw conceptual idea can be presented but also a broader approach is necessary.

5.1. Management of distributed data

The requirements prescribe that NExT needs to account for some sort of distributable data repositories that can be accessed by NMR experimenters from different teams eventually located in different companies or institutions. The requirements furthermore describe how collaboration between teams is expected to be handled and focuses on the notion of recursive structures with adequate transparency in terms of collaboration between different companies and institutions as well as different organizational levels therein. We first look at how we can meet the requirements in terms of distribution and collaboration in general. Second, we concentrate on the detailed aspects of creating, publishing, retrieving and modifying the corresponding data entities.

5.1.1. Distribution and collaboration

Collaboration requirements are limited to sharing process model elements and the execution of specified processes by a corresponding execution engine. From a conceptual point of view, it thus is feasible to think of collaboration as some type of data sharing. Our concept defines a generic component named Repository that can be used for three distinct functionalities: (i) to represent a data repository, (ii) to transparently act as proxy for a conglomerate of the latter, and (iii) to be able to represent a private workspace. By accounting for these distinct functionalities, the requirements in terms of distribution and collaboration can be met satisfactorily. The following figure illustrates the main principle.

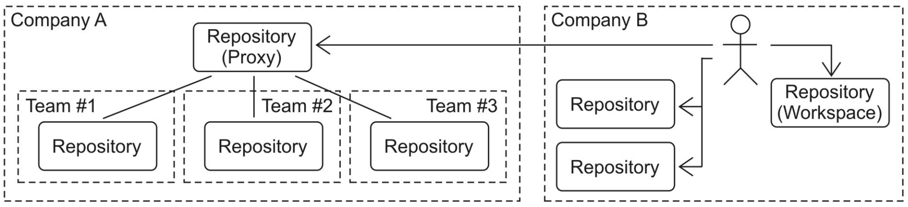


Figure 10. A repository component for three functionalities

The requirements prescribe that different teams (eventually from different companies or institutions) need to be able to collaborate without the need for some type of a central communication point. The concept in this respect foresees that each team hosts an arbitrary number of repositories and that these can optionally be opened for remote access. Several teams in this sense can share their data among one another by mutually opening some of their own repositories for community wide access. Different read and write access policies thereby are imaginable. For example, if the teams primarily want to share their completed work but are not so much interested in collaboratively elaborating new NMR projects, they can open their repositories for read-only access. Being able to read from other repositories thereby leads to two major benefits: First, teams are no longer limited to reason upon their own data but can do so upon a community-wide virtual data basis. Hence, the likelihood to find suitable data elements during the first and second phase of the NMR project's lifecycle (see chapter 2.1) can significantly be increased. Second, the teams as well may be allowed to run particular services or entire process workflows if they additionally are given the required execution rights.

The prescribed notion of a workspace can reasonably be provided by a slightly modified repository. Except for the persistence functionality, a workspace behaves in the same way as a repository illustrated above. Thinking in terms of inheritance, we conceptually either can look at a generic repository as a subtype of the workspace with extended functionality or can look at the latter as a subtype of the former with specifically applied restrictions. A user on the other hand may still treat them as totally different components.

A proxy repository is used by the concept in order to account for the hierarchal structure of teams within companies or institutions. The requirements prescribe that on each hierarchical level, it must be possible to abstract from any subordinate structure. With other words, if an NMR experimenter from company B wants to accesses a repository from company A, it need not concern about company A's internal structure. The repository in question does not necessarily need to exist in reality. It as well may internally be represented as a conglomerate of multiple team-wide repositories as is depicted in figure 10. The introduction of such a proxy has the benefit that the system does not explicitly need to care about recursion and transparency but that such are inherently provided.

5.1.2. Creating data elements

When defining a NMR project, the user starts out with an element of type Case that references a ProcessPlan and that again references an arbitrary number of Experiments. These three elements are yet useless and do not represent any precious information. An Experiment is mainly specified by its attached ExperimentalSteps and their further referenced composite and atomic Processes. The entire structure of a modeled Case, a ProcessPlan or an Experiment respectively is described in appendix A.1. We realize that the definition of an Experiment can be seen as a recursive collection of strongly interdependent model elements. Thereby, a single data element is useless because it does not and cannot hold any semantic information. This fact in mind, the NMR experimenter needs to have the notion of a virtual element basket. In this regard, data elements within a basket are considered to be treated as one unit as they tend to be heavily interrelated. An element basket has a unique identifiable name and is created in the user's private working area. Per default, NExT automatically puts each Case together with its ProcessPlan and Experiment into a separate basket. Subsequently added elements are put into the same basket if not told otherwise. A Case can indirectly reference data elements from different baskets; furthermore, particular elements can arbitrarily be moved from one basket to another.

5.1.3. Publishing data elements

Publishing in principle is the process of making an element accessible to other users. In the previous discussion, the notion of a so-called element basket was mentioned. A user in this regard does not want to share a single element but a particular set of elements that is referred to as an element basket.

When the user shares some of her work, she publishes it into a data repository. Simply speaking a data repository is a place where data can be persistently stored and later be queried and retrieved. NExT in this sense uses an arbitrary number of repositories that independently run in the form of deamon-like services that are distributed onto different machines within a LAN or WAN. Consequently, a data element is referred to as *published* if it resides in a data repository instead of the local working area. Depending on the specific user rights of a particular repository, other users have more or less restricted access to the contained data. We will come back to user rights when we look at the aspect of modifying data elements shortly hereafter.

5.1.4. Retrieving data elements

Retrieving is the process of reading data elements from the private working area or remote repositories. In principle, the requirements for this process are obvious and straightforward. As we deal with recursive element structures, whose elements reside in different baskets in again physically distributed data repositories the subject gets a little more complicated. The simple part specifies how the retrieval process in regard to a particular data element needs to run. The somewhat awkward part specifies how the system is expected to handle complete element structures.

The concept allows retrieving a particular data element based on its unique identification without the need for specifying at which repository (or local private working area) that element is indeed to be found. Put with other words the fact that elements are distributed onto physically separated data repositories is handled transparently. Given a particular element's unique identification, the system locates the corresponding element within the scope of the list of registered repositories. From the previous discussion we know that a single data element is hardly ever used in isolation and therefore resides in an element basket that unites closely related other elements. As furthermore, a basket is understood the smallest shareable unit, a request of a particular data element is implicitly interpreted as the request for the corresponding basket. Instead of returning the sole requested data element, the system thus returns the entire corresponding element basket.

If the requested data element specifies relations to other elements, the retrieval process not only returns the explicitly identified element but also returns all recursively referenced elements as well. In the case where these elements reside in the same element basket as the primary (root) element, the retrieval process in principle remains the same. From above discussion, we know that an element is always returned within its residing basket. If on the other hand some of these elements are not located in the same basket, the system engages into a recursive retrieval process and finally returns not only one but several elements baskets in order to account for so-called foreign referenced data elements. Note however that the recursion is only processed within the particular repository. If relations happen to point to elements in other repositories, these relations cannot be revolved. Repositories do not know of one another. In this case, the referenced entities need to be retrieved by the client manually.

5.1.5. Modifying data elements

When a persisted[12] data element is modified, the question arises how the system is going to handle the subsequent saving process. In principle, two different saving methods can be applied. While the first method represents an overwrite operation, the second deals with a version mechanism that has the primary goal to keep both the original and altered element. Each method has its individual, distinct characteristics. Depending on the concrete situation, either the former or the latter method may be more appropriate. Conceptually, we can provide both methods and thereby are able to give the user the ability to optionally define which method may needs to be applied. If the user does not care, a previously declared default method can apply.

We recall the notion of an element basket from the discussion above. Because a sole data element usually does not have a purpose or meaning when used in isolation, NMR experimenters need to have the possibility to bundle closely interrelated data elements within so-called element baskets. Thus, each element resides in a concrete basket. If a modified data element is saved by the latter method, it is important that this modified element be put into the appropriate basket. Because baskets themselves may reside in remote data repositories that are subject to specific user rights, the saving process is not always straightforward. Table 5 depicts the default saving methods for each possible situation. Overwrite reveals whether or not the original element per default is overwritten. Respectively, "Version" points out whether the modified element is assigned a new version and "Same Basket" reveals whether the modifications are stored in the same basket as the original element.

Element Origin	**Permission**	**Over-write**	**Version-ing**	**Destination**	**Same Basket**
Working area	-	x			x
Repository	read-only		x	Working area	
	writable, not owner		x	Repository	x
	writable, owner	x			x

Table 5. Default saving scenarios

Optionally, the user always has the possibility to overrule the settings specified in the table above. The process is sufficiently defined if the destination repository and the concrete saving method are specified. Because repositories are subject to individual user rights, not all combinations are indeed valid. The system yields with an error message if the requested process cannot be performed.

[12] A persisted element refers to a data element that previously has been saved in some persistent data storage but currently resides in applications memory and eventually may need to be updated from there.

5.2. Reasoning over distributed data

Reasoning is the process of executing arbitrary queries upon a well-defined data basis. If a query is executed, the reasoner engine needs to have access to the compendium of data that represents the reasoning basis. Furthermore, depending on the statement of the query (thus the type of answer that is expected upon execution) the engine may need to check every single date entity in order to be able to generate the corresponding query result. As in the context of NExT data is located in different data repositories, the question arises how we go about in order to reason upon a composite set of data repositories. In principal, two methods are imaginable: either we can locate the reasoner engine at the user's workspace and reason upon the data that currently is loaded or we can equip each data repository with a reasoner engine that queries upon the repository's contained data. Using the former method reveals that querying can only take place after the user has loaded the so-called relevant set (introduced in chapter 4.4.1) of data into her private workspace. On the other hand, using the second method reveals that a query needs to be executed at each relevant data repository and that afterwards the query results need to be aggregated appropriately. The major characteristics of the two methods are described in the table below. In the following, each listed characteristic is separately discussed in detail. Finally, we conclude which method can better meet the requirements and present a corresponding concept that can be applied in chapter 6 when we build the system architecture.

Characteristic	**Centralized reasoning**	**Distributed reasoning**
Temporary required data storage	high (–)	moderate (+)
Data transfer volume	high/moderate (–)	small (+)
Graph computation	high (–)	moderate (+)
Query language compatibility	high (+)	small (–)
Reasoning effectiveness	high (+)	moderate (–)

Table 6. Reasoning characteristics for centralized and distributed method

5.2.1. Temporary data storage

Temporary data storage refers to the volume of data that temporarily needs to be stored in order for the reasoner engine to be able to conduct the requested query. Because reasoning (in contradiction to traditional querying) not only constitutes in matching entity attribute values but also includes the analysis of relations between entities and their categorization in regard to the type within known taxonomies (or ontologies respectively) [38], a reasoner engine uses some sort of a graph representations in order to be able to infer the requested information. Temporary storage in this respect represents the storage need for temporarily graph representations. In the centralized approach, all relevant data is aggregated in a private workspace and hence a corresponding graph representation rather tends to be heavy in size. When reasoner engines are placed at individual data repositories, their resulting graph representation on the other hand tends to be much smaller than is the case for the former method. The reason is that we plausibly can assume that in order for a user to execute a meaningful and thus valuable query, it previously needs to load huge amount of data from several different data repositories. Clearly, the distributed (thus decentralized) approach in this regard is highly preferred over the centralized approach.

5.2.2. Data transfer volume

The volume of data that is transferred between a user and the relevant data repositories differs significantly. In the case of the centralized approach, all relevant data (eventually from many different, distributed data repositories) first needs to be loaded into the private workspace in order to be able to execute an arbitrary query. As a result, the transferable data volume per se is extremely high. We might be able to reduce the high volume to some moderate load, if we implemented a sophisticated update mechanism that makes sure that only data that has been outdated or is not yet available in the private workspace is being loaded. In case of the distributed approach, only the query and the query results need to be transferred between the user and the relevant data repositories. Again, we realize that from a scalability and feasibility point of view, the latter approach is highly preferred over the former one.

5.2.3. Graph computation

Graph computation tends to be high in case of the centralized approach and tends to be moderate in case of the distributed approach. Because computation directly depends on the amount of data that needs to be represented, the same argument as we discussed in the first point, namely the requirement for temporary data storage, applies. Additionally, we can argue that graph computation on the basis of individual data repositories is more economic than doing so for each individual workspace. Once the graph for a repository has been calculated, in can be used for queries from multiple users as long as data in that particular repository remains unchanged. Obviously, this characteristic especially becomes valuable when data repositories with rarely modified data are concerned as is the case for archives or *publishing-only* repositories (repositories that only serve as data containers for completed, thus published work within well-defined community boundaries).

5.2.4. Query language compatibility

The requirements prescribe that the system needs to rely on third party reasoners that are integrated in a plugin-like fashion. As different reasoner implementations may require different query languages, compatibility aspects need to be closely analyzed. In terms of the centralized approach, compatibility is not a problem. Different reasoners arbitrarily can be used for different query needs when available (plugged into the system). As far as individual query languages are concerned, either the business logic (above the system architecture) can manage them transparently to the user or the latter needs to choose the appropriate language which is feasible however not necessarily comfortable. In terms of the decentralized (distributed) method, compatibility however represents a major problem. While in the former case a query is executed at a single location, namely at the user's private workspace, in the latter case, the query is executed in isolation at each relevant repository and the resulting query answers are aggregated afterwards. What, if not all repositories have the same reasoner implemented and hence do not support the same query languages? The system needs to provide a common query language from which translators located at individual repositories can translate the common query into their reasoner's supported language. However, as defining a common query language is a very difficult task and is still subject to ongoing research, such a solution is not feasible at the time of writing. In summary, when the distributed approach is applied, reasoning upon a defined set of repositories is not necessarily possible. Indeed, such is only possible, if the defined set of repositories at least support for one mutually compatible query language.

5.2.5. Reasoning effectiveness

As we already have noticed when we talked about the motivation in chapter 2.3, reasoning over distributed data repositories reveals the problem of how to infer indirect knowledge that is not contained at one place but eventually distributed over multiple repositories. OWL/RDF allows assigning attribute values to a particular data entity at different places. It handles attributes as triples of a subject, a property and an object. The subject represents the data entity to which the attribute value is assigned. The property represents the specification of the attribute. Finally, the object represents the attribute value that can either be a character string, a date, a number, etc. (direct value) or a reference to yet another data entity. Because every entity of the process model is represented by an OWL individual, an NMR researcher can specify a particular attribute in one repository whereas the subject (thus, the data entity to which this attribute is applied must not necessarily be located in that same repository. When reasoning is at stake, this realization is crucial to the outcome of a query result. A reasoner engine obviously can only infer knowledge that emerges from its given data basis. Hence, if reasoning also needs to account for data entity attributes that are not stored in the same repository, there obviously is no other solution but merging all relevant data into one storage container and perform reasoning thereupon as is done by the centralized approach. In this particular situation, the distributed approach is highly handicapped, as it is not able to offer such functionality by any means.

5.2.6. Conclusion

A decision for either of the two presented methods is not trivial. The distributed approach provides a promising solution for the first three characteristics as far as scalability and feasibility are concerned. However, this approach is not able to adequately account for the last two characteristics that seem to be as important. On the other hand, the centralized approach fails to feasibly account for the first three characteristics, but yet is able to deal with the last two characteristics. Because the centralized approach totally fails as far as scalability in terms of storage and computation are concerned, we reasonably need to think about applying the distributed approach and figuring out how the last two characteristics can be solved adequately.

In terms of the compatibility of query languages among different data repositories, we already argued that such can be overcome by making sure that each repository does at least offer one default reasoner implementation. Furthermore, if our concept is flexible enough, a future available common query language (CQL) may be possible to be integrated.

The inability of the decentralized method to cope with reasoning over data entities whose attributes may reside in different repositories, can be neglected, when we take into consideration the relatively rare situations such may be needed. Both, the requirements specification (chapter 4) and the concept for the management of distributed data (chapter 5.1) assume that the definition of data entity attributes together with their referring entities usually reside in the same *element basket* (see chapter 5.1.2) thus as well in the same data repository.

The underlying concept to be followed by the system architecture in order to account for the prescribed reasoning capability principally has been shaped throughout above discussion. For the sake of completeness, we present the main design-oriented and technical aspects that the system architecture needs to account for.

- plugin-like integration of different reasoners
- independence of data repository and reasoner
- keeping different reasoners implementations transparent
- modeling different query representations

Furthermore, a broad overview of the concept following the distributed (decentralized) approach is depicted in the figure below.

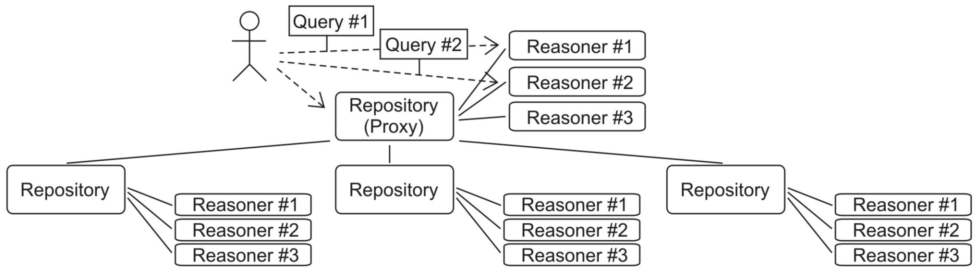


Figure 11. Reasoning concept overview

5.3. Versioning

The requirements prescribe the implementation of a versioning mechanism that is capable to distinguish between the notion of versions and variations. In the following, we elaborate a versioning concept that directly leans on the outlined idea from the requirements chapter. First, we explain how we deal with different versions and what difficulties may arise when the system is distributed and aspects like accessibility and performance become relevant. Second, we demonstrate how we can account for the notion of variations. We will realize that by offering support for variations we not only will meet the requirements but also will gain a promising solution for one of the major difficulties in terms of versioning in distributed environments.

5.3.1. Versions

Versioning takes place on the basis of process model entities. Each entity is assigned a concrete version number that reveals to a particular point in its modification history. The higher the version number the more modification steps in general have been performed on a particular data entity. After the modification of a particular data entity, that entity can be saved by either the same or the following next version indication. The concept does not define in what situation a new version is required and in what other situations the same version indication may be kept.

Note that from a software engineering point of view this is an important characteristic. By leaving the decision up to components that eventually will make use of the versioning concept, separation of concern [32] is accurately applied.

The concept treats versions as modified copies of the initial data entity. The relation between two concrete versions of the same data entity is sufficiently determined by two descriptive attributes. While the scope attribute defines the common intended representation, the version attribute reveals the relative point in regard to the modification history. The scope value can be an arbitrary URI. The importance is that it keeps the same for all versions of a concrete data entity. Given a particular data entity of a particular version, all other available versions of the same entity can then be inferred by looking for entities with the same scope value. Figure 12 illustrates the use of the scope value.

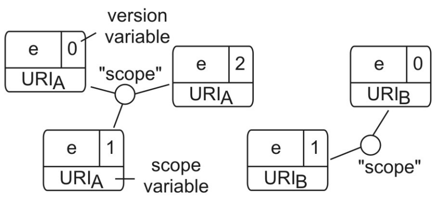


Figure 12. Versions within a scope

Among entities of the same scope, the version attribute value is unique. While the initial (original) entity is assigned the lowest version, subsequently modified entities are assigned increasing numbers thereof. The concept assumes that for every entity, that does not represent the origin or the latest version the preceding and following version is inferable by comparing the version attribute among the entities of the corresponding scope. Figure 13 illustrates that this procedure is feasible as long as we can assume that a new version is always created from the current, latest version. Creating a version from some intermediate entity leads to the implicit creation of a branch [12], which hence violates the just made assumption.

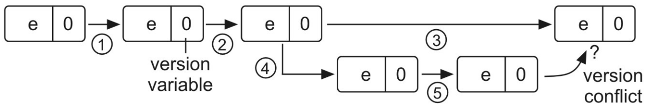


Figure 13. Branches conflict version history

The process of creating a new version consists to two specific tasks that need to be performed on the underlying entity. First, the entity needs to be assigned a separate unique identification (UID). The reason therefore is obvious. Every version is treated as a separate element that coexists with preceding versions in the same environment. Hence when an entity is assigned a new version it unavoidably also needs to change its current UID. Second, the attribute that holds the current version obviously is to be changed to hold the new version number.

Because versions are represented by independent data elements, versioning becomes a localized aspect (localization principles are discussed in [32]). System components not interested in versioning can safely ignore what we have discussed. They can treat a data element as if it did not support versioning at all. While we may not fully grasp the extent of this potential at this early time, we surely will do so when we discuss the concrete system architecture in the next chapter.

There is one major problem that we face when applying the presented concept to the system architecture of NExT. While the concept assumes that a new version is only created from the latest available version it does not account for the fact that such may just not be feasible. In the case of NExT the assumption cannot be met due to two distinct reasons. First, data is subject to some sort of access control, which prohibits certain users from accessing certain data. As a result, the latest version is not necessarily accessible by the user who starts creating a new version. When the system queries the data repositories for the latest version it finds the highest numbered version that access is provided to the current user. Obviously, that version does not necessarily represent the actual latest version.

Second, data eventually is distributed over multiple data repositories of which most need to be accessed over the network. As a result, accessibility is no longer guaranteed but subject to typical network, corresponding aspects that we do not want to discuss in here.

In order to solve this problem, the concept introduces an additional attribute that each data entity is to provide if it accounts for the above versioning mechanism. The additional attribute is a pointer to the data entity of the previous version and gives the system an additional path tracking mechanism. In the case where two data entities are assigned the same version number, the system can optionally distinguish the versions by their root path. In this respect, the decision whether branches make sense in a particular contexts or whether they happen to exist by accident and need to be corrected is left up to the user. In the latter case, entities with peer versions need to be manually merged such that the result is again one single version path as is expected the normal case. The following figure illustrates the discussed aspect and shows how so-called duplicate peer versions (entities of the same version indication but unconsciously created by different users with probably different access rights) can uniquely be tracked.

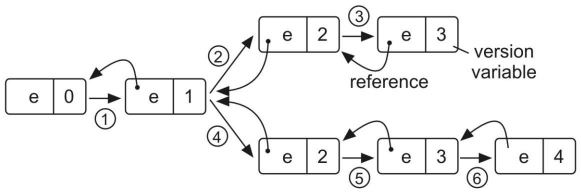


Figure 14. Duplicate (peer) versions

5.3.2. Variations

The difference between a version and a variation primarily lies in their semantics. A version as discussed above accounts for the history of modifications that take place as a data element is subsequently altered. A variation on the other hand accounts for the fact that in certain situations modifications not only lead to a next version of the original data element but also do lead to the representation of a slightly different entity than the corresponding original. As in such a situation, the difference is only small and the modified entity still has strong relations to the original entity, it is not accurate to store it as a fully independent data entity.

The herein described concept looks at variations as an extended functionality of the previous discussed versioning capability. In principle, a data entity that represents a specific version optionally indicates a specific variation. Concerning the versioning path a variation in combination with a version scope defines a separate version space[13] where versioning again starts from the beginning. For example, if we have a data entity that represents version 7 and now decide to create a variation the version of the newly defined variation will simultaneously be set to the lowest (initial) version, which usually is represented by the number 0.

While a version only can be created from the current latest version, the creation of a variation is not limited by any aspect. As a result, any intermediate version can serve as the basis for a new variation. Note also that while the attribute value for the next version is well determined, such is not the case when defining a variation. In fact, any arbitrary value (such as an arbitrary character string) can be used for the specification of a particular variation as long as the value is guaranteed to be unique within the corresponding version scope.

Figure 15 depicts a typical situation for the usage of a variation. On the left hand side, the modification history of a data entity e is shown. While the original entity is assigned version 0 the latest version is 7. On the right hand side, a variation of name "A" is depicted that is created on the basis of version 4 of the original entity. The circled numbers refer to the sequence of actions. Note, although version 7 already exists at the time of the creation of the variation "A", that variation is not forced to emerge from the latest version but is free to choose any existing point within the modification history.

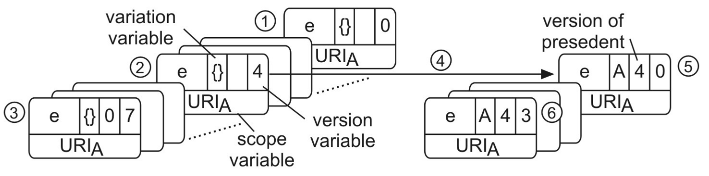


Figure 15. Definition of variations within a version and/or history path

5.4. Annotation

NExT will probably be developed in incremental steps. Hence, a first version will eventually only contain a small part of the compendium of envisioned features. Gradually, NExT will then be extended and improved. As far as annotation is concerned, we definitely must not speculate about the final set of implemented features. Chapter 2 – among other topics – talked about the

[13] A version space has the analogous meaning of a name space as defined for the concepts of XML or the internet domain names. Interested readers are referred to [41].

power as well as future compelling urge for metadata and thereby revealed that annotation, as we understand it in the context of NExT, may only be a very little part of the overall big picture.

In the following, we present a concept that describes how generic data entities can be assigned generic metadata. The concept leans onto the ideas above and treats generality as its major goal. First, we introduce the concept's overall idea and talk about the relation between OWL, the OWL-S API, and Java as the used programming language in the upper layers of the envision system architecture. In the second part, we elaborate the generic object structure that shall offer the prescribed flexible annotation mechanism. Thereby, the main interesting aspect certainly is the way we deal with multiple inheritance above the data persistence layer. Remember, Java does not offer any support for multiple inheritance as far as classes and objects are concerned [39]. In the last section, we translate the generic concept onto the level of NExT. Clearly, this will be the time when we will find out to what extent the concept will help us meeting NExT's requirements in terms of flexibility and extensibility.

The coarse-grained system architecture of NExT described in [1] specifies the use of OWL and the integration of the corresponding OWL-S API from Mindswap that eventually will be responsible for the mapping of OWL concepts to corresponding Java objects. One of the major reasons why OWL has been chosen for the persistence of application data is its inherent support for ontologies and semantics. The herein presented concept directly follows this initial intention. As a result, it starts right at the lowest level of the system architectures and from there steadily goes up to the business logic or the NExT core [1].

5.4.1. Multiple inheritance in the data representation layer

In the data representation layer the concept uses multiple inheritance in order to decouple the primary data from its corresponding metadata. A generic data entity inherits from two concrete classes. While one inherited class accounts for the primary data representation, the other class accounts for the corresponding metadata. In the following, we shall refer to the former as primary class and the latter as metadata class.

The primary class can be of any type. This reveals that the concept does not make any assumption as far as a generic data entity is concerned. While in the generic case, the base type of the primary class probably is an OWL individual, in the case of NExT, the base class is a Process from the process model or an arbitrary subclass thereof.

The metadata structure on the other hand is encapsulated in a separate class and thus independent of the concrete data entity it eventually will refer to. If we somewhat abstract from the notion of multiple inheritance and look at the two classes as two scopes for individual evolution we can compare the concept's fundamental idea with the bridge design pattern [40]. Indeed all major characteristics of the bridge pattern are also true for the presented concept if they are translated correspondingly.

Let us look at an example that illustrates how a simple data entity can be defined in OWL (see figures below). We assume that "entities" refers to the ontology for the primary data structure and that this ontology defines a class Entity. In order for Entity to be aware of corresponding metadata, it is defined as a subclass of Metadata that shall be defined in a separate ontology named "metadata". Figure 16 shows how an instance of type Entity is defined with both the attributes of the primary class (property1 and property2) as well as the attributes of the secondary class (creator and date). Primary data in this respect is decoupled from metadata in terms of separate ontologies. Note that the attributes property1 and property2 refer to the "entities" ontology and that the attributes creator and date refer to the "metadata" ontology.

Figure 17 shows an alternative way for describing the same situation. It further decouples the primary data from its corresponding metadata by moving the latter part into a separate entity scope. OWL allows for the definition of data elements in a distributed fashion. While one construct serves as the basis and is defined by the rdf:ID an arbitrary number of additional constructs can refer to the same base by describing the corresponding relation with the rdf:about construct. Although the metadata attributes are defined in a separate entity construct, the technically still refer to the same entity, namely the entity named E1. Obviously, this second method has one major advantage. It allows for the replacement of different metadata definitions while keeping the primary data unchanged. If sometime in the future the application is to switch from one metadata concept to another, it may do so by only adapting the metadata constructs. Thereby one appropriate and efficient technique is the use of XSLT [41]. XSLT allows the definition of data transformations encoded XML. The Entity construct that defines the metadata for E1 could thus be transformed into the required construct of another metadata ontology.

```
<rdf:RDF
    xml:base="urn:repository1"
    xmlns:md="urn:metadata"
    xmlns:entities="urn:entities">
    <entities:Entity rdf:ID="E1">
        <entities:property1>value1</entities:property1>
        <entities:property2>value2</entities:propertye>
        <md:creator>Author 1</md:creator>
        <md:date>2006-03-05</md:date>
    </entities:Entity>
</rdf:RDF>
```

Figure 16. Encoding a simple data entity

```
<rdf:RDF
    xml:base="urn:repository1"
    xmlns:md="urn:metadata"
    xmlns:entities="urn:entities">
    <entities:Entity rdf:ID="E1">
        <entities:property1>value1</entities:property1>
        <entities:property2>value2</entities:property2>
    </entities:Entity>
    <entities:Entity rdf:about="E1">
        <md:creator>Author 1</md:creator>
        <md:date>2006-03-05</md:date>
    </entities:Entity>
</rdf:RDF>
```

Figure 17. Alternative encoding of a simple data entity

The essence of using multiple inheritance in the persistent data storage layer is (i) that it gives us maximal flexibility in terms of evolution on either side, and (ii) that it inherently accounts for the fact that primary data should have nothing in common with its corresponding metadata. Note at this point that also the aspect of how to deal with metadata in regard to the definition of ontologies is solved. While one or multiple ontologies can define different primary classes, yet another and totally separate ontology can define the structure for the metadata.

5.4.2. Partial or full support for annotation

The OWL-S API supports annotation in a generic way. It assumes that an OWL construct (i.e. a generic data entity as depicted in figure 17) may define arbitrary annotation attributes. It does not make any assumptions as to how such attributes are defined in terms of ontologies and their class hierarchies. Hence, the API does not distinguish between an attribute describing primary data and an attribute that describes metadata as far as the transparent data representation layer is concerned.

In the layer above where data entities are bound to corresponding Java objects the previous flexible approach is significantly restricted. An attribute representing annotation data must be assigned a direct value such as a date, some number, or some arbitrary character string. References to other data entities are no longer supported. As long as we talk about typical annotation information such as the name of an author, a creation date, and so on, this limitation does not border much. Things though change if we are about to account for any type of metadata and treat annotation information as some specific subset thereof.

5.4.3. A work around or a reasonable extension

Let us first look at how the API treats annotation attributes in its corresponding Java objects and secondly look at possible solutions to avoid the built-in restriction. Figure 18 depicts the OWLIndividual class which acts as the base class for generic data entities. The class defines two sets of methods: one to manage regular attributes and the other to manage annotation data. Note at this point that in the domain of OWL, attributes are called properties[14]. While in case of managing regular attributes, a method expects either an object of type OWLDataValue or OWLIndividual, in case of annotation attributes support is only built in for objects of the former type. It becomes clear that the announced restriction exactly is due to this small difference. An object of type OWLDataValue represents a direct value such as a number, an arbitrary character string or a well-formatted date. Annotations are forced to be defined by such a construct but cannot refer to other entities that respectively would be represented by an object of type OWLIndividual.

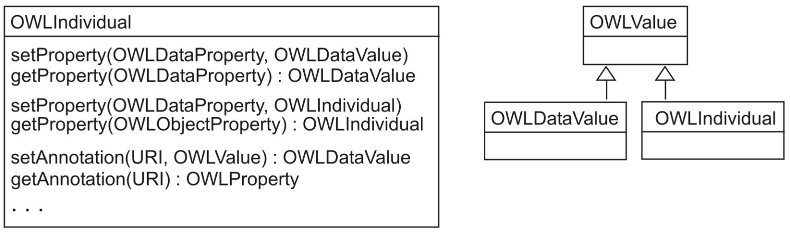

OWLIndividual
setProperty(OWLDataProperty, OWLDataValue)
getProperty(OWLDataProperty) : OWLDataValue
setProperty(OWLDataProperty, OWLIndividual)
getProperty(OWLObjectProperty) : OWLIndividual
setAnnotation(URI, OWLValue) : OWLDataValue
getAnnotation(URI) : OWLProperty
. . .
OWLValue
OWLDataValue
OWLIndividual

Figure 18. The binding base class of the OWL-S API

[14] For the sake of consistency, throughout this thesis, we refer to properties of the OWL domain as attributes.

There are two obvious solutions to avoid the built-in restriction. We either can adapt the OWLIndividual class or can use the methods destined for regular attributes as well for the management of metadata. Almost obvious, the former solution is clearly favored over the latter solution. Nevertheless, the major arguments are given in the following list.

- Because the API already supports the management of annotation information to some degree, it seams reasonable to extend it in order to account for the discussed additional requirement.
- Using methods for contradictionary intentions always is a poor design decision. As a result, we should not use methods that are destined for the management of primary data for the handling of metadata as well.
- Adapting the OWLIndividual class is not an endless burden but a rather trivial task. The set of methods for the management of annotation information only needs to be extended by one additional method that in contradiction to the existing "getAnnotation" method needs to expect an object of type OWLIndividual instead of OWLDataValue.

5.4.4. Multiple views of a data entity

We argued that the notion of multiple inheritance reveals a promising approach in order to decouple primary data from its corresponding metadata. However, in contradiction to OWL, Java does not support multiple inheritance in terms of classes [39]. As a result, the concept foresees the notion of multiple object views. Analogous to multiple inheritance in the data representation layer discussed in 5.3.1, a Java object is to support two distinct views. One view accounts for the primary data representation and the other view respectively accounts for the corresponding metadata. The following figure intends to clarify the illustrated concept.

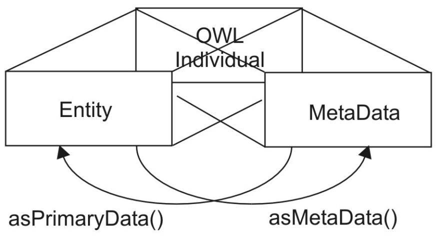


Figure 19. Multiple views hide the primary objects from the OWS-S API

An arbitrary client that works with data entities must not necessarily be aware of the fact that a data entity eventually has attached metadata. On the other hand, clients that are interested in metadata will call the method asMetaData and respectively will get the corresponding view that will be in charge of handling the entities corresponding metadata. Both views are represented by independent classes whose only requirement is that they relate to one another by their defined methods asPrimaryData and asMetaData. Since we virtually deal with the same constellation as in the data representation layer the corresponding design characteristics do not change either[15].

Note additionally that due to the fact that the relation between the two views is bidirectional, we can also think of a scenario where a client solely is focused on metadata. In this case, the client namely does not need to be aware of the fact that the MetaData class it is dealing with eventually relates to some specific entity holding the corresponding primary data. Nevertheless, as soon as this client happens to share entities with a client B, which in contradiction to the former, is interested in the primary data, the relation from the class MetaData towards the class Entity becomes crucial.

5.4.5. Getting the whole picture

The concept's underlying idea is to make use of multiple inheritance in order to gain the required flexibility and extensibility in terms of annotation capabilities. So far, we looked at the particular envisioned solutions for each of the relevant layers.

In the data representation layer, we noticed that OWL supports the notion of multiple inheritance out of the box. In terms of object binding, we addressed the integration of the OWL-S API. While the API's build-in support for annotation per se is promising, we nevertheless noticed that for the handling of metadata some small modifications are necessary but feasible. The top layer is represented by a typical data access layer. The concept showed that a business client should not directly rely onto the OWL-S API. The reason is that the API's defined business objects (data representing objects) do not support the concept's underlying idea of the separation of primary data and its corresponding metadata. With the introduction of the notion of multiple views, a promising solution was found in this aspect.

Finally, the following figure illustrates how the elaborated solutions fit together in the broader perspective. On the left hand side, the architecture for the generic approach is presented. Note that Entity in this respect stands for any imaginable type of data entity. On the right hand side, the architecture for the specific case of an NMRExperiment is described.

[15] For the discussion of the concrete design characteristics, please refer to chapter 5.4.1.

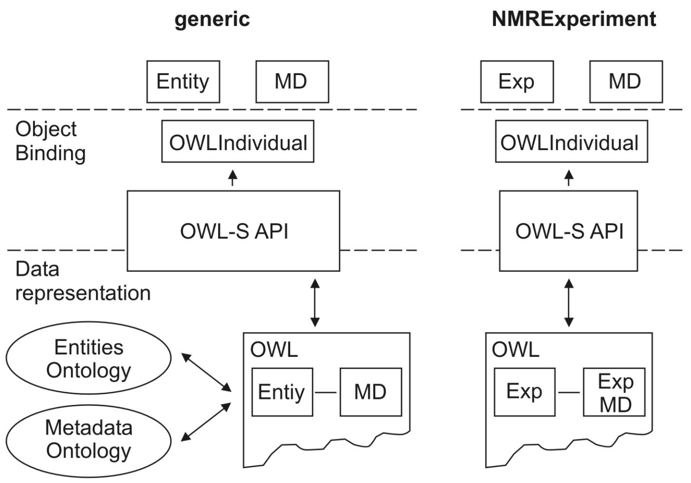


Figure 20 Using multiple inheritance to assign metadata to genetic data entities

5.5. Authentication and Authorization

In this section, we look at the security aspects as far as authentication and authorization is concerned. Being aware of the prescribed requirements outlined in the previous chapter, we now face the job of finding suitable concepts that are capable of meeting these requirements. First, we look at authentication and authorization in the specific context of NExT and thereby set the focus on different imaginable trust relations. Based on this acquired knowledge we then present four distinct login procedures that collectively claim to be able to cope with every revealed trust relation from the first discussion. The third part presents a mechanism that allows the system to manage connections between a client and multiple data repositories in a flexible and transparent fashion. Last but not least, we discuss how connections can be secured. Two fundamental concepts are feasible but incorporate very different characteristics. We will find out which of them will best suit the prescribed requirements and also can feasibly be integrated into the overall system architecture.

5.5.1. Authentication and authorization in the context of NExT

The requirements specify that NExT needs to handle data in user's private working areas and distributable data repositories. The former represents a transient data container, which is used in the sense of a desktop. Data elements that are used on a permanent basis when working on a project are temporarily stored therein. The latter represents a persistent data container that can be shared among multiple users. While a user may run a private data repository on her private machine, most other repositories though serve as collaborative platforms and run on specific dedicated server machines distributed over a LAN or WAN. Here is the point where the discussion about login concepts starts to become relevant.

Depending on the type of trust, which is mutually offered between a client and a data repository, different authentication and authorization concepts are adequate and feasible. Figure 9 depicts the relevant types of trust in respect to the different networking scopes. We look at the three types of trust and in the next section will discuss what authentication procedures are adequate and feasible.

	Local machine	Single admin. domain	Multiple admin. domains
trusted client	☑	(☑)	☒
trusted authentication	☑	(☑)	☒
trusted service / trusted authorization	☑	(☑)	☒

Table 7. Trust relations between a client and a service

(i) A trusted client application is an application that can be trusted in terms of its behavior. Such is the case if the service can legitimately assume that the client application is accurately configured and authentic; thus, not modified in terms of its underlying source code. (ii) When a data repository service trusts the claimed identities of a connecting client, we talk about trusted authentication. Note at this point, that from the system point of view, a physical user (an NMR experimenter) can have several different identities. One identity uniquely identifies the representing physical user; additional and optional identities represent affiliations to defined user groups. (iii) Analogous to a trusted client, a trusted service can legitimately be assumed to behave in the expected sense and to represent a non-modified system component.

If the client application connects to a data repository service located on the same machine, the communicating components trust in all three aspects. The explanation therefore is rather trivial. Because the two communicating components run on the same machine, they both inherently trust the same authority, which in this situation is the owner of the machine. If such a machine is used by multiple users, the common authority shifts from a regular user to the corresponding machine administrator.

If the client application connects to a data repository service that is located on a dedicated service machine but still resides in the same administrative domain, trust in all three aspects is subject to specific characteristics of the system environment.

Clients and services from different administrative domains principally cannot trust one another. A service cannot trust the client and vice versa, a client cannot trust the service as long as their authorities are not the same or do not bilaterally trust one another in the first place.

5.5.2. Four login procedures

The previous discussion reveals that NExT obviously needs to support different authentication procedures in order to account for the different revealed trust constellations. During a profound analysis of NExT's context, four distinct login procedures were identified. The first procedure accounts for the situation where a communicating client and a data repository do fully trust one another. A verification of the client's claimed identities in this case is needless (figure 21a). The second procedure considers the typical situation where the client trusts in the service but vice versa, the service does not trust in the client. As a fact, the service does not accept the counterpart's claimed identities[16] until these are approved by an appropriate authentication process (figure 21b). The third procedure is a modification of the second one. Instead of managing a list of authorized users and performing the necessary authentication, the repository service trusts the official authentication service from the client's administrative domain and delegates both user management and authentication to it. If you are familiar with the generic principles of token-based authentication mechanisms, you see the parallels that are pursued in this context. Interested readers are referred to appendix B.1 where token-based authentication in respect to Kerberos is discussed. The procedure is especially suitable if the number of users from the remote administrative domain is high and a corresponding user management would be expensive (figure 21c). Last but not least, the fourth procedure introduces the use of a third party digital certificate infrastructure. It relieves the system from handling any sort of authentication procedure by itself. Details about the functioning of a generic digital certificate infrastructure can be found in appendix B.2. The basic concept of the four login procedures is depicted in the following figure.

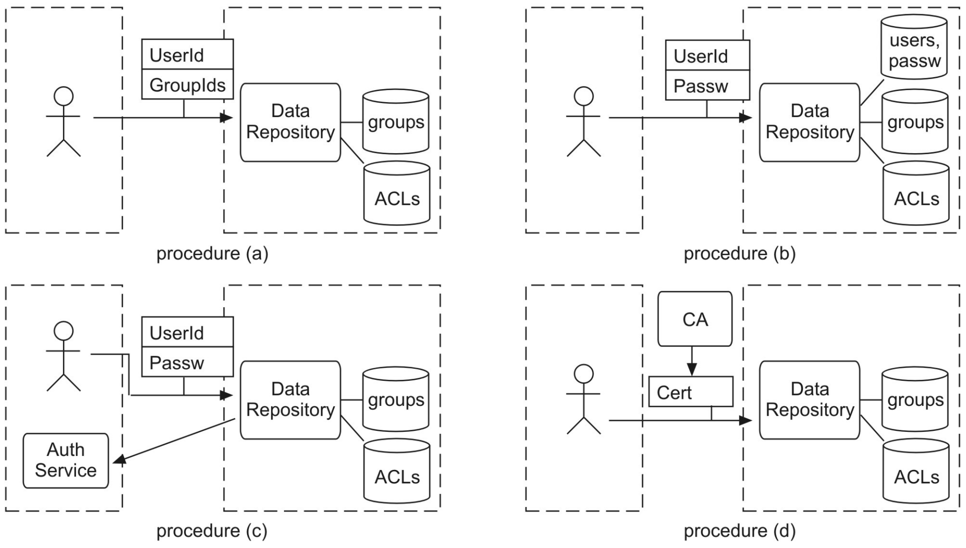


Figure 21. Different authentication concepts

5.5.3. Connections management

As outlined in the motivation and resumed in the requirements chapter, it is very likely that NMR researchers will work with remote data on a regular basis. Because such data is stored in data repositories that can only be accessed by authorized users, a NMR researcher has to

[16] A client is allowed to represent multiple identities. A system usually assumes that a client has one unique and mandatory identity and optional identities in terms of user group affiliations.

authenticate herself as frequently as remote data is requested. By rights, the requirements therefore prescribe that login procedures at data repositories be performed automatically and foremost transparent.

The concept that is chosen to meet this particular requirement is depicted in figure 22 and explained in the following.

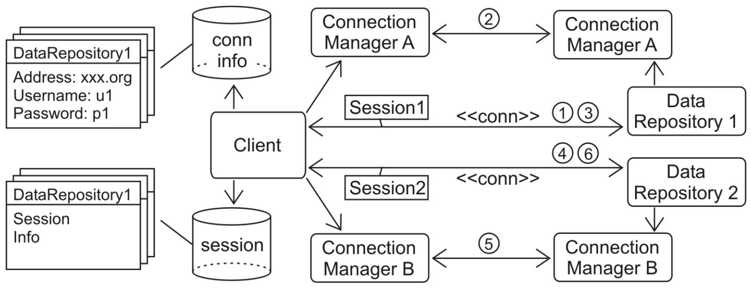


Figure 22. Transparent login at remote data repository

From the discussion about the distribution of data in chapter 5.3.1, we know that the client keeps a list of registered data repositories. For each such repository, the list holds the unique identifying name and the information that is used in order to successfully login at the remote location. For example, a record that describes a data repository that is restricted to authorized users, at the minimum, holds the address of the remote service, the username and the corresponding password. In the figure, this list is represented by the database named "conn info".

The concept allows a client to hold connections to multiple different data repositories simultaneously. In the figure, established connections are depicted by bidirectional arrowed lines between the client and the two data repositories named with the stereotype[17] *conn*. Note that the notion of a connection in this context does not refer to the low-level connectivity but refers to the fact that two parties have successfully established a connection context. The technical aspects as how and by what concrete transport protocol a low-level connection is established, is not discussed at this point. In the figure, a connection context is captured by the notion of a session. The database called "session" illustrates the fact that a client at all times keeps track of its current open connections. A session record has a unique identifier and optionally contains further implementation specific attributes. Note that the inside structure of a session is subject to a concrete implementation.

What yet remains to be discussed is the function of the so-called connection managers (CM) and the procedure by how a connection actually is established. As you already may have guessed, these two aspects go hand in hand. The concept accounts for the fact that repositories be free to implement any standard or individual type of authentication procedures. Different authentication procedures are implemented by different connection managers. A repository furthermore does not itself perform the authentication but delegates it to a connection manager. In this sense, a data repository keeps as many different connection managers as it wants to offer different

[17] A stereotype is a distinct UML construct. Further information can be found in [42].

authentication procedures. When a client contacts a repository, it does not know in advance what concrete authentication procedure it will have to engage. Depending on the user and its originating domain, the repository service answers the requested authentication procedure and the port at which such is available. The process by which the service infers the requested procedure does not have to be deterministic. The client checks whether it has support for the requested authentication procedure and if yes creates a corresponding connection manager. At this point, a pair of compatible connection managers has been set up. One resides at the repository and the other resides at the client. The client's connection manager now takes the initiative and engages into the authentication procedure with the announced remote authentication manager. Upon successful authentication, the connection manager at the repository creates and exchanges a connection context that is presented in the form of a previously introduced session. At this point, the communication between the two connection managers ends and the control is given back to the client and repository service. For the lifetime of the session, the client can now directly talk to the data repository and can initiate the appropriate requests. In the figure, the circled numbers reveal the sequence of actions.

Due to the fact that the client keeps a list of its registered data repositories with corresponding contact and authentication relevant data, the client is able to autonomously engage into the establishment of a connection. Given that the user has completely defined all her data repositories as part of her preference settings, the system will handle connections fully transparent. Note that the concept as is does not account for failure transparency. Due to time constraints, this aspect was intentionally ignored. Nevertheless, the concept's openness will eventually support almost any form of corresponding extension.

5.5.4. Secure connections

The previous two discussions were about connecting to remote data repositories. While the first discussion was about different authentication procedures, the latter focused on means how to establish a session based connection and how such can be managed over its lifetime. What yet remains to be discussed is how we can secure an established connection between a client and a remote data repository. In the case of NExT, we deal with client applications on the one side and different service applications on the other side. A common and standard security concept that can be applied to both the client as well as the service applications is thus highly appreciated. We present two fundamental security concepts. Afterwards we decide which of them we actually want to use for the envisioned system architecture.

A concept of hierarchical layers

The first concept is based on the notion of a hierarchy of layers. While the top layer deals with the primary functionalities, the second layer is in charge of all security aspects and subsequent layers care about the basic network communication and usually are classified by the layer defined by the ISO OSI (Open Systems Interconnection) [15] model. The layered structure is depicted in the following figure.

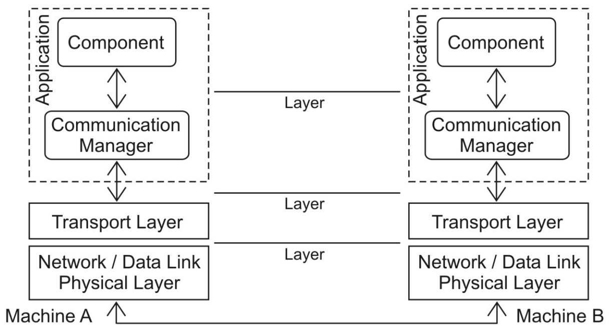


Figure 23. Hierarchical layer based concept

Client and service applications reside in the top layer. Abstractly they can be regarded as generic software components from which we only can infer that they need to communicate over the network and eventually engage into a bilateral session-based communications with other components from remote locations.

The second layer consists of a communication manager that acts as the coordinator of messages from local components to remote counterparts. Given a message, it transparently delivers it to the defined receiver at a remote location. Depending on its configuration, the type of message and the sender it optionally can encrypt the given message. Encryption thereby is subject to a preliminary establishment of a secure connection with the corresponding communication manager. The concept does not define how a secure connection is actually to be established. It assumes that a concrete implementation can make use of appropriate standard technologies of the third layer.

The third layer represents a bridge to the well-defined Transport Layer of the common ISO OSI model. An implementation hereby is encouraged to make use of standard low-level technologies. Later when we talk about the concrete system architecture, we will see that Java has some nice frameworks ready that right fit into this layer. The nice thing is that by using such a framework we do not even have to write one single line of code.

A concept of separated concerns

This second concept is based on the principle of separated concerns. Our goal is to find a way to encrypt messages that are sent over the network. In order to do that but not to conflict with additional concerns we must only care for the bare encryption procedure. Let us see how we can do that.

A generic software component that may represent a client or service application relies on a security manager (SM) that is able to establish a security context with a peer counterpart at a remote location. A security context thereby represents some kind of a shared key that can later be used by the component in order to encrypt/decrypt any required messages. Given a concrete established security context, the SM furthermore provides the functionality to encrypt and/or decrypt any given message. While figure 24 depicts the static setup, figure 25 describes the sequence of actions that need to take place in order for a component to securely send a message to some remote component.

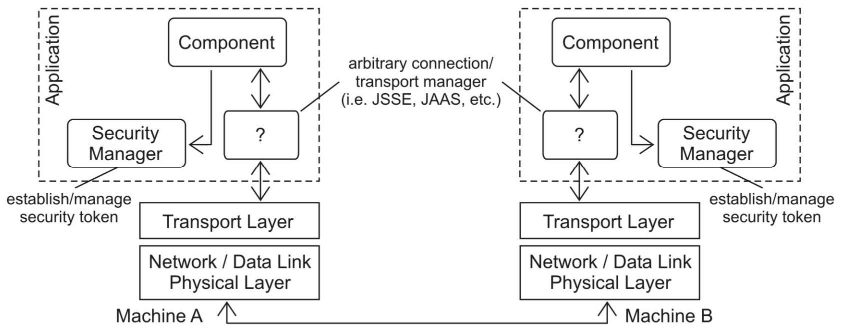

Application
Component
Security Manager
?
arbitrary connection/ transport manager (i.e. JSSE, JAAS, etc.)
establish/manage security token
Transport Layer
Network / Data Link Physical Layer
Machine A
Machine B

Figure 24. A concept of separated concerns

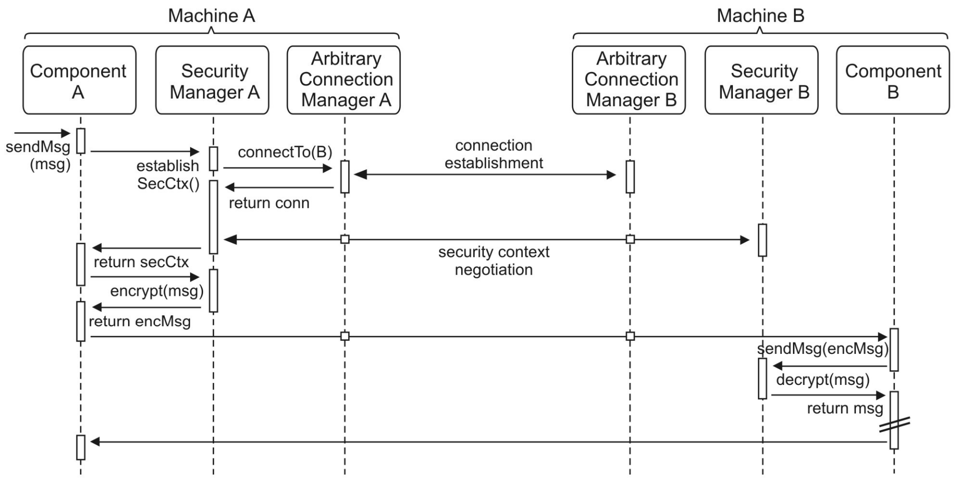


Figure 25. Sequence diagram: Security token based connection

Conclusion

Both presented solutions are generically feasible. Nevertheless, due to the architectural flexibility and the natural following of "Separation of Concerns" the second solution seams to represent the rational choice for NExT.

While solution 1 calls for a concrete hierarchy and defines part of the communication procedure it is less flexible than solutions 2 that neither assumes a concrete architectural structure nor defines how communication is to be enabled.

"Separation of Concern" is a promising design principle whose power has been widely proved both, in the industry and the academic arena in respect to the field of software engineering. In this sense solution, one is more likely to face severe design conflicts when concrete problem domains will be at stake than this is the case for the latter solution.

However, how nicely or how badly the second concept will actually fit into the overall system architecture will reveal the next chapter.

6. System Architecture

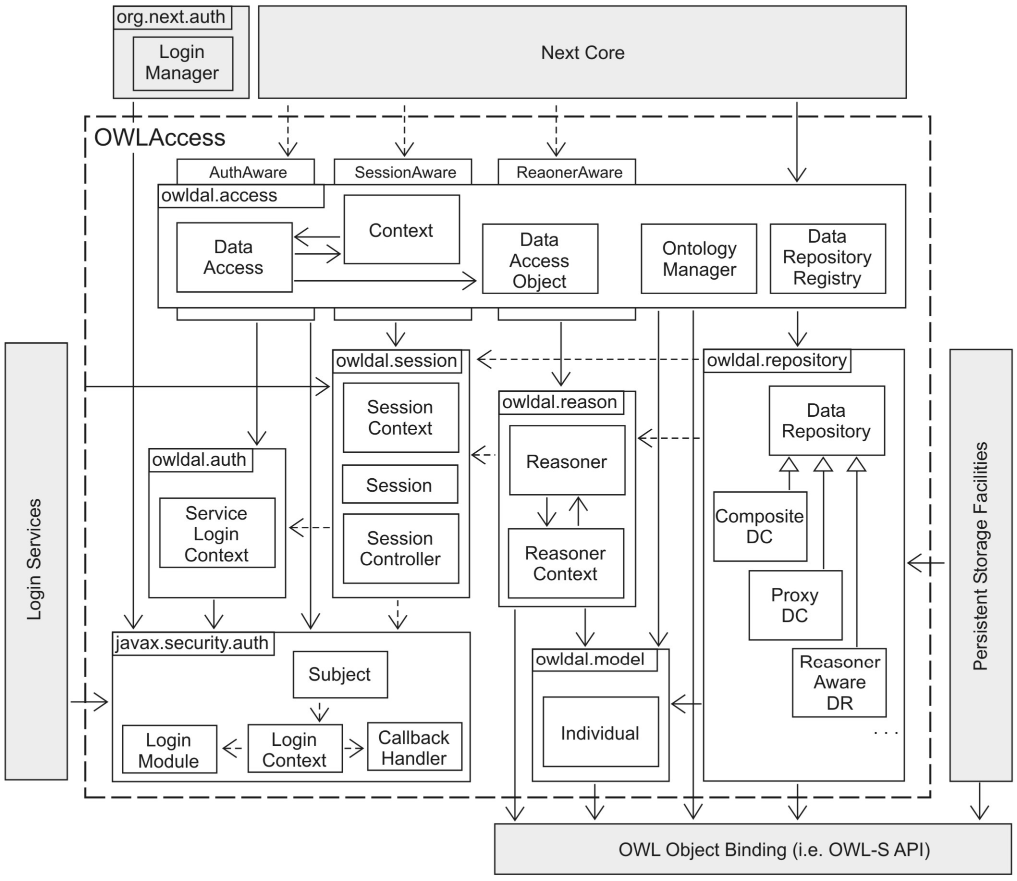


Figure 26. Overview of the OWLAccess system architecture

A coarse-grained overview of the system architecture (named OWLAccess) is presented in the above figure. The gray colored elements represent external components that either make use of the system architecture or serve for the latter as third party facilities. The figure depicts the main packages with their main components and shows their mandatory as well as optional dependencies. The arrangement of packages in term of their location is chosen by a distinct rule. They are laid out from the top to the bottom such that package dependencies only go from higher to equally or lower positioned packages. In terms of horizontal connections, an addition rule is applied. Except for dependencies with external components, dependencies only go from right to left. While on the right hand side and on the top packages tend to almost exclusively serve for the primary functionality, packages on the left and on the bottom respectively tend to serve for mainly secondary and/or supportive functionalities.

The first package from the top represents the official access point for external components that make use of the system's offered functionalities. Note that this package depicts three optional aspects. ReasonerAware refers to the optional support for generic reasoning capabilities. SessionAware represents the awareness for sessions when connections to remote locations are concerned. AuthAware finally refers to the optional support of authentication and authorization. The repository package is located on the right and – as the name reveals – serves for the definition and management of data repositories as introduced in the previous chapter. Next to it is the reasoning package, which is in charge of the reasoning aspects that profoundly were discussed in the previous chapter. Going further to the left, we come across the packages that (i) serve for the session management and (ii) enable the system for generic authentication and authorization capabilities. What now remains to be discussed are the two packages on bottom. Javax.security.auth refers to the standard authentication and authorization framework for Java and is usually referred to as JAAS[18]. The model package finally represents a container for components that serve as generic data entities.

In the following, we separately look at each of the broached aspects and explain their characteristics and detailed functioning. Figure 26 will guide us throughout this chapter and thereby will repeatedly tell us where in the overall context specific components need to be located.

6.1. Management of distributed data

The requirements in chapter 4 revealed that the majority of typical transparencies for distributed systems are crucial and therefore need adequate consideration. Additional but almost equally important requirements were discussed in regard to high flexibility and openness. In chapter 5, we looked at concepts of how data in regard to NExT's specific requirements may be best distributed and among other topics discussed reasonable storage topologies. In the following, we present the system's architectural design in terms of data distribution and persistent storage. The presenting design claims to reasonably cover all of the discussed aspects and in this sense does not make any remarkable curtailments.

6.1.1. Data Access

The main access point to the system and its functionality is provided by the DataAccess interface. DataAccess acts as the transparent façade to the underlying components and their interactions. As a result, a client such as the core from NExT initially creates an object of this type and subsequently uses that for all data centric operations.

As a façade, the DataAccess interface mainly provides delegating functionalities and performs highest-level coordination. In this sense, it acts as a typical singleton [40]. Let us have a look at the concrete offered functionalities. The interface defines four major methods that we briefly like to discuss in order to get the overall picture. The first method allows the creation of a Context

[18] JAAS (Java Authentication and Authorization Service) represents the equivalent of the PAM framework [49] for the Java arena. An overview of PAM and JAAS are presented in the appendixes B.3 and B.4 respectively.

that we subsequently shall refer to as the data access context (DAC). Whenever the façade is used by a client in order to execute some data centric operations a Context is always requested. We will look at the particular characteristics in the discussion that follows.

The second and third method are typical access providers. They respectively provide a RepositoryManager and an OntologyManager. The RespositoryManger keeps track of the currently registered repositories and allows adding or removing them at all times. We remember that this functionality represents the features of being able to descriptively define existing repositories as part of the user's personal settings. The OntologManager represents a controller of the currently available defined ontologies. Note that this second component heavily depends on the former one. The list of available ontologies directly depends on the available repositories.

The fourth method answers so-called data access objects or DAOs [43] as they usually are referred to. A DAO is an object that offers operations in correspondence to a particular data entity. We describe their functionalities as well as their specific characteristics in a separate discussion in chapter 6.1.3.

6.1.2. Data access context

The data access context (DAC) follows the notion of a workspace as introduced in the previous chapter. A DAC in this sense has two major functionalities. First, it acts as the data container for permanent working data. With the help of a DAO data entities from repositories are loaded into this container. Once a data entity resides in this symbolic container, it can be manipulated. That is, it can be modified and put into relation to other loaded data entities. At some point in time, the client decides to update the current work in the corresponding repositories. In the same way as data entities were loaded into the container in the first place, they as well are written back in the form of an update procedure.

Second, a client may look at the DAC as a black box [44]. The system requests a DAC for almost all communication with its client. It thereby uses it in order to store information that is needed to keep track of the client's actions but of which the latter must not be aware of. In contradiction to relational data structures, OWL based data structures tend to be significantly more complex in terms of their data entities' interdependences. As a fact, simple CRUD[19] operations may result in tedious dependency tracking algorithms in order to guarantee data integrity.

Apart from the discussed aspects above, the DAC has two more functionalities that may be of interest at this point. First, the DAC offers methods to define the default repository and default ontology. Both settings improve the work convenience and support further transparency aspects. Temporarily working on different – maybe even distributed – repositories thus is as simple as appropriately switching the state of the default repository. Analogously, the same is possible in terms of the available ontologies.

Second, the DAC allows for the definition of a so-called reasoning base. Reasoning capabilities are transparently provided upon all registered repositories and their contained ontologies if not stated otherwise. While this feature certainly is welcomed in the majority of

[19] CRUD is the abbreviation for the four distinct data management operations: Create, Read, Update and Delete.

situations, there still may be other situations where this same feature may be anticipated as rather painful. We remember the phase cycle of a typical NRM experiment project introduced in chapter 2.1. While in phase one and two the researcher usually queries her whole spectrum of data, in phase 4 she usually only wants to gather and manage data within the project's main dedicated repository. The reasoning set in this context allows for the concrete definition of repositories and ontologies respectively that will be used as the data basis for reasoning requests.

6.1.3. Data entities and their data access objects

An extensible and flexible data access layer must be compatible with a wide range of data entities. The less restrictive a system is in regard to postulating characteristics in terms of generic data entities the more flexible and convenient it is.

The system architecture expects a generic data entity as long as it complies with the Individual interface from the model package. The Individual defines a single method in order to give the entity a global unique identification. Apart from that restriction, the entity basically can have any arbitrary internal structure. It can have as many attributes as needed, and also can define any interdependencies among data entities of their own as well as foreign types.

The introduction of a data access object

The system handles the corresponding flexibility by introducing the notion of a so-called data access object or DAO. The underlying idea is that for each individual data entity there is a distinct dedicated object that is responsible for that corresponding CRUD operations. With other words, for every concrete type of data entity there also needs to exist a corresponding DAO that is in charge of handling its generic as well as individual data related operations.

The system architecture defines a generic DAO from which every specific data access object needs to inherit and which defines the generic interface upon which affecting system components may eventually rely. The design principle that underlies this construct is the template pattern [40]. While the system relies onto the defined methods of the generic DAO, their concrete implementations may significantly vary among concrete DAOs. Let us have an example in order to get this straight. One of the defined methods from the generic DAO is the method update that is used in order to store the modifications of an entity in its originating ontology and repository respectively. When the client requests a specific data entity to be updated, the system does not need to care about the concrete operations but solely can delegate this operation to the corresponding DAO. The DAO on the other hand is free to perform whatever adequate task in order to serve the client in its best way. The main concern of a concrete DAO is usually to make sure that data consistency is guaranteed [43].

The Relation between data entities and corresponding DAOs

In order for the system to be able to rely on the adequate DAO when being given a data entity of a specific type, a binding between the data entity types and their corresponding DAOs is required. The DataAcces provides a corresponding configuration capability. At startup or re-initialization it reads a textual binding definition that (i) defines the classes to be used to represent the particular data entities and (ii) the classes to be used to represent the corresponding

DAOs. As a result, an implementation is neither needed to be recompiled nor requested to be stopped when new or modified data entity types are available.

The following figure shows the specific data entities and corresponding DAOs in respect to NExT's process model. Note that all of the specified process model entities defined in [1] can be integrated without any modifications or adaptations.

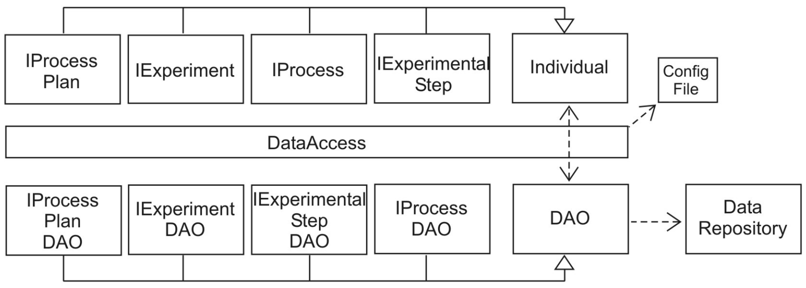


Figure 27. Data entities and its corresponding DAOs

6.1.4. Data Repositories

DataRepository represents the basic interface for a data repository as it initially was introduced in chapter 5.1 when we talked about the basic concepts for the management of distributed data. Because each repository can be understood as a self-running component that is independent from the system as far as its internal behavior and its offered functionalities are concerned, the system architecture followed the principle of decorators (see decorator pattern in [40]) that optionally can be assigned when they are needed. While one DataRepository may run as part of the NExT client and represent a private working space, another DataRepository may be implemented as a proxy that points to some remote repository. Needless to mention, both of them naturally offer different types of functionalities. If we take into consideration further aspects such as reasoning capabilities, authentication, authorization, session awareness and the like, it becomes even more obvious that the compendium of these functionalities should certainly not be requested by every single type of DataRepository.

The following figure illustrates the use of the decorator pattern in the mentioned context. There are eight distinct decorators that can be applied to a specific type of DataRepository independently from one another. The function of each of the illustrated decorators is discussed in the following.

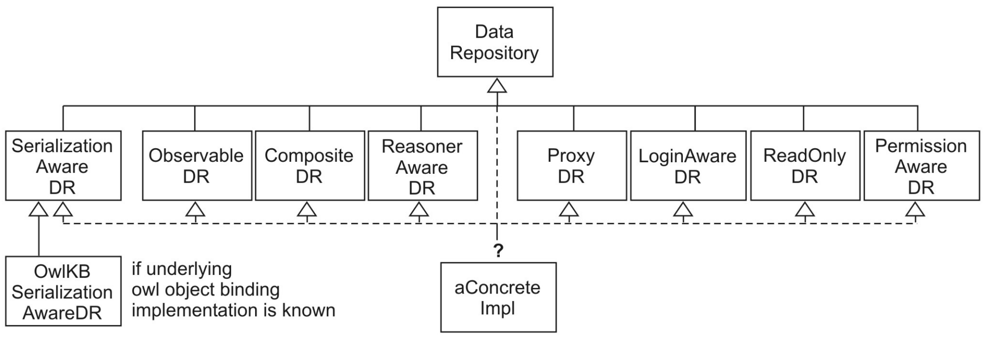


Figure 28. A data repository with different functionalities

We start discussing the most obvious decorators first and then go on to decorators whose functions are less apparent.

ReasonerAwareDR

The ReasonerAwareDR is a specification of the basic DataRepository and offers functionalities in terms of powerful reasoning mechanisms. The interface defines two additional methods. While the first method returns a list of available reasoners identified by their distinct names, the second method offers a handle to access one of them in order to start an arbitrary reasoning request. We will discuss reasoning in chapter 6.2.

ReadOnlyDR

The second decorator that we look at is the ReadOnlyDR. As its name already infers, this decorator defines a restriction to the basic DataRepository. While a generic DataRepository is assumed to offer all of the standard CRUD operations this particular decorator restricts the use of all of them but reading. A read-only DataRepository can be used for different purposes. The system architecture mainly thinks of the following two scenarios. First, an institution may have archives that are actively maintained by dedicated system administrator that work outside of the scope of OWLAccess. In order for them to let an arbitrary application access their archived data with highest security restrictions in terms of modifications, they may build a ReadOnlyDR. Not only will this interface provide for a convenient implementation due to the wide restriction in terms of expected offered functionalities, it also provides the required security aspects.

Second, data described in OWL/RDF originally was destined to be made accessible over a web server. Indeed this is also the main concept that the OWL-S API is based on. So far, we almost only focused on data that was subject to modifications on a regular basis and thus looked for means to be able to conveniently cope with the corresponding problems. Nevertheless, the assumption that all data needs to be regularly modified is not accurate. There indeed are situations where data once written need not be modified anymore. Here is one major use scenario. OWL based applications almost never come without underlying domain ontologies that define their distinct data model. A domain ontology thereby is understood to remain unchanged over relatively long periods of time. In fact, they only need to be changed if their underlying data model is adapted to new or changing business requirements. The choice to publish them by a web server under a well-known URL becomes a promising solution. Nowadays, existing OWL based domain ontologies almost all are accessible over the web browser. A system such as NExT in this case may build a read-only data repository in order to include its published domain ontologies.

PermissionAwareDR

The PermissionAwareDR is an extension to a regular DataRepository and offers authorization capabilities on either the scope of the whole repository, the scope of ontologies or the fine-grained scope of single data entities or Individual as they are referred in the system architecture. We realize that this decorator accounts for the prescribed authorization requirements. On the one hand, authorization needs to be open in terms of their specification granularity. On the other hand, the functionality is to be provided by an optional feature. We will come back to this topic when we separately talk about authentication and authorization in chapter 6.5.

ProxyDR

The ProxyDR is used in order to wrap another data repository whose accessing time and or initialization procedure may be delayed for arbitrary reasons. The system architecture mainly foresees its usage for wrapping data repositories from remote locations. In this sense, the proxy provides a surrogate or placeholder for the remote repository and is able to keep specific control mechanism transparent from the rest of the system components. The interested reader can find a detailed discussion of the proxy design pattern [40] in the adequate literature. Shortly hereafter, we will look at the possibilities that the use of such proxies can offer in terms of the discussed distribution topology in chapter 4.3.2.

LoginAwareDR

A DataRepository that requires some sort of a login procedure prior to be able to access the contained data, implements the LoginAwareDR interface. The interface provides a method that returns a corresponding SessionContext. The SessionContext respectively will be in charge of managing the eventually required authentication procedure and subsequently sets up a session that can be used to refer to a performed login procedure. The detailed purpose of the SessionContext as well as a profound discussion about the sequence of actions that take place in this specific situation is subject to the separate discussion about authentication and authorization in chapter 6.5.

CompositeDR

The CompositeDR follows the composite design pattern [40] and thus primarily allows composing tree-like structures of recursive related data repositories. This decorator accounts for the fact that several repositories from different levels of a company's or institution's organizational hierarchy may need to be treated uniformly when being accessed from some external point. The underlying concept to this particular design was discussed in chapter 5.1.1 where we looked at distribution topologies. By using a CompositeDR, the distribution of subordinate repositories can be kept transparent to its clients. The system architecture foresees its main usage when providing a unified view onto closely interrelated repositories from the business logic perspective. While particular repositories may be regarded identical in terms of their usage and their representing data, they however may not necessarily reflect identical implementations. Indeed, they in the majority of situations may (i) offer totally different functionalities in terms of providing decorators and (ii) may be located in geographically distributed areas. The resulting diversity of individual repositories is nicely hidden by a CompositeDR.

ObservableDR

The ObservableDR represents a DataRepository that allows notifying its clients about its internal state changes. The interface closely leans onto the generic Java Observable from the package java.util [39]. However, in contradiction to the latter, the ObservableDR only is defined by its offered functionality (thus its interface) but not by a concrete class that as well determines part of the internal mechanism. Nevertheless, due to fact that the concept from the Java reference implementation is identical to this decorator's followed underlying concept, a concrete implementation still may decide to utilize the Java ready-to-use construct. In this case, the implementation can conveniently account for the offered functionality of the ObservableDR but does not have to deal with the particular concerns and insights of the observer pattern [40] as such.

The system architecture foresees two main situations where the ObservableDR interface can be used efficiently. First, a data repository is concerned with the management of its containing data entities in respect to their persistent storage. A data repository can decide to implement its own persistence mechanism but may as well want to rely onto third party services such as triple stores[20] or OWL object binding frameworks. While the first choice may be appropriate for first version implementation purposes and their testing phases, the latter choice obviously is the preferred go when a flexible and open system is to be built. By using an ObservableDR when connecting a DataRepository with a corresponding persistent storage facility the resulting coupling becomes least stressed and offers means for flexible interchangeability.

Second, a CompositeDR as it was introduced above manages a dynamic set of subordinate repositories. When the set of referenced subordinate repositories will change, clients to the CompositeDR may need to be informed in advance in order for them to be able to react adequately in terms of required preliminary actions. For example, most clients need to be notified when available repositories are removed. If they are not, they may not be able to guarantee data integrity in the first place.

SerializationDR

Last but not least, the SerializationDR supports an important additional functionality when it comes to the persistent data storage by external (third party) components. As data at a repository is defined in OWL/RDF, data interchange between a DataRepository and an arbitrary persistent

[20] A triple store [54] is a persistent data storage facility for data defined in Semantic Web languages such as RDF or OWL.

storage facility is best achieved with OWL/RDF serializations. When data is to be read from the persistent storage into the repository the latter requests a data stream which it can use to populate its internal data container. Vice versa, when the data repository needs to persistently store its containing data, it uses a stream into the opposite direction. The SerializationDR defines two methods that provide support for the described mechanism.

In the situation where the system is allowed to refer to a particular underlying framework data interchange between a DataRepository and a persistent storage facility must not necessarily be performed by a neutral serialization mechanism. Instead, the interchange may take place on a higher and more efficient data representation level. The OwlKBSerializationDR represents such a solution when the usage of the OWL-S API can be assumed.

6.1.5. Distribution mechanism for data repositories

The transparent and flexible distribution of data repositories is based on distinct collaboration patterns in terms of data repositories and their concrete applied decorators. In the following, we first look at the abstraction that the system architecture provides onto the registered data repositories of a client. Thereby we foremost will realize the main duty of the DataRepositoryManager. Second, the discussion focuses onto the abstraction that a repository proxy uses in order to handle the variety of different kinds or repositories it may need to target. In the last part, we finally get the chance to look at how the system is able to build flexible and recursive structures of local and/or remote data repositories.

The abstract view onto registered repositories

Figure 29 illustrates a typical system topology setup. The DataRepositoryManager, which is mainly used by the DataAccess, treats its maintained set of repositories as generic DataRepository objects regardless of their individual supported functionalities. As a result, any client that uses the DataRepositoryManager in order to refer to an appropriate repository fully abstracts from the distribution as well as the variety of different repository representations. Furthermore, because a generic DataRepository can optionally implement any of the eight above discussed decorators the system also allows for specific transparencies in regard to each decorator's additional functionality. In the figure, all three repositories implement the decorators ProxyDR, LoginAwareDR and ReasonerAwareDR. The first decorator is needed since the three repositories act as proxies towards remote services. The LoginAwareDR interface is required because we assume that the remote repositories permit access only to authorized clients. Finally, the ReasonerAwareDR makes sense as otherwise, a client would not be able to infer about the content of a particular repository. Note that the depicted scenario nicely reveals how a combination of decorators can be used to build powerful system constellations that in the same time are able to follow all major design principles [32].

The abstract view from a proxy towards its connected target repository

In the same fashion as the DataRepositoryManager abstracts from the actual functionality of its assigned repositories, a proxy analogously abstracts from the remote repository's functionality by solely relying on the interface of a SessionAwareDR. The session is used in order for the remote repository to be able to keep track of the connections to its current clients. However, if a remote repository does not rely on preliminary login and eventual authentication procedures, it may as well provide its functionalities through the basic interface of a DataRepository.

Tree-like repository hierarchies

The figure finally also depicts how the initially claimed capability for tree-like hierarchies of independent repositories can be achieved. The gray colored remote data repository not only acts in the sense of a SessionAwareDR but additionally also acts as a CompositeDR. As a fact, it transparently refers to a set of subordinate repositories that by themselves may again be implemented as ProxyDRs and/or CompositeDRs. We can imagine how the illustrated recursion could almost endlessly be followed and how an almost unlimited structure of interdependent repositories may arise. Note also, that the recursion is not restricted to local repositories but as well may recursively span over geographically distributed locations.

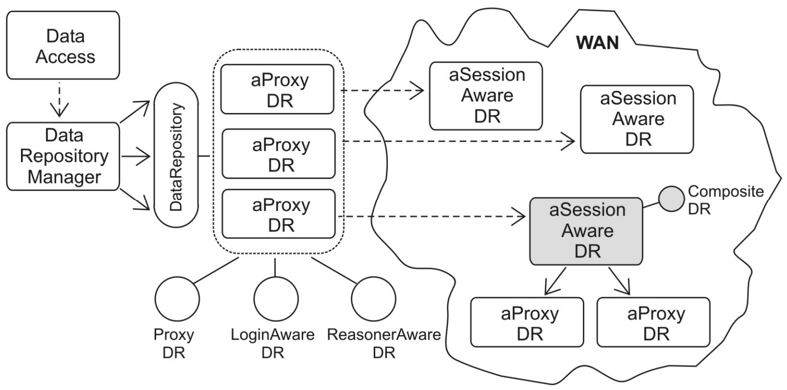


Figure 29. Distribution of data repositories

We may not have realized the real power of each single decorator when it first was introduced. The main reason therefore is that the majority of decorators indeed do not provide any substantial gains in terms of their appliance to basic DataRepositories but only do so when they are wisely combined. The above usage scenario hopefully could give the required understanding in order for us to grasp the real power that this architecture exhibits.

In the next section (chapter 6.2), we will look at the reasoning functionality that yet represents another challenge for the system. We then will realize that the underlying principle does not significantly differ from the one that we have discussed in here. In addition, we will note that reasoning indeed becomes one of the most crucial functionalities altogether.

6.1.6. How all fits together

The entire process of a request such as a data-centric operation upon a particular data entity within a particular ontology and repository is described by the following sequence diagram. The intention thereby is to show how the components and design principles introduced throughout this subchapter about the management of distributed data fits together to one coherent architectural picture.

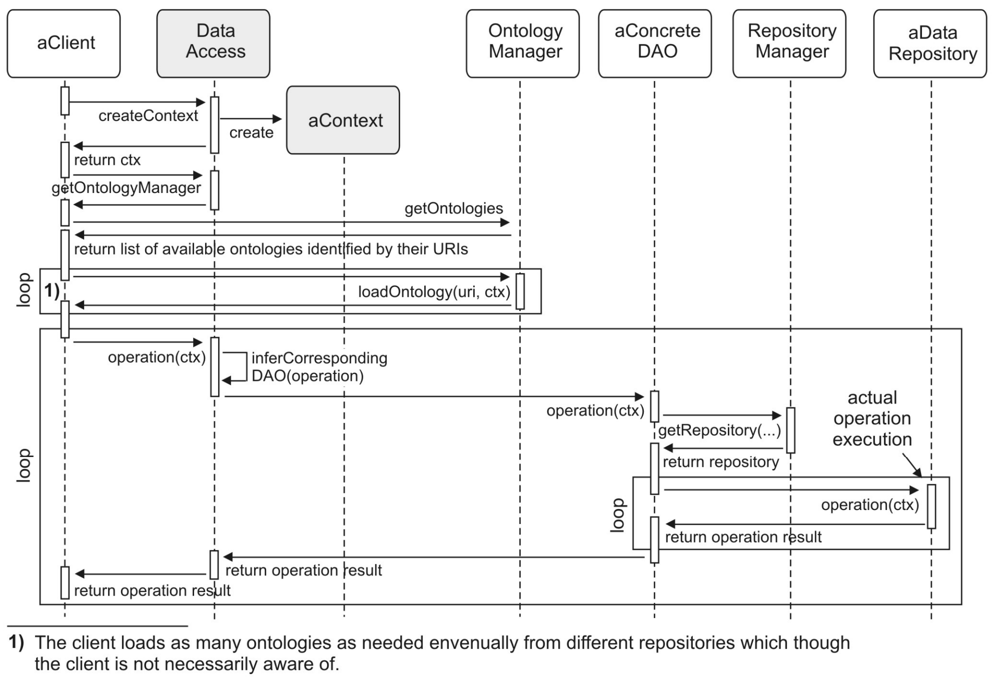


Figure 30. Sequence diagram: From the client to the data repository

The two major components that an external client such as the NExT core is concerned with are the DataAccess and the corresponding Context (Data Access Context, DAC). The sequence diagram starts with the creation of a DAC. The Client thus addresses the DataAccess in order to provide it with a new Context that subsequently will be used in order to communicate with the system. Once in possession of a personal DAC the Client starts figuring out what ontologies it has available. It again addresses the DataAccess that passes it the OntologyManager, which is in charge of all ontology based operations. By asking the latter about the currently available ontologies, it answers a list or their unique names. Thereby the list contains the compendium of ontologies from the currently available repositories. Note besides that we assume that the available repositories were preliminarily registered as part of some regular initialization task that is not covered by the diagram. Aware of all available ontologies, the Client now is able to load any number of them into the Context. Obviously, it will not just go ahead and load of them but distinctly may want to choose a small number that covers the data upon which it subsequently needs to work on. The diagram depicts this process as a loop of loading requests destined to the OntologyManager. Once the data retrieval process has completed the Client is able to start its actual work. We assume that the Client needs to modify and/or create some data entities and thereafter needs to update them in the corresponding data repositories. The sequence diagram abstracts from specific data operations such as a modification, a removal, etc. and depicts the update process as a repeated loop of generic data centric operations. If we look at how the Client is involved into the update process, we realize that its sole duty is to make the decision what actions need to be performed upon its holding Context. Everything else is transparently managed by the system. When the DataAccess receives an operation request, it infers the type of data entity that is involved and grasps the corresponding DAO. The operation request is then forwarded to the latter. As we know from previous discussions, the DAO has the knowledge of how to perform the operation in terms of their fine-grained steps. Hence, what from the client's perspective is treated atomic, not necessarily needs to be performed alike in lower level contexts. Once the operation is completely executed by the DAO the action flow together with the corresponding operation result is turned back to the DataAccess where it is further returned to the original requestor, the Client.

For the sake of simplicity, the scenario skipped the reasoning aspect. Nevertheless, we can easily think of a slightly modified scenario that hence as well accounts for the previously profound discussed reasoning capability. Instead of loading particular ontologies from the OntologyManager, the Client could as well request a ReasonerContext and subsequently execute queries in order to find out what ontologies and/or data entities are needed. In this situation, the Client uses the reasoning capability in order to efficiently infer the relevant set of data it needs to work on.

6.1.7. Conclusion

The previous discussion was held under the topic of how to deal with the management of distributed data. We started at the top of the architectural structure and looked at the way an external client communicates with the presented system. We learned that there exists a façade-like component that is in charge of handing out the relevant components. Among others, the client deals with a RepositoryManager and an OntologyManager. While the former is used to infer about the registered, thus available, repositories, the latter is in charge of providing information concerning available ontologies and their particular content. We further have learned that the system makes use of he widely known principle of so-called DAOs (data access objects). For every individual type of data entity there is a corresponding DAO which is in charge of handling the particular data-centric operations such as create, update, delete and the like. The distribution of data across the network is managed by data repositories. We realized that because a particular repository is able to provide an arbitrary set of additional functionalities and does so in a fairly transparent way the system is able to cope with each prescribed requirement in terms of flexibility, openness and scalability. We concluded the discussion with an overall usage scenario that was destined to reveal how all the mentioned aspects and components fit into one coherent structure. Apart from the additional aspects that follow, the system architecture seems to represent a robust and coherent design. The actual usability though can only be proven by an appropriate implementation. We yet stay tuned for the chapter about the implementation that eventually will reveal as to what extent the envisioned principles will hold true.

6.2. Reasoning over distributed data

The querying functionality is considered an optional feature. An implementation of this system architecture is thus allowed to fully ignore reasoning throughout all the potentially affected components. In the following, we show how the system architecture is able to manage the separation between the primary functionality and reasoning as some sort of additional feature, thus secondary functionality.

First, the main components in charge of enabling the system for reasoning capabilities are introduced. Because reasoning is treated as an optional feature and is strongly separated from the primary functionality, modeling a sole reasoner component almost obviously is not feasible. We will realize how the notion of a reasoner context will enable the envisioned goals. Second, a separate discussion about the characteristics of the latter is given. The reasoning topic finally is concluded with a profound usage scenario that eventually reveals how the elaborated components fit together and how reasoning as such is applied onto the overall system architecture.

6.2.1. Main components

The architecture defines four main components that are responsible to enable the required reasoning capabilities. The following figure depicts these components and reveals their interrelations. A corresponding explanation is given hereafter.

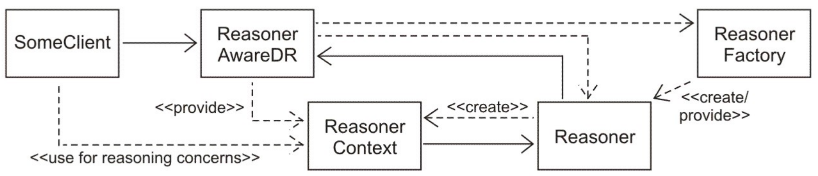


Figure 31. The main components in charge of the reasoning functionality

The ReasonerAwareDR is a generic DataRepository with additional functionalities to provide reasoning capabilities. The additional method that a ReasonerAwareDR in comparison to its generic repository provides is a handle to its corresponding ReasonerContext. The ReasonerContext serves as the actual enabler for reasoning processes. Its existence furthermore allows decoupling the reasoning functionality from the primary concerns of a DataRepository. A client such as the DataAccess, which acts as the façade to the overall system, does not directly deal with a particular data repository but instead deals with the corresponding ReasonerContext when it comes to reasoning tasks. The ReasonerFactory is used by the data repository in order to retrieve a particular Reasoner that is determined by its unique identifying name. The system architecture in this sense assumes that multiple different third party reasoners may coexist and that the client, prior to the execution of a query, chooses a particular implementation from a provided list of available concrete reasoners. Note additionally that only by the notion of a list of available reasoners, the distribution of data repositories onto multiple machines within various administrative domains becomes feasible. The system must not assume that all connected data repositories provide the same reasoner implementation.

Once the ReasonerAwareDR has received a concrete Reasoner, it offers the Reasoner a handle onto its internal data representation in order for the Reasoner to be able to access the corresponding data. When a client needs to reason upon the repository's data, it addresses the ReasonerAwareDR in order to ask for a corresponding ReasonerContext that can be regarded as a session-like connection to the Reasoner. In this situation, the ReasonerAwareDR transparently forwards the request to the Reasoner. The latter is able to create the requested ReasonerContext and immediately returns it to the ReasonerAwareDR from which that is finally returned to the client. We note that a client never directly gets into contact with the Reasoner. The Reasoner is kept transparent behind the ReasonerAwareDR and the ReasonerContext.

6.2.2. The notion of a reasoner context

We may reasonably wonder as to why the system introduces the notion of a ReasonerContext but does not let the Reasoner itself be responsible for the functionality that is offered by the former. After all, it seems as if the sole ReasonerContext's duty is to act as an intermediate between a client and the concrete reasoner implementation that is represented by the Reasoner component. The decision for the illustrated design is not arbitrary but is grounded upon multiple aspects. The introduction of the ReasonerContext namely has following advantages. It (i) inherently allows using established sessions between a client and a data repository, (ii) it enables the Reasoner to eventually run query processes asynchronously which may provide significant performance and stability gains, (iii) it allows the Reasoner to handle query requests in a queuing system, and (iv) it makes it possible to eventually manage requests by some sort of a priority mechanism. In the following, we provide a brief explanation for each of the broached aspects.

i In one of the previous discussions, we looked at the functioning of a proxy that connects to a remote data repository. We mentioned that in most situations the remote repository will only allow access to authenticated clients. As a fact, an authentication procedure as discussed in chapter 5.5.2 will preliminarily be necessary. The result of a successful login is usually a bilaterally established session context that is used by both parties to refer to the initial authentication process. A ReasonerContext can be created on a per connection basis and thus be assigned to individual clients.

ii Running query requests in an asynchronous mode is adequate because (i) the corresponding reasoning process may require substantial processing power and (ii) the number of simultaneously connected clients is uncertain but may tend to be rather high than low. If reasoning runs asynchronously the client does not actively wait for the result to be returned but disconnects from the reasoner and either repeatedly asks the latter about the state of the process until that turns out to be completed or is notified by the latter after the process has ended. Once the client knows that the reasoning process has completed, it again connects to the reasoner and picks up the deposited result. Because the same reasoner simultaneously may be used by multiple clients, the communication between a client and the reasoner must be identified by some sort of a context. Otherwise, they cannot communicate in the described requested fashion. We realize that only by providing the notion of a ReasonerContext, asynchronous execution becomes possible in the first place. The choice as to whether or not reasoning is indeed to be executed asynchronously still is up the implementation of a particular reasoner.

iii As reasoning generally absorbs potential processing power, a reasoner – once put into a multi user system – needs to account for some sort of balancing mechanism in order to prevent from crashing. Queuing [11] thereby is an appropriate and feasible mechanism. The ReasonerContext can be used as the queuing element. In this sense, it is used in the same fashion as lightweight processes (threads) [45] are managed by common operating systems.

iv Priority management usually depends on the existence of a queuing mechanism. Its potential in regard to the management of reasoning requests is obvious. As soon as a service is shared among multiple actors, the notion of some priority rules in regard to the type of task or the requestor's identity becomes apparent. The reasoner in this case may want to schedule incoming query requests on the basis of various aspects in order to provide maximal efficient performance anticipation. Again, we might refer to the analogy of the way processes or threads are managed by common operating systems.

6.2.3. A usage scenario

Up to now the underlying mechanism and the main components of the system were introduced. The actual sequence of actions that take place when the client initiates a query request is best illustrated on a sample usage scenario. The following sequence diagram depicts the main action flow of a typical scenario starting at the client and proliferating over a couple of different intermediates down to a concrete ReasonerContext. While we discuss the interactions between the affected parties some of the already mentioned aspects will be repeated in order to deepening the overall understanding. Readers we substantial practice in reading a sequence diagram eventually may want to skip the following explanation and only may study the depicted provided diagram.

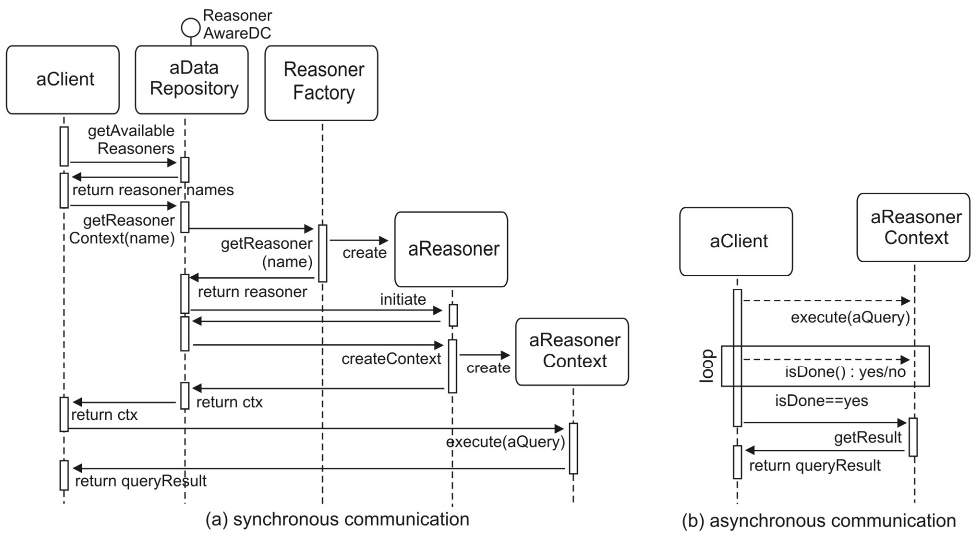


Figure 32. Reasoning on a data repository

The depicted scenario in the sequence diagram above starts with a request for a ReasonerContext by some arbitrary client. From the initial discussion in this subchapter, we know that the client is either the DataAccess that acts as the façade to the overall system or the DataRepositoryManager. Both are controlled by an external client such as the NExT core and are used as intermediaries to the transparent underlying functioning of OWLAccess. In this respect, we can think of the depicted client as an arbitrary external client although such generally would not contact a repository directly but would do so over certain indirections.

Note that in order for the Client to request a concrete ReasonerContext it must provide the unique name of the underlying reasoner implementation that the latter transparently can rely on. Prior to sending a query request, the Client therefore consults the available list of reasoners that the corresponding DataRepository provides.

When the DataRepository receives a request for a particular ReasonerContext the DataRepository contacts the ReasonerFactory in order to be provided the appropriate Reasoner. Hereafter the DataRepository calls upon the Reasoner to create a new ReasonerContext. Finally, the DataRepository returns the newly created ReasonerContext back to the initial requestor.

The Client at this point owns a ReasonerContext upon the data contained in the DataRepository. The actual query procedure depends on the protocol of the ReasonerContext. In case reasoning is implemented by a synchronous algorithm, the Client requests the execution of a query by the appropriate method and waits upon the result to be returned. The concrete action flow for this situation is illustrated in the continuation of the discussed sequence diagram. The case in which reasoning takes place in an asynchronous mode is depicted in the additional diagram to the right (figure 32b). While the process of announcing the query request is identical to the former method the subsequent action flow significantly differs. Once the Client has submitted its query request, it engages into an interrogation loop during which it repeatedly checks the state of the reasoning process at the ReasonerContext. When the ReasonerContext eventually answers that the process has finished, the Client exists the loop and in a final step picks up the query result. Note that because the query request is encapsulated into a context but is not directly sent to the actual reasoner, aspects such as the installment of a queuing and/or priority mechanism as discussed above are nicely hidden from the requestor. With other words, the architecture does not prescribe how reasoning in the end is to be performed but only defines the protocol that a reasoner implementation has to comply with. The defined interface thus is said to be complete and neutral[21].

6.3. Versioning

The system architecture strictly follows the versioning concept that was introduced in the previous chapter. The capability for versioning thereby is regarded as an optional feature and is fully decoupled from the primary functionality. In the following, we look at the main components of the architecture and discuss their interdependence as well as their integration into the overall system. Although the presented concept is extremely powerful, the corresponding architectural design is not. Indeed, due to the fact that the concept aimed at maximal decoupling, only very

[21] Completeness and neutrality are discussed in chapter 4.1.

few components will be affected by the introduction of this additional feature. The subchapter finally ends with an overall conclusion.

6.3.1. The main components

The overall system architecture depicted in figure 26 at the beginning of this chapter describes the model package with a single element, namely the Individual. Although the illustrated class diagram does not claim to provide a detailed view onto the system but rather intends to present an overall and thus simplified picture, it does an accurate job when presenting the model package. Indeed, the system principally does not rely onto any other interface than the Individual when referring to data entities. As a result, a versioning mechanism can be introduced in a fully transparent way as long as the mechanism is based on the defined Individual.

The following class diagram depicts the main components that are responsible to provide the system with the additional versioning capability. The system defines an optional interface named VersionAware that bases on the Individual and provides all the functionalities in order to enable the envisioned characteristics of the versioning concept presented in chapter 5.2. The major methods defined by the VersionAware interface are depicted in the class-like box to the right.

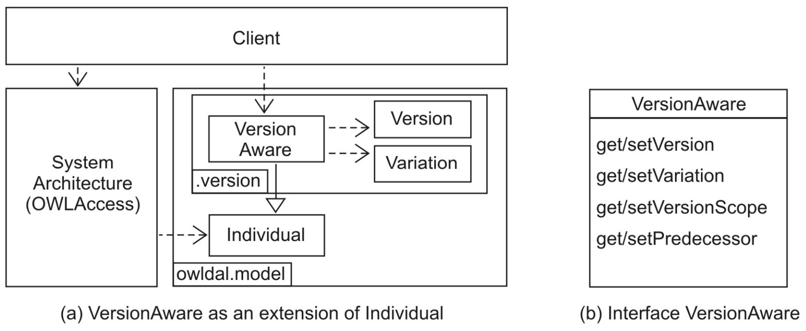


Figure 33. Version and Variation as subclasses of Individual

Any concrete type of data entity that implements the VersionAware interface becomes aware of its version as well as optional variation. In contradiction to the discussion in chapter 6.1.3 where we assumed that a specific data entity inherits from the basic Individual, an entity now obviously needs to inherit from the VersionAware instead. Nevertheless, since VersionAware only defines some fairly simple, descriptive functionality, its application to domain specific data entities is unproblematic.

The simplicity of this design is indeed astonishing. There is not even one other component that needs to be extended in order to provide the concept's envisioned functionality. Nevertheless, in case we want to provide some corresponding convenience functionalities we though might want to extent the DAO in order for it to be able to handle operations such as the creation of a new version or the retrieval of a specific version in a transparent fashion. The system architecture therefore defines an optional interface that can be implemented by the generic DAO in order to become version aware.

6.3.2. Conclusion

The architectural design that accounts for the required versioning capability is straightforward. The overall principle is based on the inheritance from the generic Individual. Because the entire system solely relies onto an Individual when referring to various data entities, the integration of the versioning concept becomes a fairly easy and natural task. Up to some extent, we can reasonably argue that the herein achieved simplicity is not only the result of the promising underlying concept but may – even more – be the result of the continuously forced focus on transparency, openness and scalability.

6.4. Annotation

Like in the case of versioning, the functionality for annotation is strictly decoupled from the overall system architecture and follows the annotation concept elaborated in the previous chapter. We remember that we extended the initial annotation concept in order to be able to also serve generic metadata concerns and that the underlying principle is based on the notion of multiple inheritance.

In the following, we present the corresponding architectural design. We first look at the main components and their interactions with the overall system architecture. In the second part, we provide an unbiased conclusion.

6.4.1. The main components

The main components that enable the system to manage generic metadata are described in figure 34. The notion of two separate views onto a virtual data entity is realized by two interfaces that both inherit from the generic Individual. One interface serves for the view onto the primary data representation and hence the second interface serves for the view onto the corresponding metadata. Because both inherit from the generic Individual, the overall system architecture does not have to care about their explicit functioning. Indeed, the system can treat either view as a regular data element and thus is principally not affected by the additional support for the management of generic metadata.

The illustration on the right hand side of figure 34 depicts the two interfaces with their defined methods. As both of them only define one distinct method, they can easily be applied to any particular domain specific data entity. Dublin Core Metadata Initiative [13], FOAF [46] or VCard [34] represent common used standards for describing data elements in terms of annotation information. The illustration reveals that either of them easily could be implemented by an extension of the generic Metadata interface. Note at this point that the three standards are only mentioned in order to reveal the architecture's flexibility. They of course could as well be replaced by any other standard or application specific data model.

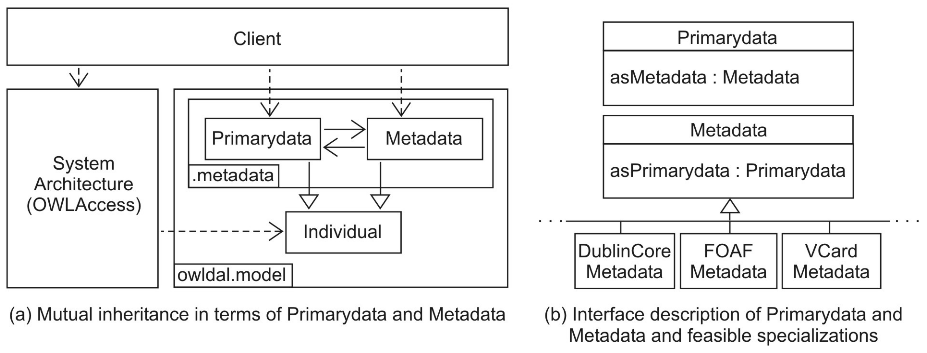

Client
System Architecture (OWLAccess)
Primarydata
Metadata
.metadata
Individual
owldal.model
(a) Mutual inheritance in terms of Primarydata and Metadata
Primarydata
asMetadata : Metadata
Metadata
asPrimarydata : Primarydata
DublinCore Metadata
FOAF Metadata
VCard Metadata
(b) Interface description of Primarydata and Metadata and feasible specializations

Figure 34. Primarydata and Metadata as extensions of the generic Individual

The metadata concept elaborated in chapter 5 goes one step further than the majority of currently known metadata standards. It namely does not make the common assumption that a metadata attribute be restricted to only represent a direct value such as an arbitrary character string, a date or some number value. Instead, the concept foresees that an attribute as well can represent a relation to another data entity. An obvious example is the typical assignment of the author's name and contact information. While it is common to define the author by its name and its address in the form of one simple character string, we as well could imagine providing the same information by a distinct relation to a corresponding author entity. The latter method avoids the creation of redundant information and thus is far more flexible when it comes to reasoning aspects. Furthermore, and in some situations even more important, it also eases the burden of general data maintenance [47].

Because both Primarydata and Metadata inherit from the generic Individual and the system architecture does not distinguish between either of them, the additional capability for relational metadata attributes is inherently enabled. Nevertheless, we need to be aware of the fact that allowing metadata elements to build relations among one another will eventually result in complex interdependencies that otherwise could not evolve. We furthermore notice that in this regard we eventually also may need to provide metadata specific DAOs in order to be able to manage the metadata level in the first place. As soon as metadata elements start to represent independent entities and thus are no longer bound to a distinct corresponding entity from the primary data level, they can only be managed if the system provides a corresponding DAO[22].

6.4.2. Conclusion

The definition of the two interrelated views, described by the annotation concept from the previous chapter, revealed a fairly easy job. Due to the fact that the entire system architecture refers to the basic Individual when referring to arbitrary data entities, the two views could directly be inherited from the latter and thus be kept simple.

[22] The relation between a data entity and its DAOs is discussed in chapter 6.1.3

Another discussion point was the handling of relational attributes in terms entities of type Metadata. Once more, because the system architecture solely relies onto the basic Individual interface and therefore does not distinguish between either view (Primarydata or Metadata) defining a concrete specialization of the Metadata interface can be done without limitations. In summary, we can reasonably say that the initial requirements could all be met satisfactorily.

6.5. Authentication and Authorization

The discussion about authentication and authorization from the last chapter revealed that a lot of different particular aspects need to be taken into consideration if we are about to design a system that not only guarantees authentication and authorization in a limited one-machine environment but as well does so in a widely distributed system. In the following, we illustrate how the system architecture introduced at the beginning of this chapter can be extended in order to feasibly account for these very features.

We start with an overview of the involved components and subsequently look at major aspects that emerged from the elaboration of the security concept from the previous chapter. Among others, we discuss how the system can account for the connections management by making use of a common session based architecture. We also look at how transparency in regard to various types of login procedures can be handled and how thereby the separation of identification, authentication and authorization comes in as a precious and promising concept. As always, we finish the subchapter with an overall conclusion.

6.5.1. The main components

The major components that enable the system to support authentication and authorization are depicted in figure 35. For the sake of comprehensibility, the figure uses distinct colors in order to highlight the three major aspects that are identification, authentication and authorization.

The yellow colored elements are responsible to manage generic identification concerns. As we already know from the discussion of the management of distributed data in one of the previous chapters, identification is taken care of by a session construct. While in general, there are different incentives that call for the creation of a session, we only use them in order to be able to refer to a distinct bilateral communication context that is established between a client and a server.

The red colored elements support the system for authentication capabilities. We note that the major functionality herein is delegated to the JAAS [48] framework. JAAS stands for Java Authentication and Authorization System and represents the reference implementation of the PAM [49] framework for Java.

Finally, the blue colored elements enable the system architecture to cope with the required authorization capability.

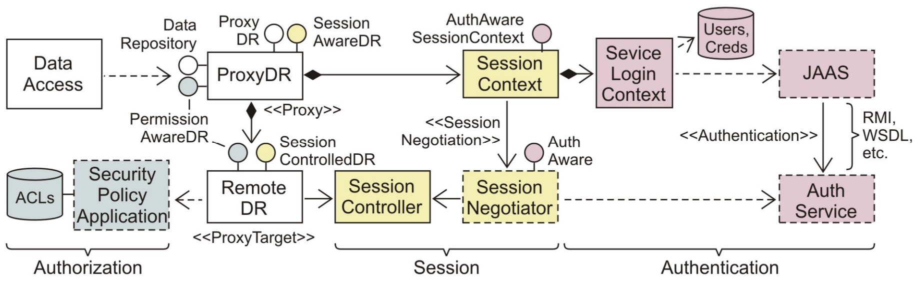


Figure 35. Main components for authentication and authorization

In the following, we look at each depicted component and discuss its major aspects as well as its interactions with its counterparts.

We already are familiar with the DataAccess, the ProxyDR and the RemoteDR. They represent the components that serve for the primary functionality. A detailed discussion about their individual characteristics and their interdependencies was already provided in chapter 6.1. Hence, we can directly address the more specific components that serve for the three distinct aspects.

6.5.2. Session management

The SessionContext is owned by the ProxyDR and acts as a delegator for the session management between its owner and the remote party. At the remote location, a similar constellation is anticipated. The RemoteDR owns a delegator (the SessionController) which keeps track of the current sessions. When a client contacts the RemoteDR in order to deposit an operation request it shows the previously established session. The RemoteDR subsequently shows the provided session to its SessionController that is able to validate it. If the session turns out to be valid, the RemoteDR starts the regular process in order to perform the requested action. On the other hand, if the session turns out to be unknown or outdated the RemoteDR answers with an appropriate failure message. Note that the connection between the proxy and the remote repository not always needs to be based on a session. A session is only necessary if the remote repository implements the SessionControlledDR, as is the case in the depicted scenario. Also, notice that due to the fact that the RemoteDR act as a SessionControlledDR, the proxy on the client location needs to be provided with the corresponding functionality and hence needs to implement the depicted SessionAwareDR interface.

By now, we know how sessions are being used by the parties that are in charge of the primary functionality. Nevertheless, we do not yet know how a session is being established in the first place. Let us therefore look at the SessionContext and its corresponding peer, the SessionNegotiator. When the SessionContext is told by its ProxyDR to return a current session, two possible actions might follow. In the first case, the SessionContext already holds a valid session and thus immediately returns it to the ProxyDR. In the second case, the SessionContext obviously is not yet in the control of a session and therefore needs to engage into the corresponding establishing process. Here is the point where the SessionNegotiator finally comes into play. It represents an independent service that like the RemoteDR is accessible by the client over a well-defined interface. While the RemoteDR is in charge of the primary functionalities, the SessionNegotiator on the other hand is in charge of the yet missing session establishment.

When the SessionContext needs to establish a new session, it addresses the destined remote SessionNegotiator that in turn starts the corresponding negotiation process with its counterpart. Note that the exact process by which a session is being established is not defined by concept but is subject to an individual session negotiation protocol. In the simplest case a specific negotiation protocol can be represented by a simple one-way call where upon the SessionNegotiator returns an independently created session identification. On the contrary, another protocol may be based on several bidirectional message flows. The latter is usually the case if a session constitutes of shared secret keys and such need to be established during the described creation phase [15].

The following figure summarizes above discussed aspects and shows a typical action flow. The abstractly covered process of opening a session at the SessionContext will be illustrated in details when we talk about the authentication aspect in chapter 6.5.4.

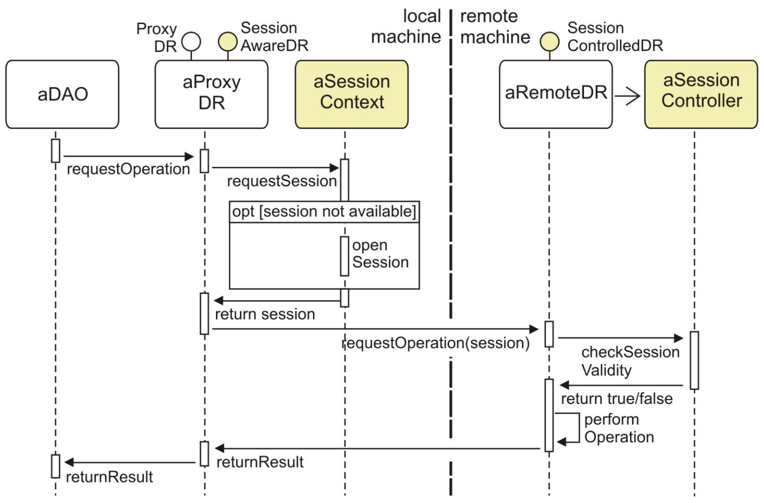


Figure 36. Establishing a session between a client and a remote repository

Using sessions in the described fashion above enables the system to rely onto identities without the need to care about their credibility. We so far discussed how a session is technically established and how it is delivered to agents such as the ProxyDR or the RemoteDR. We though did not talk about how the negotiation process decides to whom and in what circumstances a session is being granted and thus bilaterally established. We so far assumed that we can trust both the client with its SessionContext and the server with its SessionNegotiator. As long as this assumption holds true there is no need to authenticate one another. The negotiation of a session can be done regardless of whether or not the claimed identities are indeed accurate. Note that this aspect represents a necessity if we are about to provide authentication services as additional, fully decoupled and transparent features.

6.5.3. Authorization

The functionality for authorization is provided by yet another specific decorator of the generic DataRepository. Figure 35 shows that both, the ProxyDR and the RemoteDR implement the

PermissionAware interface that provides them with the appropriate additional functionalities. When the proxy addresses its target and requests the execution of a particular data operation such as an update, a deletion or the like, the repository at the remote location first grasps the simultaneously provided session and gives it to its SessionController. Given a concrete session, the controller is able to infer the corresponding identity. Knowing the identity of the requesting client the RemoteDR subsequently addresses a destined security policy application in order to infer the permission that is granted to the current client. The security policy application depicted in figure 35 is in possession of ACLs (Access Control Lists) [15] that correspond to the contained data in the RemoteDR. Note at this point, that the system architecture silently assumes that permissions are being described by a DAC (Discretionary Access Control) [50] system.

If the security policy application answers a permission that allows the execution of the requested operation, the RemoteDR starts the regular process in order to perform the client's request. On the other hand, if the inferred permission does not allow for the execution of the requested operation the RemoteDR answers with an appropriate failure message.

The granularity by which permissions can be defined is subject to the concrete decorator that the RemoteDR presents to its client. Figure 37 shows the three different kinds of available interfaces. PermissionAwareDR represents the root interface and accounts for the granularity of a per repository basis. A client in this case can be granted permissions that collectively apply to all data contained in the repository. OntologyPermissionAwareDR accounts for the granularity of a per ontology basis and EntityPermissionAware respectively account for the granularity of a per data entity basis. Depending on specific requirements of particular data repository, any combination of the presented interfaces can be applied.

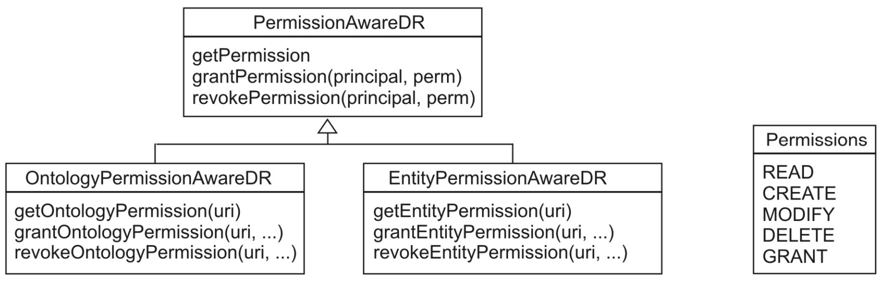

PermissionAwareDR
getPermission
grantPermission(principal, perm)
revokePermission(principal, perm)
OntologyPermissionAwareDR
getOntologyPermission(uri)
grantOntologyPermission(uri, ...)
revokeOntologyPermission(uri, ...)
EntityPermissionAwareDR
getEntityPermission(uri)
grantEntityPermission(uri, ...)
revokeEntityPermission(uri, ...)
Permissions
READ
CREATE
MODIFY
DELETE
GRANT

Figure 37. Permission on various granularities

6.5.4. Authentication

Authentication is the process of confirming the claimed identity of some communicating actor. When we looked at the session mechanism, we realized that the negotiation process is based on the assumption that both parties trust one another and that therefore authentication is not required. As soon as this assumption does not hold true anymore, the bilateral negotiation process of a session is only possible when the communicating parties have preliminarily authenticated one another.

As a result, the SessionContext can optionally be configured in order to account for the appropriate authentication procedure. In this case, the SessionContext implements the optional AuthAwareSessionContext interface that represents a decorator to the already known basic

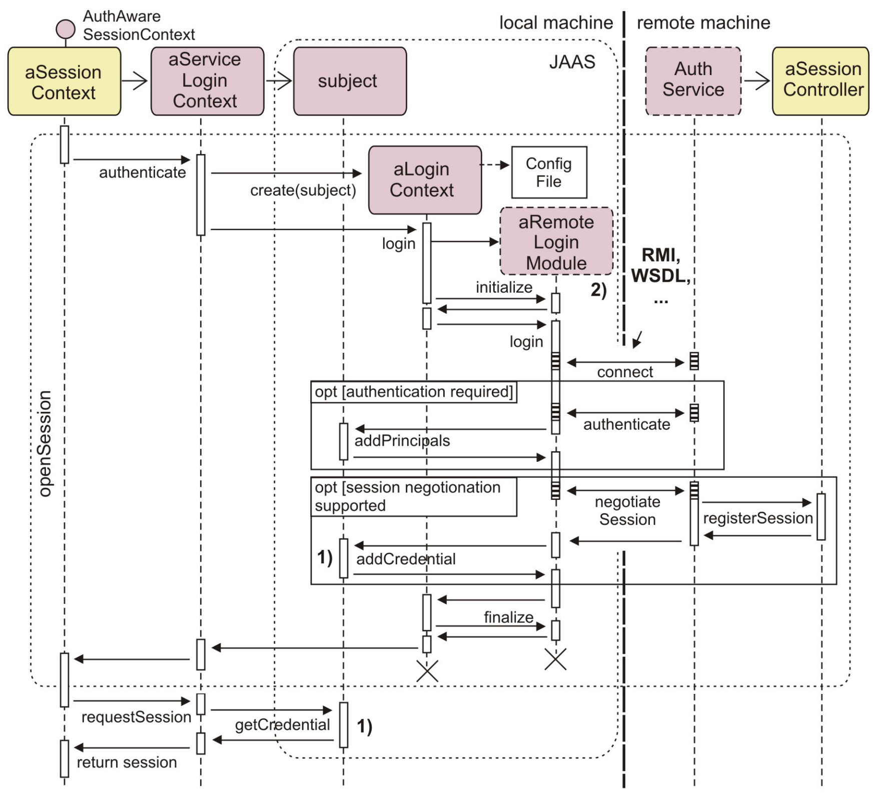
SessionContext. AuthAwareSessionContext defines the additional functionality to declare a corresponding ServiceLoginContext (SLC). Analogously to the delegation between the ProxyDR and the SessionContext, the SLC acts as a delegator for the latter as far as authentication procedures are concerned. When the SessionContext is requested to create a new session and thus implements the AuthAwareSessionContext interface, it first addresses its corresponding SLC in order to perform the required authentication process and only afterwards engages into the process of establishing a session as described above.

The underlying concept of the SLC is straightforward. The SLC namely can be seen as a typical intermediate between the SessionContext and the JAAS framework. Having access to the client's credentials, it basically does nothing more than initiating a login request on behalf of the SessionContext or ProxyDR respectively. Once the authentication procedure has successfully finished, the SLC immediately returns the action flow back to the SessionContext.

When the SessionContext and its peer, the SessionNegotiator (SN) engage into the establishment of a session and the SN implements the optional interface AuthAware, negotiation will only be accepted by the latter, if the connecting client preliminary was successfully authenticated at the corresponding AuthService. The AuthService thereby can represent any type of standard or specific authentication service that analogously to the SN or the RemoteDR is accessible by the client. A primitive authentication service may be represented by the rlogin or telnet program on a UNIX workstation [49]. On the other hand, powerful mechanisms such as a login over SSH[23] or Kerberos [16] is feasible as well. Note that the SN in this situation needs to be able to communicate with its corresponding AuthService in order to find out what clients recently have been successfully authenticated.

A lot of standard token-based security services couple the authentication and session negotiation process in order to be able to establish a secure connection. Such for instance is the case with Kerberos. The sequence diagram in figure 38 depicts how the integration of such a security system can be achieved. Note that due to the fact that the session is already created by some JAAS component (RemoteLoginModule), the SessionContext does not need to engage in to the previously discussed session negotiation process but easily can get the session from its corresponding ServiceLoginContext. Note also that the components subject, LoginContext and RemoteLoginModule are specific JAAS components that we did not discuss. Readers not familiar with the basics of JAAS may first have a look at the appendix B.4.

[23] SSH is a common protocol which can be used as a general purpose cryptographic tunnel. At the same time, it represents a widely used remote login application that directly relies onto the SSH protocol [15].



1) The ServiceLoginContext relies on the subject's assigned credentials when looking up the current session. Thus, a session is kept as a credential and can be retrieved any time that is needed.

2) The RemoteLoginModule is a typical authentication procedure specific module in the sense of a PAM or JAAS framework. For Linux a simple concrete application would be telnet or rlogin. Obviously, also a full-fledged application such as Kerberos is possible instead.

Figure 38. Transparent authentication and session negotiation

The above explanation was held on a suitable elevated abstraction level in order to be able to focus on the principles and the major collaboration patterns among the introduced components. As a result, the explanation does not claim to give any insight as far as the actual integration of the standard JAAS framework is concerned. The interested reader is referred to the appendix B.3 and B.4 where both PAM and JAAS are covered. Readers interested in the concrete functioning of JAAS as a generic security framework nevertheless are referred to appropriate literature mentioned in the reference chapter ([48, 36]).

6.5.5. Conclusion

The presentation of the session management revealed that it enables the separation of authentication and authorization by providing a common denominator for the representation of the identity of a communicating client. Authorization is achieved by yet another optional decorator for the generic DataRepository. Finally, as far as authentication is concerned, the system calls for the delegation onto the widely used JAAS framework that enables an application to virtually be able to deal with any type of concrete authentication facility. In summary, it can reasonably be said that all corresponding requirements from the chapter 4.7 are successfully met.

7. Implementation

On of the goals of this thesis was to elaborate a system architecture that is able to adequately solve the deficiencies introduced in the motivational chapter. We hence studied the corresponding requirements, then looked at concrete underlying design principles and in the previous chapter finally presented a complete system architecture. An additional goal of the thesis however was to develop a reference implementation that primarily has the two goals of (i) serving as an actual proof-of-concept prototype and (ii) providing a running system that – as is – can be plugged into overall NExT's system architecture described in [1]. In the following we look at the developed reference implementation but instead of profoundly elaborating the generally rather straightforward implementation aspects, we only focus on some of the specific additional design considerations.
We first engage into the discussion of factories and singletons concerning their co-existence. The goal is to sensitize for their distinct differences and to reveal in what situations they best be applied. Second, we discuss the debate about whether or not to make use of underlying, thus somewhat hidden, implementations. We in this section will find out how the reference implementation can profit from the two powerful frameworks represented by the OWL-S API and the Jena framework. In the third and the fourth section, we finally look at two concrete implementation aspects. On the one hand, we look at the concrete integration of the Pellet reasoner [51] and thereby try to point out with how little effort the integration is accomplished. On the other hand, we highlight the straightforwardness of the implementation of the versioning and annotation functionality that eventually results from the system architecture's underlying concepts. Security aspects regarding authentication and authorization are not specifically covered, as they are principally delegated to the standard JAAS framework.

7.1. Factories and singletons

The use of factories and singletons is not mutually exclusive although this sometimes is misleadingly believed by programmers rather new to the use of design patterns [40]. A factory is used in order to hide the creation and composition of objects that apply to defined interfaces of a system architecture [40]. A component that uses a particular counterpart hence should not rely onto that concrete implementation but should only rely onto the provided interface. As a result, if the required counterpart does not yet exist, the former needs to have a third party component (thus a factory) that it can ask for the appropriate creation.

A singleton is used in order to make sure that a particular object only exists once for the entire system and that it therefore cannot be created multiple times [40]. In contradiction to a factory, a singleton however cannot hide its concrete implementation towards its clients. Hence, if a component relies onto an interface that is implemented by a singleton, the component is not able to create an arbitrary object that offers the required interface if it does not know the corresponding concrete implementation. If the component needs to be able to refer to a singleton without knowing the underlying implementation, the singleton additionally needs to be wrapped into a factory that accounts for the necessary indirection.

Being aware of above realization, the reference implementation uses singletons always in combination with a factory. In addition, it carefully distinguishes between *singletons de-facto* and *singletons de jure*. All *singletons de facto* are defined by regular classes that principally may have multiple instances. In such situations, the factory is responsible to guarantee for the inherent singleton restriction. It makes sure that once a particular instance has been created, that very same is returned by any subsequent request. *Singletons de jure* on the other hand are implemented as regular singletons. In this situation, the singleton restriction is directly controlled by the class implementation. While the difference between the two design methods is small, a careful distinction nevertheless can be precious. Two major aspects are given hereafter. First, a careful distinction undoubtedly improves the readability and the clarity of the overall system and hence improves maintainability. Second, a careful distinction usually also improves testability [32]. When the creation of testing environments is at stake, singleton-like components oftentimes need to be specifically composed and/or configured. In yet other situations, it is desired to be able to intentionally relief the inherent singleton constraint. As a result, from this second point of view, it is desirable to have as little regular singletons as possible. However, such is only achievable if each potential singleton is analyzed in terms of its actual requirements.

7.2. Underlying implementations

The discussion about the use of underlying implementations by the technique of explicit casting of a generic into a more concrete type, which only can be assumed but is not clearly specified by the corresponding interface, can be endless. The subject basically deals with the question of how much a client component is allowed to interpret an interface's described functionalities within an assumed context that reveals additional implicit assertions. Design by contract [32, 44] does indeed not necessarily forbid to reason upon an underlying implementation as long as that can reasonably be inferred and as long as the implementation adequately declares its additional resulting dependencies. The first argument reveals that the assumption must not be arbitrary but most by some means be legitimate. For example, the determination of the particular used implementation may be defined by the system requirements, or may be sufficiently defined by a superior system architecture that serves as the definition of the environment wherein the particular implementation is placed. The second argument reveals that an implementation must somehow specify all its dependencies that cannot directly be inferred from the applied system architecture. Many implementations do not document the functionality of their components but in this respect refer to the corresponding documented interfaces from the system architecture. Indeed such technique is legitimate and also efficient as it keeps documentation at a well-defined location. However, it though fails when assumptions not explicitly asserted by the interfaces are made. An implementation in this situation thus must document whatever aspect is not yet covered by the interfaces.

The reference implementation heavily relies onto the underlying implementation of the OWL object-binding framework. In this respect, it namely relies onto the OWL-S API and its underlying Jena framework (figure 39a). However, this design decision can be regarded legitimate because the implementation accounts for both above discussed aspects. The assumption is not arbitrary but is well documented in NExT's coarse-grained system architecture described in [1]. Second, the implementation adequately documents the defined classes in terms of assumptions and assertions that cannot be inferred from the corresponding interfaces (system architecture). As far as the second point is concerned, the implementation goes even one step further and defines internal interfaces in order to be able to define more precisely the components' functionalities. The reference implementation thus again is partitioned in the definition of particular interfaces and their corresponding class implementations. Figure 39b depicts the relation between the system architecture and the implementation with additional interface definitions. Note that the implementation of course does only define additional interfaces where the system architecture reveals somewhat too generic.

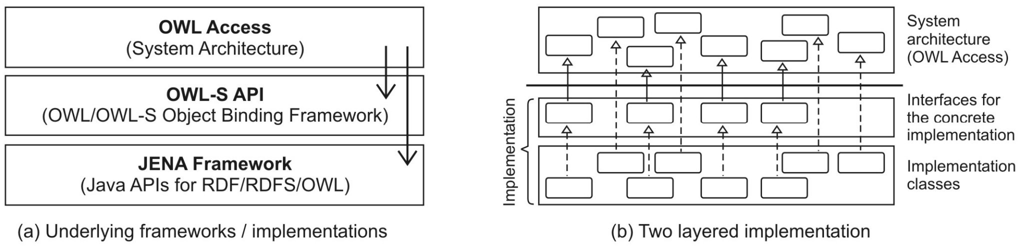

OWL Access
(System Architecture)
OWL-S API
(OWL/OWL-S Object Binding Framework)
JENA Framework
(Java APIs for RDF/RDFS/OWL)
(a) Underlying frameworks / implementations
Implementation
System architecture (OWL Access)
Interfaces for the concrete implementation
Implementation classes
(b) Two layered implementation

Figure 39. Use of underlying implementations

7.3. Reasoning with Pellet and OWL-S API

In this section, we show how the reference implementation in particular deals with the integration of the Pellet reasoner that is compatible with the OWL-S API. Let us first resume the basics of the envisioned concept presented in the previous two chapters as far as reasoning is concerned and only thereafter look at the concrete implementation.

Due to the fact, that reasoning needs to be performed over distributed data and because data volume tends to reach magnitudes that cannot be easily transferred from one remote location to another, the concept prescribes that reasoning is to take place separately at each repository. We profoundly discussed this topic in chapter 5.2. A major design aspect in this situation is represented by the mechanism by which arbitrary reasoners get access to repositories' internal data representations. As different reasoner implementations may use very different reasoning mechanisms, they also tend to require very different object oriented graph representations upon which reasoning can eventually be performed. As a result, the concept foresees that arbitrary reasoners do not directly access the repositories' internal data representation but instead manage their own, thus private and redundant, data representations. A data repository implements the ObservableDR decorator (interface) in order for the reasoner to be able to get informed about changes to the internal data representation. The update process between a reasoner and a repository is then performed over generic OWL/RDF serialization. Hence, the data repository also is assumed to implement the OwlSerializationProviderDR decorator, which provides reasoners with the necessary functionality to read updated and/or newly created data entities and ontologies respectively.

When using the Pellet reasoner, above summarized mechanism however can significantly be simplified. Pellet depends on the Jena and OWL-S API as far as the object oriented graph representation and part of the inference mechanism is concerned. Collaboration between the repository and the Pellet reasoner can be based on their common underlying frameworks and thereby must not go the rather tedious detour over generic OWL/RDF serialization interchange. Moreover, the reasoner as well does not need to manage its private data representation. Figure 40 depicts the resulting simplified architecture.

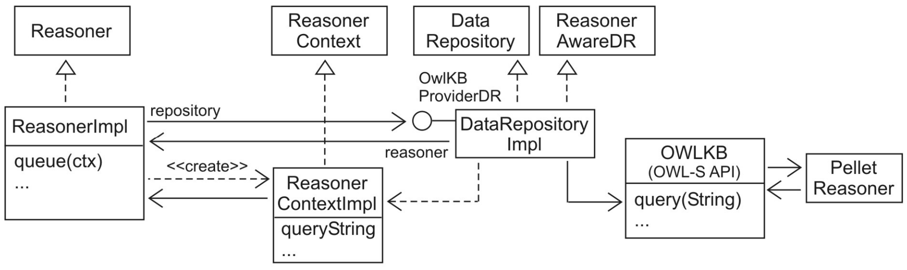


Figure 40. Implementation of the pellet reasoner

In contradiction to the generic collaboration, the depicted architecture uses the possibility to infer about the underlying implementations, which are the OWL-S API and the Jena framework. The reasoner communicates with the repository over the OwlKBProviderDR interface instead of the generic OwlSerializationProviderDR. As a result, the reasoner gains direct access to the repository's internal data container that is represented by an OWLKnowledgeBase, a particular interface from the OWL-S API. Clearly, serialization over OWL/RDF as well as the installment of a notification infrastructure is no longer necessary. When the reasoner receives a query request, it just gasps the corresponding OWLKnowedgeBase upon which it directly is able to run the necessary inference procedures. The ReasonerContext is no longer involved into the reasoning procedure but solely serves for the temporary storage of the RDQL query (QueryString) and the eventual query result. When the execute method of the ReasonerContext is called, the ReaonerContext takes the provided query (QueryString) and stores it in some internal variable. It afterwards switches into phase WAINTING and puts itself into the reasoner's internal queue. When the reasoner eventually picks up the context in order to perform the requested query, it (i) switches the phase of the ReasonerContext from WAITING to RUNNING, then (ii) reads the temporarily stored query string from ReasonerContext's internal variable, (iii) executes it upon the data representation of the repository, (iv) stores the query result in the ReasonerContext, and (v) finally switches the phase from RUNNING to DONE. Clearly, the implementation is straightforward and seamlessly comprehensible. For the sake of completeness, figure 41 shows the reasoner's infinite loop that it entered after initialization in order to manage internal queue of query requests.

```
public void run() {
   ...
   ReasonerContext ctx;
   String query;
   while(queue.hasElements()) {
      ctx = queue.pop();
      query = ctx.getQueryString();
      ctx.result = getRepository().getKB().execute(query);
      ctx.setPhase(ReasonerContext.DONE);
   }
   ...
}
```

Figure 41. Management of queued query jobs by the Reasoner

While the above demonstration does not reveal any novel or otherwise captivating information, it nevertheless is able to reveal some of the inherent potential that the two aspects (i) reuse and (ii) delegation to underlying implementations can eventually bring about.

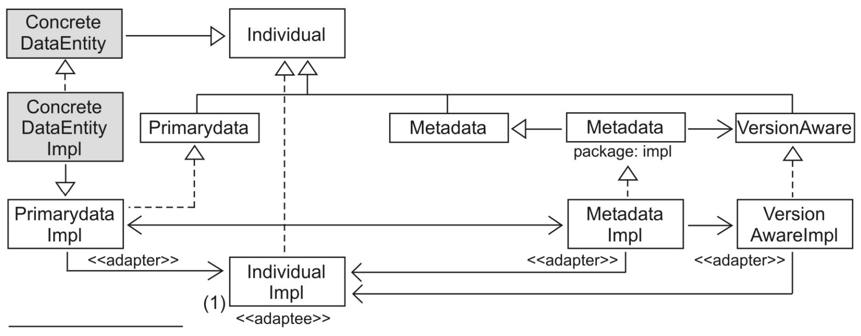
7.4. Annotation and versioning in one go

In the following we look at the implementation of the versioning and annotation functionality that both reveal to have many similarities and thus can somewhat be addressed in one go. In chapter 5, we realized that the prescribed generic annotation functionality can best be accounted when looking at annotation specific data as generic metadata, thus data about data. We then elaborated a concept based on two separated views that can optionally be applied to a generic data entity, which – in terms of the system architecture – is defined by the Individual interface. The elaborated concept for the versioning functionality on the other hand is based on the notion of a decorator for the Individual interface. Although both concepts follow different goals and were separately elaborated, they nevertheless do share quite a lot in common. The following list illustrates the major similarities.

- Both aspects in principal deal with data about data, thus metadata.
- Versioning and annotation can be regarded as extensions to the generic data entity defined by the Individual interface.
- Both functionalities need to be provided optionally when such are adequate and/or needed
- Their functionalities in both cases are defined by interfaces that inherit from the generic Individual and hence can be regarded as decorators
- The definition of particular data entities (such as a NMRProcess, NMRExperiment, etc) must not be affected by either of the two functionalities

The reference implementation mainly focuses onto the fourth and fifth point. It decides to use the adapter pattern [40] in order to be able to encapsulate a genetic data entity within a wrapper once for metadata purposes and once for versioning purposes. The latter subsequently is being bound to the former as is done with Primarydata and Metadata, which was discussed in chapter

6.4.1. The reference implementation defines a slightly more specific Metadata interface that is named identical but that is yet put into another package. The only difference between the two interfaces is that the latter additionally defines a method that allows the client to get a VersionAware view onto the corresponding data entity. The advantage that comes with this design decision is that versioning related data can be treated as metadata of some special kind. The interface of a particular data entity (such as an NMRProcess or NMRExperiment) as a result is not scattered with versioning functionalities (methods) but in this respect is kept clean and adheres to the principle of separation of concerns [32]. Because both functionalities are defined by separate interfaces, they still are decoupled as far as extensibility (subclassing) and interdependence is concerned. If for example, NExT decides to use the Dublin Core Metadata Initiative [13] in order to account for its annotation needs, it uses the DCMetadata interface and a corresponding implementation thereof (chapter 6.4.1). As VersionAware is implemented by a separate view onto the corresponding data entity, it thereby is not affected by the illustrated extension. Instead of independently extending the Metadata interface, we as well could independently extend the VersionAware interface. A concrete data entity implementation by default would inherit from the basic IndividualImpl. However, if the entity needs to support either or both of the discussed functionalities (annotation and/or versioning) it instead can inherit from the provided PrimarydataImpl and with this is given the additional method that allows a client to ask for the Metadata and VersionAware view respectively. Since PrimarydataImpl also uses the adapter pattern in order to wrap the generic IndividualImpl, inheriting from either of the two implementations (IndividualImpl or PrimarydataImpl) does not matter when creating a concrete data entity. Note, that while the former characteristic especially accounts for forth point, the latter adequately does so for the fifth point. The illustrated architecture is summarized in the figure below.



1) PrimarydataImpl and MetadataImpl both bind the same IndividualImpl leading to two different views onto the concrete Individual representation.

Figure 42. Versioning and annotation functionality in one go

8. Conclusion

8.1. Summary

The thesis elaborated a coherent and fine-grained architecture for specific aspects of NExT's overall system architecture described in [1]. Speaking more precisely the goal of the thesis was to focus onto following somewhat independent aspects:

- Elaborating an architecture and a corresponding reference implementation of a generic OWL/RDF data access abstraction layer.
- Defining a scalable and flexible reasoning mechanism that allows reasoning over data distributed onto self-contained data repositories. Thereby allowing manual and programmatic querying of data.
- Providing a flexible and evolutionary annotation concept to be used for data entities of the NExT process model.
- Elaborating a versioning concept that both, suits the inherent requirements within the NExT context, and represents a feasible and natural approach in regard to the underlying concepts of OWL and RDF.
- Extending the NExT architecture with adequate functionalities for authentication and authorization in its inherently distributed environment.

The thesis started with a motivation of above focusing aspects and revealed some of the most obvious deficiencies that can be inferred from the overall NExT system architecture. Afterwards, a brief digression was made which allowed introducing a visionary system that claims to solve all the problems discussed throughout the course of the thesis and that furthermore claims to account for problems that will arise as the notion of the Semantic Web and the ongoing trend to ubiquity steadily increases in terms of importance and global acceptance.

The primary part of the thesis discussed the mentioned focusing points and for each of them elaborated a feasible solution. While the reasoning aspect was treated as a de-facto mandatory functionality of the designed data access layer, versioning, annotation and security (authentication and authorization) were considered optional and thus discussed separately. In the following, each architectural design aspect is briefly summarized.

A data access layer needs to exhibit different transparency aspects. A discussion about the concrete requirements revealed that adequate consideration of eight transparencies for distributed systems is needed. It furthermore turned out, that aspects such as flexibility, openness and scalability are as important and thus needed to be equally considered. In order to define the underlying design concepts, the lifecycle of a typical NMR experiment was profoundly studied. The result was the elaboration of three distinct problem fields. (i) The granularity of shareable data reveals that sharing may take place on different abstraction levels. (ii) The frequency of publishing versus retrieving addresses performance related considerations in terms of imaginable data processing concepts. (iii) The topology of data repositories highlights the notion for distributed, self-contained services for persistent data storage.

In order for the system architecture to account for the mentioned aspects above, principles such as modularity, information hiding, separation of concerns, pay as you go, and design by contract turned out to be crucial. One of the major design aspects dealt with the modeling of data repositories that may provide different functionalities in different situations and additionally may be located on remote machines. Here, the decorator design pattern came in as a suitable and handy approach. As far as reasoning functionalities are concerned, yet another key design consideration was necessary. A thorough discussion about the inherent characteristics of reasoning over OWL/RDF described data, revealed that it is not feasible to preliminarily collect all relevant data from the corresponding repositories in order to subsequently start reasoning upon them. The system architecture as a result decided to locate reasoner engines directly at their corresponding data basis (thus at individual data repositories) and to provide an adequate infrastructure which allows to use the former in a maximal transparent fashion. The required transparency was achieved by introducing an additional indirection, which was represented by a so-called reasoner context.

The elaboration of the concrete requirements for versioning in the context of NExT revealed that the notion of a version has to account for two different aspects. First, a version is assigned to a particular data entity in order to be able to refer to its individual modification history. In this sense, any time an entity is modified; it is stored by a new version. Second, a version also is assigned when a copy of an existing data entity is used to represent a variation to its original. The underlying versioning concept accounts for the notion of a version in order to refer to the first aspect and accounts for the notion of a variation in order to refer to the second aspect. While the concept is simple and straightforward, it reveals one major problem. As different versions and/or variations of a data entity are not necessarily stored in the same data repository but may be distributed across the network, it is not always possible to infer the correct subsequent version number. The thesis discusses two solutions that claim to overcome the mentioned deficiency.

The underlying concept that eventually claims to provide the system with a flexible and scalable annotation capability is based on the notion of two separate views. While the first view accounts for the primary data, the second view addresses the management of annotation or more precisely generic metadata. In the data representation layer the OWL's capability for multiple inheritance is applied in order to naturally account for the definition of the two separated views. In the higher level where Java is assumed to be used as the primary programming language, multiple inheritance is no longer possible and thus needs to be replaced by an adequate design concept. The two views are defined as mutually depending decorators to the generic data entity which is modeled as an Individual. The two major benefits that result from this design decision are simplicity and transparency. The latter benefit leads to the fact that except for two localizable components the system is not affected when support for metadata is optionally added.

Depending on the type of trust that is mutually anticipated between a client and a remote data repository, different authentication and authorization concepts are adequate and feasible. The thesis in particular elaborated four fundamental login procedures that each may be applied for a distinct anticipating trust relation. In order for the elaborated security concept to best possibly account for all four login procedures, it needed to focus on the strict separation of the notion of identity, authentication and authorization. Another major design decision was encountered when mechanisms concerned with establishing and maintaining secure connections were discussed. The question was whether the architecture should go for either a connection-oriented (focusing on hierarchical layers) or a token-based (focusing on separation of concerns) approach. For the sake of openness and flexibility, the latter was chosen.

8.2. Future Work

Future work is mainly represented by improvements and regular extensions to the elaborated system architecture. The most evident tasks are presented in the following. The order is arbitrary and therefore does reflect neither the feasibility nor the importance of the relative aspects.

8.2.1. Web of Trust

A web of trust can successfully be provided by integrating the PGP concept that relies on public key cryptography. Data entities could be encrypted in a similar way as is done with E-Mail messages. As the system architecture prescribe that data entities be stored in OWL/RDF format, the de facto standard RDF vocabulary WOT (Web of Trust ontology) [31] could thus be integrated. In this sense, the Individual interface would need to be extended by an optional decorator as analogously is done for versioning (chapter 6.2) or annotation (chapter 6.4).

8.2.2. Digital Rights Management

When we looked at the underlying design concepts in chapter 5, we – among other topics – talked about authentication and the corresponding handling of data access rights. Because the requirements only described primitive requirements, a traditional DAC mechanism could be applied. However, if we wanted to extend the foreseen functionalities and provide a standard, flexible and foremost generic data access control mechanism, we may want to integrate the corresponding W3C standard Open Digital Rights Language (ODRL) [52] that claims to account for any current and probably future rights management requirement. As the ODRL rights definition is based on an XML application, we may preliminarily need to define a corresponding OWL/RDF vocabulary. Although, such may represent a tedious task, it nevertheless is feasible.

8.2.3. Extensible and interoperable entity identification

By the notion of self-contained data repositories, the system architecture offers a flexible and interoperable way to distribute data over the network. Due to the inherent lifecycle of a typical NMR experiment, data entities not necessarily reside in the initial repository but may change its location on a regular basis. Unique entity identification within the scope of the entire network is crucial and needs to be appropriately accounted for. As the system architecture currently relies onto the user to define a globally and all-time unique identification, such a control system is totally missing. DOI (Digital Object Identifier) [53] is an ISO standard framework that accounts for the mentioned deficiency by specifying a standard numbering syntax and a generic resolution service. While it is obvious that the system architecture needs to be extended for some kind of an appropriate control mechanism, it may – but does not have to – integrate the introduced ISO standard.

8.2.4. Ontology translation mechanism

Different companies and institutions may specify individual and/or proprietary ontologies in order to describe their specific application and/or problem domains. Situations where two different ontologies do specify similar or even identical concepts are more than possible to occur. In order for two such companies to share their data efficiently, an ontology translation mechanism will be needed. In this sense, the system application could be extended to implement the corresponding translation mechanism from Edutella [20, 21] that defines so-called wrappers that are put atop of their agents. We already covered the Edutella's overall architecture when we looked at related work to this thesis.

8.2.5. Common query language

A common standard query language will be required as soon as we want to query over multiple data repositories and cannot assume that the repositories all provide the same query languages. When we looked at the reasoning capability of the system architecture, we noticed that reasoning over multiple repositories indeed is only possible if the repositories can agree upon a common reasoner. With the notion of a common query language from which wrappers are able to translate the query into the reasoner's individual query language, this limitation is no longer anticipated. As a query usually results in an answer data set that is returned to the requestor, not only the query language but also the data model eventually needs to be translated. As is with the ontology translation, the system architecture again may want to make use of the promising solution from the Edutella [20, 21] framework. Edutella defines both, a common query language (ECQL) and a common data model (ECDM). Additionally, it built a first prototype that serves as a proof-of-concept.

8.2.6. Repository workload balance

The system assumes that the number of data repositories is steadily augmented as data load increases. The architecture however does not account for a corresponding workload balance mechanism that can transparently move data between specified repositories in order to guarantee that particular repositories be not overloaded. Obviously, a good place to implement such a mechanism is at proxy repositories that act as composite repositories and transparently manage a specified set of subordinate data repositories. Eventually, the system architecture could thus define another specific decorator that then can be applied when adequate.

8.2.7. Triple store

The current reference implementation stores OWL/RDF data in regular files without concentrating on aspects such as scalability, performance and the like. The implementation in this respect only provides a simple solution that allows proving the architectural design. The system architecture however foresees the integration of a third party triple store [54]. While the corresponding interfaces are already defined, a concrete implementation is yet pending.

8.2.8. JXTA

As illustrated in chapter 2.7.1, data repositories could eventually be implemented as JXTA services in order for the system to further improve its flexibility and interoperability. JXTA represents a novel communication framework for peer-to-peer systems and – among many other aspects – accounts for functionalities such as agent notification, agent grouping, agent discovery and agent communication through arbitrary network firewalls. As far as communication between proxies and their target data repositories is concerned (see chapter 6.1.4), JXTA could represent a promising standard and open solution.

A. Appendix A - NExT Application

A.1. NExT's process model in a nutshell

From an abstract point of view, conducting an NMR project does not differ from any other business workflow. Whatever task needs to be performed at a specific point in sequence can be described as a process. NExT's process model defines three fundamental types of processes that can be assigned to abstract hierarchical levels.

A ProcessPlan represents the root of the hierarchy. It defines a collection of Experiments and the sequence in which these are to be performed. Experiments in this sense represent the second hierarchy level. Each of them formally describes the workflow of an NMR project in terms of their required tasks. The third (thus lowest level) characterizes the tasks themselves. A task that recursively can be subdivided is modeled by a CompositeProcess. Respectively, a task that represents the smallest conceivable unit is modeled by an AtomicProcess. The following figure depicts the described process model as it is used for the NExT system.

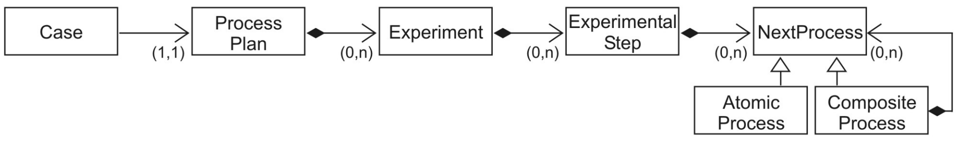


Figure 43. NExT's process model

Apart from the above described process hierarchy, the process model introduces an additional entity that is represented by the so-called Case. A Case essentially is wrapped around the ProcessPlan and is able to record the workflow's concrete pathway during execution. Additionally, it also keeps track of the state modifications that result from the sequential execution of the inherent recursively contained processes. Once a project will have been successfully completed, the Case is put into a Case Base [1] where it is used for future reasoning purposes. A detailed discussion of the NExT process model can be found in [1].

A.2. NExT's coarse-grained system architecture

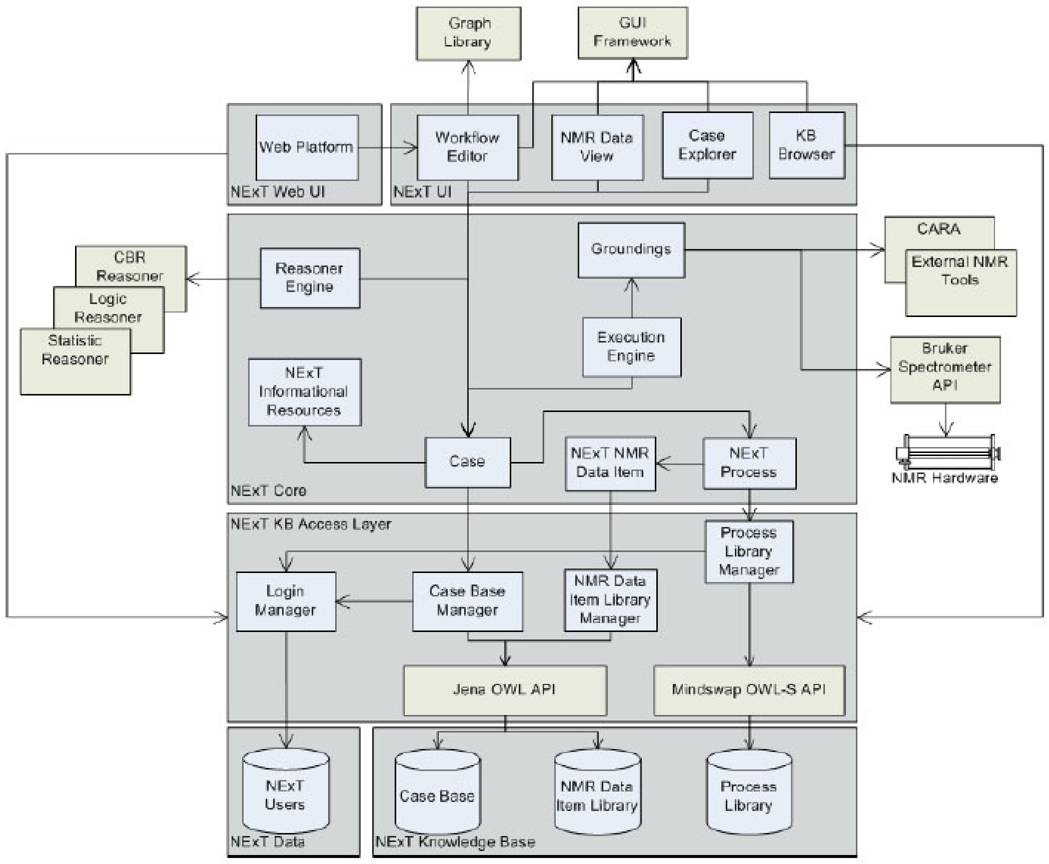


Figure 44. NExT coarse-grained system architecture [1]

B. Appendix B – Used Technologies

B.1. Token based authentication in single and multiple realms

Kerberos [16] is a widely used authentication technology whose concept is based on session-oriented service granting tickets that are being issued between a user and a so-called Ticket Granting Server (TGS). The technology is widely used because it allows a client to negotiate with its Ticket Granting Server without the need of transferring plain security tokens. Kerberos hence does not require the establishment of secure connections between clients and servers [16].

The discussion herein focuses on the fundamental authentication procedure but does not look at the specifics in terms of the higher conceptual components such as the notion of a Ticket and a Ticket Granting Server. Readers interested in the detailed concept of Kerberos are referred to appropriate literature [16].

Kerberos relies on the notion of security tokens that are established between communicating actors. Due to this fact, authentication can only be provided within an established, well-defined realm unless further effort is being made. Let us first discuss the notion of a realm and its characteristics for a generic token-based authentication system. In the second part, we then look at what specific solution Kerberos offers in order to avoid the mentioned limitations.

When different parties need to negotiate with one another but cannot trust their claimed identity, they need to authenticate themselves against one another. A token based authentication system pursues the notion of negotiated temporary passports that actors can show when communicating with one another in order to avoid repetitive and burdensome authentication processes. In this sense, a token is used as a bilateral passport.

The idea is to engage into a preliminary authentication phase where each actor negotiates a bilateral token with its counterparts. Once a token is established by an appropriate authentication procedure, it subsequently can be used between the two parties as long as they both accept the token to be valid. An actor A, who shares a bilateral token with actor B, can safely communicate by attaching the corresponding token to each message that is destined for actor B. So far so good. However, the real question is how authentication between two untrusted actors can be established in the first place.

Authentication can only be proved by pursuing adequate tests. The most common procedure nowadays is the use secure passwords. In this case, an actor that requests a counterpart to be authenticated asks for the counterpart's identification and corresponding password. Authentication in this case succeeds if the provided information matches with a preliminary stored pair of an identification representation and a corresponding password. As a result, authentication can only be performed if the participants have previously shared their credentials among one another.

As long as communication among actors is done in a hub-like fashion, where there is one single server and an arbitrary number of clients, the preliminary proliferation and subsequent maintenance of the clients' passwords (credentials) can feasibly be done. In such a constellation, the server typically keeps a list of registered clients with their corresponding credentials.

In an environment with multiple servers, the management of bilaterally accepted credentials is no longer feasible. As a result, a third party actor is introduced to globally manage the credentials of all the relevant actors. In such a constellation both, clients and servers, register themselves at the central actor and fully trust in that functioning. The scope in which the third party actor provides its services is what we referred to as a realm. We realize that the concept of temporary passports inherently assumes a well-defined environment in which at least one actor is to be trusted collectively. Kerberos assumes that the Ticket Granting Services is trusted by all components and that tickets be accepted as artifacts that have the power to prove an actor's claimed identity.

Note that this concept basically follows the procedure that we, as humans, use to deal with authentication. If a person wants to pass the customs from one country to another, the customs official does not call up some governmental office in order to prove our claimed identification. The official instead requests the person to show his or her passport in order to prove his or her identity. Also, note that the passport in this case exhibits the same rationale as we elaborated above. This passport can only prove the identity as long as it will be accepted by the authorities. The realm in terms of a personal passport is therefore defined by the set of accepting countries.

Kerberos offers a solution to avoid the discussed limitation that token-based authentication systems inherently exhibit. The underlying idea is based on trusted delegation. Ticket Granting Servers (TGS) [16] from different realms can be configured to trust one another. In this case, they bilaterally accept certain types of their issued tickets and thereby gain the possibility to authorize local clients to be able to access arbitrary cooperating TGS in different realms.

Let us look at the required sequence of actions that take place when two parties (client and/or server) from different realms engage into a bilateral communication. Let us assume that a client from realm A ($client_A$) needs to contact a server that is located in realm B ($server_B$). Following steps in this situation are necessary. First, $client_A$ authenticates at its local Ticket Granting Server (TGS_A) and requests a ticket that will be accepted by the Ticket Granting Server at realm B (TGS_B). Having received such a ticket, $client_A$ is able to connect to the TGS_B. In order to prove its identity it does not engage into a corresponding authentication procedure but provides its previously issued ticket. The TGS_B examines the provided ticket and infers that it has been established by a trusted (thus cooperating) TGS, namely the TGS_A. Hence, the passport is accepted and is able to prove $client_A$'s identity. Because $client_A$ intends to communicate with $server_B$, but that server only accepts tickets issued by its local TGS (TGS_B), our $client_A$ needs to call on TGS_B in order for the latter to subsequently create an appropriate ticket; a so-called Service Ticket (ST) [16]. Once $client_A$ is in possession of a regular ST for a particular service at $server_B$, it is able to communicate with the latter for the time of the ticket's validity. Note that due to the fact that $client_A$ receives a regular ST for the realm B, the targeted server ($server_B$) does not become aware of the actual remote location of $client_A$. We realize that although the sequence of action is bit of a pain, the underlying principle however is straightforward and well comprehensible. Kerberos names the described mechanism Cross Realm [16]. Readers interested in the details are referred to appropriate literature [16, 15].

B.2. Digital certificates

A digital certificate [15] acts as an electronic credential and verifies that the person presenting it is truly who she actually claims to be. In this respect, a certificate is similar to a passport. Both establish an individual's identity, contain a unique number for identification purpose, and have a recognizable issuing authority that verifies the credentials as authentic. In the case of a certificate, a Certificate Authority (CA) [15] functions as the trusted, third party that issues the certificate and verifies it as an authentic credential.

In contradiction to traditional authentication methods, digital certificates allow users to communicate securely without having prior access to a shared secret key. The method is based on two mathematically related keys. One key is globally proliferated and is referred to as the public key. The second key is kept secret by the owner of the certificate and yet is referred to as the private key [15]. The underlying concept foresees that a piece of information can be encrypted with either the private or the public key. Once some data is encrypted, it however can only be decrypted by the opposite key. If for example the owner encrypts a message with her secret key and sends it over the network, any peer that receives the message is basically able to open and read it. In order to do so, the receiver (i) request the sender's public key from a recognizable certificate authority and (ii) opens (thus decrypts) the message with the received key. Due to the fact that the message could be decrypted be the received public key, the sender implicitly was authenticated. No other than the sender's private key must have been used to encrypt the message in the first place. As long as the private key is kept secret and the CA is trustworthy, two arbitrary clients can bilaterally authenticate one another without having to exchange secret tokens in advance.

The main advantage of digital certificates in correspondence to other authentication methods is its broad use potential and its scalability aspect [15]. First, because a certificate is based on a private and a public key, a certificate not only can be used for user authentication but as well can be used for data integrity and data privacy. Second, the use of digital certificates for authentication purposes scales in terms of both, the number of users and the number of applications. Once a user has been issued a certificate, that very certificate can be used for as many applications as one might think of. With other words, once the required infrastructure for public key cryptography is setup, it can serve for all current as well as future applications.

Obviously, there as well is one main disadvantage. Each client and each service needs to be supplied with an individual certificate. However, the derived distribution and maintenance process is difficult and cumbersome. Certificates rely on private keys whose security is crucial. The concept indeed is only trustworthy as long as the private keys are not comprised. As a result, any single private key needs to be treated with special care. Second, Certification Authorities (CA) must be careful when issuing certificates. It is their responsibility to make sure that certificates (i) are not issued to wrongful principals and (ii) are not used in an illegal or insincere way over time. In summary, we reasonably conclude that issuing certificates is not possible unless there is a considerable bureaucracy involved that though is cumbersome and probably also rather expensive.

B.3. PAM

PAM (Pluggable Authentication Modules) [49] is a software library written in C programming language and is primarily destined for UNIX systems. Its purpose is to offer a standardized API for authentication, authorization and users management. While current implementations still are exclusively available for UNIX-like systems such as (AIX, Solaris or Linux), the underlying principle has been widely adopted for other platforms and/or programming languages. JAAS [48] for example is the Java equivalent that is further explained in appendix B.4. In the following, PAM is explained in term of its applied concept. Readers interested in particular implementations or its concrete usage, are referred to appropriate literature [49].

B.3.1. PAM Architecture

The core component of the PAM framework is a twofold interface, which at the one hand is used by applications in order to ask for suitable authentication modules that they can use as generic delegators, and on the other hand, is used as a central coordinator for the plugin of arbitrary authentication modules. While the former is called front-end and usually is referred to as the PAM API, the latter is called back-end and referred to as PAM SPI (Service Provider Interface). The simplified view of the architecture is given by the following figure.

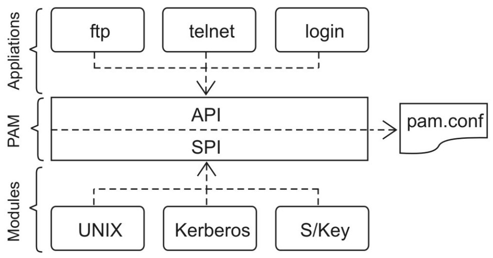


Figure 45. PAM architecture [49]

The figure shows the relation between typically potential applications (ftp, telnet or login), the twofold API and arbitrary authentication modules (UNIX, Kerberos, S/Key). When an application calls the PAM API, it transparently loads the appropriate authentication module, which is configured in the corresponding configuration file. Subsequently, any request from the application is transparently forwarded to the module where it then is performed appropriately. Analogously, corresponding operation results are transferred back to the initial requestor, which is an arbitrary application. The SPI is used by the modules in order to be able to (i) get back to the requestors by means of specific callbacks and (ii) to communicate with other modules if such is required.

Clearly, the architecture achieves full transparency between the two parties (applications and modules). As applications can rely onto the standard PAM API when they need to integrate authentication functionalities, they need neither concern about particular authentication procedures nor about concrete security aspects. On the other hand, a programmer team working on a particular authentication module such as Kerberos or S/Key can safely abstract from particular applications and their individual characteristics. Hence, they can concentrate on their primary work and solely rely onto the standard SPI. A system administrator can use both, applications and authentication modules independently from one another. For example, if a particular business application lives over a long period of time (such is usually the case for legacy systems) and the company eventually changes its global security strategy and for example decided to switch from S/Key to Kerberos, the system administrator can leave the business application untouched but only needs to adjust the corresponding configuration.

B.3.2. Module interfaces

Apart from the illustrated authentication functionality, PAM additionally offers support for account, session and password management [49]. Hence, a pluggable module as depicted in figure 45 does not necessarily need to offer authentication services but as well may particularly be dedicated for one of the these additional aspects. PAM, as a result, defines four independent interfaces that can be applied by modules when appropriate. The four interfaces are depicted in the following table.

Interface	**Offered functionalities**
Authentication Management	Offers pam_authenticate() function to authenticate a user, and pam_setcred() to set, refresh, or destroy a user's credential
Account Management	Offers pam_acct_mgmt() function to check whether authenticated users should receive access to their accounts in terms of expiration and/or access hour restrictions.
Session Management	Offers pam_open_session() and pam_close_session() functions for session management and corresponding logging.
Password Management	Offers pam_chauthtok() function to change a user's password

Table 8. PAM module interfaces

B.4. JAAS

JAAS (Java Authentication and Authorization Service) [48] represents a set of APIs that enables Java programs to efficiently and conveniently deal with authentication and authorization. The architecture is primarily based on the standard PAM (Pluggable Authentication Module) framework described in appendix B.3. In the following, we present an overview of the JAAS architecture and reveal how standard authentication processes are being performed. Note that several simplifications are made in order to be able to keep the discussion in an adequate bound. Interested readers are referred to appropriate literature [48, 36].

B.4.1. JAAS architecture

The JAAS architecture strictly follows the underlying principle of the PAM framework. Indeed, each component found in the PAM framework can directly be mapped onto a particular component in the JAAS architecture. While PAM offers an API to generic applications such as ftp, telnet or login, JAAS's equivalent is defined by the LoginContext interface that is destined to be used by components of arbitrary applications that need to deal with authentication and/or authorization aspects. PAM's SPI (Service Provider Interface) is represented by the LoginModule interface and is applied to particular module implementations. Finally, the configuration capability is addressed by the Config interface. An implementation thereof is prescribed to read a similar configuration file as is used in the PAM framework. While the PAM configuration references particular C modules such as pam_unix.so or pam_skey.so, the JAAS configuration references Java classes that implement the LoginModule interface. The JAAS overall architecture is depicted in figure 46a.

B.4.2. Subject, Principals and Credentials

Apart from the already introduced components LoginContext, Config, and LoginModule, which each relate to an equivalent in the PAM framework, JAAS additionally defines a Subject which is as important and which we will look at in the following.

The Subject represents the user and or the service application that eventually needs to be authenticated. In the following, we use the expression agent in order to refer to a user and/or service application respectively. Because an agent may have several identities that it uses for different situations, the subject contains an arbitrary number of Principals. Each Principal thereby represents a distinct identity. For example, if a Subject happens to represent a student "Alice Bar", the Subject may hold two principles: "alice.bar", the login-name that she uses in order to enter her mailbox, the intranet, etc., and "99-999-999", her unique student id.

A Subject furthermore can be assigned an arbitrary number of public and private Credentials. Thereby, a Credential represents an arbitrary security related attribute. A Kerberos ticket or a public key certificate (X.509, PGP, etc) are typical public credentials. A password or a private key on the other hand are typical private credentials.

B.4.3. Authentication process

When an application component (ApplComp) needs to authenticate an agent, it creates a LoginContext by providing a realm and a CallbackHandler. The realm is a regular character string and refers to a section in the configuration file where the individual security policy of the ApplComp is specified. The CallbackHandler later is used by the LoginModule in order to communicate with the ApplComp transparent fashion. Once the LoginContext is successfully created and initialized, the ApplComp executes the provided login method and waits until the method returns or a LoginException is thrown. In the former case, authentication succeeded and the ApplComp can retrieve the authenticated Subject from the LoginContext. Note that at this point the Subject holds the authenticated Principals as well as optional Credentials. In the latter case, authentication failed and the ApplComp may eventually raise a similar exception or may print an appropriate failure message.

When the login method is called, the LoginContext consults the Config component in order to find out what LoginModule it has to use in this particular context. The LoginContext uses the given realm in order to be able to refer to a concrete security policy. In the next step, the LoginContext instantiates the appropriate LoginModule and provides it with a newly created Subject as well as the CallbackHandler which it requested from the ApplComp in the first place. The login action is then forwarded to the LoginModule that needs to perform the necessary procedures in order to authenticate the agent. We assume that our LoginModule represents a simple application that authenticates on the basis of a username and a password. As a result, the LoginModule's first action is to request the agent's username and corresponding password. Easy to guess, it uses the provided CallbackHandler and executes the handle method with appropriate arguments. The handle method eventually returns a username and a password. If the validation process succeeds, the LoginModule creates a Principal for the provided username and assigns it to the provided Subject. As our module happens to be very simple, there are no Credentials to be added. However, if for example the module represented an authentication facility that relies on public key cryptography, it additionally created appropriate public and private Credentials and assigned them to the Subject in order to provide the necessary certificate keys. After the Subject is adequately updated, the login method returns the action flow back to the LoginContext. On the other hand, if the validation process failed, the method answers with a LoginException. Remember that the subsequent actions for both situations (success and failure) have already been covered above and thus need not be repeated again. The discussed interactions are partially illustrated in Figure 46b.

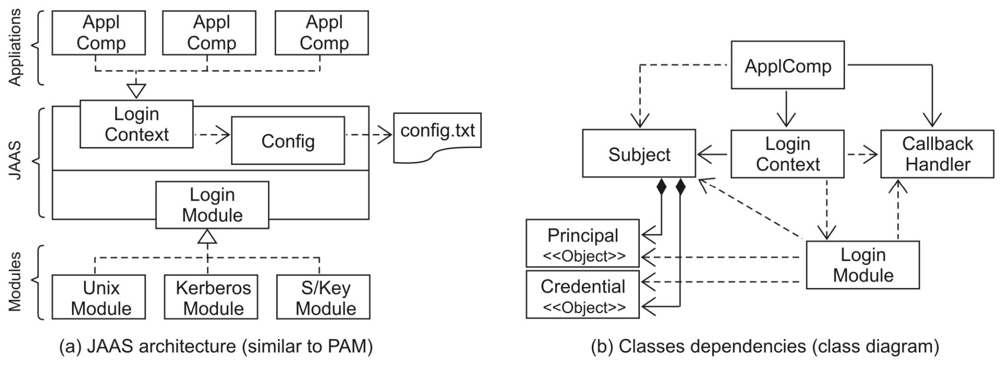


Figure 46. JAAS architecture and overall class diagram

List of Tables

List of Figures

References

[1] Michael Dänzer: *NEXT – The NMR EXperiment Toolbox*, diploma thesis, Department of Informatics, University of Zurich, Switzerland, 2005

[2] Website of the OWL-S API: http://mindswap.org/2004/owl-s/api – last time validated on March 22, 2006

[3] Website of the Mindswap (Maryland Information and Network Dynamics Lab Semantic Web Agents Project) group: http://www.mindswap.org/ – last time validated on March 21, 2006

[4] Official W3C OWL website: http://www.w3.org/2004/OWL/ – last time validated on March 21, 2006

[5] Official OWL-S language specification: http://www.daml.org/services/owl-s/ – last time validated on March 21, 2006

[6] Website of the Jena OWL API: http://jena.sourceforge.net/ – last time validated on March 21, 2006

[7] Official W3C RDF website: http://w3.org/RDF/ – last time validated on March 22, 2006

[8] G. Antoniou, Grigoris Antoniou, Frank Harmelen: *A Semantic Web Primer*, MIT Press, 2004

[9] James Hendler, Tim Berners-Lee, Eric Miller: *Integrating Applications on the Semantic Web*, Journal of the Institute of Electrical Engineers of Japan, Vol. 122(10), 2002, pages 676-680

[10] Official Semantic Web website: http://www.w3.org/2001/sw/ – last time validated on March 22, 2006

[11] Andrew S. Tanenbaum, Maarten van Steen: *Distributed Systems: Principles and Paradigms: International Edition*, Prentice Hall, 2003

[12] Jennifer Vesperman, *Essential CVS: First Edition*, O'Reilly, 2003

[13] Official website of the Dublin Core Metadata Initiative: http://dublincore.org – last time validated on March 22, 2006

[14] Frank Manola, Eric Miller: *RDF Primer*, 2004, http://w3.org/RDF/rdf-primer/ – last time validated on March 02, 2006

[15] Andrew S. Tanenbaum: *Computer Networks: Fourth Edition*, Prentice Hall, 2003

[16] Jason Garman: *Kerberos: The Definitive Guide: Cross-Platform Authentication & Single-Sign-On*, O'Reilly, 2003

[17] Official website of the JXTA project: http://www.jxta.org – last time validated on March 22, 2006

[18] Schahram Dustdar, Harald Gall, Manfred Hauswirth: *Software Architekturen für verteilte Systeme*, Springer, 2003

[19] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywook, Jean-Christophe Hugly, Eric Pouyoul, Bill Yeager: *Project JXTA 2.0 Super-Peer Virtual Network*, http://www.jxta.org/project/www/docs/JXTA2.0 protocols1.pdf – last time validated on March 22, 2006

[20] Official website of the Edutella project: http://edutella.jxta.org – last time validated on March 22, 2006

[21] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, Tore Risch: *Edutella: A P2P Networking Infrastructure Based on RDF*, http://edutella.jxta.org/reports/edutella-whitepaper.pdf – last time validated on March 22, 2006

[22] Website of the Matchmaker project hosted at the Intelligent Software Agents Lab, Carnegie Mellon University, PA: http://www.cs.cmu.edu/~softagents/daml_ Mmaker/matchmaker.html – last time validated on March 22, 2006

[23] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, Katia Sycara: *Semantic Matching of Web Services Capabilities*, http://www.daml.org/services/owl-s/pub-archive/ISWC2002-Matchmaker.pdf – last time validated on March 22, 2006

[24] Michael Wooldrigde: *An Introduction to MultiAgent Systems*, John Wiley & Sons, 2004

[25] M. M. Lehman, L. A. Belady: *Program Evolution: Process of Software Change*, Academic Press, London, 1985

[26] Fabio Casati, Harumi Kuno, Vijay Machiraju, Gustavo Alonso: *Web Services: Concepts, Architectures and Applications*, Springer, 2003

[27] Official website of the UDDI project: http://www.uddi.org – last time validated on March 22, 2006

[28] Alfarez Abdul-Rahman, *The PGP Trust Model*, 1996, http://www.cs.ucl.ac.uk/ staff/F.AbdulRahman/docs/pgptrust.html – last time validated on March 22, 2006

[29] Simson Garfinkel: *PGP: Pretty Good Privacy*, O'Reilly & Associates, 1994

[30] Official website of the Semantic Web Interest Group: http://www.w3.org/2001/sw/ interest/ – last time validated on March 22, 2006

[31] Web or Trust RDF Vocabulary: http://xmlns.com/wot/0.1/ – last time validated on March 22, 2006

[32] Shari Lawrence Pfleeger, Joanne M. Atlee: Software *Engineering: Theory and Practice: Second Edition*, Prentice Hall, 2001

[33] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar, Neveen Srinivasan: *Automated Discovery, Interaction and Composition of Semantic Web services*, 2003, http://www.cs.cmu.edu/~softagents/papers/websemantics2003.pdf – last time validated on March 22, 2006

[34] Official website of VCard and VCalendar hosted by the Internet Mail Consortium (IMC): http://www.imc.org/pdi/ – last time validated on March 22, 2006

[35] Reference description, version 1.1, of the Dublin Core Metadata Element Set: http://dublincore.org/documents/dces/ – last time validated on March 22, 2006

[36] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, Roland Schemers: *User Authentication and Authorization in the Java Platform*, published in the proceedings of the 15th annual computer security applications conference, Phoenix, AZ, 1999, http://java.sun.com/security/jaas/doc/acsac.html – last time validated on March 22, 2006

[37] Martin Glinz: *Software Engineering I*, University of Zurich, 2003

[38] Michael K. Smith, Chris Welty, Deborah L. McGuinness: *OWL Web Ontology Language Guide*, 2004, http://www.w3.org/TR/owl-guide/ – last time validated on March 21, 2006

[39] Bill Joy, Guy Steele, Gilard Bracha, James Gosling: *The Java Language Specification: Third Edition*, Addison Wesley, 2005

[40] Erich Gamma. Richard Helm. Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Design*, Addison-Wesley, 1995

[41] Elliotte Rusty Harold, W. Scott Means: *XML in a nutshell: A Desktop Quick Reference: Third Edition*, O'Reilly, 2004

[42] Grady Booch, James Rumbaugh, Ivar Jacobson: *UML2 and the Unified Process: Practical Object-Oriented Analysis and Design: Second Edition*, Addison-Wesley, 2005

[43] John Cupri, Dan Malks, Deepak Alur: *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2001

[44] Dominik Gruntz, Stephan Murer: *Component Software: Beyond Object-Oriented Programming: Second Edition*, Addison Wesley, 2002

[45] William Stallings: *Operating Systems: Fifth Edition*, Prentice Hall, 2004

[46] Official website of the Friend of a Friend (FOAF) project: http://www.foaf-project.org – last time validated on March 21, 2006

[47] Serge Demeyer, Stephane Ducasse, Oscar Nierstrasz: *Object-Oriented Reengineering Patterns*, Morgan Kaufman Publishers, CA, 2003

[48] Sun's official JAAS website: http://java.sun.com/products/jaas/ – last time validated on March 22, 2006

[49] Vipin Samar, Charlie Lai: *Making Login Services Independent of Authentication Technologies*, http://java.sun.com/security/jaas/doc/pam.html – last time validated on March 22, 2006

[50] National Computer Security Center: *A Guide to understanding Discretionary Access Control*, 1987, http://www.radium.ncsc.mil/tpep/library/rainbow/CSC-TG-003.html – last time validated on March 22, 2006

[51] Website of Pellet: http://www.mindswap.org/2003/pellet/ – last time validated on March 21, 2006

[52] Renato Ianella: *Open Digital Rights Language, Version 1.1*, 2002, http://www.w3.org/TR/odrl/ – last time validated on March 22, 2006

[53] International DOI Foundation: *DOI Handbook, Version 4.2.0*, 2005, doi:10.1000/186, http://www.doi.org/handbook_2000/DOIHandbook-v4-2.pdf – last time validated on March 22, 2006

[54] Yeh Ching-Long, Lin Ruei-Feng: *Design and Implementation of an RDF Triple Store*, 2002, http://datf.iis.sinica.edu.tw/Papers/2002datfpapers/sessionB/B-3.pdf – last time validated on March 22, 2006