AGH University of Science and Technology

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

Institute of Computer Science

Jakub Wach

Collection and storage of provenance data

Master of Science Thesis

Supervision
Dr. Marian Bubak

Consultancy
Bartosz Balis

Krakow, June 2008

Abstract

The subject of this thesis is collection and storage of the provenance data in a Grid system. Provenance is defined as *derivation path of a piece of data*. Nowadays Grid systems are equipped with tools and components forming collaborative space for science, called virtual laboratories. These modern scientific environments allows for executing *in silico* experiments in such disciplines as biochemistry, astronomy or quantum physics. In each of those cases, scientists are highly interested in resemblances between experiments and results, tracing data entities or attaching metadata to obtained results. All these requirements can be fulfilled by tracing, storing and querying provenance in the system.

This thesis presents **PROToS** - provenance tracking system designed to meet specific requirements of the ViroLab virtual laboratory. It is based around semantic modelling of provenance and system's data and motivated by the Semantic Grid vision. Apart from design and implementation of the PROToS, also integration in challenging environment of the ViroLab is presented.

Contents of this thesis is organized in chapters, as follows. First chapter introduces subject of this work, presenting motivation and objectives to be achieved. Second chapter describes background of the thesis, that is provenance, the ViroLab system and its virtual laboratory. Third chapter presents most important existing provenance systems along with brief analysis. Fourth chapter defines requirements for provenance tracking system to be created. Fifth chapter overviews architecture of the system. Also, identified use cases and project organization are presented. Sixth chapter details PROToS design and implementation, describing also technologies used. Seventh chapter shows PROToS environment in the ViroLab virtual laboratory along with examples of real-world provenance usage. Eighth chapter is devoted to project's status and future work.

Key words

provenance, grid, semantic web, ontologies, semantic integration, data mining, ViroLab, virtual laboratory

Acknowledgements

I would like to express my gratitude to dr Marian Bubak, supervisor of this thesis for his invaluable help, commitment and time.

This work was made possible thans to the ViroLab and CoreGrid projects.

http://www.virolab.org/ http://www.coregrid.net/

Contents

Chapter 1

Introduction

This chapter provides a rationale for this work. First section contains brief motivation for research in provenance tracking field. Second one presents list of objectives to be filled. Finally, the organization of this thesis is given.

1.1. Motivation

Tracking origins and derivation paths of data (the provenance) in large scale, high level system is recently gaining more interest. It is especially emphasized since e-Science has become popular. The e-Science term [67], popularized by John Taylor means new kind of scientific research, backed by the next generation infrastructure. It is typically identified with Grids, offering virtualization of resources [70] and collaboration over virtual organizations [62]. A number of initiatives emerged to extend science Grid systems with needed capabilities. An example could be the myGrid [74] project. In this system, application data, workflow templates and annotations including provenance information are stored in the common information repository. Provenance in myGrid is generated from workflow execution events and involves limited semantic in form of ontology annotations. Other attempts to provenance tracking include Karma provenance framework [72] and Virtual Provenance Data Model [77]. However, all these solutions have drawbacks and limitations. In most cases they are too narrow-minded or tightly integrated with particular system, being hardly usable in different environments. Moreover, the future of e-Science lies in the *Semantic Grid* [59] so its ideas should be also incorporated.
Thus, need for profound research in this area still exists. So research should end with design of provenance model and accompanying system suitable for broad range of Grid systems of new era - the e-Science.

1.2. Objectives

Goals of this work of primary importance can be summarized as follows:

1. research and design of semantic provenance model applicable in wide range of Grid systems, enabling not only gathering of provenance data but also complex mining queries over the data
2. perform research on requirements and possible applications of provenance system in collaborative science space for bioinformatics, like the ViroLab virtual laboratory
3. design and implementation of the provenance tracking system for the ViroLab virtual laboratory environment
4. research and design of provenance system integration within the ViroLab environment, involving necessary external components and interfaces

The provenance tracking infrastructure being developed, should be based on current state of the art solutions, following best patterns and fixing identified shortcomings. Research done in this field is summarized in section 3.
Work of this thesis is bound to the ViroLab and its specific requirements. Yet, provenance model to be designed should be generic enough to allow provenance tracking in other environments. Therefore, additional source of provenance usage scenario should be studied, as those from First Provenance Challenge [60]. Insight in various provenance usage scenarios is given in sections 4.1 and 2.3.2.
What is more, designed and implemented provenance tracking component has to be integrated in productional, final ViroLab system. Thus, has to prove itself in challenging and technically advanced environment of Grid system. System's architecture shall be prepared for implementing necessary reliability and performance improvements easily. Also, implementation with industry standards and proven libraries should help system achieve this goal. Design and implementation details are covered in section 6. Technologies behind system's prototype are presented in section 6.2.
In summary this work concentrates on

- performing thorough study of provenance modelling and tracking requirements in modern virtual laboratories
- design and implementation of provenance tracking system prototype, fulfilling the ViroLab user's requirements

1.3. Organization of this document

The remainder of this thesis is organized as follows. Chapter 2 is devoted to background of the thesis, introducing provenance, the ViroLab system and its virtual laboratory. Chapter 3 gives overview of existing provenance systems and presents brief analysis of theirs strong and weak spots. In chapter 4 requirements for the ViroLab provenance system are defined. Next chapter - 5 contains overview of system's architecture along with use cases and project organization in Maven2. Chapter 6 is completely devoted to details of the PROToS design and implementation, describing also technologies used along with explanation for choices made. In chapter 7 examples of provenance real-world usage are presented. Moreover, description of PROToS integration in the ViroLab virtual laboratory is given. Last chapter, number 8, contains information regarding project's current state and future work to be done. Appendices contains as follows: A - detailed guide to system's configuration, B - sample deployment of PROToS, C - administrator manual and D - data storage / retrieval manual.

Chapter 2

Background

This chapter presents background of this thesis work - the provenance tracking system. First section introduces new tools for e-Science, virtual laboratories. Successive section presents overview of the ViroLab and its virtual laboratory. Next, basic information about provenance and its usage is given. Finally, Grid technologies constituting virtual laboratories are presented.

2.1. Virtual laboratories

Virtual laboratory can be defined as multiple, integrated components forming collaborative space for science. Before the ViroLab, few other project adopted this approach for conducting experiments, processing workflows and constructing Grid applications. Below we present state of the art in the virtual laboratory area. **Kepler**.

Kepler [22] is a system created for constructing workflows. Most important feature of Kepler is advanced environment for visual building and execution of workflows. They are described in the MoML language, destined for modeling workflows as clustered graphs. This representation brings following advantages:

- Implementation independent - being based on XML, MoML is designed to work with any tool.
- Semantic independent - MoML itself does not carry semantic information about interconnections between components, offering "director" mechanism instead.
- Integrated with Web - as MoML is based on XML with syntax similar to the commonly used HTML.
- Rendering support - MoML models can contains annotations (hints) for rendering utilities.

Workflows in the Kepler are constructed from following component's classes: directors, actors, relations and ports. Actors encapsulate functionalities, as calls to Web Services and Globus Jobs executions. Moreover, actor can encapsulate whole other workflow.
Kepler's virtual laboratory serves good users without technical knowledge, enabling simple drag and drop workflow construction. Albeit easy to use, this approach is not suitable for more complex workflows / experiments, containing many loops and branches. **Taverna**. Taverna [34] is experiment construction workbench used in the

myGrid project. Experiments in Taverna are written with usage of the Simple Conceptual Unified Flow Language (Scufl). This language enables describing conceptual tasks as a single entities without implementation particulars. Scufl workflows are built of services and Taverna environment provides following means of accessing them:

- WSDL files from local file system
- Services used in already existing workflows
- Standard UDDI registry
- WSDL files from remote locations, pointed out by URL
- Specific myGrid registry, called GRIMOIRES

This virtual laboratory is very popular, with more than 1000 services available. Nonetheless, it can be pointed out that script-based experiment definition can be much more productive than visual drag-and-drop one. **Triana**. Triana [35] is a

problem-solving environment, enabling easy workflow construction by using user-friendly GUI. It allows for drag-and-drop building blocks of the workflow and define connection between them. Also, user can edit workflow blocks and set adequate parameter values. Workflow elements can take form of local operations, Grid jobs and remote Web Service calls. Also, dynamic WS discovery and invocation is possible.
In conclusion Triana is simple and easy to use, but shares same disadvantages as other "drag-and-drop" workflow construction environments.

2.2. ViroLab

2.2.1. Introduction

ViroLab [73], site: [37] is EU-funded project number 027446 from 6th Framework Programme.
Its main mission is to develop a virtual laboratory enabling decision support in viral disease treatment. Main ideas behind this project are:

- Integrating distributed medical knowledge to facilitate research and treatment in virology diseases field. Nowadays, many large clinical patient databases are available. These can be used in various tasks, as discovering drug susceptibility. Great challenge of the **ViroLab** is to provide uniform, user-friendly access to the data for members of the medical scientific community.
- Providing users with complex tools taking advantage of available medical data. These vary from simple drug ranking to very advanced automata model of the HIV-1 co-receptor tropism. Every application is to be available in user-friendly

and unified manner. What is more, one of most important feature of the **ViroLab** is combining applications in workflow-like experiments.
- Basing on Grid architecture. Virtualization of the hardware, computing infrastructure, databases and services - in summary provides powerful environment for running bioinformatic applications. Bridge to infrastructure like clusters and EGEE grid offers enough computing power.

The **ViroLab** project organization is split on several work packages, listed as follows:

1. **Project Management**
   Takes care of financial and administrative management of the ViroLab consortium. Ensures on-schedule execution and communication with sponsor - EU Commission.
2. **Virtual Organization**
   Responsible for building security infrastructure for the Grid, presentation layer and middleware.
3. **Structure of the ViroLab virtual laboratory**
   Most important work package, developing virtual laboratory, concerning such aspects as uniform data access, user session management, experiment execution, resource brokering, user collaboration and provenance.
4. **RetroGram: Virtualization, Enhancement and Individual Based Interpretation**
   Handles development of the distributed decision support system, based on existing drug ranking software. Includes tools for detailed studies on patients and obtained treatment results.
5. **Population and epidemiological based interpretation system**
   Responsible for carrying development of expert rules for clinical patient treatment. Will also validate the ViroLab results basing on epidemiological studies.
6. **Dissemination**
   Takes care of presenting ViroLab results to the public.

As stated, virtual laboratory is critical work package from end-user's point of view. Also, this is where provenance tracking system lies. Thus it is presented in separate section 2.2.2.

2.2.2. Virtual Laboratory

The **ViroLab** virtual laboratory (**Vlvl** [66] and [57], main site: [38]) is tool for collaborative planning and executing *in-silico* experiments. Enables sharing, annotating, discussing and saving results of these experiments. With provenance tracking enabled, mining results traces and exploring resemblances between experiments is possible. Virtual laboratory provides tools for for writing experiments's plans (**Experiment Planning Environment, EPE**) and managing experiment's execution (**Experiment Management Interface, EMI**).
Experiment is main concept behind Vlvl. It is defined as process combining data and services (activities) processing that data to obtain results. Data and services are not restricted to a local machine, but can come from multiple, distributed resources. In the Vlvl, experiment's lifecycle has following stages:

- **Planning**
  In this stage, *Developer* creates and delivers valid **experiment plan**, containing experiment identification (name and version), local input files and libraries, legal information and most important - **experiment script**. It is a program written computer programming language, interpreted by the VLvl components. Current version of the Vlvl uses JRuby [21]. Script defines services and data used along with control flow. It can be said that script constitute heart of the experiment. VLvl provides also **Experiment Repository** for storing and versioning experiment plans.
- **Execution**
  In this stage *Experiment Users - Scientists and Virologists* performs experiment according to defined textbfplan. It is done by executing provided **script** in one of two modes:
  — local - requiring local installation of the runtime software. Script is passed to the runtime by command-line tool.
  — remote - allowing to run experiment on remote runtime server by using the ViroLab portal tool (EMI) or development tool (EPE). In this case it is required that experiment plan is available in the Experiment Repository.

  Different execution modes adds required flexibility to the Vlvl environment, fulfilling needs of any type of user and organization. For example, local execution is suitable for testing purposes whether remote mode is convenient for long-running complex experiments.
- **Result management**
  In this stage user can evaluate, annotate and store outcome of his experiment. This is very important as enables strong collaboration between scientists and lays foundations for tracking provenance.

To sum up, experiment lifecycle defined for the Vlvl supports collaborative work of all types users, from developers to clinicians.

Fig. 2.1 depicts abstract layers of the **Virtual Laboratory**.

This conceptual architecture consists of following:

- **Users** of the system, acting in experiment lifecycle stages described above.
- **Interfaces**, representing tools dedicated to particular user's groups. These include mentioned earlier EPE - used by Developers, EMI and application-specific components running inside the ViroLab portal, used by Scientists and Virologists.
- **Runtime**, constituting bridge between interfaces and various services, both computational and data. Runtime components allows for selecting resources and use them in experiment's execution.
- **Services** performing computations and accessing distributed data sources. First type can point to Web Services, WSRF, components or grid jobs. The latter provide access to relational databases, files and other sources, all in unified and vitalized way.
- **Infrastructure** layer constitutes physical layer where all services run. Virtual laboratory supports multiple solutions, ranging from single PC machines to large Grid testbeds as **EGEE** and **DEISA**.

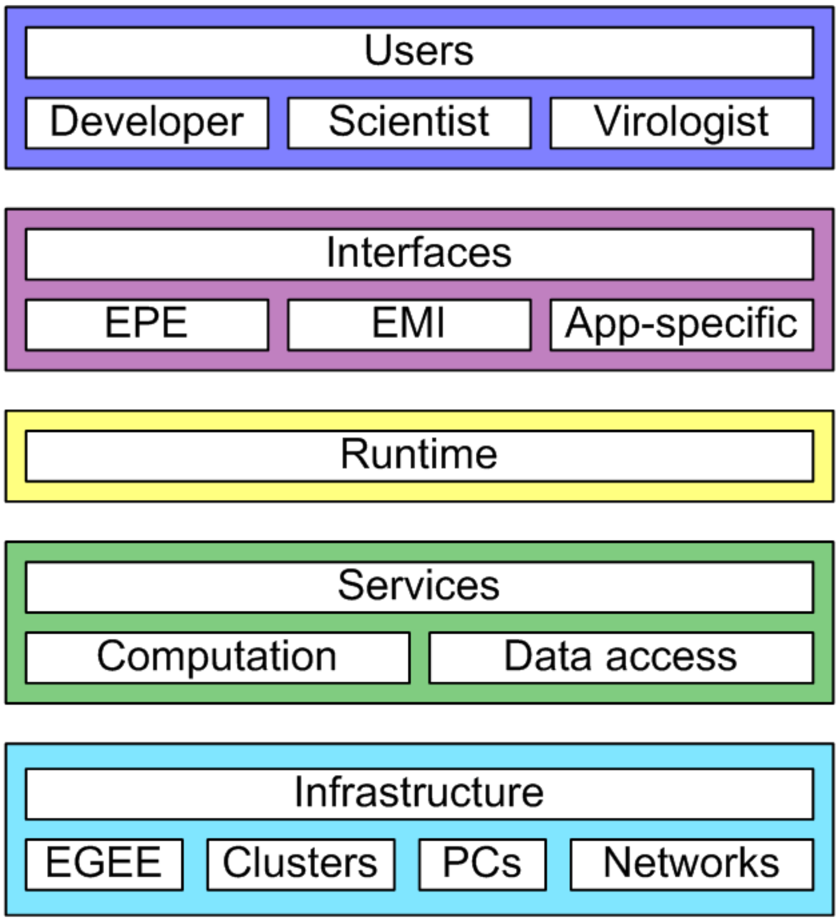


Figure 2.1. Conceptual layers of the ViroLab virtual laboratory. Figure does not reflect real, complex architecture of the VLvl, but rather presents how components are grouped.

Above description presents only background of the ViroLab virtual laboratory. In fact, its architecture is far more complex. Thorough design and manual of the VLvl is to be found on the web [38].

2.2.3. Virtual Laboratory Applications

The ViroLab Design Deliverable document contains detailed description of some important applications, prepared to run in the ViroLab virtual laboratory environment. These are very important, being first source for later requirements analysis and specification for various ViroLab's sub-systems, as provenance tracking.

- Rule-based Decision Support System (Drug Ranking System; DRS)
  This application helps clinical virologists chose drugs most efficient for treating patients. It is done by using publicly available, high quality databases relating virus genotype to drug-susceptibility. To obtain personalized healthcare for a patient, virologist should only enter list of virus mutations and use one of available rule sets. Application also allows for commenting rule sets and results, thus sharing knowledge between application's users. Rule sets evolve in time, so one of appli-

cation's features are automatic updates. On each request, the DRS will contact remote databases, check for rule set's updates and eventually download them. What is more, the DRS application can also be used as shingle step in more complex experiments.

- From Genotype Information To Drug Resistance Interpretation
  Scenario for this application extends the Drug Ranking System's one. It allows virologists to interpret bare HIV RNA strands. Application's steps include:
  — translation of the nucleotide sequence to the amino acid one. Results will be available in some popular formats, for user's convenience.
  — comparison of the nucleotide sequence to reference strains. As as result, mutations per-gene will be obtained. Reference sequences shall be obtained from external databases.
  — identification of the HIV virus subtype, based on amino acid sequences obtained in previous step.
  — drug resistance prediction, handled by the DRS application.

  Also, virologists would like to attach overviews and statistics to results obtained in the application. Finally, summary of all mutations per codon should be provided. As in case of previous application, also this has to deal with different data formats. Therefore, conversion to the common format will also be provided.
- Establishing Large Databases of HIV Sequences
  This application concentrates on serving HIV sequences data from various, distributed sources. Scenario of this application reuses some components from the previous application. It consists of three main steps:
  — data is **gathered** from sources available in the Grid
  — data is **processed** - this is where previously described components enter. Main transformations to be applied are mutations and substitutions identification, sequence alignment and subtyping.
  — data is **exported** to common format and made available to system's users.
- Data Retriever (and applications accessing hospital data)
  The application is build to gather various hospital data and present results as combined datasets available on the Grid. What is more, these datasets are to be presented in an unified format.
  Main purpose of this application is accessing internal databases of hospitals. These are scattered and built with different standards. Also, various security restrictions apply, mostly related to protection of patient's personal data. Proposed solution overcomes these issues with so-called sandbox environment.

2.3. Provenance

2.3.1. Definition

From Merrian-Webster Online Dictionary [43]:

Provenance - origin, source. Comes from French - *provenir* to come forth, originate, from Latin *provenire*, from *pro-* forth + *venire* to come more at.

In computer science applications, provenance is defined as:

The provenance of a piece of data is the process that led to the data.

Other definition is:

A derivation path of a piece of data.

In complex Grid systems, as the ViroLab, provenance could include almost everything action of user or component, such as:

- experiment's call to a computational service, including type and identifier of arguments and obtained results
- data load and store, realized by specific services and including particulars of used data source (type, protocol, physical machine, geographical location)
- internal calls of system's runtime, involving such details as class and implementation of used Grid Object, SQL/HQL/OQL query sent to a data source or concrete computing resource used (host, port, architecture, OS..)
- abstract events describing complex workflow concerning particular domain, as *'New drug ranking in the DRS application'*
- Gird monitoring data, involving scheduling time, service call performance and others
- User's actions involving GUI controls, defining abstract actions as *'experiment run'* or *'result save'*
- parts of applications scenarios, as attaching annotations to data is sometimes treated as a provenance (*metadata*)

All items listed above could be reckoned as provenance definition for the complex virtual laboratory system.

2.3.2. Possible applications of provenance in virtual laboratories

Virtual laboratory as described in section 2.2.2 can constitute first base for possible provenance application scenarios.
As stated in respective section, **experiment** and its lifecycle are most important concepts behind virtual laboratories. Following scenarios presents how provenance could be applied along with experiment usage.

1. **Data trace**. When an experiment is executed, outcome of some processing could be saved as result. Next, obtained results could be used in consequent experiments as input of various services, producing another results. When provenance tracking is enabled, such (common) scenario builds data graph, where data entities are vertices and processing constitutes edges. Using gathered provenance data, user can receive answer for queries like:
   - *from what pieces of data this piece was derived?*
   - *how often particular service was called to obtain this piece of data?*
   - *what pieces of data were derived from particular piece / pieces of data?*
   - *how many operations (service calls) were required to obtain data X from data Y ?*

   Browsing provenance data graph can reveal much more information about data and service dependencies than shown with example queries. What is more, powerful techniques as statistical analysis are able to explore even greater level of detail.

2. **Experiments resemblances**. Many experiments conducted in virtual laboratory will be from particular domain, as HIV infection treatment. Using provenance data gathered from many experiments of one type, user is able to discover similarities between different traces. For example, following queries are possible:
   - *what are most common data entities used for particular operation?*
   - *how many steps are typically required to obtain data X ?*
   - *which experiments were conducted on data from particular location (hospital)*
   - *who conducts experiment on patients with particular disease* (hence: who could help solve similar problems in treatment)

   As presented, these queries are typically enhanced with similarity operators ('like', 'typical' and so on).
3. **Annotation storage**. Ability to send provenance data at any stage of experiment's lifecycle could be used for attaching annotations to actions, data entities and service calls. Metadata attached could vary from text descriptions of experiments or results to rating of application's accuracy. The latter, when used properly, can add another level of usefulness to virtual laboratory applications. For example, previous queries could be rewritten as:
   - *what are highest ranked data entities used for particular operation?*
   - *what are best operations required to obtain data X?*
   - in bioinformatics domain: *what is the best treatment for particular disease?*

   This way, simple data and services annotations are enriched with semantic meaning.
4. **Experiment repeat**. Provenance record of the experiment execution can be used for later repeat of exact or similar experiments. Even if experiment's plan is lost, with usage of the provenance it can be restored.
5. **Experiment replay**. Exhaustive gathering of provenance data, as described in previous section leads to full record of processes. Using this data, user is able to perform smart experiment replay, starting from chosen time point. What is more, specific virtual laboratory component could be designed to perform such replay automatically. Such usage could speed up complex experiments execution, by starting new computation after time-consuming part.

Apart from virtual laboratory, another source of possible provenance applications can be the Provenance Challenge [28]. Scientists from all around the world, involved in provenance development agreed that mining over large sets is useful and required. To test capabilities of existing provenance systems, specific FRMI (Functional Magnetic Resonance Imaging) workflow was defined along with set of useful queries concerning this workflow.
Workflow is composed of following operations:

- **align_wrap** - compares new image with reference one, determining how new image should be adjusted to match reference brain.
- **reslice** - transforms new brain image according to the parameter set - output of the align_wrap operation.
- **softmean** - all images resliced by previous operation are averaged into single image.
- **slicer** - creates 2D atlas data set from the averaged image.
- **convert** - transforms 2D atlas data set to specific graphical atlas image.

Paper [60] summarizes works done by the First Provenance Challenge. Queries presented in this work are very good examples of what scientists require from the provenance system side. Example queries follows:

- *Find the process that led to Atlas X Graphic* (thus retrieving full provenance of given piece of data)
- *Find the Stage 3, 4 and 5 details of the process that led to Atlas X Graphic* (thus retrieving partial trace)
- *A user has annotated some anatomy images with a key-value pair center=UChicago. Find the outputs of align_warp where the inputs are annotated with center=UChicago.*
- *A user has annotated some atlas graphics with key-value pair where the key is studyModality. Find all the graphical atlas sets that have metadata annotation studyModality with values speech, visual or audio, and return all other annotations to these files.*

As shown, Provenance Challenge queries are quite similar to those presented for VLvl.
In summary quick analysis of possible provenance applications in typical scientific-driven Grid environment shows provenance's great potential. Furthermore, some applications are necessary in modern virtual laboratories and can not be achieved without full provenance tracking.

2.4. Grid computing

All contemporary virtual laboratories are Grid-aware. This is because Grid-enabled Service-Oriented Architecture integrating and virtualizing resources fits best requirements of automated application creation.
History of the Grid dates back to the 1990s, when scientists put their interest in the new idea of a virtual supercomputer. First significant summary of new infrastructure appeared in 1998, when Ian Foster and Carl Kesselman published [68]. They defined the Grid as:

> ..a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities.

Since then, may other Grid system types emerged, as Data Grid, Collaboration Grid or Network Grid. Principles that should be followed by all Grid systems were defined by Foster in his next article [61]. He lists following rules:

- Built with open, general standards in such fundamental aspects as resource discovery and access, authentication and authorization. This is crucial to achieve system that is inter operable and available in global scale. What is more, standards assure that system is general-purpose, not application specific, thus able to integrate and use multiple resources.
- Delivering miscellaneous qualities of service from every dimension as security, performance and reliability, to meet user requirements. This should allow system to be much more usable than simple combination of it's elements.
- Integrating and coordinating resources from different control domains, taking care about such issues as security, payments or policy settings.

Contemporary definitions, as those provided by Rajkumar Buyya in [58] tends to view Grid as an distributed infrastructure, that enables integration, aggregation and sharing of various, autonomous, geographically distributed resources (as computers, networks or data), chosen dynamically in order to fulfill users' quality of service demands.

Typical architecture of the Grid system, designed to accomplish all goal mentioned above, is divided into four virtual tiers, as presented in Fig. 2.2. Brief description follows:

1. Applications. Grid applications are created with usage of services provided by lower middleware tiers to access resources and perform computations. Nowadays, many applications are developed as portlets and deployed in grid portals, such as Gridsphere [14]. Those portlet containers provides additional level of user-oriented services, as programming interfaces to common resources and security entry point.
2. User middleware. Uses lower tier - system middleware to provide higher level services required by the user. This includes application development tools, as compilers or debuggers with necessary libraries. Provides also so-called resource brokers, managers for Grid resources and processes.
3. System middleware. This tier does most of the work that identify system to be a Grid. Provides uniform method of accessing distributed resources from the heterogeneous fabric tier. Takes care about resources discovery and registration. Manages computational processes, scheduling and optimizing them to achieve best service and resource utilization. Finally, assures quality of service demanded by the system user.
4. Fabric. This tier makes up physical background of the Grid system. It consists of networks connecting physical machines (of any kind, form PCs to supercomputers) and data sources. Everything in this tier is basically called a resource.

As stated before, middleware tier is the most important one in mission to provide a fully-fledged Grid system. Thus, standards concerning this tier have been established. Initial one was created by the Open Grid Forum and called Open Grid Services Architecture (OGSA, [24]). It is based on Web Services technologies (as SOAP and WSDL) and addresses key services, as security, execution and resources management, information and data. Same organization published in 2003 new standard - Open Grid Services Infrastructure (OGSI, [25]). OGSI was meant to constitute infrastructure layer for OGSA, by essentially extending standard Web Services with statefulness. Later this standard became obsolete in favor to the Web Services Resource Framework (WSRF, [46]). This new standard, introduced by the OASIS in 2004 is in fact family of complex specifications defining what operations could be implemented by Web Services to become stateful. This complexity, especially concerning identification of the WSRF-enabled services with WS-Addressing [45], raised great deal of controversy and resulted in slow adoption of the standard in the Grid community.
Describes standards would have little impact without proper, open implementations. There are several Grid frameworks, implementing those standards or subsets. Most notable are listed below.

- Globus Toolkit [11]. Established in 1995 and now developed by the Globus Alliance, Globus Toolkit is oldest and most popular open source framework. Currently at version 4, offers support for OGSA / OGSI, WSRF, WS-Management and stateless

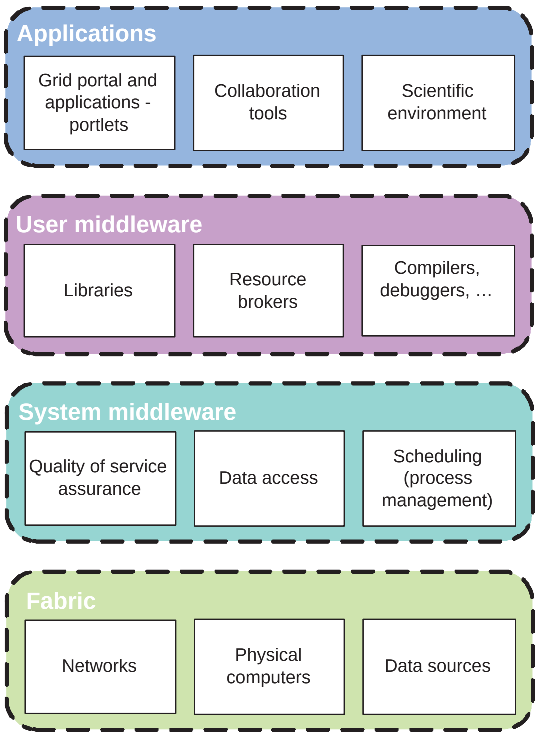


Figure 2.2. Typical Grid system architecture virtual tiers. This is rather virtual model presenting which components should be present in a Grid system and they should be grouped.

Web Services technologies (WSDL, SOAP). Primary use of the Globus is development of computational grid middleware and grid based application requiring stateful Web Services.It consists of four components:
— Grid Security Infrastructure (GSI), based on X.509 Certificates, Public Key Infrastructure and SSL to provide authorization, resource authentication, encryption and single sign-on for Grid services.
— Globus Resource Allocation Manager (GRAM), a uniform interface to various local schedulers (as LSF), providing remote execution features.
— Monitoring and Discovery Service (MDS), used to publish and discovery resource properties, as nodes capabilities.
— GridFTP - an extension of the FTP protocol for reliable, secure data management in the Grid environment.

At present many large, productional Grid systems use Globus in middleware tiers, for example CERN grid an US TeraGrid.

- UNICORE [36]. UNiform Interface to COmputing REsources (UNICORe in short) project was initiated in 1997 as an middleware solution alternative for the Globus Toolkit.From version 6 it partially supports WSRF standard, including WS-Resource-Lifetime but without full implementation of WS-Notification. UNICORE architecture is divided on three layers:
— User, accessing resources by running the UNICORE Client on a local machine. It's interface is made from Job Preparation Agent and Job Monitor Agent.

  These components are used to compose, submit and check status of jobs running in the UNICORE-enabled Grid.
  - Server, bound to a specific organization, defining so-called Usite, managing connected UNICORE Clients. Usite consists of Gateway (entry point for Clients), User database, Incarnation database (defining commands suitable for every available target system) and Network Job Supervisor.
  - Target System, offering computational power and resources. Target systems are organized as Vsites. Each Vsite consists of two components: Target System Interface and Batch Subsystem. Those components cooperate in executing passed jobs on systems belonging to a Vsite.

  Since UNICORE development was funded by the UE, many other UE funded projects makes use of it. Among others, most notable ones are OpenMolGRID, EUROGIRD or fresh Chemomentum project [6].
- Gridbus [13]. This project was founded by the University of Melbourne with focus on eBusiness application (thus name GRID BUSiness). Nowadays is open source project, developed by multi-institutional consortium. It provides technologies for various applications, ranging from cluster economy to portals and simulation. Current version support WSRF-compliant services along with standard, stateless Web Services.
  Gridbus applications includes such projects as NeuroGrid, HydroGrid and Australian Virtual Laboratory.

What should be expressly noted, many present-day Grid projects uses only parts of mentioned middleware solutions, as OGSA-DAI components from the Globus or simply WSRF implementation. This is because developers have to build solutions for particular needs, as virtual laboratories for e-science. Those Grids not necessarily fits into middleware model proposed by off-the-shelf solutions, having need for extra components and functionalities, like a provenance tracking system.

Chapter 3

Overview of provenance systems

First section of this chapter is devoted to overview of existing provenance-enabled systems and solutions. Each important system is briefly described. Second section provides discussion of mentioned systems with strong and weak spots emphasized.

3.1. Existing provenance systems

As stated before, provenance tracking in Grid systems is becoming very hot topic lately. This section contains brief analysis of existing systems and theirs weak points. It is based mainly on papers referred in text.

1. **myGrid** [74] and [76]
   In myGrid system, provenance is represented with usage of semantic web technologies: RDF and ontologies. Data is collected on different levels: process, data, organization and knowledge. Main sequence of provenance generation is composed of two stages:
   - Workflow execution generates events.
   - Postprocessing annotates logged provenance events with semantic concepts. They are taken from ontology description of services used in workflow execution.

   What should be expressly noted, authors of myGrid concentrates on provenance browsing capabilities. Easy construction of complex mining queries is a little bit difficult in this model. Furthermore, browsing as implemented in myGrid does not allow queries to cross provenance domain and query for application data.
2. **Virtual Data Provenance Model** [77]
   This work concentrates on creating fully-functional provenance model with high querying capabilities. Provenance information is presented as falling into one of two types:

- *prospective* - describing workflow (modeled as procedure) to obtain piece of data.
- *retrospective* - describing execution environment of a procedure (runtime properties, resources used).

Authors of the VDM state that only *prospective* provenance constitutes trace of a piece of data. However they also find that *retrospective* information is required for complete overview of the data. Information about data process environment is of great value in data preparation and analysis in science-driven systems.
In fact, VDM defines provenance of a piece of data as a functional procedure that was used to produce it and can be used to reproduce it. Moreover, data - procedure association fidelity allows for later re-execution of process leading to particular piece of data. In addition, metadata can be associated with datasets, procedures, arguments, calls and workflows. Those annotations take form similar to the RDF [29], that is triples subject – predicate – object. Subject is one of the five entities to be annotated, predicate is the name of metadata entry and object contains actual value. All data is stored in relational model, therefore information about operations (such as argument) is represented by pure strings. Also, datasets used by procedures (workflows) are stored by names. This way, all semantic information about data is lost.
As far as querying capabilities are concerned, authors of the VDM distinguish three query types:

- *virtual data relationship queries* - core queries of the model. Focuses on prospective procedures and retrospective logs of procedure's calls. Serves queries such as: *find procedures and calls by given name*, *find calls by given procedure*, *find jobs running for more than specified time* - retrospective one.
- *annotation queries* - queries making use of annotations capabilities of the VDM. What is more, with this query type, user can select VDM objects (as dataset or workflow) annotated with specified meta data entry. Servers queries such as: *select all annotations for defined object* or *find all objects with annotations having defined predicate.* present)
- *lineage graph queries* - making use of lineage relationship, as described in [75]. Can be derived for all data entities. Serves complex queries such as: *find datasets derivation path* or *find all ancestors of some dataset.*

Model allows for more complex queries as combinations of above types. These are described as *Provenance Queries in Multiple Dimensions.* Unique feature of the VDM is the ability to **update** provenance database. Provenance data can be modified or enriched with new information, such as procedures or annotations. Queries of this type are referred to as *Modification and Composition Queries.*
All queries can be expressed in commonly used SQL language, which is an advantage of using relational model for provenance data.

3. **PROVENANCE project** [64]
This very interesting project aimed to define provenance suited for SOA and build adequate architecture on the top of the definition.
Authors presents concept of *p-assertions.* Each *p-assertion* represents stage in a

process and is to be submitted by some actor involved in the process. Following groups are distinguished:
- interaction p-assertions - document data flow between actors
- relationship p-assertions - document data flow within actors
- actor state p-assertions - documents internal state of an actor

Albeit presented model is interesting from theoretic point of view, it should be pointed out that it is rather suited for being background of more complex model.

4. **Karma provenance framework** [72]
This framework is destined for workflows in SOA-based systems. In short, it allows collecting following two of provenance information:
- process - known also as workflow trace. Documents workflow execution and service calls particulars. Can be used for example to monitor workflow progress in distributed Grid environment.
- data - derivation paths of a piece of data. Documents services, parameters and input information that contributed in creation of every available piece of data. This derivation data is gathered across all workflows executed in the system. Also, services that used particular piece of data are part of the trace. Data provenance could be used in determining quality of obtained information.

Karma was designed to met domain requirements of the Linked Environments for Atmospheric Discovery (LEAD) [71], while preserving good performance in workflow-oriented environment. Karma introduces notion of *activity*, taking place at different levels of execution - workflow, service or application. Every part of each level along with every data entity is identified by globally unique *ID*. For user's convenience, Karma generates three types of XML provenance documents, based of mentioned activities:
- workflow - activities of particular workflow execution
- process - activities of particular service or application call, including input and output
- data - applications that created or used particular piece of data, across all executed workflows

Those provenance documents are created on the fly by the Karma. All recorded activities are stored in relational database. System provides also simple graphical tool for viewing and navigating provenance graphs (documents).
In [72] Authors provides also performance evaluation of the Karma as compared to PRServ solution [65]. They reckon Karma as being faster, but restricted to LEAD applications. The latter should be expressly noted, as virtual laboratory applications defines slightly different requirements.

3.2. Discussion

All previously described models have certain limitations. For example, myGrid solution does not offer easy way of adding metadata information to provenance records. Also, however provenance data is enriched by ontologies, it only concern services in very narrow scope. What is more, myGrid system querying capabilities can not span multiple domains - in example provenance and application data.

As far as the PROVENANCE project is concerned, model proposed by final report is quite low level and generic. Albeit this could be seen as a feature, it lacks many particulars that would definitely enhance level of provenance information stored and query capabilities. As already stated, it rather should be part of a broader, more complex provenance model/framework.
The VDM proposition is interesting in assuming particular computation model, separating model's elements - as experiment, operation, input and so on. This strongly enhances level of provenance details captured. Yet, it seems that semantic web modelling with ontologies would be more adequate. Ontologies precisely capture environment modeled. What is more, separation to concepts (models) and individuals (registries) in ontologies are parallel to VDM's idea of prospective and retrospective provenance. Finally, ontology-driven model would support more complex queries, enabling mining over provenance data.
In conclusion listed solutions either focus on low-level models or do not support well complex, mining queries over various repositories. Especially, queries spanning multiple domains are not covered by existing models. Enriching data with semantic information seems to be good road to follow in design of new, appropriate model.

Chapter 4

Requirements specification for provenance sub-system

This chapter provides requirement specification for emerging provenance system along with some insight into its features. First section formulates requirements - both functional and non-functional, on the basis of ViroLab's applications analysis. Next section contains architecture assumptions made to fulfill those requirements. Finally overview of provenance tracking environment inside virtual laboratory is presented.

Based on the detailed study of Virtual Laboratory applications, two important types of provenance data available in the system were identified.
These are:

1. annotation provenance: Extra information about some piece of data, depends on specific application requirements. Exists solely as annotations, doesn't provide additional reasoning information.
2. actor provenance (event): A record of some action taken by an actor in system, using Virtual Laboratory application. Could be connected with creating new piece of data or changing existing piece. Enables Provenance sub-system to trace process that led to some piece of data in details, including creating or changing other system data involved in this process.

Emerging system should make possible recording those two types of provenance data. What is more, it should be done in a way convenient for medical applications, primary concern of the ViroLab project.
As it was already stated in the Chapter 3 - Related Work, provenance is very fresh topic in the Grid systems. Therefore, only few systems approached the matter until now. The VL provenance sub-system should base on research done in that projects

and adapt some current approaches, but with specific ViroLab's requirements kept in mind.

4.1. Requirements

Requirements for the VLvl provenance system were gathered by detailed analysis of described earlier, real world VL applications. Possible provenance use in these applications is summarized below.

- Drug Ranking System (DRS)
  Identified provenance tasks includes annotating results with meta data and tracking statistic of the DRS, like most used rule set or typical mutations for specific patient group. The latter can add much functionality to the DRS application desired and used by clinical virologists, thus extending application use cases.
- From Genotype To Drug Resistance
  Application's use cases incorporate provenance for storing annotations on various application stages. Other provenance usage in the application includes tracking and mining origins of gathered results.
- Establishing Large Databases Of HIV Sequences
  One of possible provenance task is to store information about the process of gathering data. Such information could be later used to perform fast updates. It should be pointed out, that such usage would generate massive amounts of provenance data.
- Data Retriever (and applications accessing hospital data)
  Application description, as found in the Deliverable document, suggests that provenance tracking system shall be used to store execution history data. This could include tracking data requests along with events specific to applications running in the sandbox environment.

Functional requirements

Gathering, storing and mining provenance data constitute main responsibilities of the provenance tracking system. From end-user point of view, convenient and potent provenance querying capabilities are even more important.
Therefore provenance sub-system has to fulfill following requirements:

1. Architecture should support recording both annotations and events (actor provenance) in the VL virtual laboratory
2. Provenance information recording should be fine-grained, allowing to capture even smallest pieces of data
3. Storage should be able to hold recorded provenance data indefinitely, without risk of information loss
4. User should be able to query for both stored annotations and actor provenance
5. System should allow querying for data annotations for particular piece of data or set of conditions
6. System should also allow querying actor provenance for a particular piece of data. Such query should retrieve whole trace of the piece of data. Depth of the trace shall depend on the user-defined query.
7. Query capabilities of the system should allow user to perform mining on a prove-

nance data. In example, no only queries for simple traces should be allowed but also queries about resemblances between various data traces. Also, statistic queries as stated in VL applications description have to be supported without additional processing.

8. User should be capable of accessing provenance information from specified time scope.
9. Provenance system architecture should allow users to save, reuse and share their mining queries. This requirement is crucial as virtual laboratory is designed as collaborative environment.
10. Architecture should provide means for preparing backup of provenance information stored. Output format should be configured and some optimization of the process, as compression should be considered.

Non-functional requirements

As stated in the introduction, recorded provenance data is immutable, and therefore storage space requirements will grow infinitely in time. What is more, complicated queries processing so large amounts of data could be very demanding task. Thus, also some performance requirements have to be fulfilled. In the grid systems and especially in the ViroLab, security issues are very important. Although provenance data will not contain any sensitive information, query construction and result rendering could access restricted domains. Therefore, security requirements should be kept in mind.

All above aspects of the provenance tracking system imposes following requirements on the architecture:

1. Storage back end of the provenance tracking system should be largely scalable
2. Space used for provenance data should be minimized, but without sacrificing query performance (thus compression is not recommended)
3. Architecture of the storage should be distributed, using multiple nodes to achieve big storage space
4. Data storage should be reconfigurable, allowing storage nodes to be revoked or added in run-time
5. External system interfaces should provide seamless integration with other components, by using some remote communication technology
6. Run-time configuration, management and monitoring of the whole provenance component should be possible remotely
7. Architecture design should take into consideration characteristics of distributed systems, as synchronization of multi-threaded processing.
8. Transmission of sensitive data, as patient-related records, should be authorized and encrypted
9. Query processing should be efficient in handling queries that cope with large amounts of data
10. Complicated query processing characteristics should be taken into account in architecture design, to avoid possible performance penalties

4.2. Preliminary assumptions

Profound analysis of the requirements listed in previous section along with research on existing provenance have been done. It led to some preliminary assumptions, that have to be made in order to design robust and appropriate provenance tracking system architecture.
These assumptions follow:

1. Built with ontologies
   Formally, ontology represents set of concepts from specific domain along with properties and relations between those concepts. This can be used to model some knowledge about given system, for example provenance, defined as processes (actor events) and meta-data. What is more, concept of separate ontology domains can be used to separate events generated in different system applications. Also, required annotations of data entities and operations can be easily modeled using ontologies. What should be noted, ontologies allow reasoning on gathered data. So feature could be used by the provenance tracking system to extract more fine-grained information, fulfilling adequate requirement. Usage of ontologies adds also much flexibility to the system. In a research project with ever-changing requirements this feature is not to be underestimated. This assumption goes in line with recent research in the virtual laboratory area. As it was pointed out in [59] and [63], semantic and knowledge services in Grid systems are of paramount importance.
   Current standard for modeling ontologies is the OWL language. There are many existing standardized solutions designed for processing and reasoning over OWL ontologies. Using such tools would definitely speed up implementation and boost reliability of the provenance system.
2. XQuery support -
   Lately XQuery language has become the standard for mining XML based data. With high-level, SQL-like and English-resembling syntax it is very easy to learn and use. These features make XQuery perfect fit for the 'user friendly querying' requirement. Additionally, the language itself is very powerful, allowing for nested constructs of the basic FLOWR (for, let, where, order by, return).
   What is important, XQuery is official W3C recommendation and ultimately will be supported by major DBMS engines.
3. XML storage
   Thorough research on available XQuery support has revealed that in order to use the language a native XML DBMS must be used. This is because all existing XQuery APIs are able to access only data contained in files, not data sources. So architecture would be very inefficient.
4. Support for ontology languages
   Apart from user-friendly XQuery, provenance system should also support other query languages. Because of the ontology nature of the data, the most desired are RDF specific ones. Particularly, RDQL and SPARQL languages are broadly used and most developed. What is important, they are supported by the industry-standard OWL solutions, as Jena Ontology Framework.
5. Distributed storage
   Research on the provenance usage in other project, described in the Related work

section has led to follow 'Separate store pattern' from the Provenance project. Also, in order to maintain satisfying performance, storage architecture will be distributed with respect to ontology descriptions.

6. Interoperability and high performance
To satisfy interoperability requirements, provenance system will use Stateless Web Services interface to external components. However, Web Services suffer from some performance issues. Thus, where applicable, other communication middleware will be used. In example, Java Remote Method Invocation seems to be good solution for inter-system communication.

4.3. System's environment overview

Fig. 4.1 depicts PROToS with provenance environment. It is composed from identified system components, cooperating in provenance tracking, storing and using.

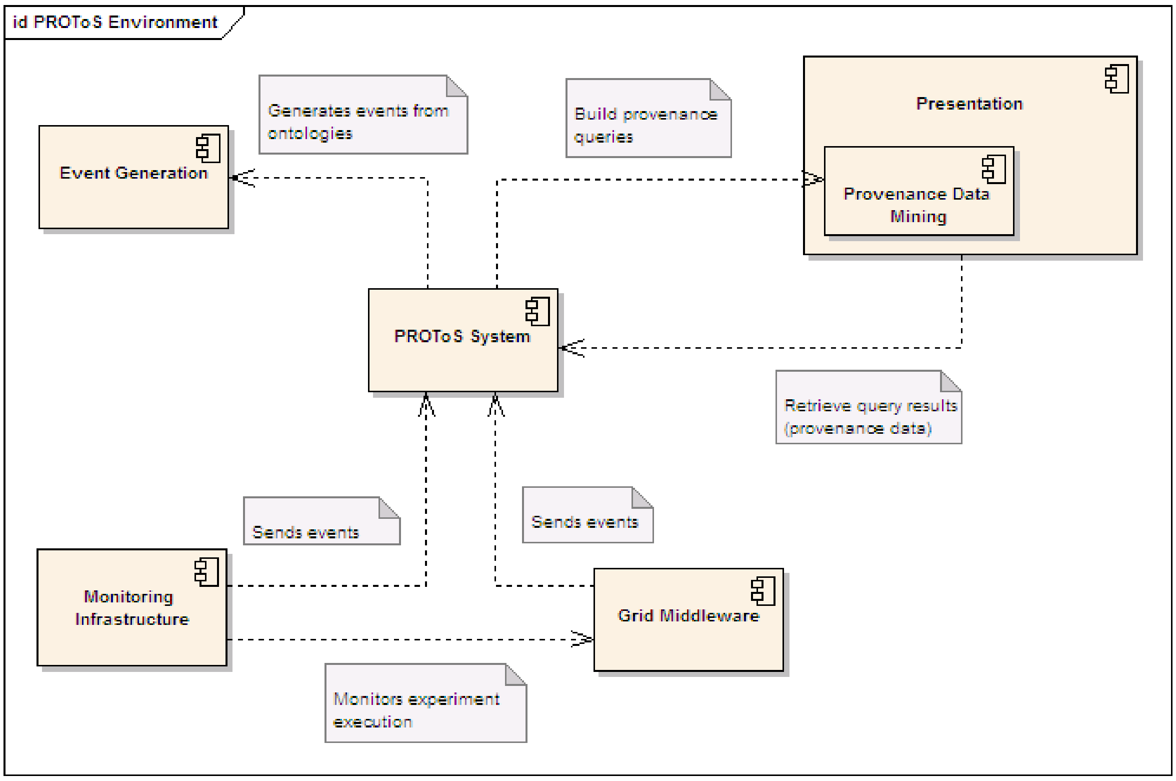


Figure 4.1. Environment of the PROToS. Figure includes all external components of the virtual laboratory, cooperating in tracking of provenance data. Most important omponents are Monitoring - feeding PROToS with data and Presentation - processing provenance queries.

Brief description of the environment follows:

- **PROToS System**
Central point of the environment. Provides core capabilities as interfaces for provenance data storage and retrieval. It's architecture is covered in great details in chapter 5.

- **Event Generation**
  The component responsible for generating events from domain ontologies. As stated before, PROToS system is assumed to make use of ontology models for describing provenance events and annotations. These ontologies have to be converted into Java classes, that can be instantiated and send to notify PROToS.
  Key assumption is that most of provenance events can be modeled as single OWL language classes. Every single-class event should stick to a chosen model, as Java-bean. Therefore, process of translating OWL-based ontology to Java-based package can be easily automated.
  Such tool is present in the system for pure convenience reasons.
- **Monitoring Infrastructure**
  This virtual laboratory sub-system is responsible for monitoring experiment execution in the Middleware. Thus, each stage of the execution could be modeled in specific ontology and send to the PROToS, as a piece of provenance data. Quality and amount of provenance data provided by the monitoring infrastructure does not depend on experiments being executed. Accordingly, the component is crucial in order to collect as much provenance as possible.
- **Grid Middleware**
  Middleware component models part of the virtual laboratory that executes user's experiments. During execution, user's scripts can communicate with miscellaneous services, called Grid Objects. PROToS can also act as Grid Object, therefore experiments are able to store script - specific provenance data. Events to be send from inside scripts shall be modeled in application-specific ontologies, concerning separate domains as protein folding.
- **Provenance Data Mining**
  This component, part of the Presentation, enables users to create provenance and data mining queries in an user-friendly way. Analysis of the virtual laboratory revealed that enormous amount of provenance data will be generated by the system. Thus, to utilize data in common user practice, specific tool is required. What is more, typical user will not possess technical knowledge, so the component shall allow complex provenance mining is simple manner.
- **Presentation**
  Presentation component defines user interface of the virtual laboratory. Execution of the queries built with Provenance Data Mining tool along with other applications will definitely include calls to the PROToS system. Furthermore, some application-specific Presentation sub-components may allow user to store additional meta-data in the provenance sub-system.

Provenance tracking environment, as depicted above, is solely first - draft of components necessary for adequate PROToS operation. Current, real-world implementation of these, as found in the VL virtual laboratory can be found in section 7. Also, whole description of the provenance tracking in the ViroLab virtual laboratory is in [53].

Chapter 5

PROToS architecture

This chapter provides brief overview of the PROToS architecture. First section contains dictionary of terms used in this and consecutive chapters. Next section describes core concepts behind the ViroLab's provenance tracking and PROToS architecture. Third section presents component architecture overview of PROToS. Following one presented use case diagrams for identified PROToS scenarios. Last one describes project organization, done with Maven2 tool.

As stated in previous chapter, main goal of this work is to develop solution for handling provenance in virtual laboratory of Grid-enabled system. Thus, simple name was chosen - PROToS. It stands for PROvenance Tracking System. From now on, the system will be referred to with this simple name only.
PROToS is designed from the very beginning to prove itself in challenging environment of the modern virtual laboratories. As stated in the requirements before, system for tracking provenance in the Vlvl should boast with high performance, huge storage space and such user-friendly features as remote configuration. What is more, being a research project, it should also be flexible enough to embrace changing requirements and allow swift replacement of implementations along with algorithm's advancements. This chapter provides first insight into PROToS architecture and core concepts, designed to fulfill above description.

5.1. Dictionary

Table 5.1 summarizes most important definitions and acronyms used in successive sections of the documentation. Some of them were used before, or relate to technologies presented in last section 6.2. From now on, all of them will be referred to without explanation.

Table 5.1: PROToS dictionary

Acronym	**Definition**
Vlvl	ViroLab virtual laboratory. Part of the VirLab grid system.
DGE	Data Gathering Engine. PROToS component responsible for gathering provenance data.
DRE	Data Retrieval Engine. PROToS component responsible for processing queries and returning results.
DSS	Distributed Storage Supervisor. PROToS component responsible for storage management
SP	Storage Peer. PROToS component acting as single storage endpoint. Element of the PROToS distributed storage.
SSN	Storage Super Node. PROToS component responsible for management of storage endpoints.
Event	Instance of a Java class that represents some system event such as user action or new data annotation. Events are delivered by the monitoring infrastructure to the DGE component of PROToS.This name is also used sometimes to describe internal PROToS representation of delivered event instance, that is an ontology component.
Provenance event	System event that provides some provenance information. Depends on application
DO	Domain Ontology. Set of ontology components, that describes some domain of Grid application. For simple applications there can be only one Domain Ontology. See 'preliminary assumptions' for more information about ontologies in PROToS.
XQuery	Modern, industry-standard language for querying XML data.
FLOWR	For - Let - Where - Order By - Return. Basic query construct in XQuery language
RDF	Resource Description Framework. W3C specification for metadata information and information modeling. Based on subject - object - predicate triplets.
OWL	Web Ontology Language. Industry standard for ontology description, based on RDF.
IoC	Inversion of Control. Design pattern, foundation of Java lightweight component technologies.
DI	Dependency Injection. Design pattern, being form of the Inversion of Control. Implemented in such frameworks as Spring [31] and widely used in PROToS applications.
WS	Short for stateless Web Services.

5.2. Core concepts

This section presents core concepts, lying in the foundations of the PROToS architecture. These concepts strictly follows and extends design assumptions made in the section 4.2.

1. **Ontology models**
   As stated in respective section, OWL-based ontologies are ideal for modeling provenance knowledge in systems like the ViroLab. Analysis of provenance applications in the VLvl has led to separation used ontologies on three main groups:
   - **Data models**
     Models describing data entities available in the system. Should model application's specific entities (as drug rankings in the DRS application) along with domain (as Patient). Also execution input and output should be taken into account as separate entities (as Image or Report).
   - **Domain specific models**
     Various applications running in VLvl can originate from different domain. As far as the ViroLab is concerned, domains are related to virusology and patient treatment, but next generation projects can span on very different domains. Domain ontology is to model specific knowledge of an application. For example in the Drug Ranking Application, grid object call can be modeled as *New Drug Ranking* PROToS event with *used rule set* and *tested nucleotide sequence* properties. Range of these properties should point to adequate concepts from common **data** model. Usage of domain specific models allows for very fine grained provenance tracking. Nonetheless, use of application specific models depends on experiment's script write, what can be viewed as a drawback.
   - **Experiment model**
     Models provenance events bound to generic experiment execution. Every experiment, regardless of execution domain, is composed of calls to grid objects, data reads and writes. Also some metadata, as user login or experiment script version can be attributed to each execution. All above can be modeled in one common ontology. So model is not able to capture domain-specific knowledge, as semantic types of grid object's arguments, but can be applied to execution of every script, without effort from script's author.

   OWL language chosen for modeling PROToS provenance ontologies possess very useful feature - ability to establish relation between concepts from different models. So-called object properties should be used to link concepts from experiment, data and domain models. Fig. 5.1 presents how example models could be joined.
   As depicted, data model defines one concepts, root for data entities hierarchy. This concept is linked to the **Service call** concept from generic experiment model, as input and output data range. Because generic call can take as argument any data, root concept has to be used instead of one of specific, concrete entities (**Data A**, **Data B**). Those concepts are used by the **Domain call** from domain model. It is possible because the concepts model particular call of some grid object, thus argument types are known by the time ontology is written. Generic experiment model contains also **Experiment** concept, aggregating execution calls and carrying

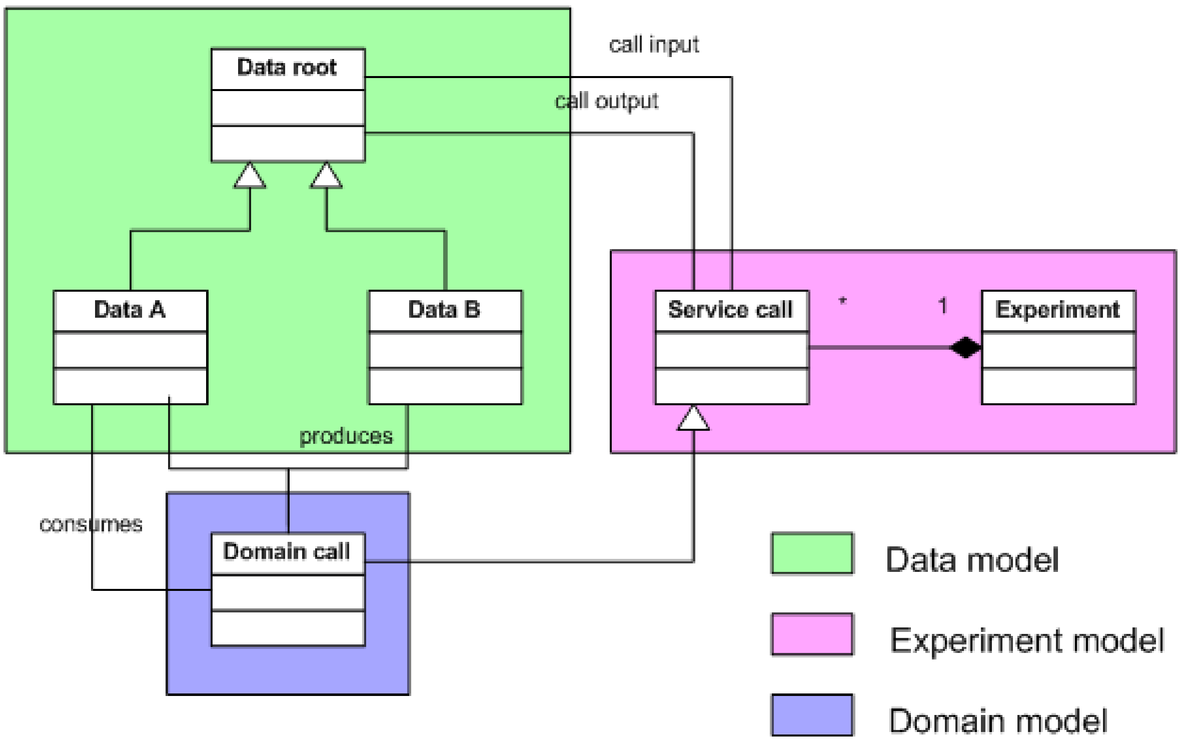


Figure 5.1. Example PROToS ontology models. Diagram contains example for each group - data, domain and experiment. Also, model's concepts are linked by object properties and inheritance relation.

experiment's meta data. What should be additionally noted, various models are linked not only by object properties, but also by inheritance - example **Domain call** and **Service call** concepts.

2. **Hierarchical storage**
In section 4.2 design assumption of distributing the storage was made. This concept extends the idea with defining hierarchy in the storage.
First level of the storage consist of **Nodes**. Each node manages one or more assigned domain ontologies and is directly connected to the storage root. **Node** is designed to act as 'thick' member, that is performing part of required storage/retrieval processing. Second level is composed of **Peers**. Each **Peer** is designed to be 'thin' member, providing only storage space and a basic API for manipulating data. Groups of **Peers** are to be assigned to a **Node**. Thereby, storage is organized in tree-like structure. Example storage organization is depicted on Fig. 5.2.
Proposed design takes advantage of ontology foundations of the PROToS. Provenance usage projections for the Vlvl shows that typical queries span only few domain ontologies, often only one. In this storage schema, such queries are to be handled by one **Node**. Therefore, distributed query and results merging algorithms are not used. Performance gains in those cases are quite huge. What is more, clear separation on processing (Node) and storing (Peer) components allows for efficient utilization of available system resources.
3. **Distributed processing**

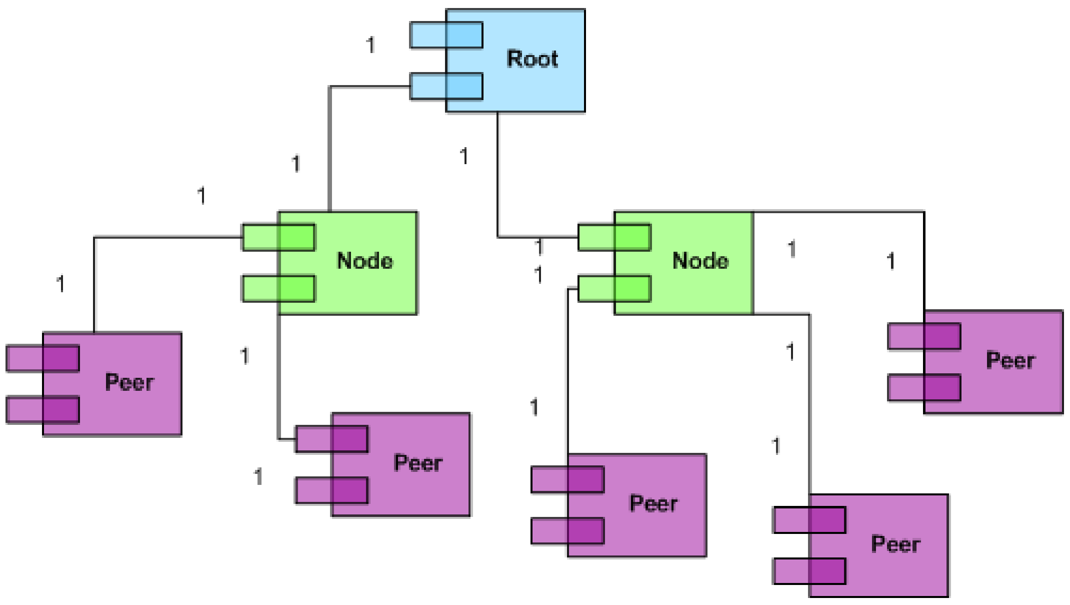


Figure 5.2. Example three-level storage hierarchy of the PROToS. Storage depicted consists of one root, two nodes and five peers creating tree-like structure.

Projected usage characteristics of the provenance sub-system requires PROToS to have single access point, preferably as stateless Web Service. Yet, performance plays great role according to requirements defined in 4.1. This issue is to be addressed by distributing processing, which goes nicely with **hierarchical storage** concept.
Physically, PROToS architecture distinguishes three types of components:

- **Core application**
  Acts as single entry point for the PROToS, exposing external interfaces (producer, consumer). Manages **Nodes** and handles general system configuration. Performs computations required for distributed querying and result merging. Also, processing specific to ontology models, as incoming provenance data validation is to be handled by the **Core**. Queries and event processing specific to certain domain ontology is entirely delegated to adequate **Nodes**. **Root** component from diagram 5.2 is to be implemented by the **Core application**.
- **Node application**
  As stated previously, manages assigned domain ontology models and **Peers**. Handles all processing related to in-house models. **Node application** implements **Node** component from diagram 5.2.
- **Peer application**
  Implements component **Peer** from diagram 5.2. Acting as simple storage facility contains as little logic as possible. Key assumption is that **Node** should use computation resources only and **Peer** should use storage resources.

5.3. Architecture overview

Diagram 5.3 summarizes architecture of the PROToS system, outlined in section 5.2. Components are presented along with main external, exposed interfaces.

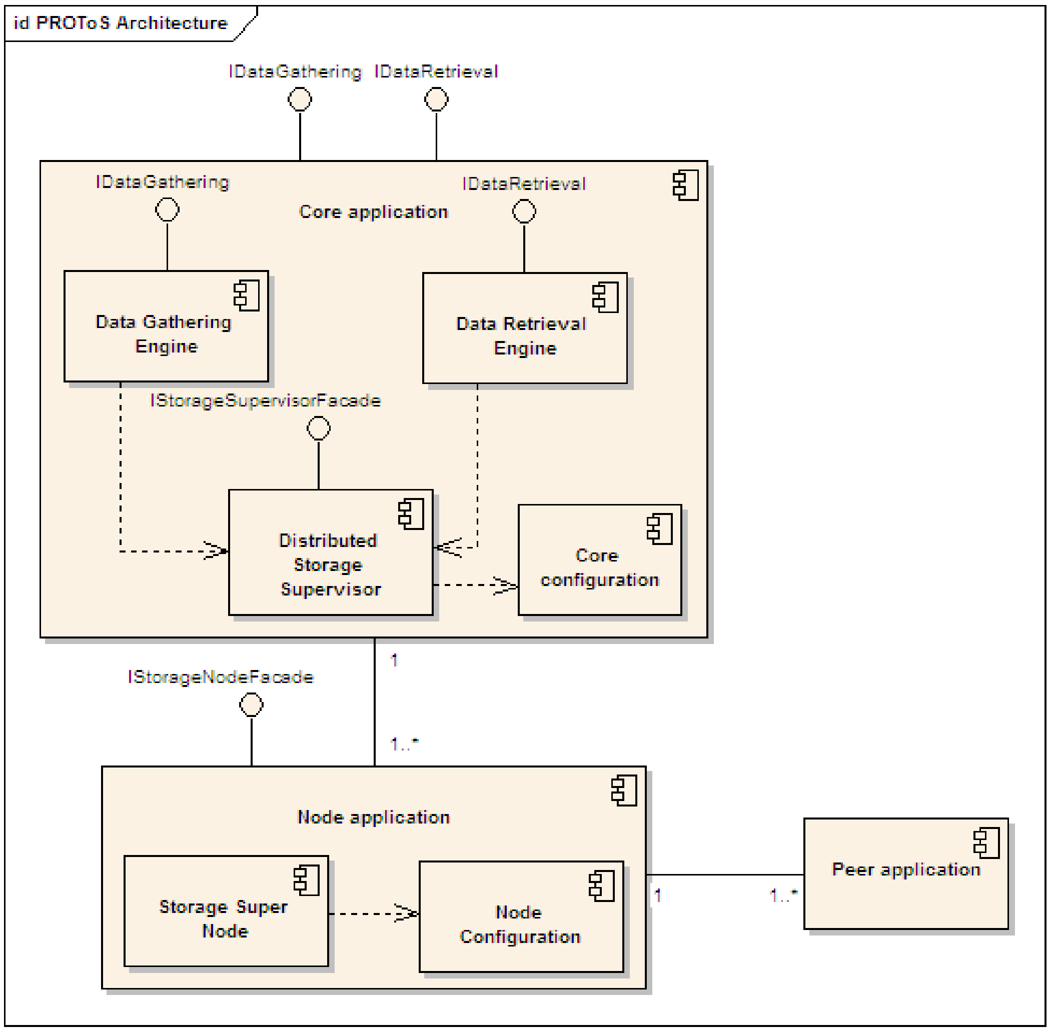


Figure 5.3. Proposed PROToS architecture as UML component diagram. Diagram depicts core components - applications and sub-components encapsulating particular functionalities. Also, main interfaces are presented, with IDataGatehring and IDataRetrieval constituting system's external interface.

Depicted system's components fall into one of two groups:

- **applications** - components to be deployed on physical machines, as runnable Java JAR or WAR archives. Short descriptions of application's roles are to be found in previous section 5.2.
- **software components** - encapsulating interfaces and implementations for specific functionalities. Packed together forms applications of the PROToS system. Short descriptions for these components follow:

— **Data Gathering Engine** - DGE in short. Responsible for accepting, validating and handling PROToS events delivered to the *IDataGathering* interface it exposes.
— **Data Retrieval Engine** - DRE in short. Responsible for accepting, validating and handling queries along with formatting and packing results. Also, backup functionality is to be provided by the **DRE**. Exposes one interface - *IDataRetrieval.*
— **Distributed Storage Supervisor** - DSS in short. Responsible for managing Nodes available for the PROToS system, processing complicated multi-domain queries and results merging. Acts as a mediator between PROToS distributed storage and other **Core** components. In fact, DSS is most important and complex component of the PROToS. Exposes interface *IStorageSupervisorFacade*, used by the **DRE** and **DGE** components to communicate with **Nodes**.
— **Core configuration** component, responsible for handling remote and local access to configuration of the **Core** components.
— **Storage Super Node** - SSN in short. Responsible for managing ontologies and Peers designated for this **Node**.
— **Node Configuration** is responsible for handling remote and local management of SSN's configuration.

Presented components are mostly implemented as separate software components (modules). Details of mapping between components and modules can be found in section 5.5.2. Detailed design and implementation descriptions of all PROToS components and modules are to be found in respective sections of chapter 6. Overview of the PROToS architecture was also given in [56].

5.4. PROToS Use Cases

PROToS system at it's final stage shall provide functionalities fulfilling previously defined requirements. Use cases capturing these are presented on diagrams 5.4, 5.5, 5.6 and 5.7.
Listed use case diagrams feature actors, described below:

- **Administrator**
  Grid system user, responsible for deployment and installation of the PROToS system, or deployment of new applications in PROToS-enabled environment. The latter task would involve configuring Domain Ontologies for new applications.
- **External component**
  Components interested in mining provenance data stored inside PROToS. As depicted in Fig. 4.1, these are **Presentation** and **Provenance data mining** components. Apart from shown, other Grid system components could be also interested in provenance. For example, there are some ideas about using the data in application optimization sub-systems.
- **Monitoring Infrastructure**
  Part of the Virtual Laboratory responsible for sending system events to PROToS. Described in section 4.3.
- **Grid Middleware**

Virtual Laboratory component, sending domain-specific events to the PROToS. Covered in section 4.3.

Use cases specific to particular PROToS components and applications are presented in consecutive sections.

5.4.1. DRE component

Fig. 5.4 presents use case specific to the DRE component.

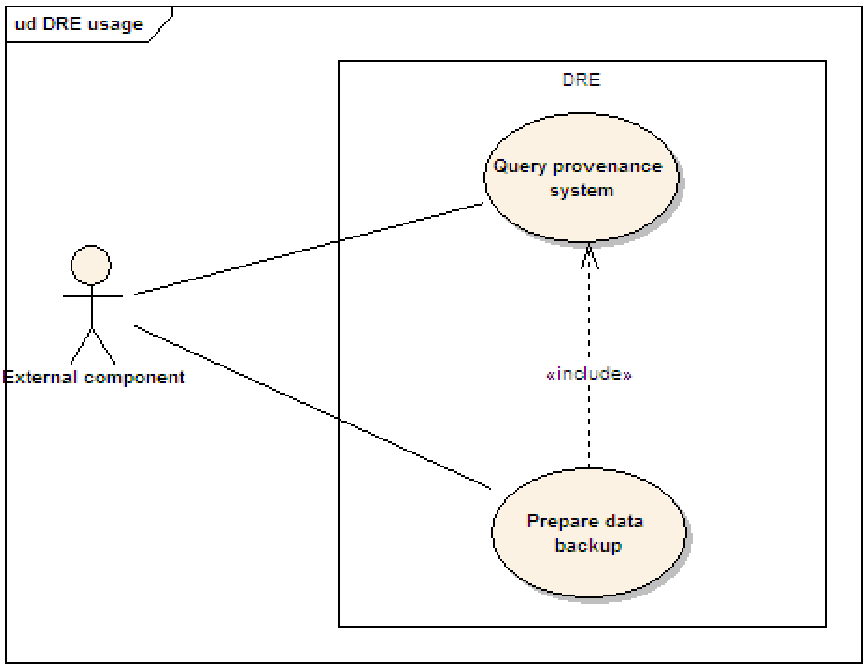


Figure 5.4. DRE component Use Case UML diagram. Figure presents core use cases, bound to DREs functionalities.

Actor in this use case is an external component interested in querying provenance. It acts in one of following scenarios:

1. **Query provenance system**
   In this scenario actor submits provenance mining query to the DRE and retrieves the result. The query is in one of the supported languages. Component's configuration is complete and defines requested format of returned result.
2. **Prepare provenance data backup**
   In this scenario actor submits request for full PROToS data backup. The format and other particulars of the backup process are configured.

5.4.2. DGE component

Diagram 5.5 depicts use cases for the DGE component.

Actor in this use case can be either Monitoring Infrastructure or Grid Middleware component. It can act in following scenario:

1. **Submit PROToS event**
   In this scenario actor submits new PROToS event to the DGE component. The

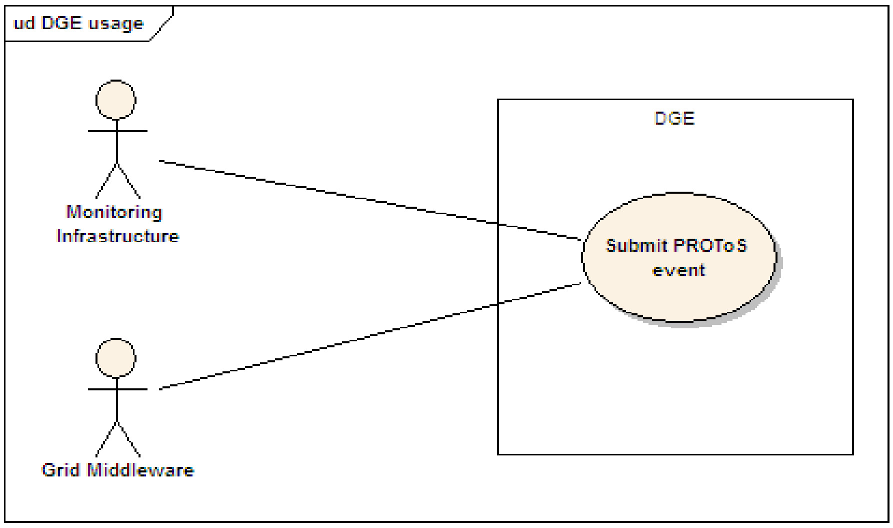


Figure 5.5. DGE component Use Case UML diagram. Figure depicts core component's functionalities.

event is configured, valid and contains all necessary data for PROToS storage. DGE component is also configured to handle incoming event's type.

5.4.3. Core application

Fig. 5.6 presents use cases in PROToS Core application configuration. Particular components under configuration are also marked on the diagram.

PROToS system administrator is an actor in presented use cases. He acts in following scenarios:

1. **Configure event validation**
   In this scenario Administrator configures the validation policy for incoming PROToS events. Policies differ in rigour of validation and system components involved in validation.
2. **Configure event handling**
   In this scenario actor configures handlers required for specific event's types. Administrator chooses one or more handlers from list of currently implemented and available for the DGE component.
3. **Configure Domain Ontologies**
   In this scenario actor configures DOs required for the DSS operation. Administrator can add a new Domain Ontology, remove an existing one or require system to load new versions of available and configured DOs.
4. **Configure storage**
   In this scenario, Administrator configures distributed storage of the PROToS system. Configuration is fine-grained with regard to nodes available to the DSS. It involves also configuring DOs handled by particular nodes.
5. **Configure query result formatting**
   In this scenario, Administrator configures result formatters available for incoming

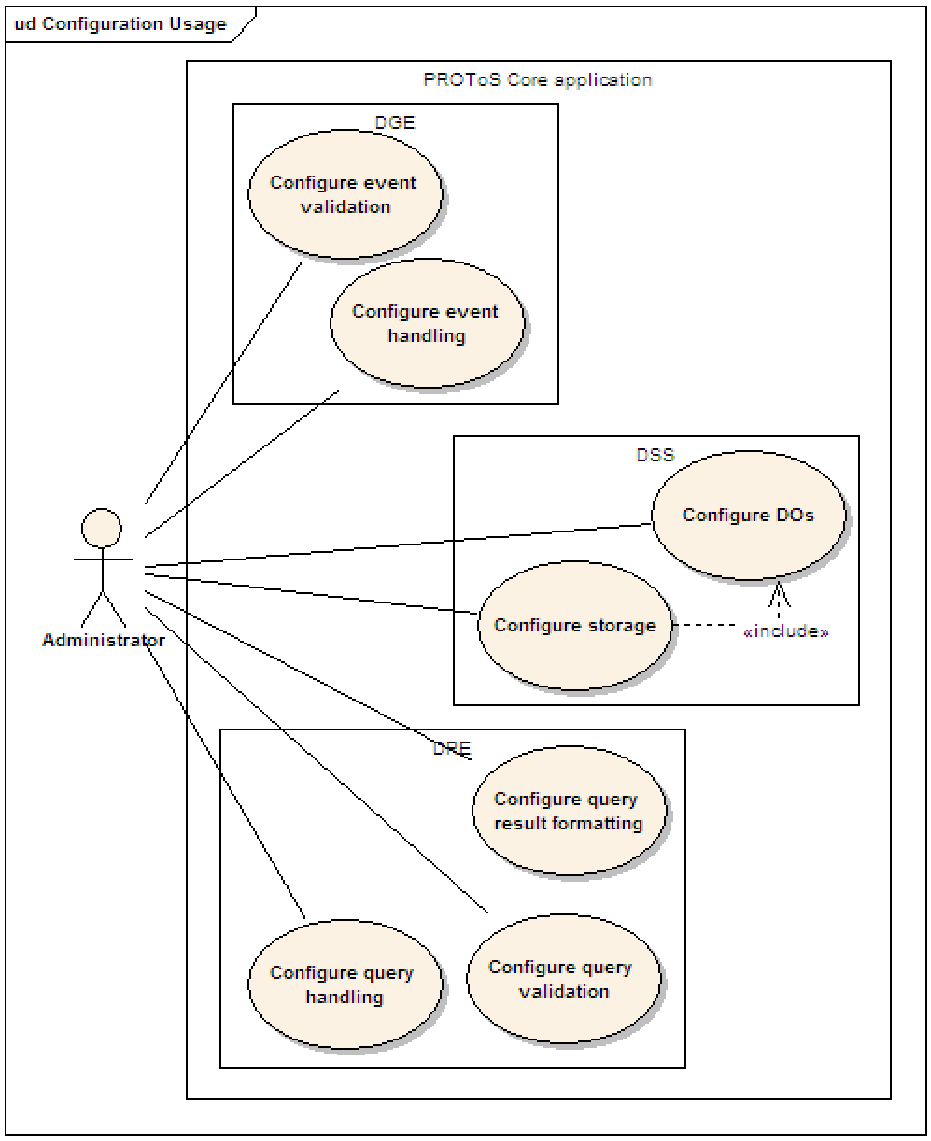


Figure 5.6. PROToS core application Use Case UML diagram. Diagram presents configuration functionalities of the component.

queries. The actor chooses one or many formatters from list of available for the DRE component.

6. **Configure query validation**
   In this scenario, Administrator configures validators active for incoming queries. The actor chooses one or many validators from list of implemented and applicable. Validators are specific to language, type and options of incoming queries.
7. **Configure query handling**
   In this scenario, Administrator configures handlers available for incoming, specific queries. The actor chooses one or many query handlers from a list of all implemented and available for the DRE. Handlers are specific to languages of queries and particulars of handling algorithm.

5.4.4. Node application

Diagram 5.7 depicts use cases for the PROToS Node applications.

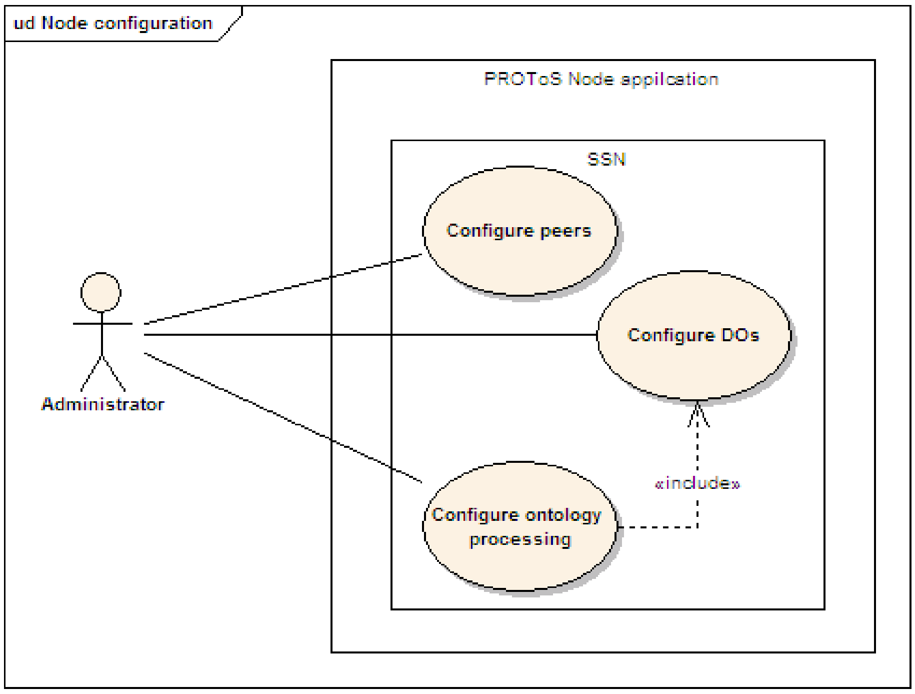


Figure 5.7. PROToS node application Use Case UML diagram. Diagram depicts configuration functionalities of the application component.

System's administrator is an actor in these use cases. He can act in following scenarios:

1. **Configure peers**
   In this scenario Administrator decides which storage peers shall be available for the Node. Peers are passed as addresses, in format understood by the node logic.
2. **Configure DOs**
   In this scenario Administrator configures Node's domain ontologies.
3. **Configure ontology processing**
   In this scenario, Administrator configures ontology processing capabilities of the Node. This includes adequate DOs. Also, such particulars as ontology reasoning state, ontology storage format and incoming ontology model transformations are set up.

5.5. Project organization

The PROToS project is organized with help from great Maven 2 [23] tool.

5.5.1. Maven introduction

Maven 2 is open-source tool for building and managing Java-based projects. It is based around concept of POM (Project Object Model) - XML file describing all vital aspects of a project. These include for example name and description of the project, developers involved, source code management and - most important - dependencies. The latter enables defining exact versions of libraries and components required by the project, thus solving major Java disadvantage (so-called 'jarmageddon'). Dependencies are fetched from central repositories, assuring unified build environments. What is more, in Maven dependencies can be defined in several **scopes**. This allows for using different libraries in testing and production environments. Each piece of code with own POM is called **module**. Maven allows for aggregation and inheritance relations between modules, defined in POMs. Also POM vital data, as dependencies, can be shared between POMs by using these relations. Build process of a module finishes with binary - called **artifact**. Each module can be configured to produce more than one artifact in chosen packaging, as Java JAR, WAR or EAR. Maven provides release mechanisms, allowing for publishing generated artifact under certain tag (group, artifact id and version) to remote locations. Therefore, work in large, distributed groups (as the ViroLab consortium) is much easier. Each build can also generate project information and documentation as Java Doc, published by Maven as set of easy to access HTML pages. All above steps are executed in clearly defined process, called **build lifecycle**. Lifecycle is made up of phases (as compile, test, package, verify, deploy). Each phase consists of goals - specific tasks representing fine piece of work (as unpacking project's dependencies). Each goal can be bound to one or more phases, allowing flexible builds. Finally, Maven offers easy to use plugins system for extending it's capabilities. Most popular and used in PROToS includes automatic code reporting (as PMD [27], test code coverage or FindBuge [10]) and continuous integration bridge. Plugin configuration allows for binding with lifecycle phases and goals, thereby achieving reliable and reproducible builds.

5.5.2. PROToS modules

Maven modules along with aggregation and inheritance between module's POMs, encourage fine-grained separation of project's components. Each module encapsulates particular functionalities, logic and APIs with clear relations to other components. Those principles applied to the PROToS project resulted in module's hierarchy depicted in Fig. 5.8.

Hierarchy shown includes only inheritance relation between modules. Each component belong to one of several groups, depending on it's function and POM type.

1. **Super POMs** group

   This group contains POMs defining common information for separate branches in PROToS hierarchy. It's packaging is always defined as POM, therefore resulting artifacts provides only XML data. Modules from the group are marked as deep blue.

   - *PROToS* defines root of PROTOs-related modules. All possible external components presented in Fig. 4.1, as 'Provenance Data Mining' should inherit this POM. Contains most basic project information, as metadata (group, version, description), artifact repositories, SCM and continuos integration, developers and license used.
   - *PROToS Core* defines branch for PROToS-specific modules. Configures com-

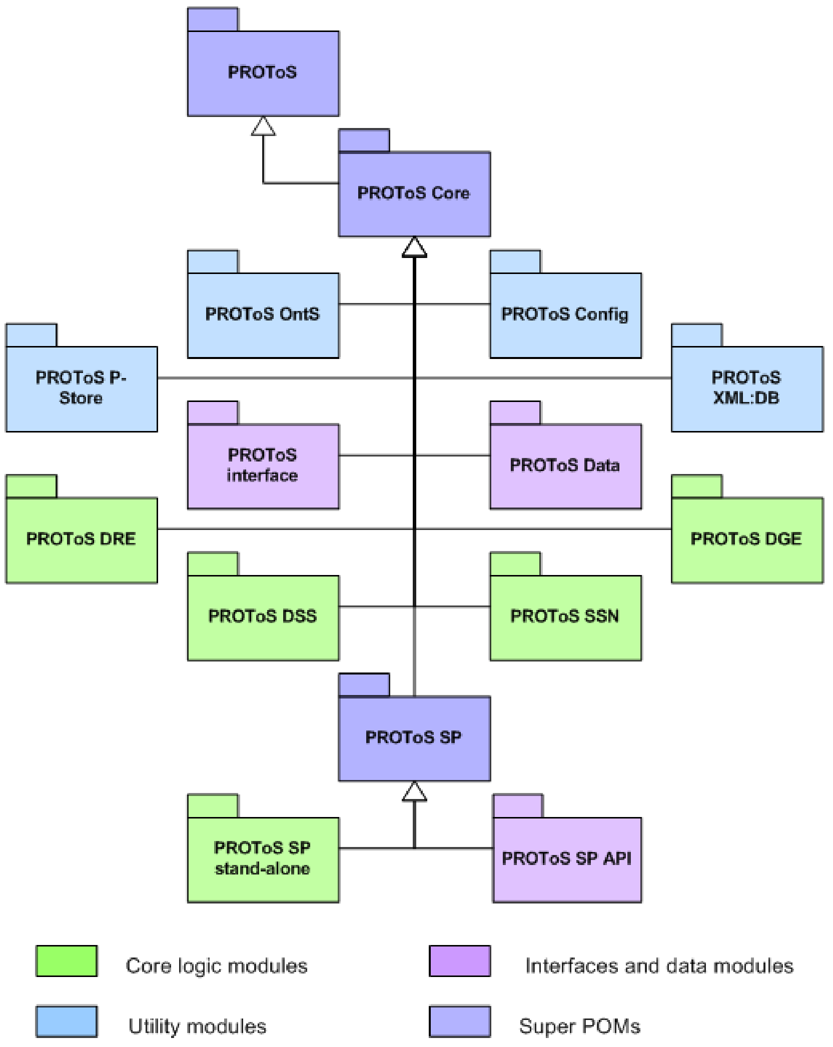


Figure 5.8. Hierarchy of PROToS modules as defined in Maven2 POMs. Diagram includes only inheritance relation between modules, defining branches in module tree. Dependencies are shown on other diagrams.

mon PROToS settings, as build plugins used (PMD, test coverage etc.) and dependencies for test environment.

- *PROToS SP* defines branch for Storage Peer components. It is required because SP is partitioned on API module and specific implementations.

2. **Core logic** group
   This group contains modules reflecting core components from architecture diagram 5.3. Modules from this group are marked as light green.
   - *DRE* contains interfaces and implementation of the Data Retrieval Engine component. Described in section 6.1.2.
   - *DGE* contains interfaces and implementation of the Data Gathering Engine component. Covered in details in section 6.1.3.
   - *DSS* contains interfaces and implementation of the Distributed Storage Supervisor component. Details provided in section 6.1.4.

- *SSN* contains interfaces and implementation of the Storage Super Node component and Node application. Covered in section 6.1.5.
- *SP stand-alone* contains implementation of the Super Peer component, based on eXist database [9]. Details of the implementation are provided in section 6.1.6.

3. **Utilities** group
   Modules from this group encapsulates common PROToS logic of particular aspect. These components were born during PROToS development, to avoid duplication of the code. Modules of the group are marked as light blue.
   - *OntS* encapsulates ontology - processing logic of the PROToS. Description provided in section 6.1.10.
   - *Config* contains configuration provider definition and implementation, used by core system components. Covered in details in section 6.1.9.
   - *P-Store* encapsulates simple persistence utility, used by core components for state management. Described in section 6.1.11.
   - *XML:DB* encapsulates data model and logic for working with databases over XML:DB API, as SP endpoints. Covered in details in section 6.1.12.
4. **Interfaces and data models** group
   Contains components encapsulating common data models or interface definitions. Modules from this group are marked as violet.
   - *Data* module contains definition of the common data model for communication with PROToS system. Covered in section 6.1.7.
   - *Interface* module encapsulates logic and configuration required for the Core application. Described in details in section 6.1.8.
   - *SP API* contains definition of interfaces that should be implemented by the SP incarnations. Described in section 6.1.6.

Fig. 5.9 presents simplified dependencies between PROToS modules, as defines in respective POMs.

Only direct dependencies are shown in the diagram because dependency relation in Maven is defined as transient. That is, all core components are dependent on the *Data* module, but only *OntS* and *SP API* import it. As depicted, Core and Node application's components all import utility logic modules.
In summary, without usage of Maven tool, not only organization but also development of such large project as PROToS would much more difficult. Necessary Maven overhead of writing XML configuration - POMs - is definitely justified.

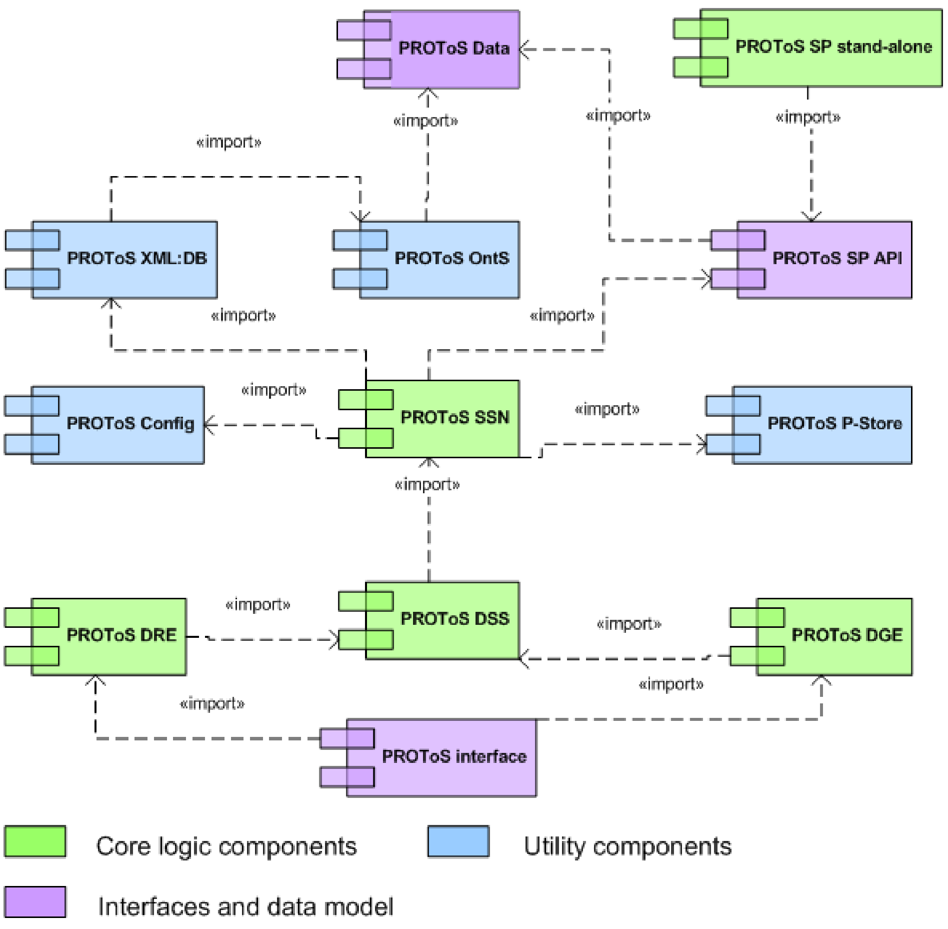


Figure 5.9. Interdependencies in PROToS modules hierarchy. Dependencies are transitive, hence only direct dependencies are depicted.

Chapter 6

PROToS design and implementation

This chapter contains detailed description of PROToS design and implementation. Each section reflects one software component (Maven2 module) as defined in project organization. Last part of the chapter is devoted to technologies used along with explanations.

6.1. Modules detailed design

This section contains detailed description of PROToS design and implementation. For convenience, section organization reflects modules hierarchy, as depicted in Fig. 5.8.

6.1.1. Conventions

Following subsections contain detailed descriptions of PROToS components. Each subsection contains component's dependencies as UML component diagram and component's design as UML class diagrams. Component's Descriptions and used diagrams make use of following conventions:

- If class depicted in UML diagram does not list methods or list some private attributes, it is implied that this class has accessor methods for every protected and private attribute. All accessors use Java Bean manner, that is *propertyX* has accessors named *setX* and *getX*. This rule is introduced to make diagrams more clear.
- All component's dependency diagrams contains only external dependencies. That is only exclusively dependant components outside of the PROToS are shown.
- In descriptions of component dependency diagrams, notions of a 'component', a 'library' and an 'artifact' are used alternatively. All of them denote a software component.

6.1.2. Protos-dre component

Description

This component encapsulates logic for data retrieval functionality, as executing provenance - mining queries and preparing backups. It also handles formatting and returning results in user-defined formats. The component heavily depends on the distributed storage, represented by the **DSS** component.

Design

All source code of the *protos-dre* component is placed in packages **pl.cyfronet.virolab.protos.dre.***.
Fig. 6.1 depicts core, external interfaces of the **DRE** component.

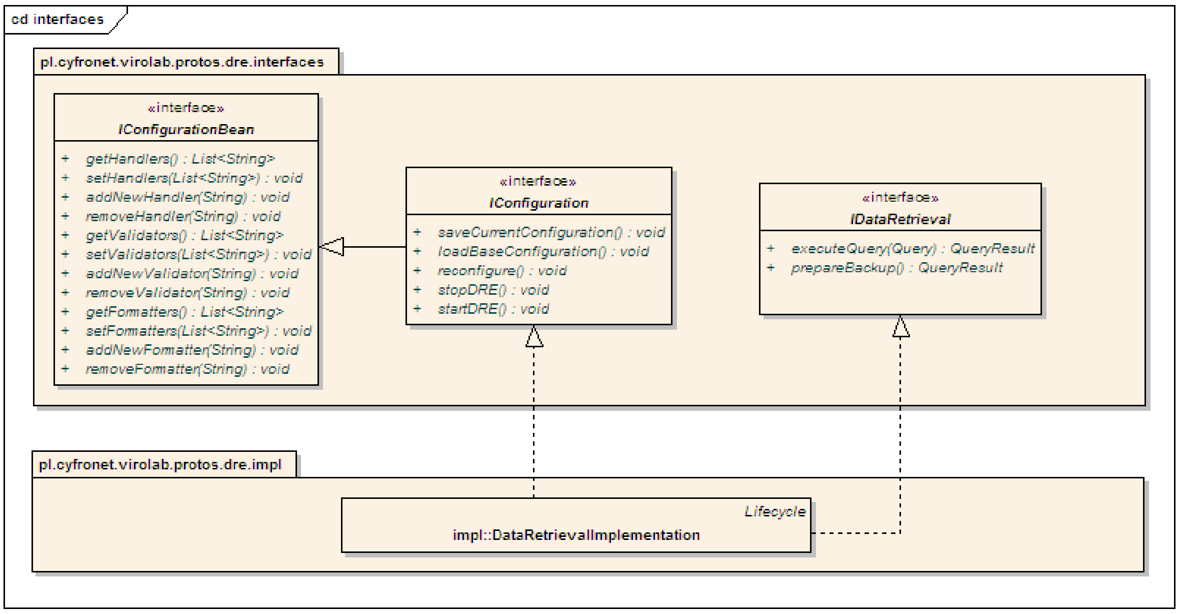

cd interfaces
pl.cyfronet.virolab.protos.dre.interfaces
«interface»
IConfigurationBean
+ getHandlers() : List<String>
+ setHandlers(List<String>) : void
+ addNewHandler(String) : void
+ removeHandler(String) : void
+ getValidators() : List<String>
+ setValidators(List<String>) : void
+ addNewValidator(String) : void
+ removeValidator(String) : void
+ getFormatters() : List<String>
+ setFormatters(List<String>) : void
+ addNewFormatter(String) : void
+ removeFormatter(String) : void
«interface»
IConfiguration
+ saveCurrentConfiguration() : void
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ stopDRE() : void
+ startDRE() : void
«interface»
IDataRetrieval
+ executeQuery(Query) : QueryResult
+ prepareBackup() : QueryResult
pl.cyfronet.virolab.protos.dre.impl
Lifecycle
impl::DataRetrievalImplementation

Figure 6.1. UML class diagram of DRE's core interfaces. Diagram models behaviour of the component in main three aspects: configuration attributes, configuration model and retrieval functionality. Also, reference implementation of DRE is shown.

Following list comprises descriptions of these interfaces:

- **IConfigurationBean**
  This interface defines methods for manipulating configuration properties of the DRE. In fact, interface sticks to the Java bean model (thus name suffix) with additional, convenience methods (as *addNewX* and *removeX*). The interface is described in great details in the configuration section A.
- **IConfiguration**
  Interface defining methods for handling actual component configuration. Sticks to the stateful component model, described in separate section A.2.
- **IDataRetrieval**
  Defines essential behavior of the component. First method is used to execute

queries, whether second allows for preparing backup of the whole storage. These methods operate with usage of common data model, described in section 6.1.7.

Those interfaces are implemented by the **DRE** facade. This is shown in Fig. 6.2, where class **DataRetrievalImplementation** is reference implementation of the **DRE** facade.

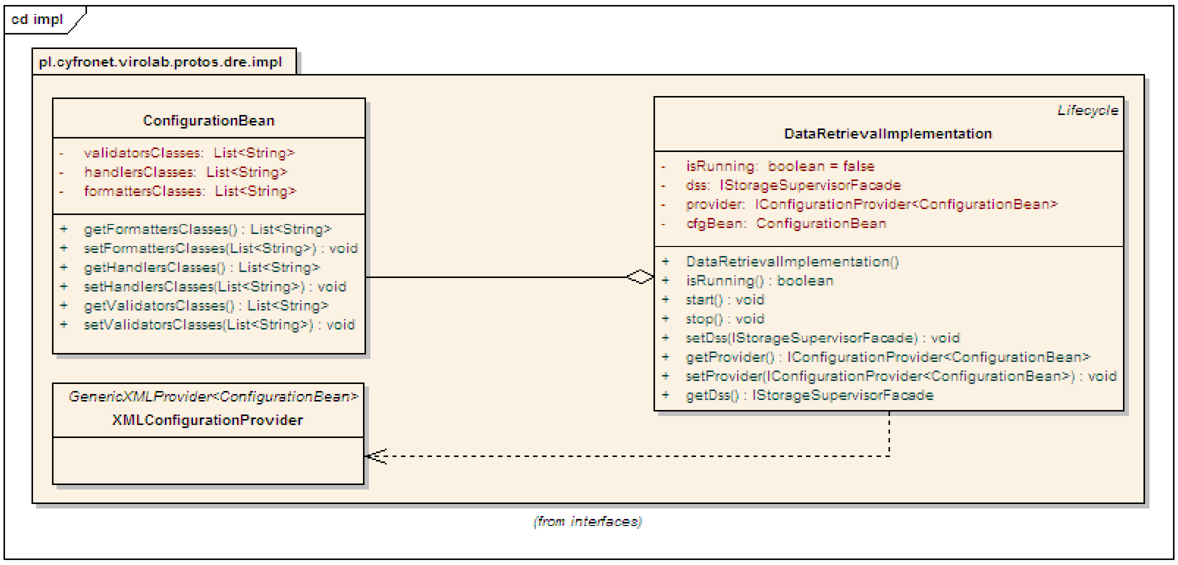

cd impl
pl.cyfronet.virolab.protos.dre.impl
ConfigurationBean
- validatorsClasses: List<String>
- handlersClasses: List<String>
- formattersClasses: List<String>
+ getFormattersClasses() : List<String>
+ setFormattersClasses(List<String>) : void
+ getHandlersClasses() : List<String>
+ setHandlersClasses(List<String>) : void
+ getValidatorsClasses() : List<String>
+ setValidatorsClasses(List<String>) : void
GenericXMLProvider<ConfigurationBean>
XMLConfigurationProvider
Lifecycle
DataRetrievalImplementation
- isRunning: boolean = false
- dss: IStorageSupervisorFacade
- provider: IConfigurationProvider<ConfigurationBean>
- cfgBean: ConfigurationBean
+ DataRetrievalImplementation()
+ isRunning() : boolean
+ start() : void
+ stop() : void
+ setDss(IStorageSupervisorFacade) : void
+ getProvider() : IConfigurationProvider<ConfigurationBean>
+ setProvider(IConfigurationProvider<ConfigurationBean>) : void
+ getDss() : IStorageSupervisorFacade
(from interfaces)

Figure 6.2. DRE implementation as UML class diagram. Diagram presents particulars of reference implementation (apart from core interface's methods) along with configuration-specific classes - property holder and provider.

Apart from interface, the **DRE** facade also defines following methods:

- Methods *start*, *stop* and *isRunning* come from the Spring **Lifecycle** interface and are delegated to generic, defined in **IConfiguration** interface.
- Methods *getDSS* and *setDSS* are used by the DI container to inject instance of storage supervising component, vital for the **DRE** operations.
- Accessor methods for *provider* property are used to inject adequate **GenericXML-Provider** instance. It is used for configuration persistence. Reference implementation - textbfXMLConfigurationProvider - is also depicted in Fig. 6.2.

Class **ConfigurationBean** also shown on 6.2 encapsulates configuration properties of the component. Facade delegates accessor methods defined in the **IConfigurationBean** to this class.
Core **DRE** functionalities includes handling various queries, validating them and formatting results. Model for classes responsible for mentioned operations is depicted in Fig. 6.3.

Model includes:

- Interface **IQueryResultFormatter** that should be implement by all classes processing query results. Results will be passed through all active instances in order defined by the list.

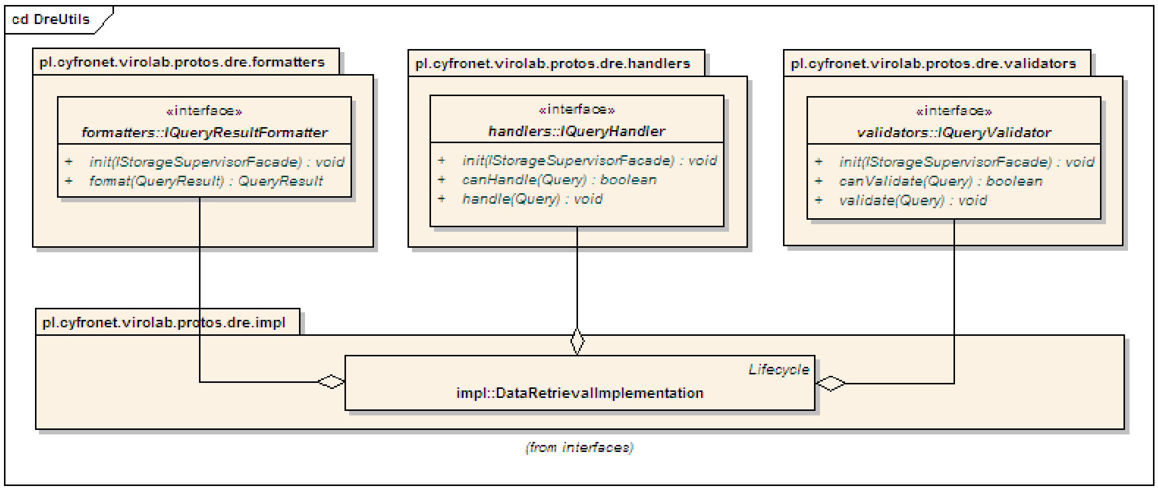

cd DreUtils
pl.cyfronet.virolab.protos.dre.formatters
«interface»
formatters::IQueryResultFormatter
+ init(IStorageSupervisorFacade) : void
+ format(QueryResult) : QueryResult
pl.cyfronet.virolab.protos.dre.handlers
«interface»
handlers::IQueryHandler
+ init(IStorageSupervisorFacade) : void
+ canHandle(Query) : boolean
+ handle(Query) : void
pl.cyfronet.virolab.protos.dre.validators
«interface»
validators::IQueryValidator
+ init(IStorageSupervisorFacade) : void
+ canValidate(Query) : boolean
+ validate(Query) : void
pl.cyfronet.virolab.protos.dre.impl
Lifecycle
impl::DataRetrievalImplementation
(from interfaces)

Figure 6.3. UML class diagram depicting DRE's utilities. Classes are partitioned on three functional groups: query handlers, query validators and result formatters. Each group - interface and implementations is encapsulated in separate package.

- Interface **IQueryHandler** that should be implemented by all classes handling particular query type and language. Query will be passed for handling only if *canHandle* method will return true.
- Interface **IQueryValidator** that shall be implemented by all query validators. Query will be validated by a class only if *canValidate* method will return true. Decision is to be based on Query attributes: language, type and options.

All itemized interfaces contains method *init*, used by the **DRE** to pass **DSS** instance.

Dependencies

Fig. 6.4 depicts external dependencies of the component.

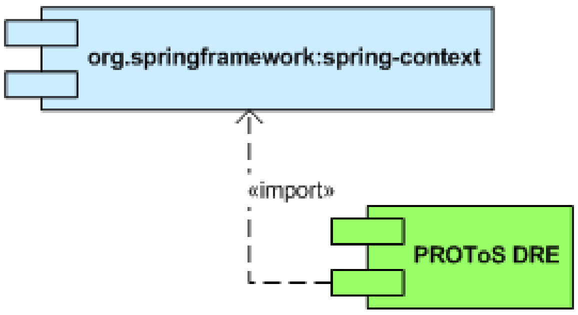


Figure 6.4. External dependencies of the PROToS DRE component. Depicted one - spring-context - is used for tight integration with Spring's lifecycle.

External module *spring-context* provides lifecycle interface method, for better integration with the Spring. This way, methods for starting and stopping **DRE** component will be automatically called by the container when needed (as when starting and stopping whole container).

6.1.3. Protos-dge component

Description

The *protos-dge* main responsibilities are data gathering functionalities, as accepting, validating and storing incoming provenance data. In regard of event storage, the component is dependent of the **Distributed Storage Supervisor** (DSS) component.

Design

Source code packages of the component begins with the **pl.cyfronet.virolab.-protos.dge** prefix.
Fig. 6.5 presents essential interfaces of the **DGE** component.

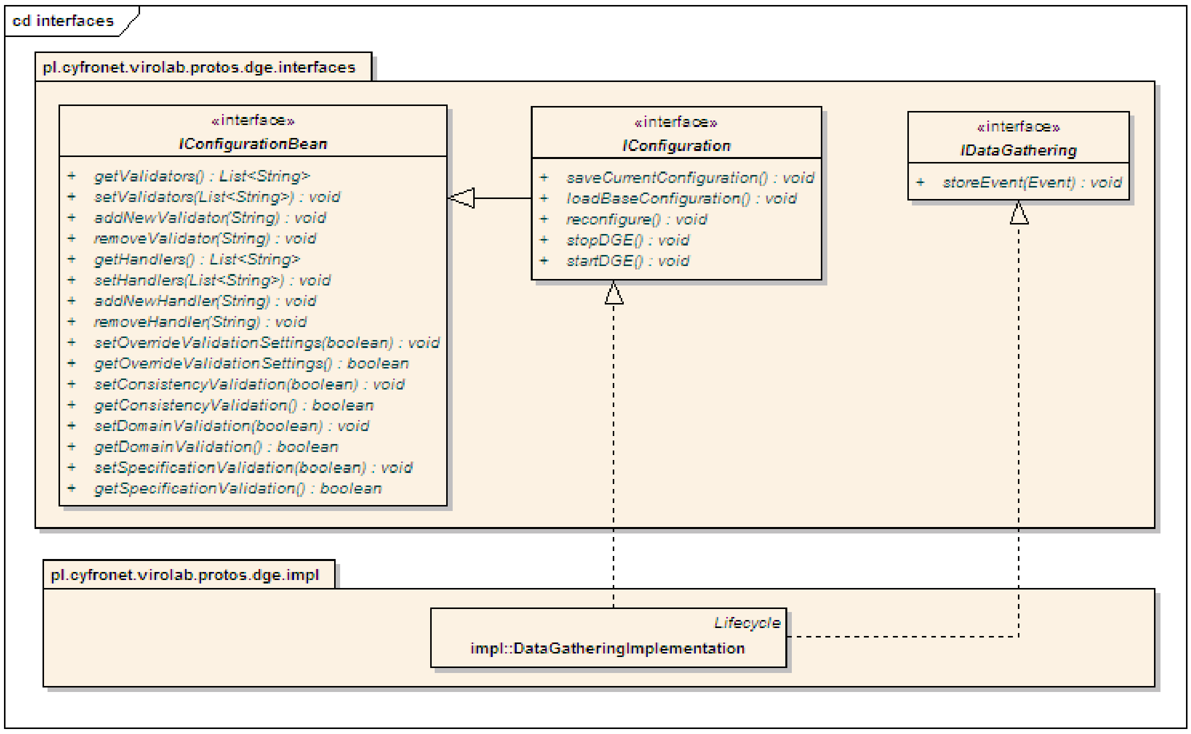

cd interfaces
pl.cyfronet.virolab.protos.dge.interfaces
«interface»
IConfigurationBean
+ getValidators() : List<String>
+ setValidators(List<String>) : void
+ addNewValidator(String) : void
+ removeValidator(String) : void
+ getHandlers() : List<String>
+ setHandlers(List<String>) : void
+ addNewHandler(String) : void
+ removeHandler(String) : void
+ setOverrideValidationSettings(boolean) : void
+ getOverrideValidationSettings() : boolean
+ setConsistencyValidation(boolean) : void
+ getConsistencyValidation() : boolean
+ setDomainValidation(boolean) : void
+ getDomainValidation() : boolean
+ setSpecificationValidation(boolean) : void
+ getSpecificationValidation() : boolean
«interface»
IConfiguration
+ saveCurrentConfiguration() : void
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ stopDGE() : void
+ startDGE() : void
«interface»
IDataGathering
+ storeEvent(Event) : void
pl.cyfronet.virolab.protos.dge.impl
Lifecycle
impl::DataGatheringImplementation

Figure 6.5. DGE interfaces on UML class diagram. Interfaces represent core functionalities: configuration setting, configuration management and data gathering. Reference implementation facade of the component is also depicted.

Description of interfaces functionalities follows:

- **IConfigurationBean**
  Interface defining how component's configuration properties could be altered. It is composed of Java-bean accessors and additional, convenience methods, as *addNewHandler*. The **DGE** configuration properties are covered in details on the section A.
- **IConfiguration**
  This interface adds stateful configuration behavior to the **DGE** component. Thorough explanation of the stateful model is in section A.2.

- **IDataGathering**
  Core interface of the component, defining it's most vital function. Operates on common data model, described in 6.1.7.

Fig. 6.6 depicts reference **DGE** facade, implementing all essential interfaces covered above.

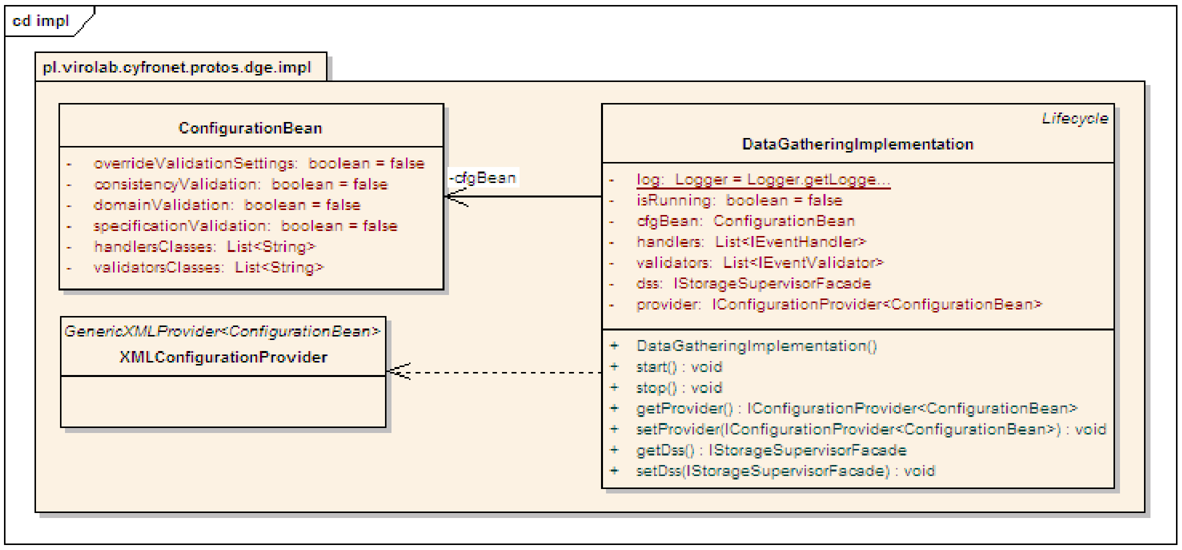

cd impl
pl.virolab.cyfronet.protos.dge.impl
ConfigurationBean
- overrideValidationSettings: boolean = false
- consistencyValidation: boolean = false
- domainValidation: boolean = false
- specificationValidation: boolean = false
- handlersClasses: List<String>
- validatorsClasses: List<String>
-cfgBean
GenericXMLProvider<ConfigurationBean>
XMLConfigurationProvider
Lifecycle
DataGatheringImplementation
- log: Logger = Logger.getLogge...
- isRunning: boolean = false
- cfgBean: ConfigurationBean
- handlers: List<IEventHandler>
- validators: List<IEventValidator>
- dss: IStorageSupervisorFacade
- provider: IConfigurationProvider<ConfigurationBean>
+ DataGatheringImplementation()
+ start() : void
+ stop() : void
+ getProvider() : IConfigurationProvider<ConfigurationBean>
+ setProvider(IConfigurationProvider<ConfigurationBean>) : void
+ getDss() : IStorageSupervisorFacade
+ setDss(IStorageSupervisorFacade) : void

Figure 6.6. DGE implementation UML class diagram. Reference implementation along with configuration utilities (attribute holder, provider) are presented. Also, non-interface methods and attributes of the component are shown.

Additional methods of the **DataGatheringImplementation** class are:

- Accessor methods for *DSS* are used to inject instance of storage supervising component.
- Accessor methods for *provider* property are used to inject **GenericXMLProvider** instance. The provider is used for managing configuration persistence.
- Methods *start*, *stop* and *isRunning* come from the Spring **Lifecycle** interface. These methods provide tight integration with Spring's stateful component model. The implementation delegates *start* and *stop* calls to generic *startDGE* and *stopDGE* methods.

Implementation package contains also following classes:

- **XMLConfigurationProvider** is simple extension of the **GenericXMLProvider** handling DGE's ConfigurationBean.
- **ConfigurationBean** encapsulates configuration properties of the component. It's persistence is handled by the **XMLconfigurationProvider**.

As stated in the description, DGE responsibilities includes handling and validating of incoming events. For these functionalities specific interfaces are used. They are presented in Fig. 6.7.

Interfaces descriptions:

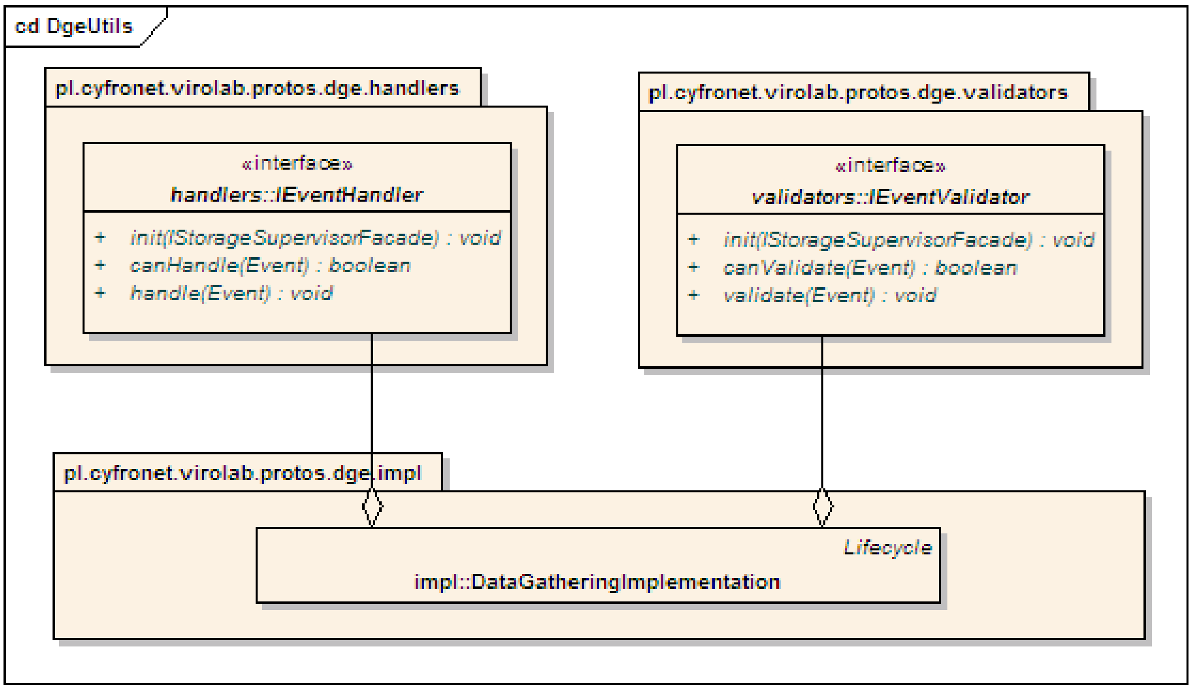

cd DgeUtils
pl.cyfronet.virolab.protos.dge.handlers
«interface»
handlers::IEventHandler
+ init(IStorageSupervisorFacade) : void
+ canHandle(Event) : boolean
+ handle(Event) : void
pl.cyfronet.virolab.protos.dge.validators
«interface»
validators::IEventValidator
+ init(IStorageSupervisorFacade) : void
+ canValidate(Event) : boolean
+ validate(Event) : void
pl.cyfronet.virolab.protos.dge.impl
Lifecycle
impl::DataGatheringImplementation

Figure 6.7. UML class diagram of DGE's utilities, handling two important responsibilities of the component: validating and handling incoming events. Each utility group is encapsulated in a separate package.

- **IEventHandler** is to be implemented by handlers for specific events. Event instance will be passed to the handler only if *canHandle* method returns true. Decision has to be made basing on *type* and *options* attributes.
- **IEventValidator** is to be implemented by validators for specific events. Event instance will be passed to the validator only if *canValidate* method returns true.

Every interface contains also method *init*. It is used by the **DGE** component to pass **DSS** instance, when putting handler/validator into service.

Dependencies

External dependencies of the **DGE** component are presented in Fig. 6.8.

Component *spring-context* plays same role as in the case of **DRE** module, providing tight Spring integration. Library *jena* provides ontology processing capabilities for the **DGE** logic.

6.1.4. Protos-dss component

Description

The *protos-dss* component encapsulates all logic required for maintaining, configuring and accessing distributed storage of the PROToS system. Main facade of the component acts as a mediator between storage and other components (DRE, DGE), providing ontology-based routing algorithms for provenance data. What is more, all of Domain Ontologies are to be stored in the **DSS**, for performing some ontology -

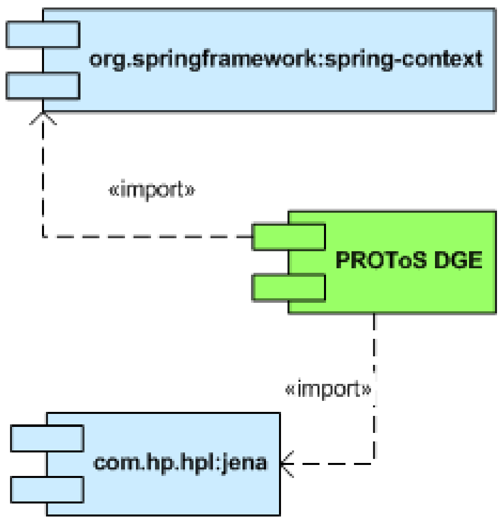


Figure 6.8. External dependencies of the PROToS DGE component. Diagram include spring-context for Spring's Lifecycle interface and Jena ontology framework.

related functionalities, as data validation. This also assures best performance of these tasks, as data do not have to fetched from distributed data store.

Design

All source code of this component is placed in the packages with **pl.cyfronet.-virolab.protos.dss** prefix.
In Fig. 6.9 all interfaces of the **DSS** component are presented. Because the **DSS** makes use of the *Facade* design pattern, all vital interfaces are extended by one super interface, called **IStorageSupervisorFacade**.

Particular **DSS facade** interfaces are described below.

- **IOntologyConfigurationBean**
  The interface defining how ontology attributes of the **DSS** can be altered. Sticks to the Java-bean model, with additional convenience method used for manipulating collections. This type of interface definition allows for Java-bean config holders that can be easily persisted. **IOntologyConfigurationBean** is covered in details in the configuration section A.
- **IConfigurationBean**
  Second 'bean'-type configuration interface of the **DSS**. This allows for changing attributes related to distributed storage and ontology reasoning. Also precisely described in the configuration section A.
- **IConfigurationAPI**
  This interface extends preceding 'bean' interfaces forming full configuration model of the **DSS** component. It also adds methods required for stateful operation of the component. Full description of the stateful model is given in the section A.2.
- **IOntologyConfiguration**
  The interface that defines methods for altering ontology state of the **DSS**. These methods are excluded from the **IOntologyConfigurationBean** for two reasons. Firstly, they are manipulating state of the component, not static configuration.

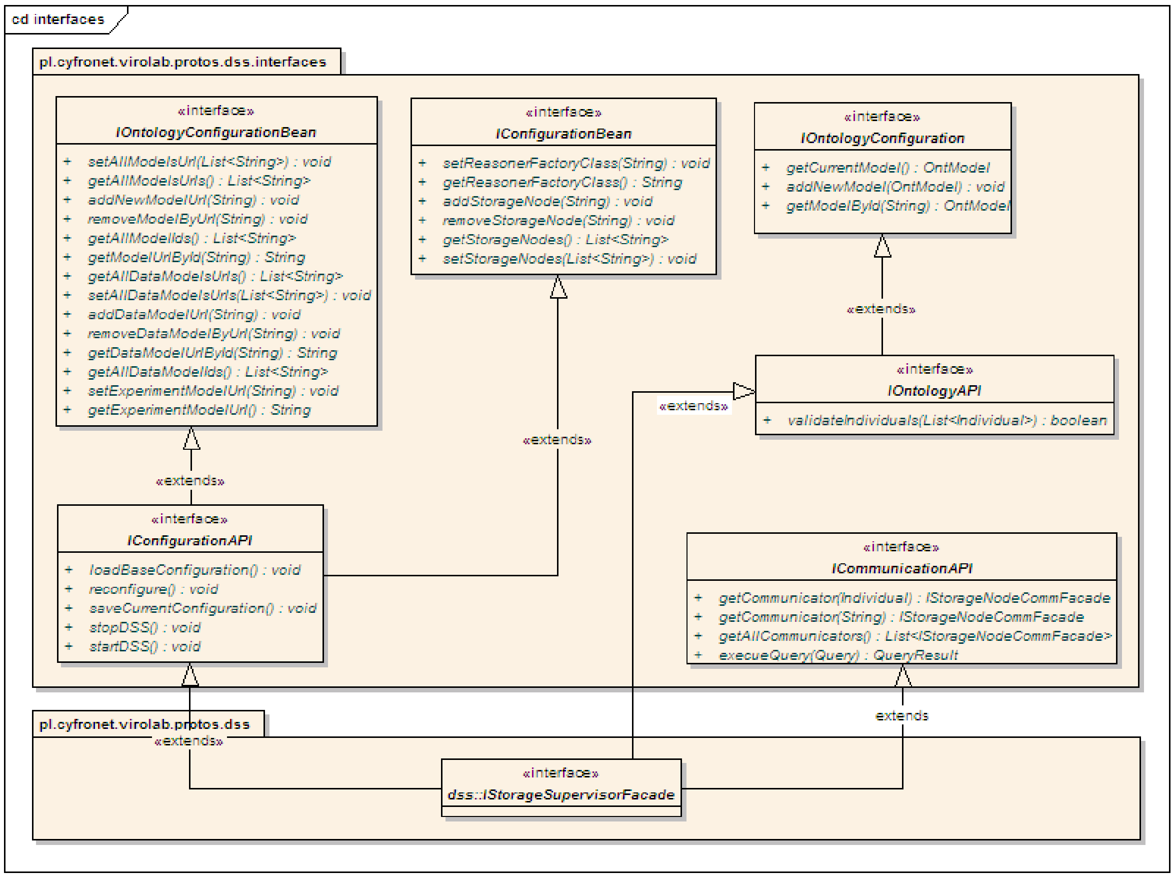

cd interfaces
pl.cyfronet.virolab.protos.dss.interfaces
«interface» IOntologyConfigurationBean
+ setAllModelsUrl(List<String>) : void
+ getAllModelsUrls() : List<String>
+ addNewModelUrl(String) : void
+ removeModelByUrl(String) : void
+ getAllModelIds() : List<String>
+ getModelUrlById(String) : String
+ getAllDataModelsUrls() : List<String>
+ setAllDataModelsUrls(List<String>) : void
+ addDataModelUrl(String) : void
+ removeDataModelByUrl(String) : void
+ getDataModelUrlById(String) : String
+ getAllDataModelIds() : List<String>
+ setExperimentModelUrl(String) : void
+ getExperimentModelUrl() : String
«interface» IConfigurationBean
+ setReasonerFactoryClass(String) : void
+ getReasonerFactoryClass() : String
+ addStorageNode(String) : void
+ removeStorageNode(String) : void
+ getStorageNodes() : List<String>
+ setStorageNodes(List<String>) : void
«interface» IOntologyConfiguration
+ getCurrentModel() : OntModel
+ addNewModel(OntModel) : void
+ getModelById(String) : OntModel
«extends»
«interface» IOntologyAPI
+ validateIndividuals(List<Individual>) : boolean
«extends»
«extends»
«extends»
«interface» IConfigurationAPI
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ saveCurrentConfiguration() : void
+ stopDSS() : void
+ startDSS() : void
«interface» ICommunicationAPI
+ getCommunicator(Individual) : IStorageNodeCommFacade
+ getCommunicator(String) : IStorageNodeCommFacade
+ getAllCommunicators() : List<IStorageNodeCommFacade>
+ execueQuery(Query) : QueryResult
pl.cyfronet.virolab.protos.dss
«extends»
extends
«interface» dss::IStorageSupervisorFacade

Figure 6.9. DSS essential interfaces as UML class diagram. Apart from reference implementation, behaviour in following aspects is modelled: common configuration attributes, ontology configuration attributes, configuration management, ontology management and distributed storage communication.

Secondly, they should be available for remote management by user, only for internal use. More on this topic is available in the section A.
Interface operates on the Jena [18] ontology data model.

- **IOntologyAPI**
  Interface extends **IOntologyConfiguration** forming unified API for ontology functionalities offered by the **DSS**. The method *validateIndividual* will only return true if passed list of individuals are consistent with DOs loaded into **DSS**.
- **ICommunicationAPI**
  Interface providing actual access to the distributed storage. Method *executeQuery* is entirely handled by the **DSS** because typical query can span multiple nodes, therefore results have to be gathered and merged. Nonetheless, processing of data storage is to be handled by single nodes. Thus, adequate methods (*getCommunicator*) returns single communicator to applicable node. Hence, better performance is achieved because part of processing is delegated to the responsible node.

Reference implementation of the **IStorageSupervisorFacade** is depicted in Fig. 6.10 in form of the class **StorageSupervisor**.

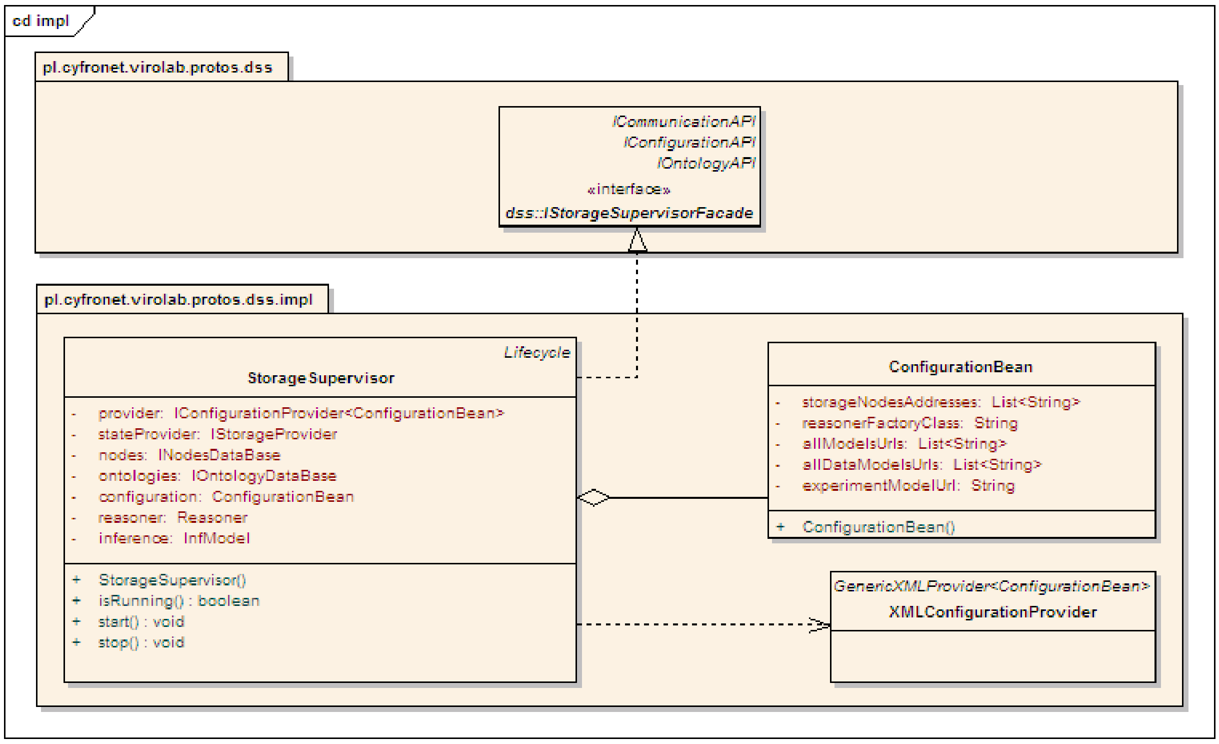

cd impl
pl.cyfronet.virolab.protos.dss
ICommunicationAPI
IConfigurationAPI
IOntologyAPI
«interface»
dss::IStorageSupervisorFacade
pl.cyfronet.virolab.protos.dss.impl
Lifecycle
StorageSupervisor
- provider: IConfigurationProvider<ConfigurationBean>
- stateProvider: IStorageProvider
- nodes: INodesDataBase
- ontologies: IOntologyDataBase
- configuration: ConfigurationBean
- reasoner: Reasoner
- inference: InfModel
+ StorageSupervisor()
+ isRunning() : boolean
+ start() : void
+ stop() : void
ConfigurationBean
- storageNodesAddresses: List<String>
- reasonerFactoryClass: String
- allModelsUrls: List<String>
- allDataModelsUrls: List<String>
- experimentModelUrl: String
+ ConfigurationBean()
GenericXMLProvider<ConfigurationBean>
XMLConfigurationProvider

Figure 6.10. UML class diagram of DSS reference implementation. Particulars of facade implementation are shown along with configuration data holder and suitable configuration provider.

Following list comprises descriptions of class methods and attributes:

- *provider* attribute contains reference to the configuration persistence utility. Suitable implementation is also depicted in Fig. 6.10 as class **XMLConfigurationProvider**.
- *stateProvider* attribute contains reference to the component state persistence manager. Reference implementation comes from the *protos-pstore* component, described in section 6.1.11.
- *nodes* attribute contains reference to the storage nodes data base. This is very important class because it's algorithms are to decide how provenance data should be scattered and balanced on the distributed storage. It's facade interface is shown in Fig. 6.11.
- *ontologies* attribute contains reference to the domain ontology models data base. It simply encapsulates logic necessary for extracting and storing various information form DOs. Reference implementation is placed in the *protos-onts* module, covered in details in the section 6.1.10.
- *configuration* fields contains instance of the **ConfigurationBean**, also depicted in Fig. 6.10. This simple container, Java bean class contains current configuration attributes of the **DSS** component.
- *reasoner* and *inference* attributes are used for semantic processing of loaded Domain Ontologies. They are used for such tasks as quick, incoming event validation.
- methods *start*, *stop* *isRunning* comes from Spring's **Lifecycle** interface. Those

methods are used for tight, seamless integration with Spring's stateful component model.

Apart from nodes data base, Fig. 6.11 depicts also one more important interface - **INodeReferenceFactory**.

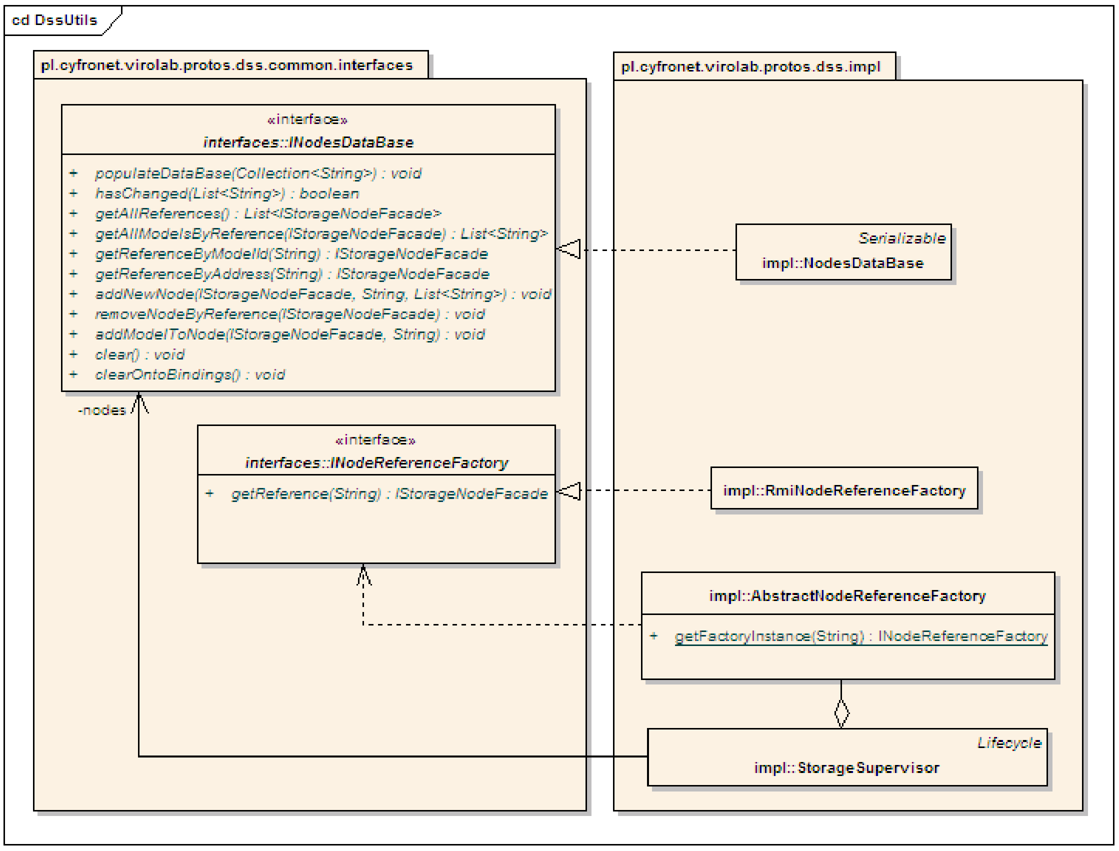

cd DssUtils
pl.cyfronet.virolab.protos.dss.common.interfaces
pl.cyfronet.virolab.protos.dss.impl
«interface»
interfaces::INodesDataBase
+ populateDataBase(Collection<String>) : void
+ hasChanged(List<String>) : boolean
+ getAllReferences() : List<IStorageNodeFacade>
+ getAllModelsByReference(IStorageNodeFacade) : List<String>
+ getReferenceByModelId(String) : IStorageNodeFacade
+ getReferenceByAddress(String) : IStorageNodeFacade
+ addNewNode(IStorageNodeFacade, String, List<String>) : void
+ removeNodeByReference(IStorageNodeFacade) : void
+ addModelToNode(IStorageNodeFacade, String) : void
+ clear() : void
+ clearOntoBindings() : void
Serializable
impl::NodesDataBase
-nodes
«interface»
interfaces::INodeReferenceFactory
+ getReference(String) : IStorageNodeFacade
impl::RmiNodeReferenceFactory
impl::AbstractNodeReferenceFactory
+ getFactoryInstance(String) : INodeReferenceFactory
Lifecycle
impl::StorageSupervisor

Figure 6.11. UML class diagram of DSS component's utilities. Classes presented include very important Nodes database (interface and reference implementation) and node reference management. The latter is build around Abstract Factory pattern, including core AbstractNodeReferenceFactory and interface for factories handling particular protocols. Also, reference implementation of factory for RMI is depicted.

Together with class **AbstractNodeReferenceFactory** and interface **IStorageNodeFacade** it forms utility for obtaining usable node references from various addresses. The utility is using *Abstract Factory* design pattern. Currently, the **DSS** component offers node reference factory implementation for the RMI - class **RmiNodeReferenceFactory**, as shown on Fig. 6.11.

Dependencies

Fig. 6.12 depicts external dependencies of the component.

Just as in the case of previous core components, there are only two of them: *jena*, used for ontology processing and *spring-context*, used for tight integration with the Spring lifecycle model.

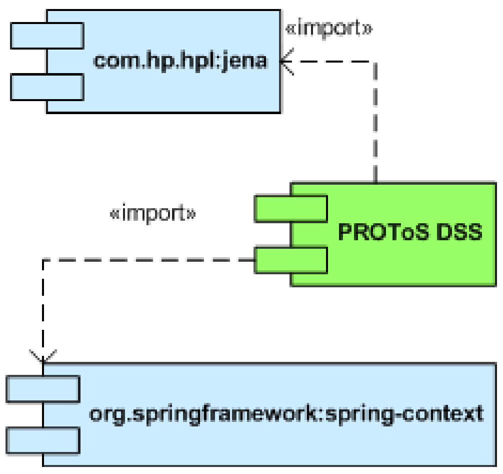


Figure 6.12. PROToS DSS external dependencies component diagram. Diagram includes Jena ontology framework and Spring lifecycle integration.

6.1.5. Protos-ssn component

Description

The *protos-ssn* module contains both interfaces for essential **Node application** functionality and implementation of the **Storage Super Node** component. From perspective of the **DSS**, whole storage node looks like one being - Node application. Under the hood, the **SSN** component manages group of **Storage Peers**. It's role also includes management of one or more DOs, therefore handling all ontology-related processing of the **Node**, as reasoning on individual registries. Basically, all work that can't be delegated to]textbfStorage Peers is to be performed here. For example, queries in languages other than XQuery have to be processed by the **Storage Super Node**.

Node interface model

Interfaces constituting Node lies in the **pl.cyfronet.protos.sn** and **pl.cyfronet.-protos.sn.interfaces** packages.
Fig. 6.13 depicts all interfaces defining required **Node** behavior. As shown, interface model makes use of the *Facade* design pattern - thus suffix **Facade** in two **SN** parent interfaces. First, the **IStorageNodeCommFacade** represents behavior that should be presented to components requesting access to storage functionalities, as **DRE**. That is why configuration interfaces are extended only by the second parent facade - **IStorageNodeFacade**. This interface is used by storage management components of the PROToS.

Following list is composed of short interface's descriptions:

- **IConfigurationBean**
  This interface defines methods for **SN** configuration attributes. Methods naming follows Java-bean model with additional convenience method (as unit operation on collection). This way implemented configuration holder can be simply, easy to use

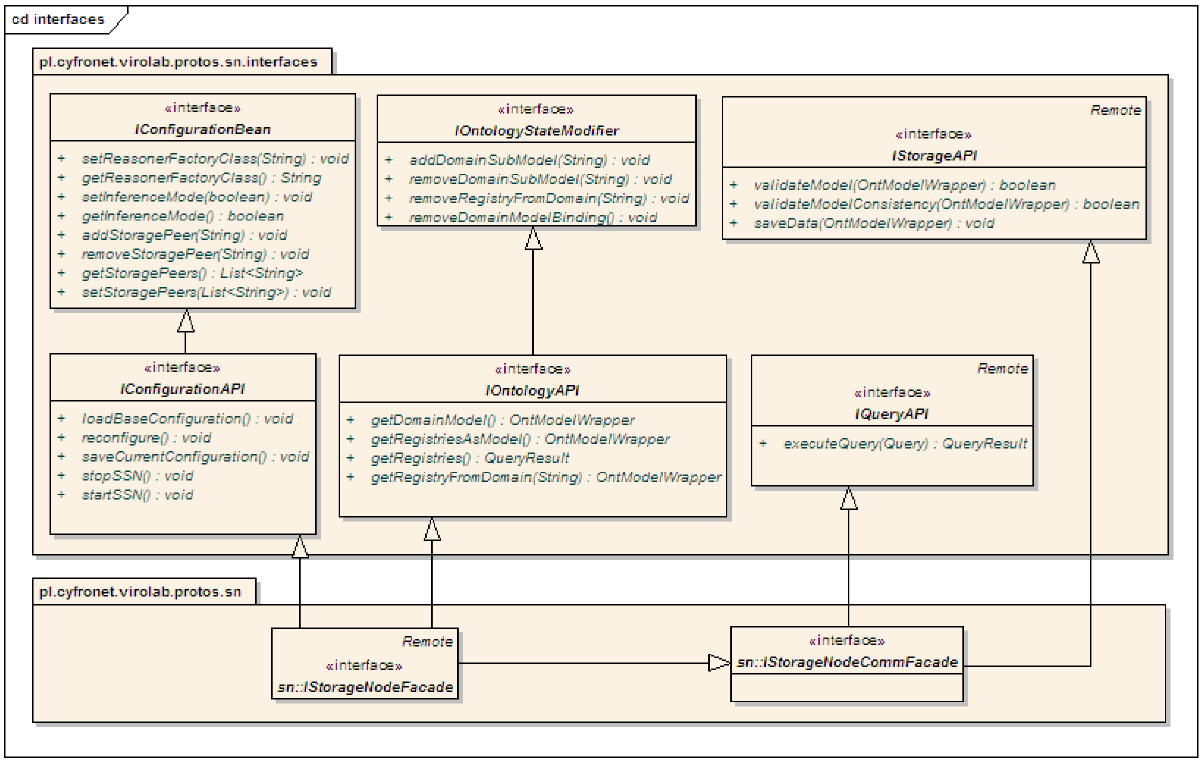

cd interfaces
pl.cyfronet.virolab.protos.sn.interfaces
«interface»
IConfigurationBean
+ setReasonerFactoryClass(String) : void
+ getReasonerFactoryClass() : String
+ setInferenceMode(boolean) : void
+ getInferenceMode() : boolean
+ addStoragePeer(String) : void
+ removeStoragePeer(String) : void
+ getStoragePeers() : List<String>
+ setStoragePeers(List<String>) : void
«interface»
IOntologyStateModifier
+ addDomainSubModel(String) : void
+ removeDomainSubModel(String) : void
+ removeRegistryFromDomain(String) : void
+ removeDomainModelBinding() : void
Remote
«interface»
IStorageAPI
+ validateModel(OntModelWrapper) : boolean
+ validateModelConsistency(OntModelWrapper) : boolean
+ saveData(OntModelWrapper) : void
«interface»
IConfigurationAPI
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ saveCurrentConfiguration() : void
+ stopSSN() : void
+ startSSN() : void
«interface»
IOntologyAPI
+ getDomainModel() : OntModelWrapper
+ getRegistriesAsModel() : OntModelWrapper
+ getRegistries() : QueryResult
+ getRegistryFromDomain(String) : OntModelWrapper
Remote
«interface»
IQueryAPI
+ executeQuery(Query) : QueryResult
pl.cyfronet.virolab.protos.sn
Remote
«interface»
sn::IStorageNodeFacade
«interface»
sn::IStorageNodeCommFacade

Figure 6.13. Storage Node interfaces as UML class diagram. Interfaces model behaviour in following aspects: configuration properties and management, ontology management, storage and querying functionalities. Also, diagram presents two Node facades: full as used by the DSS component and communicator as presented to other components.

Java bean. Thorough descriptions of interface's methods are available in Appendix A.

- **IConfigurationAPI**
  The interface defining methods of stateful component model applied to the **Node**. The model itself is covered in details in separate section A.2.
- **IOntologyStateModifier**
  Interface that defines ontology-related functionalities actually modifying state of the component. These methods basically allows for adding and removing DOs bound to the particular **SN**. The method *removeRegistryFromDomain* forces component to remove registry of individuals bound to specific DO.
- **IOntologyAPI**
  This interface defines methods for ontology processing that do not change state of the **SN** component. As opposed to previous interface, this defines only 'getter ' methods. Also, by extending **IOntologyStateModifier** forms complete ontology processing facade of the component.
- **IStorageAPI**
  Interface defining methods operating directly on storage managed by particular

SN. Apart from essential *saveData* method it contains also validation functionalities in form of *validateModel* and *validateModelConsistency* methods.

- **IQueryAPI**
  The interface that defines only one method - *executeQuery*. This functionality is detached from other storage-accessing methods because it requires much more complicated implementation and typically access to all **Storage Peers** bound to the node.

Node interfaces description requires one more remark. As depicted on the diagram 6.13, interface's methods operates on class **OntModelWrapper**, not on the generic Jena data model. This is special class that makes ontology model serializable and allows for sending OntModel instance with such protocols as RMI. Because in section 4.2 RMI was pointed as a candidate for internal middleware, such wrapper class had to be included in design.

SSN core design

All classes and interfaces of the **SSN** are placed in packages with **pl.cyfronet.-virolab.protos.sn.ssn** prefix.
Diagram 6.14 depicts essential classes of the reference **SSN** implementation design.

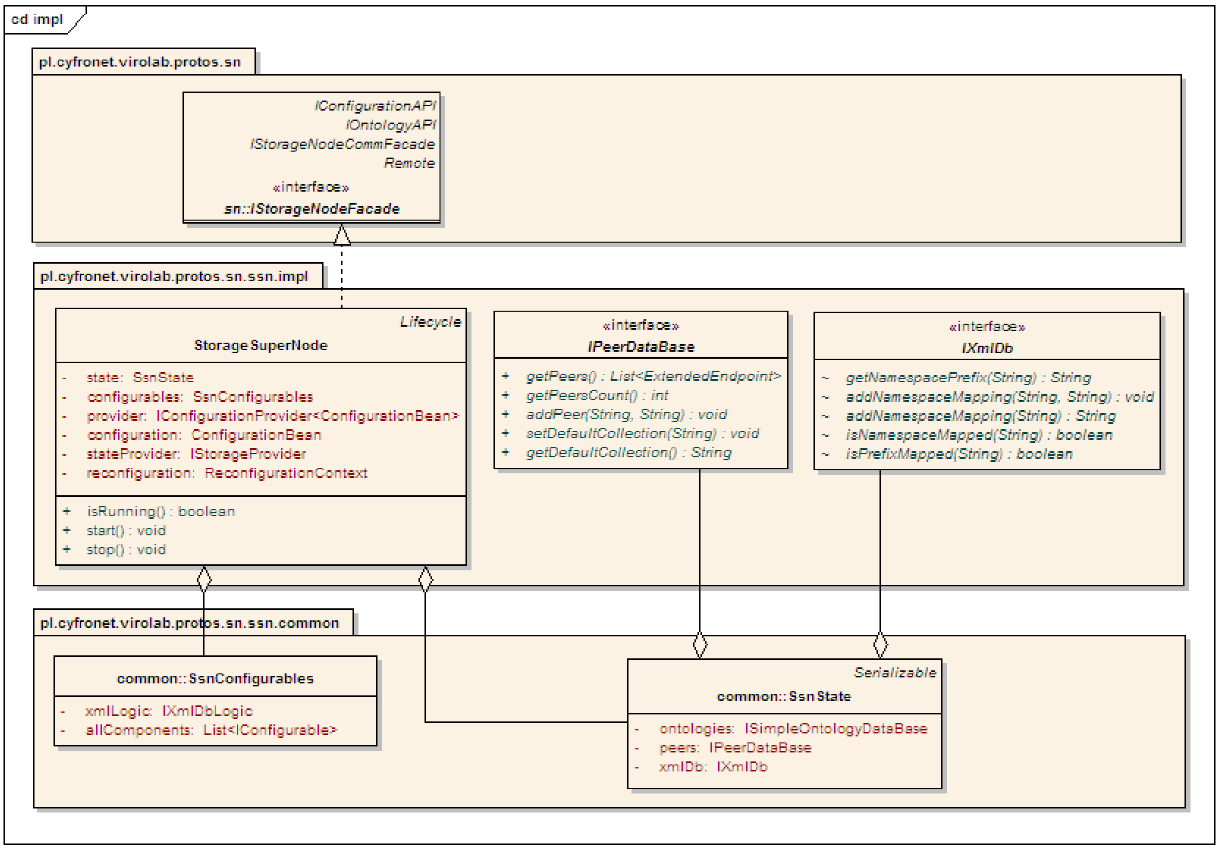

cd impl
pl.cyfronet.virolab.protos.sn
IConfigurationAPI
IOntologyAPI
IStorageNodeCommFacade
Remote
«interface»
sn::IStorageNodeFacade
pl.cyfronet.virolab.protos.sn.ssn.impl
Lifecycle
StorageSuperNode
- state: SsnState
- configurables: SsnConfigurables
- provider: IConfigurationProvider<ConfigurationBean>
- configuration: ConfigurationBean
- stateProvider: IStorageProvider
- reconfiguration: ReconfigurationContext
+ isRunning() : boolean
+ start() : void
+ stop() : void
«interface»
IPeerDataBase
+ getPeers() : List<ExtendedEndpoint>
+ getPeersCount() : int
+ addPeer(String, String) : void
+ setDefaultCollection(String) : void
+ getDefaultCollection() : String
«interface»
IXmlDb
~ getNamespacePrefix(String) : String
~ addNamespaceMapping(String, String) : void
~ addNamespaceMapping(String) : String
~ isNamespaceMapped(String) : boolean
~ isPrefixMapped(String) : boolean
pl.cyfronet.virolab.protos.sn.ssn.common
common::SsnConfigurables
- xmlLogic: IXmlDbLogic
- allComponents: List<IConfigurable>
Serializable
common::SsnState
- ontologies: ISimpleOntologyDataBase
- peers: IPeerDataBase
- xmlDb: IXmlDb

Figure 6.14. SSN implementation as UML class diagram. Diagram contains reference implementation of the full facade interface with its non-interface properties. Also, critical implementation groups - Configurables and State are presented.

Main class is the **StorageSuperNode**, implementing **IStorageSuperNode** interface, described previously. It's core attributes and methods are described as follows:

- *configuration* attribute contains reference to the holder of component's configuration data. This is simple Java bean class, convenient for persistence.
- *provider* instance field contains reference to the configuration's persistence manager. Manager is provided by the *protos-config* component, described in the section 6.1.9.
- *state* attribute contains instance of the **SsnState** class, also depicted in Fig. 6.14.
  - — **SsnState**
    This is Java-bean containing all classes that constitute component's run-time state. All instances contained have to be persisted on component's stop and restored on component's start. Currently there are three state interfaces:
  - — **ISimpleOntologyDataBase**
    Encapsulates logic necessary for performing ontology loading and storage. Interface and implementations come from *protos-onts* module and are described in section 6.1.10.
  - — **IXmlDb**
    One of many aspects of the **SSN** operations is management of incoming XML data. Coming from unified model libraries, but different sources, events in XML representation can differ in for example namespace mappings. Role of this class is to learn incoming data and enable unification before saving events in common data store. It does not depend on component's configuration but belong to state exactly because of learning in the run-time.
  - — **IPeerDataBase**
    Implementations of this interface shall manage **Storage Peers** available for the node. Core logic should implement parsing **StoragePeer** addresses and producing ExtendedEndpoint instances, allowing access to the **Peer**.
- *stateProvider* field contains reference to the state persistence utility. It's main concern is (re)storing instance of the **SsnState** class on component's lifecycle events. Interface as well as reference implementation lies in the *protos-pstore* module, described in section 6.1.11.
- *configurables* attribute contains reference to an instance of the **SsnConfigurables** class, shown in Fig. 6.14.
  - — **SsnConfigurables**
    This Java bean holds references to the configurables - classes dependent on current state and configuration. Those classes can be perfectly restored from component's state, therefore do not need to be persisted altogether with the state. **IConfigurable** interface and it's extensions are covered in section 6.1.5.
- *reconfiguration* field contains instance of the **ReconfigurationContext** class. This is main entry point of one of most important **SSN** algorithm - reconfiguration. Actual implementation decides how state (ontologies and peers) is to be set up. Current design is presented in the section 6.1.5.
- *start*, *stop* and *isRunning* methods come from the **Lifecycle** interface. It is used for seamless integration with Spring stateful component model. Those methods will be called automatically by the container when needed. Internally the **SSN** delegates those call to generic methods from the **IConfigurationaAPI** interface.

SSN reconfiguration design

Diagram 6.15 presents design of the reconfiguration model for the **SSN** component.

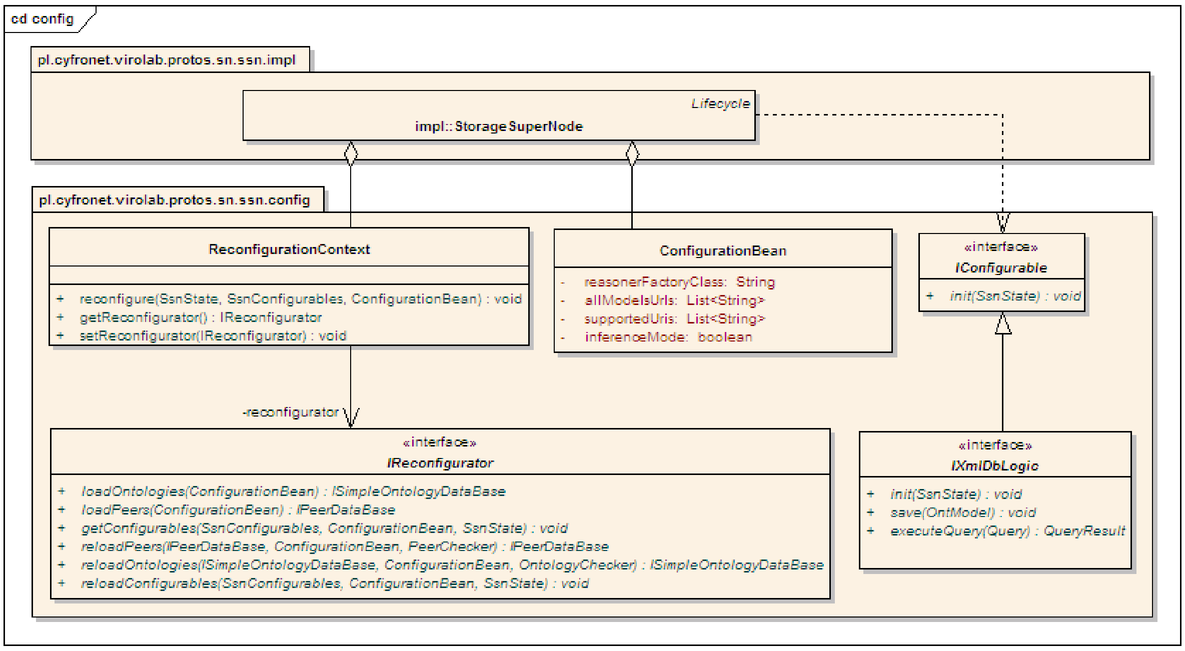

cd config
pl.cyfronet.virolab.protos.sn.ssn.impl
Lifecycle
impl::StorageSuperNode
pl.cyfronet.virolab.protos.sn.ssn.config
ReconfigurationContext
+ reconfigure(SsnState, SsnConfigurables, ConfigurationBean) : void
+ getReconfigurator() : IReconfigurator
+ setReconfigurator(IReconfigurator) : void
ConfigurationBean
- reasonerFactoryClass: String
- allModelsUrls: List<String>
- supportedUris: List<String>
- inferenceMode: boolean
«interface»
IConfigurable
+ init(SsnState) : void
-reconfigurator
«interface»
IReconfigurator
+ loadOntologies(ConfigurationBean) : ISimpleOntologyDataBase
+ loadPeers(ConfigurationBean) : IPeerDataBase
+ getConfigurables(SsnConfigurables, ConfigurationBean, SsnState) : void
+ reloadPeers(IPeerDataBase, ConfigurationBean, PeerChecker) : IPeerDataBase
+ reloadOntologies(ISimpleOntologyDataBase, ConfigurationBean, OntologyChecker) : ISimpleOntologyDataBase
+ reloadConfigurables(SsnConfigurables, ConfigurationBean, SsnState) : void
«interface»
IXmlDbLogic
+ init(SsnState) : void
+ save(OntModel) : void
+ executeQuery(Query) : QueryResult

Figure 6.15. UML class diagram of the SSN reconfiguration model. Model is based on Strategy design pattern and consists of following: ReconfigurationContext defining common algorithm steps and IReconfigurator interface, defining algorithm's particulars. Also, SSN configuration holder and configurables are part of the model.

Reconfiguration makes use of the **Strategy** design pattern. Entry point is the **ReconfigrationContext** class ('context' of the strategy). It defines following methods:

- accessors for the *reconfigurator* property. Actual implementation of the **IReconfigurator** interface is to be set by the DI container and handle particulars of reconfiguration algorithm.
- **reconfigure** method, starting reconfiguration sequence. This method takes **state**, **configurables** and actual **configuration** of the **SSN** component. It's responsibility is to check which elements of the configuration have been altered since state was set up and call adequate **IReconfigurator** methods. Also, if any changes are performed on the state, reconfigurables also have to be modified. Described checking sequence is generic and delegates all algorithm-specific functionalities to the **IReconfigurator** instance. Finally, after completion of this method state and configurables reflect actual component's configuration.

As stated previously, implementations of the **IReconfigurator** interface defines particulars of reconfiguration algorithms. Depicted in Fig. 6.15, interface defines following methods:

- *loadOntologies* method sets up ontology data base of the component. Actual implementation of the **ISimpleOntologyDataBase** interface is defined here. Also, initialization sequence from current configuration depends on chosen implementation.

- *loadPeers* method loads peers that are to be used by the **SSN**, using current component's configuration. Defines effective implementation of the **IPeerDataBase** used.
- *getConfigurables* method fills **SsnConfigurables** holder with all required implementations of the **IConfigurable** interface. Currently, only **IXmlDbLogic** is used.
  — **IXmlDbLogic**
  Implementation of this class handles saving and querying data contained in the **SSN**. It is very important, as algorithms for scattering data and merging distributed query results have very strong impact on system's performance and salability. Internally, implementations of this interface delegate XML:DB API logic to classes from the *protos-xmldb* module, covered in section 6.1.12.
- *reloadPeers* method is called when reconfiguration algorithm decides that peers have been changed. This stage of sequence can for example decide to transfer provenance data from nodes being discarded to new ones. Therefore it's implementation is also very important.
- *reloadOntologies* method is called when reconfiguration algorithm detects that ontology models bound to the **SSN** have been changed.
- *reloadConfigurables* method is called every time state (that is ontologies or peers) is changed. This is because such change have to reflected in configurables internal state. In case of the mentioned **IXmlDbLogic** this is crucial for reliable operation.

Dependencies

In Fig. 6.16, external dependencies of the **SSN** component are presented.

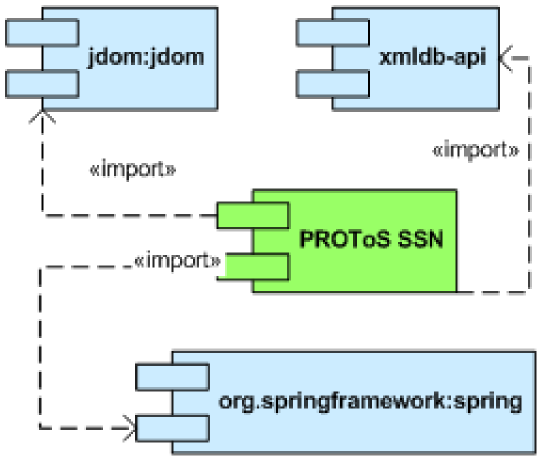


Figure 6.16. PROToS SSN external dependencies as UML component diagram. Diagram includes Spring's lifecycle integration, Jena ontology framework and XML:DB communication library.

Following list describes those libraries:

- *jdom*, used for XML processing. usage of this component is very extensive, due to processing of XML results and XML-based XQuery language.
- *xmldb-api*, used in conversion from XML:DB data model to the PROToS one.
- *spring* - in fact, it is composed from several artifacts, including *core*, *context*, *remoting* and *jmx*. Together those components provides full Spring integration for the **SSN**.

6.1.6. Protos-sp component

The **Storage Peer** component acts as end point in the **PROToS** distributed storage. From hierarchy point of view, group of **SPs** is to be managed by one **SSN** component instance.
SP components are designed to acts as simple storage units, with as little logic as possible. Basic requirement for a **SP** is to support XQuery and provide some access for configuring XML storage, as XML:DB API set. Therefore, simplest **SP** could be naive XML database, as **eXist** [9]. Also, most of the configuration is to be handled in **SP** native way and set up on component's deployment. Thus, **SP** component is not subject to the stateful component model A.2 or remote management. Vital part of the configuration - storage - is to be handled by an applicable **SSN** via XML:DB functionalities.
Currently **SP** module is composed of two components:

1. **SP API** defines basic set of features that should be implemented by full-blown **SP** implementations. These interfaces are not applicable to **SPs** wrapping native XML databases, where corresponding behavior could be achieved in native way.
2. **SP stand-alone** component contains reference implementation of the **SP** component, using eXist database for XML storage.

Described **SP** sub components are covered in following sections.

Protos-sp API component

Description

As stated before, this module contains interfaces that have to be implemented by all full-blown **SP** implementations.

Interfaces

Interfaces are placed in the **pl.cyfronet.virolab.protos.sn.sp.*** packages.
Fig. 6.17 depicts **SP** interfaces.

As shown, **SP** model follows *Facade* design pattern. Currently there is only one essential interface - **IConfigurationAPI** extended by the facade **IStoragePeerFacade**. It defines following methods:

- *loadBaseConfiguration* method loads configuration of the component from applicable persistence service. This method is required to refresh configuration if persisted config (for example XML file) was changed when component is in service.
- *reconfigure* method causes **SP** to refresh component's state using current (loaded) configuration. This method and *loadBaseConfiguration* were split up to preserve clear division between changing configuration and actual component's state.

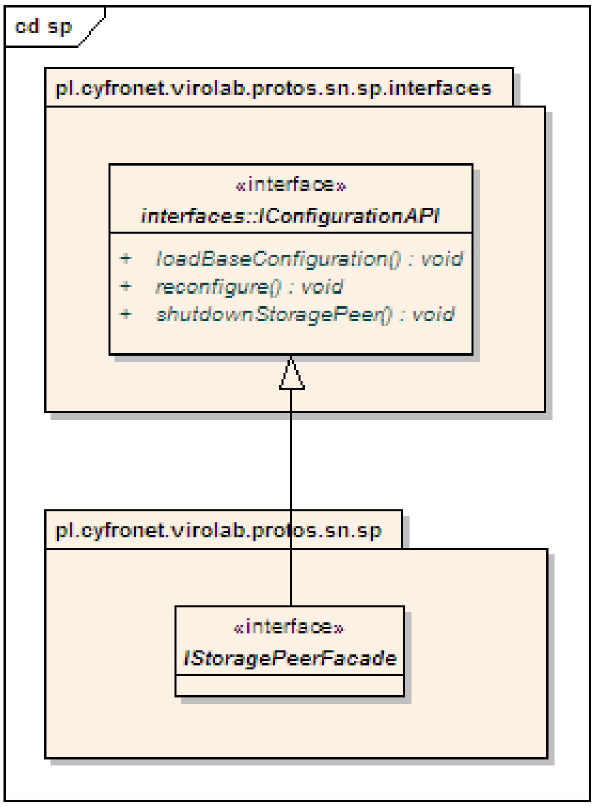

cd sp
pl.cyfronet.virolab.protos.sn.sp.interfaces
«interface»
interfaces::IConfigurationAPI
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ shutdownStoragePeer() : void
pl.cyfronet.virolab.protos.sn.sp
«interface»
IStoragePeerFacade

Figure 6.17. SP API interfaces as UML class diagram. API consists of interface defining configuration behaviour and facade to be implemented by all SP components. Facade is provided in case another aspects of SP behaviour were introduced.

- *shutdownStoragePeer* method is called when **SP** is put out of service. Should force component to persist all data in temporary storage and save it's current configuration. The method is to be called for example when **SP** is revoked from the **SSN**.

Protos-sp stand-alone component

Description

The module containing reference implementation of the **SP**. Implementation is based around instance of the *eXist* native XML db. Because it is wrapper implementation. it supports **SP API**-defined behavior in aneXist-native way. Thus, does not implement **SP API** interfaces.

Design

Source code of this component is placed in packages with **pl.cyfronet.virolab.-protos.sn.sp.simple** prefix.
Diagram 6.18 presents simplified design of the component's core.

The **ExistWrapper** class is core of the implementation. Extending Java **Runnable** interface allows for running database in separate thread. Defines following attributes and methods:

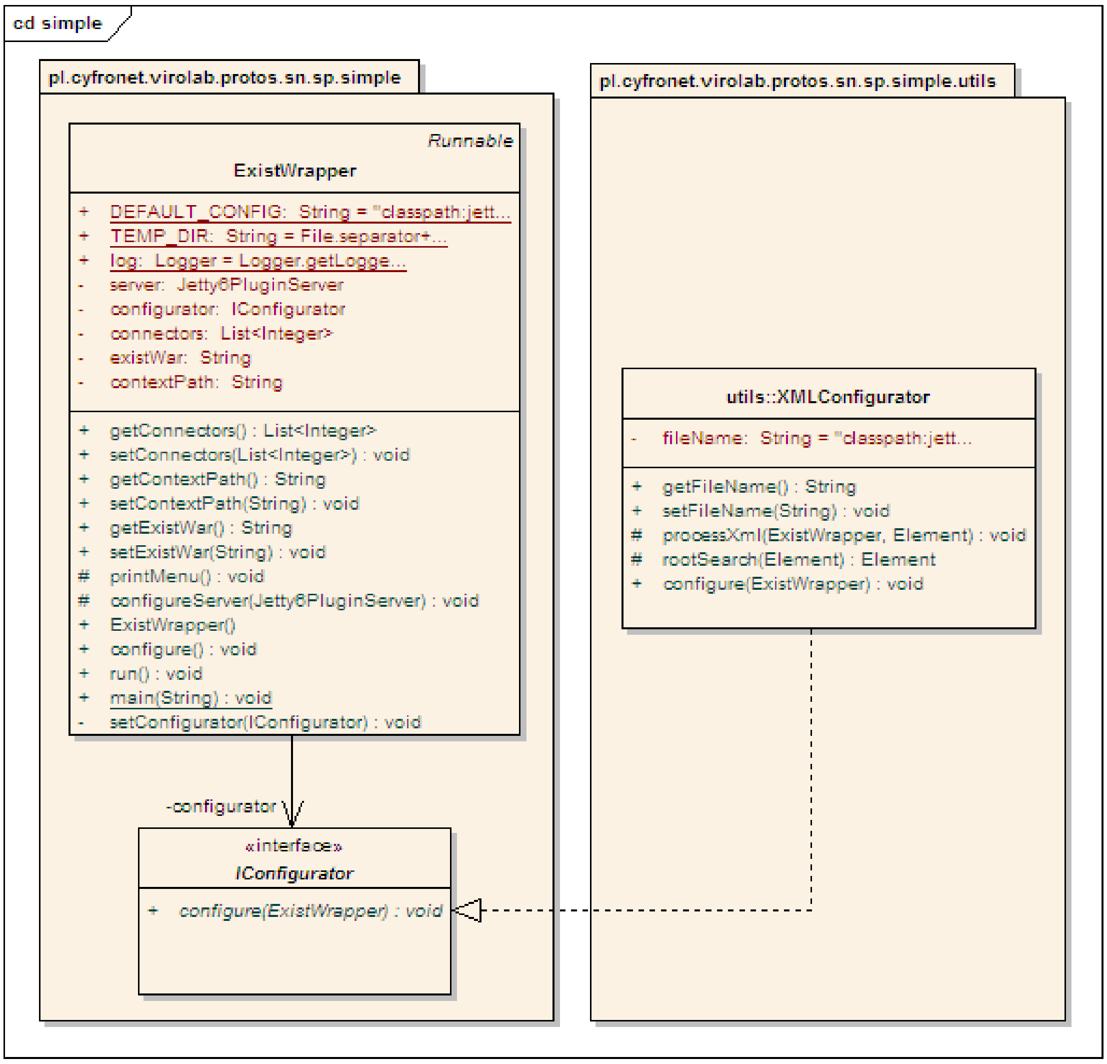

cd simple
pl.cyfronet.virolab.protos.sn.sp.simple
Runnable
ExistWrapper
+ DEFAULT_CONFIG: String = "classpath:jett...
+ TEMP_DIR: String = File.separator+...
+ log: Logger = Logger.getLogge...
- server: Jetty6PluginServer
- configurator: IConfigurator
- connectors: List<Integer>
- existWar: String
- contextPath: String
+ getConnectors() : List<Integer>
+ setConnectors(List<Integer>) : void
+ getContextPath() : String
+ setContextPath(String) : void
+ getExistWar() : String
+ setExistWar(String) : void
# printMenu() : void
# configureServer(Jetty6PluginServer) : void
+ ExistWrapper()
+ configure() : void
+ run() : void
+ main(String) : void
- setConfigurator(IConfigurator) : void
-configurator
«interface»
IConfigurator
+ configure(ExistWrapper) : void
pl.cyfronet.virolab.protos.sn.sp.simple.utils
utils::XMLConfigurator
- fileName: String = "classpath:jett...
+ getFileName() : String
+ setFileName(String) : void
# processXml(ExistWrapper, Element) : void
# rootSearch(Element) : Element
+ configure(ExistWrapper) : void

Figure 6.18. SP stand-alone reference implementation on UML class diagram. Diagram includes main eXist wrapping class along with associated IConfigurator. Also, reference configurator implementation, based on XML files, is provided.

- *configurator* attribute holds instance of the **IConfigurator**, utility for configuring component from persistence service. Fig. 6.18 contains also reference implementation of configurator - the **XMLConfigurator** class. It provides persistence service in the XML file.
- *server* attribute contains reference to the Jetty [19] servlet container instance. Server is necessary for running eXist, as it is distributed as Java web application (WAR).
- *existWar* attribute points to the place in classpath or file system, where eXist WAR is stored.
- *contextPath* attribute defines context path of the eXist, that will be part of the **SP** URL as passed to the **SSN**.
- *connectors* attribute defines how application will be available. This includes pro-

tocol, valid IP and port (if applicable). Typically, only one connector using HTTP protocol and port 8080 is configured.

Dependencies

External dependencies of the **Protos-sp stand-alone** are depicted in Fig. 6.19.

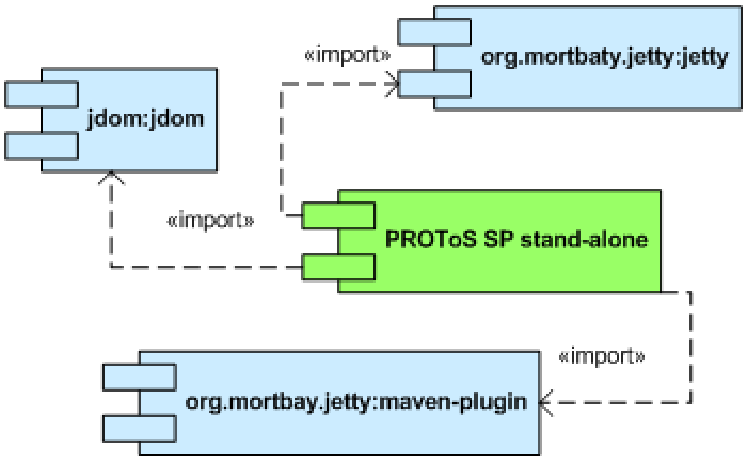


Figure 6.19. PROToS SP external dependencies as UML component diagram. Diagram includes two jetty components responsible for providing servlet container for eXist and XML JDOM library.

As a rule, library *jdom* is used for XML processing. In case of this component, stand-alone server is configured by a specific XML file. Components from group *org.mortbay.jetty* are used to setup embedded **Jetty** servlet container, hosting **eXist** database instance. Component *jetty* serves full database logic. Code from *maven-plugin* is used because it provides simple bootstrap and configuration for the container.

6.1.7. Protos-data component

Description

This component contains common data model of the PROToS system along with utilities for processing entities of the model. Model itself is composed of events, queries and query result representations. Virtually every PROToS component is dependent of this component.

Design

All source code belonging to the *protos-data* component is placed in **pl.cyfronet.-virolab.protos.common*** packages.
Fig. 6.20 presents core classes of the common data model.
Detailed description follows:

- **Event**
  This is base class for incoming ontology-enriched events, representing both actor provenance and meta data. These properties mirrors OWL features:

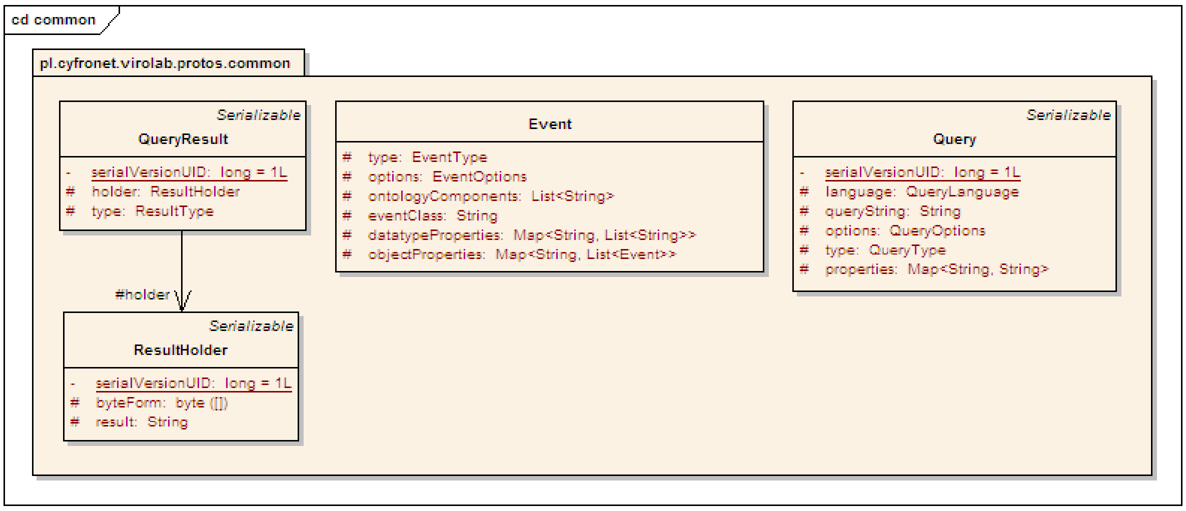

cd common
pl.cyfronet.virolab.protos.common
Serializable
QueryResult
- serialVersionUID: long = 1L
# holder: ResultHolder
# type: ResultType
#holder
Serializable
ResultHolder
- serialVersionUID: long = 1L
# byteForm: byte ([])
# result: String
Event
# type: EventType
# options: EventOptions
# ontologyComponents: List<String>
# eventClass: String
# datatypeProperties: Map<String, List<String>>
# objectProperties: Map<String, List<Event>>
Serializable
Query
- serialVersionUID: long = 1L
# language: QueryLanguage
# queryString: String
# options: QueryOptions
# type: QueryType
# properties: Map<String, String>

Figure 6.20. UML class diagram of PROToS common data model. This include classes for representing provenance events, provenance-mining queries and query results. Model is prepared to fit requirements of Web Services and RMI-type communications.

— *ontologyComponents* contains names of OWL classes or properties modeled by the event.
— *datatypeProperties* maps names of primitive type property to actual values, supporting one-to-many relation. Equivalent of the OWL *datatype property*.
— *objectProperties* maps names of complex type property to actual values, being also **Event** instances. Equivalent of the OWL *object property*.

Properties *type* and *options*, with enumerated values, sets actual PROToS type (as 'simple') and various options of the event. Property *eventClass* contains full-qualified name of the bean class representing OWL concept, when annotated data model is used. This topic is described more elaborately in 6.1.7.

- **Query**
  Represents ontology-mining queries to be processed by the PROToS. Because of requirements from various communication middlewares, as RMI, **Query** class implements standard Java **Serializable** interface.
  Provides properties to support every type of query possible. Configuration options includes *options*, *type* and *language* used. What is more, by setting *properties*, PROToS can make use of in-query variables, thus preventing 'injection'-type attacks. Variable syntax is similar to HQL, that is variable named **var** has to be included in *queryString* as **:var**.
- **QueryResult**
  Instances of this class act as a holder for the actual query result. Property *type* enables user to check actual type of result, contained in the *holder* property. because result instances are to be send through network, also QueryResult implements **Serializable** interface.
- **ResultHolder**
  Subclasses of the ResultHolder should provide means for storing and pre-processing

actual query result. Default one allows for storing result as String or byte array, with conversion logic provided in UTF-8 encoding. Being filed in the **QueryResult** also implements **Serializable** interface.

Fig. 6.21 presents full data model with helper classes and enumerations used by core classes. As far as enumerations are concerned, values are self-explaining.

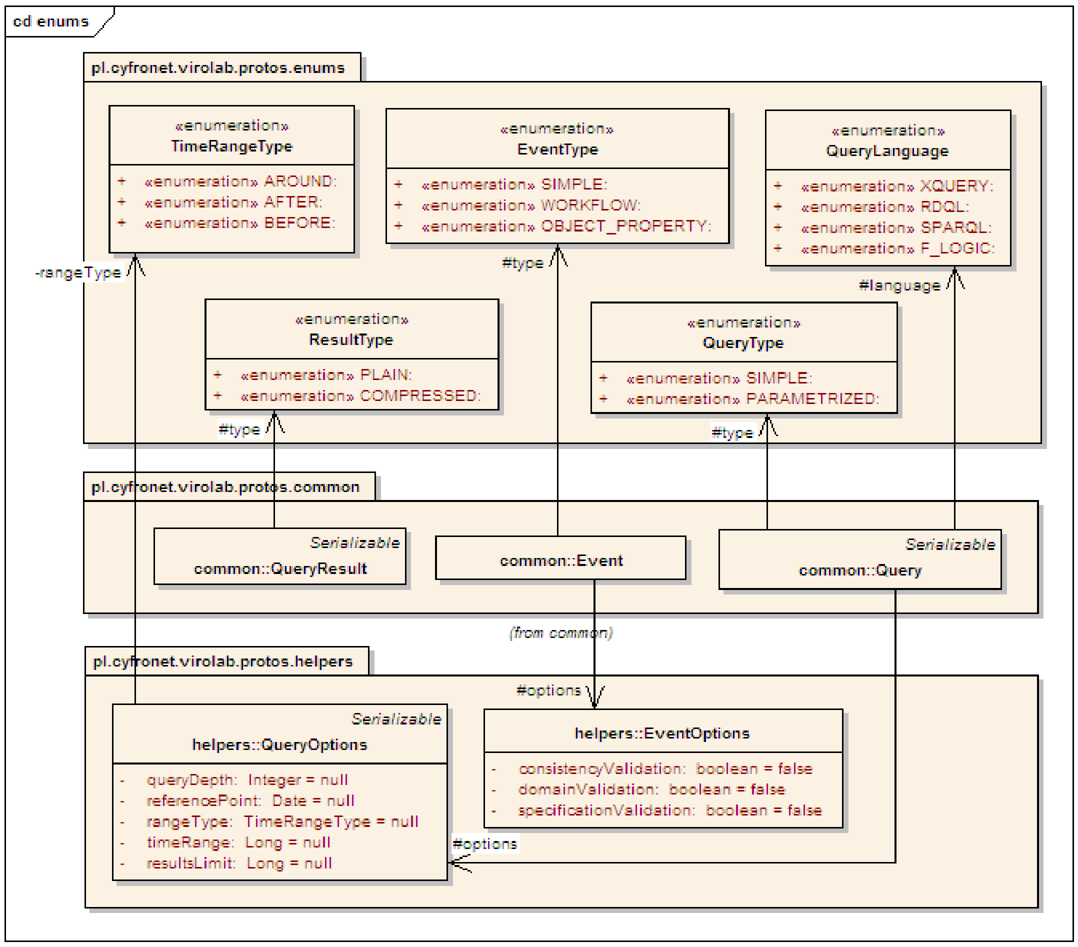


Figure 6.21. Common data model helper classes and enumerations at UML class diagram. Helpers includes options defined for queries and events. Enumerations includes for example query languages and event types supported.

Following list contains short description of helper classes:

- **QueryOptions**
  Properties *referencePoint*, *rangeType* and *timeRange* allows for choosing timeframe of data to be queried. *ResultsLimit* restricts amount of OWL entities to be returned. Finally, *queryDepth* property limits XML level that will be searched.
- **EventOptions**
  Available options changes validation that should be applied to the incoming event. These are:

— *consistencyValidation* - if set to true, system should check whether ontology information built from event is consistent with already stored ontology individuals (registries)
— *domainValidation* - if set, system shall validate event versus adequate domain ontologies
— *specificationValidation* - if set to true, additional specification checking should be performed by the PROToS

Annotated data model

As stated previously, all provenance data must be submitted to the PROToS as instances of the **Event** class. Those instances can transfer any necessary ontology information contained in source domain ontologies, as values of adequate attributes. Yet, in some aspects this model could prove itself unsuitable.
In example, events coming right from experiments, as seen in Fig. 4.1 are to be filled by experiment's writer. In order to accomplish that with default model, following steps are required:

- instantiation of bare instance of the Event class
- identification of OWL class modeling requested PROToS event and filling *ontologyComponents* with list containing URI of the class
- identification of URIs of requested class properties
- identification of type of these properties (object or datatype)
- filling adequate map property of the Even instance with entries URI-value

Above procedure seems too complicated, especially for ViroLab's users without appropriate amount of technical knowledge. Thus, annotated event model was created. It is based on the observation made in 'Event Generation' description (4.3), that every OWL class can be easily translated into Java bean. Of course, so bean still has to extend Event class. Key features of the model are summarized below:

- OWL class is represented by class extending **Event** in the Java-bean convention.
- All datatype properties are converted to built-in Java types.
- All object properties are using **Event** type.
- All getters and setters of event's properties should use adequate maps (*datatypeProperties* or *objectProperties*). This requirement comes from WebServices features. Because WSDL for PROToS services is to be created with **Event** class in the data model, WS serialization would not take additional event properties into account. Therefore, all property values that shall survive WS communication are to be stored in **Event** maps.
- Ontology-related information are to be represented as **annotations** of event's methods. Usage of annotations allows for preservation of more ontology data that usage of bare **Event** class features.
- Full qualified name of the actual event class should be stored as *eventClass* property. This will be used after WebServices transfer to determine which event class should be scanned for ontology information.

Fig. 6.22 depicts annotations defined for current data model.
Short description follows:

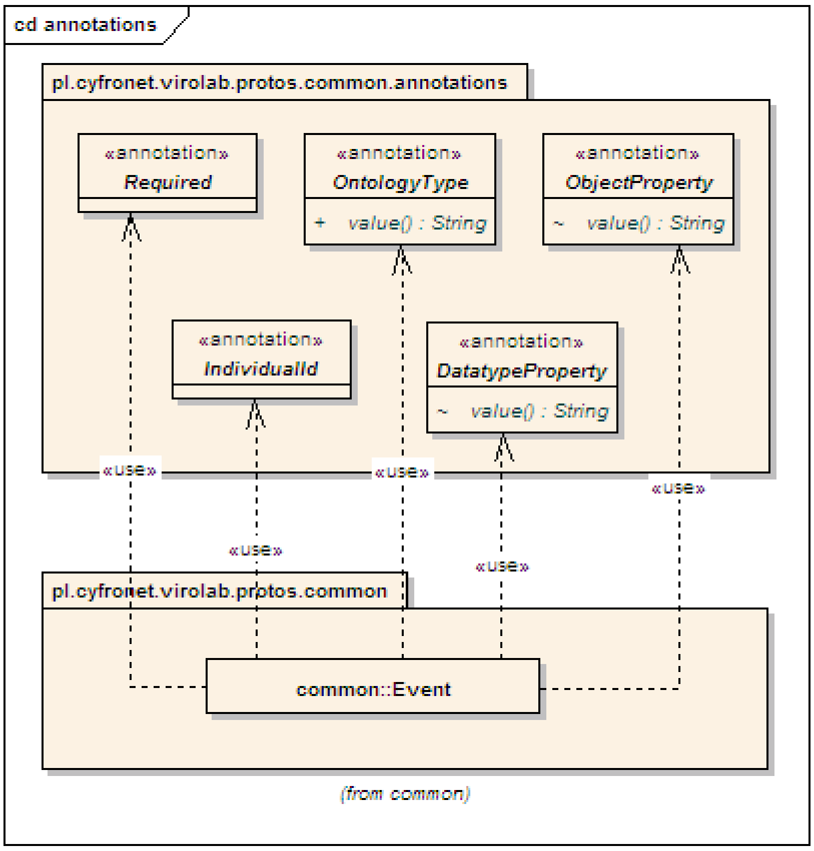

cd annotations
pl.cyfronet.virolab.protos.common.annotations
«annotation» Required
«annotation» OntologyType
+ value() : String
«annotation» ObjectProperty
~ value() : String
«annotation» IndividualId
«annotation» DatatypeProperty
~ value() : String
«use»
«use»
«use»
«use»
«use»
pl.cyfronet.virolab.protos.common
common::Event
(from common)

Figure 6.22. Current annotated data model on UML class diagram. Model defines annotations representing all needed ontology particulars: OWL class, datatype property, object property and individual id. Also, required feature used for validation purposes is included.

- **IndividualId** is used to tag property responsible for carrying ID of ontology individual.
- **OntologyType** annotates class with URI of the OWL class it is representing.
- **DatatypeProperty** tags property as being datatype.
- **ObjectProperty** tags property as being object.
- **Required** defines that some property is required by the OWL ontology model. It is an example of additional information that could not be preserved in OWL-to-Java conversion otherwise.

Implementation

In Fig. 6.23, additional utilities used for processing data models classes are presented.

Following list comprises short descriptions of utilities:

- **OntologyAnnotationProcessor**
  Core utility for processing annotated event model. Provides generic methods for extracting datatype an object properties, individual id and actual ontology type (OWL class URI) of the event.

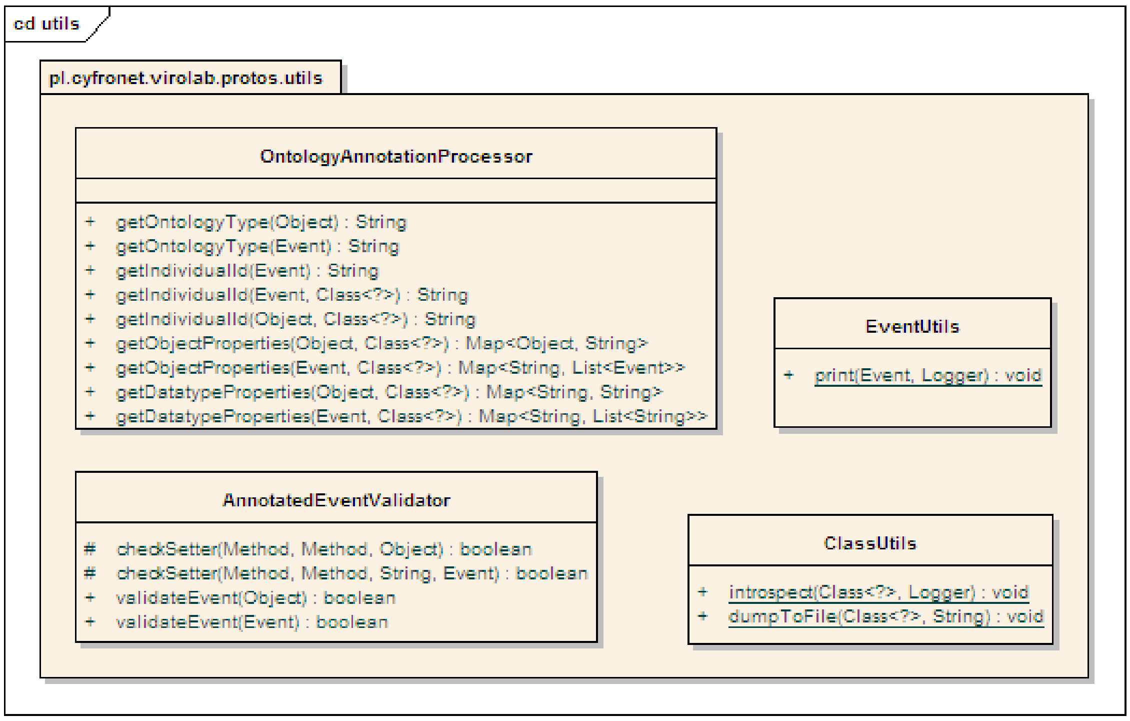

cd utils
pl.cyfronet.virolab.protos.utils
OntologyAnnotationProcessor
+ getOntologyType(Object) : String
+ getOntologyType(Event) : String
+ getIndividualId(Event) : String
+ getIndividualId(Event, Class<?>) : String
+ getIndividualId(Object, Class<?>) : String
+ getObjectProperties(Object, Class<?>) : Map<Object, String>
+ getObjectProperties(Event, Class<?>) : Map<String, List<Event>>
+ getDatatypeProperties(Object, Class<?>) : Map<String, String>
+ getDatatypeProperties(Event, Class<?>) : Map<String, List<String>>
EventUtils
+ print(Event, Logger) : void
AnnotatedEventValidator
# checkSetter(Method, Method, Object) : boolean
# checkSetter(Method, Method, String, Event) : boolean
+ validateEvent(Object) : boolean
+ validateEvent(Event) : boolean
ClassUtils
+ introspect(Class<?>, Logger) : void
+ dumpToFile(Class<?>, String) : void

Figure 6.23. UML class diagram of common data model utilities. Most important classes from this group handles processing and handling of annotated provenance events.

- **AnnotatedEventValidator**
  This utility is used to validate annotated events. Current implementation makes use only of **Required** annotation for setters. It also checks various necessary filed of generic **Event** class.
- **EventUtils**
  Introspects instance of the **Event** class, printing available info to the chosen logger. Used for testing and debug.
- **ClassUtils**
  Provides debug and testing for annotated event model. This utility helps detects run-time problems with various class loaders in managed environments.

Dependencies

Fig. 6.24 depicts external dependencies of the **Protos-data** component.

It depends only on the *commons-beanutils* library, provided by the Apache. Utilities from the library are used to manipulate bean-like events in very convenient way, unavailable in standard Java JRE API. In example, event classes extending **SimpleEvent** using PROToS annotations are processed with extensive usage of the *commons-beanutils*. In fact, all utilities for processing annotations, as **AnnotatedEventValidator** and **OntologyAnnotationProcessor** makes use of this component.

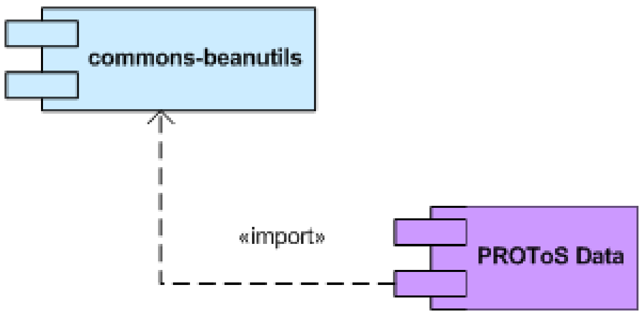


Figure 6.24. PROToS Data external dependencies as UML component diagram. Diagram includes only commons-beanutils package.

6.1.8. Protos-interfaces component

Description

Module *protos-interface* contains communicators for WS operations along with full configuration for the **PROToS Core** application. Because the application is built with an Dependency Injection container, configuration is what really matters in this component. Deep thorough reference of the DI config is given in the section A.

Design

Diagram 6.25 depicts WS communicators of the **PROToS Core** application.

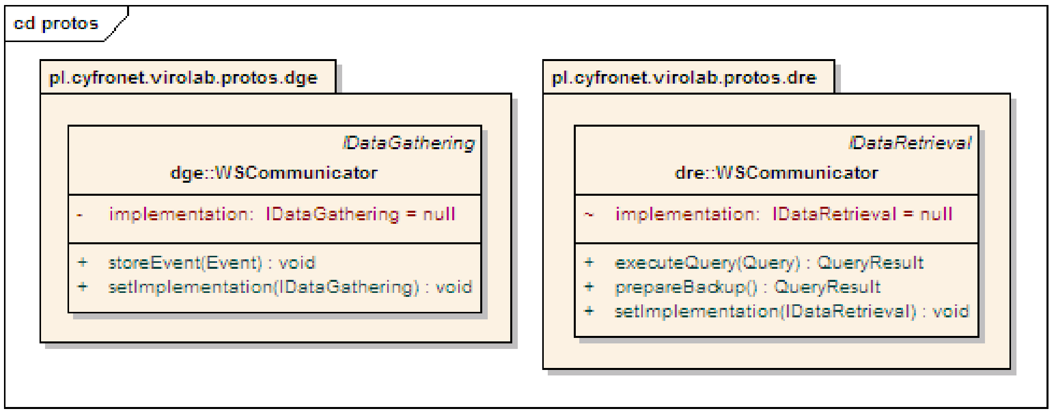

cd protos
pl.cyfronet.virolab.protos.dge
IDataGathering
dge::WSCommunicator
- implementation: IDataGathering = null
+ storeEvent(Event) : void
+ setImplementation(IDataGathering) : void
pl.cyfronet.virolab.protos.dre
IDataRetrieval
dre::WSCommunicator
~ implementation: IDataRetrieval = null
+ executeQuery(Query) : QueryResult
+ prepareBackup() : QueryResult
+ setImplementation(IDataRetrieval) : void

Figure 6.25. WebService communicators on UML class diagram. Communicators implement external interfaces - IDataGathering and IDataRetrieval and delegate functionality to adequate components (*implementation* properties).

Each communicator's design consist of following:

1. interface corresponding it's functions - **IDataGathering** and **IDataRetrieval**. These are core interfaces in adequate components, serving the functionality.
2. accessors for *implementation* attribute. It holds reference to the instance of component serving the functionality. Instance is injected by the DI container.
3. adequate entry in the configuration of the application. This is covered in section A.1.

Dependencies

External dependencies of the component are presented in Fig. 6.26.

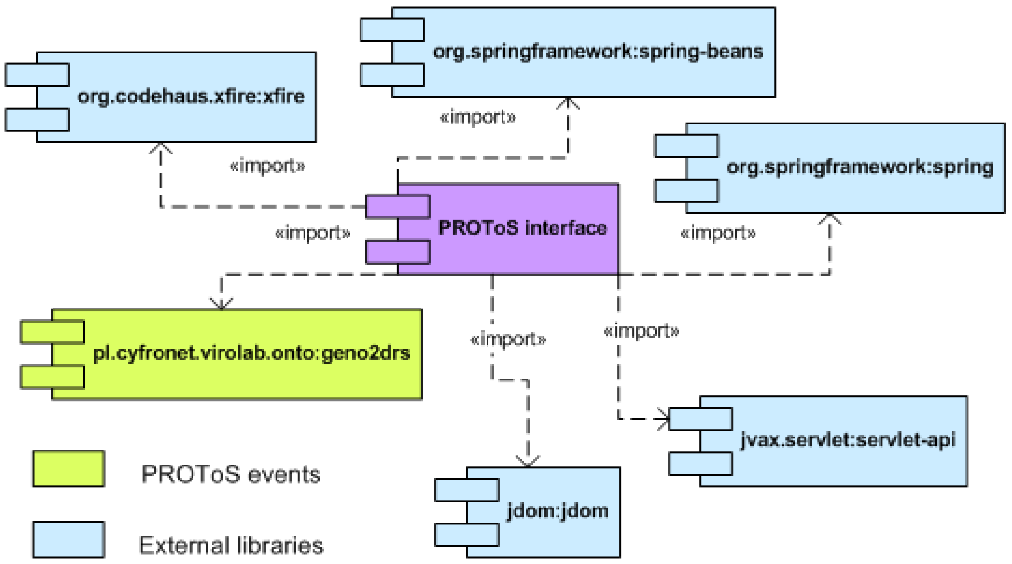


Figure 6.26. PROToS interface external dependencies on UML component diagram. Diagram includes Spring's integration, XFire WebService framework, and JDOM for XML processing. Also, geno2drs events are provided for testing purposes.

These are as follows:

- *spring-beans* and *spring* provides Spring IoC container integration, that ties whole application. Components are not referenced directly from the *Interface* code, but from the *applicationContext.xml* file - application's configuration. The *spring* component provides artifacts for such application aspects as RMI or WS integration and JMX configuration.
- *xfire* component provides all Web Services functionality. It also nicely integrates with Spring WS layer.
- *jdom* - a widely used XML processing library and *geno2drs* ontology event model are referenced by the **Interface** component solely for testing purposes.

6.1.9. Protos-config component

Description

The *protos-config* module encapsulates logic responsible for configuring component's using persistence services. Module is required to store objects in question in an editable way. What is more, implementation should allow configuring already instantiated objects. This way, some attributes can be set in object and other loaded from persistent store.

Design

Package prefix for this component is **pl.cyfronet.virolab.protos.config**. Fig. 6.27 depicts essentials of the *config* component.

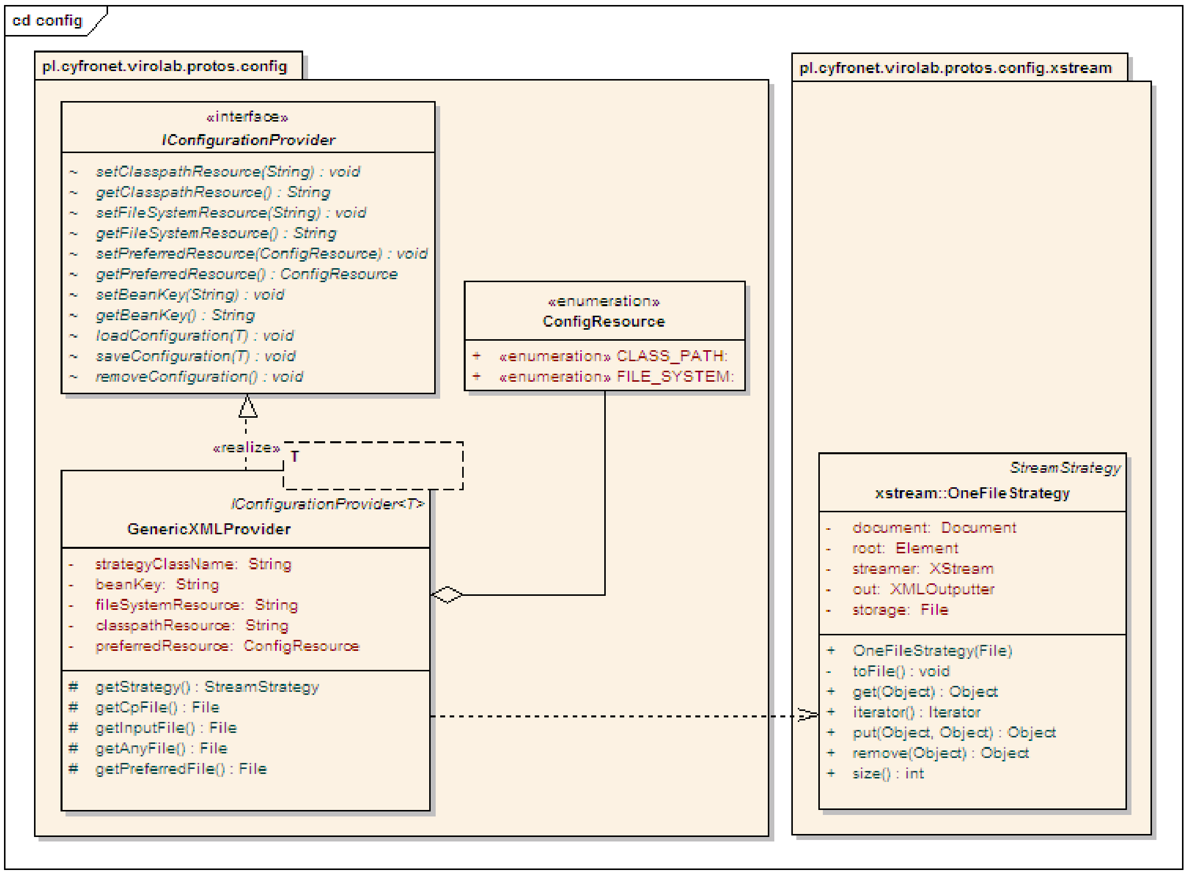

cd config
pl.cyfronet.virolab.protos.config
«interface»
IConfigurationProvider
~ setClasspathResource(String) : void
~ getClasspathResource() : String
~ setFileSystemResource(String) : void
~ getFileSystemResource() : String
~ setPreferredResource(ConfigResource) : void
~ getPreferredResource() : ConfigResource
~ setBeanKey(String) : void
~ getBeanKey() : String
~ loadConfiguration(T) : void
~ saveConfiguration(T) : void
~ removeConfiguration() : void
«enumeration»
ConfigResource
+ «enumeration» CLASS_PATH:
+ «enumeration» FILE_SYSTEM:
«realize»
T
IConfigurationProvider<T>
GenericXMLProvider
- strategyClassName: String
- beanKey: String
- fileSystemResource: String
- classpathResource: String
- preferredResource: ConfigResource
# getStrategy() : StreamStrategy
# getCpFile() : File
# getInputFile() : File
# getAnyFile() : File
# getPreferredFile() : File
pl.cyfronet.virolab.protos.config.xstream
StreamStrategy
xstream::OneFileStrategy
- document: Document
- root: Element
- streamer: XStream
- out: XMLOutputter
- storage: File
+ OneFileStrategy(File)
- toFile() : void
+ get(Object) : Object
+ iterator() : Iterator
+ put(Object, Object) : Object
+ remove(Object) : Object
+ size() : int

Figure 6.27. Config component core classes on UML class diagram. Model consists of main configuration manager interface IConfiguartionProvider along with reference implementation - GenericXMLProvider and its helper classes.

It is composed of following classes and interfaces:

- **ConfigResource**
  This enumeration defines which types of storage resources are supported by available configuration utility implementations.
- **IConfigurationProvider**
  The interface defining required behavior of all persistent configuration providers. Methods of the interface are as follows:
  - — accessors for *classpathResource* attribute, allowing to set file available in application's class path that shall be used to store objects.
  - — accessors for *fileSystemResource* attribute, allowing to set system-wide path to file that should be used as persistent store.
  - — accessors for *preferredResource* attribute, allowing to chose which resource should be used if more than one is configured.
  - — accessors for *beanKey* attribute, defining unique ID of the object that will be stored on next call to the *save* method.
  - — *loadConfiguration* method that shall load from persistent storage configuration

with ID equal to current *beanKey* and copy loaded values to applicable object's attributes.
— *saveConfiguration* method that shall read object's attributes and save them under current *beanKey* in the persistent store.
— *removeConfiguration* method that shall search for configuration with *beanKey* ID and remove it from persistent resource.

- **GenericXMLProvider**
  This is implementation of the **IConfigurationProvider** interface, that employs XML serialization of the objects. The provider reflects persisted objects searching for attributes with accessors defined as *getX* and *setX*. Thus it imposes Java-bean structure on the objects under question. Actual persistent resource handling is done by strategies implementing **StreamStrategy** interface.
- **OneFileStrategy**
  Implements **StreamStrategy** interface, storing all objects in one XML resource (file), either from class path or from file system. Uses XStream [48] to handle Object to XML mapping.

Dependencies

Fig. 6.28 depicts external dependencies of the **Protos-config**.

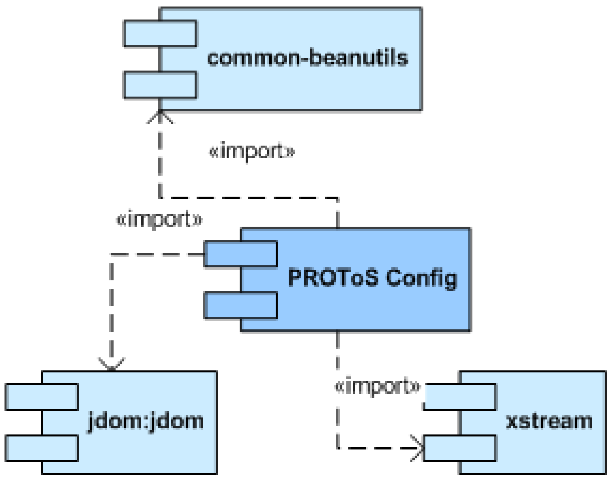


Figure 6.28. PROToS Config external dependencies as UML component diagram. Most important dependency is xstream component, responsible for object to XML mapping.

Component *common-beanutils* is used to manipulate Java beans, that are to be handled by **config** utility. Typical usage involves reflecting incoming beans for properties to be persisted or restored. *Xstream* component provides Java-to-XML (de)serialization, and the *jdom* library in used for manipulating serialized beans inside storage XML files. Reference implementation of the **Protos-config** stores all beans in one XML valid file, thus *jdom* role is very important.

6.1.10. Protos-onts component

Description

Protos-onts module encapsulates common ontology processing logic of the PROToS. As DOs are one core concepts of the system, this module is used by almost all PROToS components.
Main functionality of the *protos-onts* concentrate on following aspects:
1. retrieving ontology models from various external sources
2. extracting information from DOs, as individuals, ontology URI and others
3. providing convenience utility for manipulating ontologies represented in various models - Jena API, pure XML and PROToS data
4. providing bridges between ontology representations (Jena, XML, RDF) and PROToS annotated event model

Design

Package prefix for this component is **pl.cyfronet.virolab.protos.onts**.
Diagram 6.29 depicts main interfaces and classes of the *protos-onts* component.

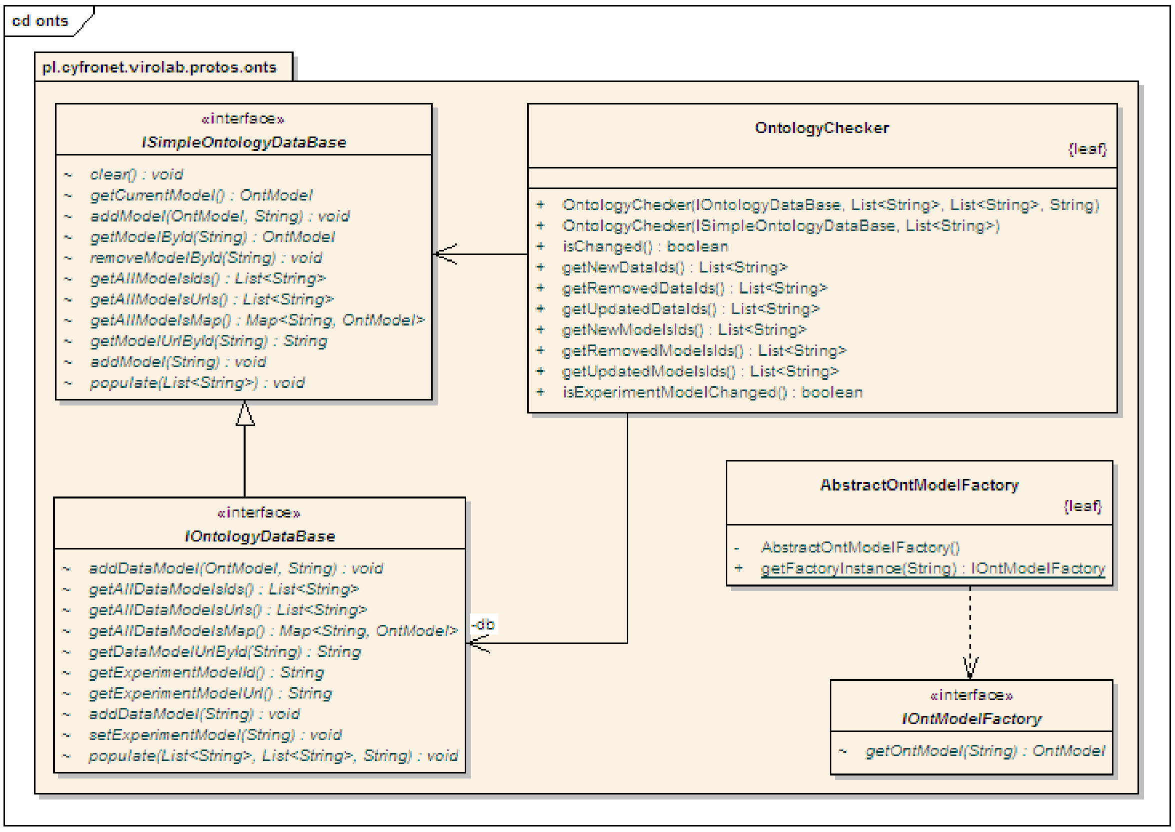

cd onts
pl.cyfronet.virolab.protos.onts
«interface»
ISimpleOntologyDataBase
~ clear() : void
~ getCurrentModel() : OntModel
~ addModel(OntModel, String) : void
~ getModelById(String) : OntModel
~ removeModelById(String) : void
~ getAllModelsIds() : List<String>
~ getAllModelsUris() : List<String>
~ getAllModelsMap() : Map<String, OntModel>
~ getModelUriById(String) : String
~ addModel(String) : void
~ populate(List<String>) : void
OntologyChecker
{leaf}
+ OntologyChecker(IOntologyDataBase, List<String>, List<String>, String)
+ OntologyChecker(ISimpleOntologyDataBase, List<String>)
+ isChanged() : boolean
+ getNewDataIds() : List<String>
+ getRemovedDataIds() : List<String>
+ getUpdatedDataIds() : List<String>
+ getNewModelsIds() : List<String>
+ getRemovedModelsIds() : List<String>
+ getUpdatedModelsIds() : List<String>
+ isExperimentModelChanged() : boolean
AbstractOntModelFactory
{leaf}
- AbstractOntModelFactory()
+ getFactoryInstance(String) : IOntModelFactory
«interface»
IOntologyDataBase
~ addDataModel(OntModel, String) : void
~ getAllDataModelsIds() : List<String>
~ getAllDataModelsUris() : List<String>
~ getAllDataModelsMap() : Map<String, OntModel>
~ getDataModelUriById(String) : String
~ getExperimentModelId() : String
~ getExperimentModelUri() : String
~ addDataModel(String) : void
~ setExperimentModel(String) : void
~ populate(List<String>, List<String>, String) : void
-db
«interface»
IOntModelFactory
~ getOntModel(String) : OntModel

Figure 6.29. PROToS onts module core on UML class diagram. Model includes ontology databases, utility for checking ontology db state and Abstract Factory - based ontology model reference retriever.

Presented classes concentrate on two aspects, described below.

- DOs storage and management
  Consists of following:
  — textbfISimpleOntologyDataBase and **IOntologyDataBase** interface defining all methods required for fully-functional DO data base. The latter extends first one adding discrimination for *data*, *domain* and *experiment* models, as used by for example *protos-dss* component. Reference implementation of main interface is provided in the **impl** package, as depicted on diagram 6.30.
  — textbfOntologyChecker class is utility for performing updates on an ontology database. Initialized with an instance of **ISimpleOntologyDataBase** or **IOntologyDataBase** and list of new DOs shows which models have been removed, added or updated.
- DO retrieval
  This functionality is designed to follow *abstract factory* design pattern. Consists of following:
  — **AbstractOntModelFactory** class decides which protocol-specific factory should be returned. Decision is based on the model address passed, as it contains protocol identifier. Adding new specific factories requires *getFactoryInstance* method to be extended.
  — **IOntModelFactory** interface should be implemented by factories specific for particular protocol (ontology source). As depicted in Fig. 6.30, currently *protos-onts* provides factories for DOs available in class path and via HTTP protocol.

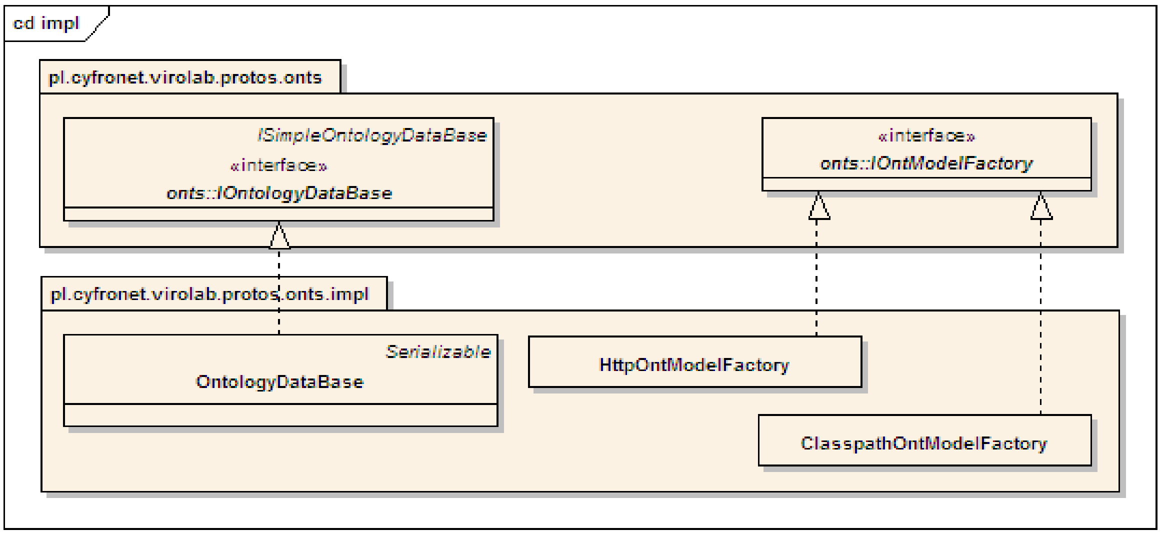


Figure 6.30. UML class diagram of onts core reference implementation. Diagram includes factories for ontology models accessible by HTTP protocol and class path.

Apart from described functionalities, main package **pl.cyfronet.virolab.protos.-onts** contains many utilities manipulating ontology models, as **OntModelSerializer** class and others.
Separate paragraph shall be devoted to the **datautils** package, containing 'bridge'

between ontology and PROToS event models. It is presented in Fig. 6.31.

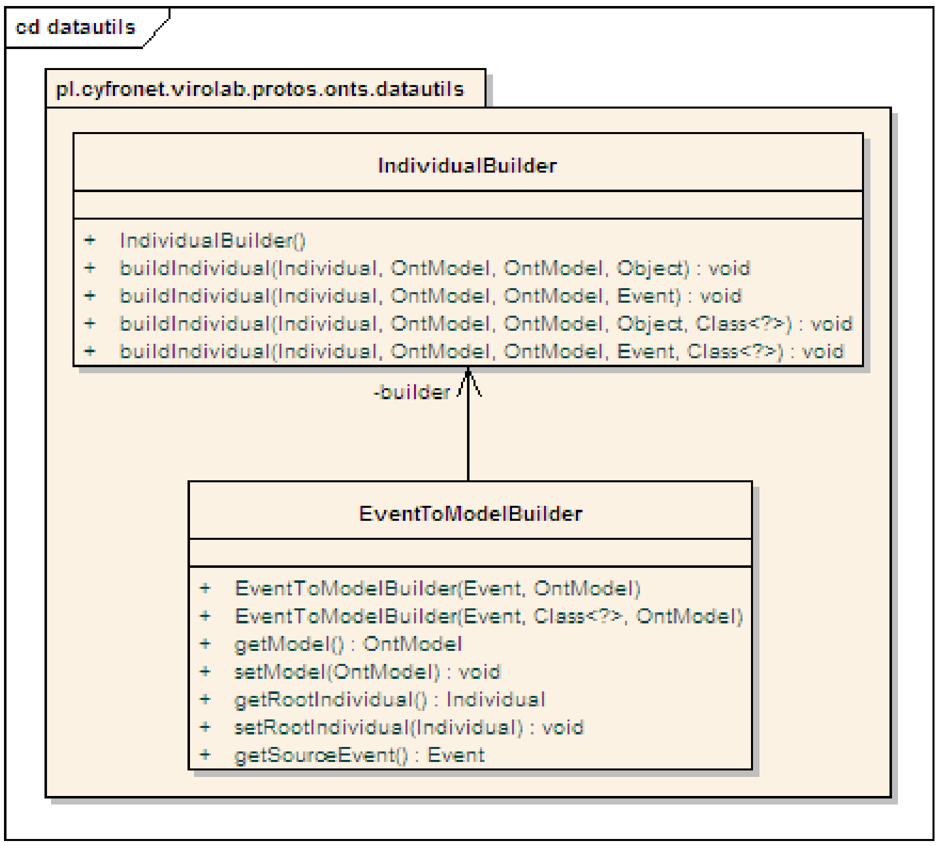

cd datautils
pl.cyfronet.virolab.protos.onts.datautils
IndividualBuilder
+ IndividualBuilder()
+ buildIndividual(Individual, OntModel, OntModel, Object) : void
+ buildIndividual(Individual, OntModel, OntModel, Event) : void
+ buildIndividual(Individual, OntModel, OntModel, Object, Class<?>) : void
+ buildIndividual(Individual, OntModel, OntModel, Event, Class<?>) : void
-builder
EventToModelBuilder
+ EventToModelBuilder(Event, OntModel)
+ EventToModelBuilder(Event, Class<?>, OntModel)
+ getModel() : OntModel
+ setModel(OntModel) : void
+ getRootIndividual() : Individual
+ setRootIndividual(Individual) : void
+ getSourceEvent() : Event

Figure 6.31. PROToS onts common data model utilities as UML class diagram. Depicted classes provides conversion from common data model (events) to Jena ontology model (Individual, OntModel classes). Only annotated events are processed by these converters.

Core class - **EventToModelBuilder** allows for building ontology model containing individuals from instance of annotated PROToS event. Constructor of the class takes two arguments:

- instance of *event* from annotated model (described in section 6.1.7
- instance of OntModel containing ontology models applicable for the event. This is required because event itself does not contain definition of classes and properties, only values for new individual.

Building is dependent on the **IndividualBuilder**, responsible for recursive building instances of Jena API **Individual** class. Methods of this class supports separation of ontology models, therefore class descriptions (DOs) are not mixed with individual registry created from an event.

Dependencies

In Fig. 6.32 external dependencies of the component are presented.

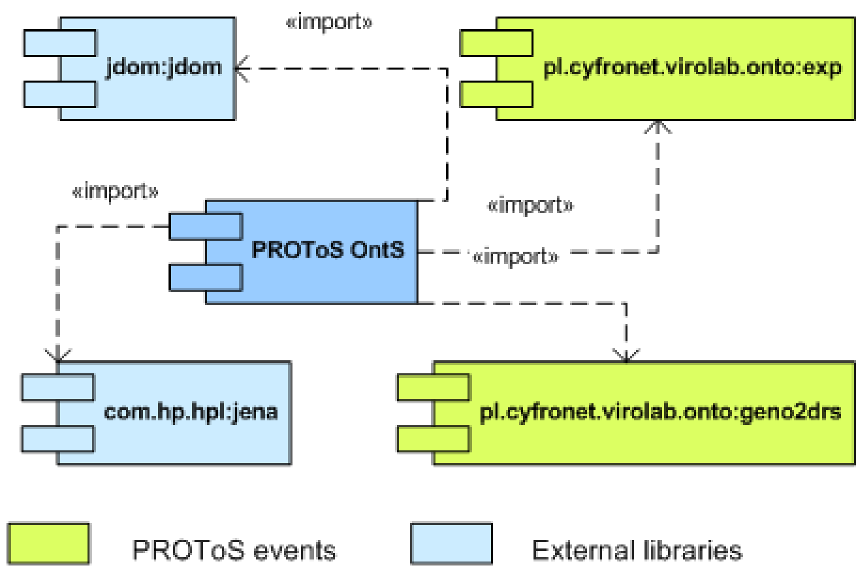


Figure 6.32. PROToS Onts external dependencies as UML component diagram. Apart from typical Jena and JDOM components, experiment and geno2drs event libraries are provided for testing purposes.

Artifact *jena*, providing OWL ontology processing is most important dependency. It is commonly used in almost all **onts** core logic classes. *Jdom* XML processor is used in parts of code dealing with the XML-RDF form of ontologies. Typical usage involves manipulation of nodes structure, as in classes **XmlStructureManipulator** and **XmlFormatUtils**.
Libraries from group *pl.cyfronet.virolab.onto* contains ontologies converted to PROToS events. They are used for integration testings purposes - usage of real-world events and event aggregates assure higher quality of the component.

6.1.11. Protos-pstore component

Description

Name of the *protos-pstore* component is an acronym for Persistent STORE. This component contains logic for loading and saving whole objects in persistent storage, such as file. As opposed to the *protos-config* component (section 6.1.9), does not copy persisted fields values to an object, but loads whole object state as was previously persisted. What is more, format of persistence is not required to be editable, but rather compact in size (for example binary).

Design

Fig. 6.33 depicts simplified *pstore* design diagram.
Diagram is composed of following classes:

- **StorageResource**
  Enumeration of possible storage resource types.
- **ResourceFactory**

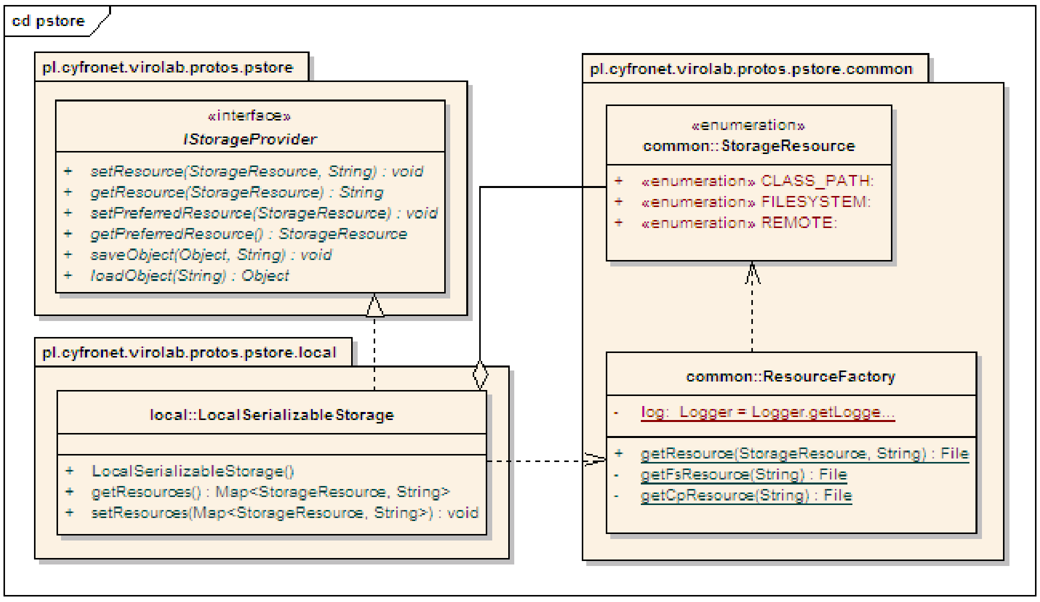

cd pstore
pl.cyfronet.virolab.protos.pstore
«interface»
IStorageProvider
+ setResource(StorageResource, String) : void
+ getResource(StorageResource) : String
+ setPreferredResource(StorageResource) : void
+ getPreferredResource() : StorageResource
+ saveObject(Object, String) : void
+ loadObject(String) : Object
pl.cyfronet.virolab.protos.pstore.local
local::LocalSerializableStorage
+ LocalSerializableStorage()
+ getResources() : Map<StorageResource, String>
+ setResources(Map<StorageResource, String>) : void
pl.cyfronet.virolab.protos.pstore.common
«enumeration»
common::StorageResource
+ «enumeration» CLASS_PATH:
+ «enumeration» FILESYSTEM:
+ «enumeration» REMOTE:
common::ResourceFactory
- log: Logger = Logger.getLogge...
+ getResource(StorageResource, String) : File
- getFsResource(String) : File
- getCpResource(String) : File

Figure 6.33. P-store core classes on UML class diagram. Diagram consists of main interface - IStorageProvider, reference implementation - LocalSerializableStorage and common helper classes.

Factory for resources. For given resource type and protocol-specific address factory returns instance of Java **File**, allowing IO operations.

- **IStorageProvider**
  The interface defining required behavior of storage providers. Defined methods allows for:
  — configuring resource type and name to be used (accessors for *resource* property). For example in case of file system resource, name of the resource to be used should take form of full file system path.
  — configuring preferred resource (accessors for *preferredResource* attribute). This is the resource that shall be used for persistence in case more than one is available.
  — persisting and loading any Java object (*save* and *load* methods). Apart object persisted, additional argument is the key - unique ID that shall identify object in persistent storage.
- **LocalSerializableStorage**
  This is reference implementation of **IStorageProvider** interface. Uses local files for storing data and Java *serializable* mechanisms for processing objects. Therefore, each object passed have to be serializable. Simply, it means that object and it's dependency graph instances have to implement **Serializable** interface or be declared as transient.
  The implementation adds accessors for map containing mappings between resource types and names.

Dependencies

External dependencies of the **protos-pstore** component are depicted in Fig. 6.34.

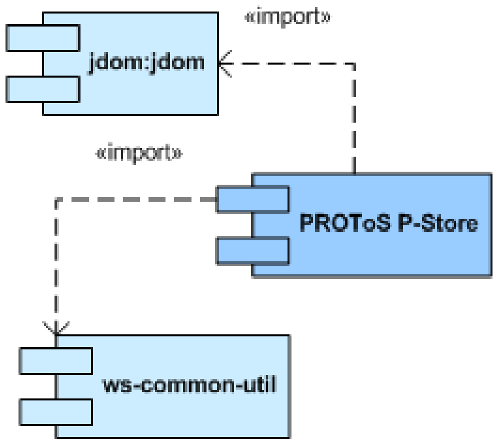


Figure 6.34. PROToS P-store external dependencies as UML component diagram. Diagram includes standard JDOM for XML processing and ws-common-util component for Base64 encoding.

As usual, *jdom* artifact is used for processing XML data - in this case, default implementation stores all passed object in one XML file. Little component named *ws-common-util* provides Base64 coding, used to nest serialized object as binary XML elements.

6.1.12. Protos-xmldb component

Description

The *protos-xmldb* component handles all XML data bases management, data and service model used by the PROToS components. Most important part is model of the PROToS XML:DB enabled endpoint, described in section 6.1.12. Endpoints are complemented by set of services and executors, constituting easy to use API. Therefore, component's using *protos-xmldb* do not have to deal with plain queries in such languages as XUpdate and XQuery. All logic is moved to the API level. It is covered in section 6.1.12. Last but not least, module provides full model for preparing XUpdate queries, shown in the section 6.1.12.

Xmldb endpoint model

Packages of the endpoint model are **pl.cyfronet.virolab.protos.xmldb** and **pl.-cyfronet.virolab.protos.xmldb.loader**.
Diagram 6.35 presents core classes of the model.

XML:DB endpoints are represented as instances of one of two following class:

- **XmldbEndpoint** This class encapsulates connection with some endpoint, as instance of eXist database. It provides access to the root collection of the endpoint (accessors for *collection* attribute) and connection management logic.
- **ExtendedEndpoint** Class extending **XmldbEndpoint**, designed specially for the **SSN** component. Allows for executing queries from *xmldb* API and accessing various services and executors for supported languages. In fact, the *execute* method

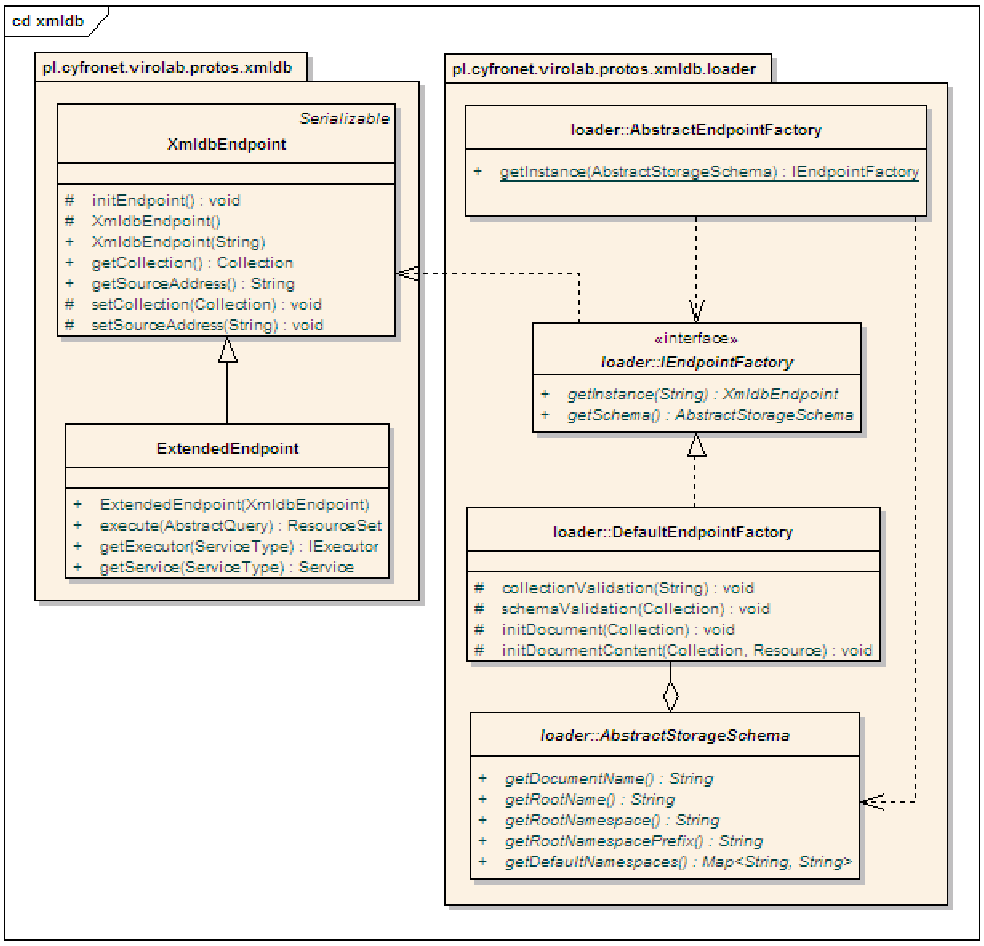

cd xmldb
pl.cyfronet.virolab.protos.xmldb
Serializable
XmldbEndpoint
# initEndpoint() : void
# XmldbEndpoint()
+ XmldbEndpoint(String)
+ getCollection() : Collection
+ getSourceAddress() : String
# setCollection(Collection) : void
# setSourceAddress(String) : void
ExtendedEndpoint
+ ExtendedEndpoint(XmldbEndpoint)
+ execute(AbstractQuery) : ResourceSet
+ getExecutor(ServiceType) : IExecutor
+ getService(ServiceType) : Service
pl.cyfronet.virolab.protos.xmldb.loader
loader::AbstractEndpointFactory
+ getInstance(AbstractStorageSchema) : IEndpointFactory
«interface»
loader::IEndpointFactory
+ getInstance(String) : XmldbEndpoint
+ getSchema() : AbstractStorageSchema
loader::DefaultEndpointFactory
# collectionValidation(String) : void
# schemaValidation(Collection) : void
# initDocument(Collection) : void
# initDocumentContent(Collection, Resource) : void
loader::AbstractStorageSchema
+ getDocumentName() : String
+ getRootName() : String
+ getRootNamespace() : String
+ getRootNamespacePrefix() : String
+ getDefaultNamespaces() : Map<String, String>

Figure 6.35. PROToS Xmldb endpoint model as UML class diagram. Diagram consists of classes representing endpoint and Abstract Factory based loaders. Also, reference implementation of factory - DefaultEndpointFactory - is presented.

is convenience utility that reflects passed query and passes it to an adequate executor. Currently supported queries and executors are depicted in Fig. 6.36.

Instantiation of endpoints is handled by specific classes, following the *abstract factory* design pattern. They are also presented on the diagram 6.35, in the **loader** package. It comprises following:

- class **AbstractStorageSchema** defining particulars of the XML storage that should be met by all endpoints.
- class **AbstractEndpointFactory** responsible for deciding which implementation of the **IEndpointFactory** should be used. Takes instance of class extending **AbstractStorageSchema** as an argument. It is used for initialization of the endpoint factory.
- interface **IEndpointFactory** that should be implemented by all endpoint factories.
- class **DefaultEndpointFactory**, the reference implementation of factory interface. Apart from instantiation of the **XmldbEndpoint** it also makes various

checks on endpoint. Conformance to schema is validated and if necessary lacking features are added.

Services model

The services model is contained inside **pl.cyfronet.virolab.protos.xmldb.-services.*** packages.
Model is depicted on the diagram 6.36. Core classes lie in the root package.

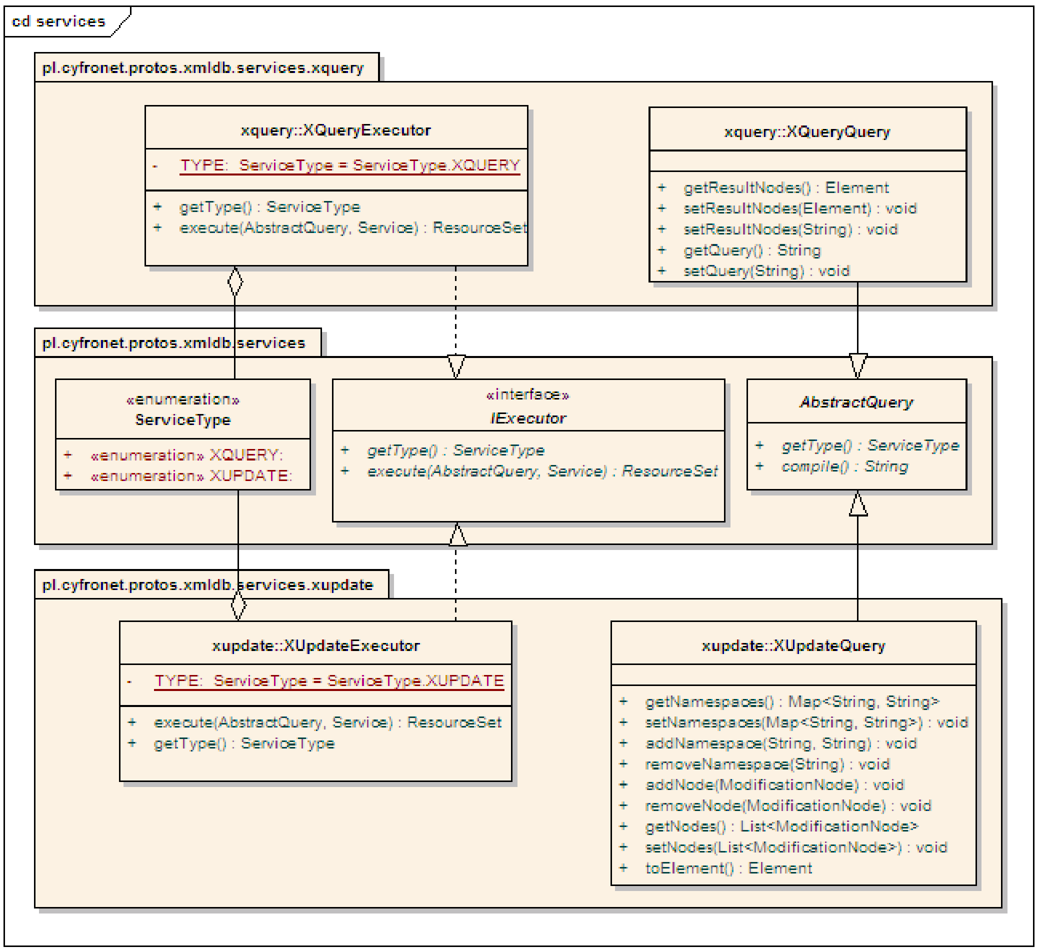

cd services
pl.cyfronet.protos.xmldb.services.xquery
xquery::XQueryExecutor
- TYPE: ServiceType = ServiceType.XQUERY
+ getType() : ServiceType
+ execute(AbstractQuery, Service) : ResourceSet
xquery::XQueryQuery
+ getResultNodes() : Element
+ setResultNodes(Element) : void
+ setResultNodes(String) : void
+ getQuery() : String
+ setQuery(String) : void
pl.cyfronet.protos.xmldb.services
«enumeration»
ServiceType
+ «enumeration» XQUERY:
+ «enumeration» XUPDATE:
«interface»
IExecutor
+ getType() : ServiceType
+ execute(AbstractQuery, Service) : ResourceSet
AbstractQuery
+ getType() : ServiceType
+ compile() : String
pl.cyfronet.protos.xmldb.services.xupdate
xupdate::XUpdateExecutor
- TYPE: ServiceType = ServiceType.XUPDATE
+ execute(AbstractQuery, Service) : ResourceSet
+ getType() : ServiceType
xupdate::XUpdateQuery
+ getNamespaces() : Map<String, String>
+ setNamespaces(Map<String, String>) : void
+ addNamespace(String, String) : void
+ removeNamespace(String) : void
+ addNode(ModificationNode) : void
+ removeNode(ModificationNode) : void
+ getNodes() : List<ModificationNode>
+ setNodes(List<ModificationNode>) : void
+ toElement() : Element

Figure 6.36. PROToS xmldb services model as UML class diagram. Diagram includes common model in form of IExecutor, AbstractQuery and ServiceType along with implementations for XQuery and XUpdate languages.

Their's functions are as follows:

- **ServiceType** enumerates services supported by the model. Currently XUpdate and XQuery are available.
- **AbstractQuery** class should be extended by queries for specific services. Every query should at least define type of service required and enable compilation to string. As presented on 6.36, *protos-xmldb* implements currently **XQueryQuery** and **XUpdateQuery**. First one is simple wrapper around query in string format, as passed to the PROToS from external components. It allows only for definition of XML nodes that should wrap query result. The latter is entry point to the XUpdate

query model, covered in section 6.1.12. Apart from construction of the actual query it also allows for manipulating XML namespaces (**Namespaces** methods) and converting query to the XML JDOM data model.

- **IExecutor** interface shall be implemented by all executors for specific services. As depicted on 6.36 current implementation support execution for XUpdate and XQuery languages. Apart from actual execution of query on service, executors perform also validation functions.

XUpdate data model

XUpdate language is widely used in the PROToS to perform maintenance of XML:DB endpoints. Queries for this functionality have to be created in the PROToS run time. Therefore, need for class-level model for XUpdate came into existence. Model is contained in the **pl.cyfronet.virolab.protos.xmldb.services.xupdate** package.
Fig. 6.37 shows very simplified class diagram of the model.

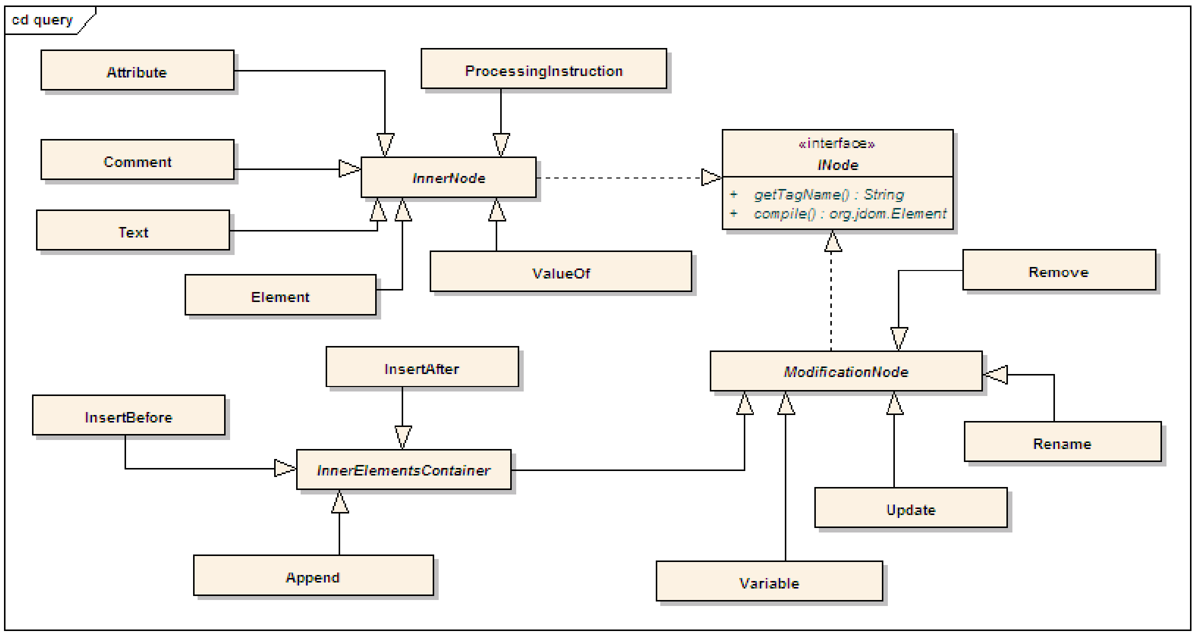


Figure 6.37. Simplified UML class diagram of the PROToS Xmldb model for XUpdate. Model allows for creating any XUpdate queries supported by specification.

Only generalization and realization properties are preserved on the diagram - aggregation, dependency and attributes were removed for the sake of clarity. As depicted, whole XUpdate specification is implemented, allowing for any type of query to be defined, compiled and send to a chosen endpoint. Because XUpdate uses XML syntax, convenient JDOM model was used for compiled representation. More elaborate description of the model could be found in the PROToS JavaDoc documentation.

Dependencies

Fig. 6.38 shows overview of the **protos-xmldb** dependencies.

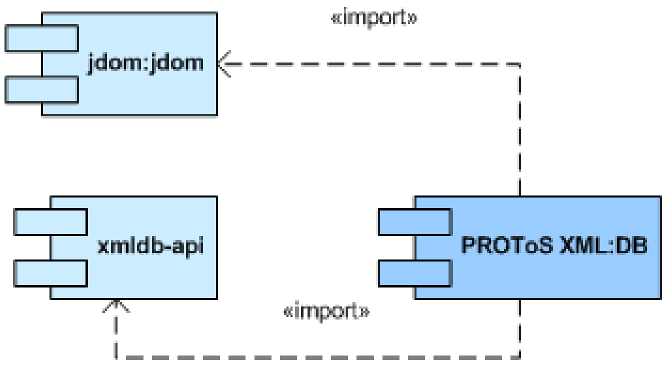


Figure 6.38. PROToS XML:DB module external dependencies as UML component diagram. Model includes JDOM for XML processing and xmldb-api for communication with XML:DB endpoints.

The *xmldb-api* component provides interfaces and data models for communication with eXist database. This include protocols for XQuery and XUpdate languages along with model for databases, collections and data sets. As a rule, *jdom* is used to manipulate XML queries and obtained results.

6.2. Technologies used

In order to achieve all goals mentioned previously, choice to use industry-standard solutions and technologies was made.

6.2.1. Stadards applied

Widely accepted standards makes PROToS more stable and portable. This kind of approach also renders system easier to understand and extend. Therefore, progress made in specific fields - as communication, implemented in adequate standards, can be easily transferred and utilized in the PROToS.

Amongst other, following standards were applied when developing PROToS:

- Dependency Injection components [8] - very light weight software components simplifying implementation switching and inter-dependency management. Following DI principles in design promotes loose coupling and separation between function (interface) and implementation. DI approach also helps by separating algorithms implementation and application configuration, moving it to the special, declarative file.
- Web Services related standards
  - Web Services Architecture [41]
  - Web Services Description Language (WSDL 1.1) [40]
  - SOAP (once Simple Object Access Protocol) [39]
- XML related standards
  - StAX - Streaming API for XML [33]. Very fast with small memory footprint XML access interface. Widely used in high-performance XML processing libraries, as XFire.

— JAXB (Java API for XML Binding) [17]. Powerfull java-to-XML mapping specification. It's implementation is used by the XFire.

- JMX (Java Management Extensions) [20]. It is the standard for remote management and monitoring of Java applications. JMX provides off-the-box tools for building modular, dynamic solutions for large, distributed systems like PROToS.

6.2.2. Solutions used

Development of such system as PROToS could not be possible without usage of external software components. Following list contains most notable technologies and solutions that have been used, with short explanation for each.

1. Dependency Injection components container - Spring [31]
   Spring is a lightweight and highly embeddable container for POJO (Plain Old Java Object) components that makes use of Dependency Injection design pattern. It enables application developer to define multiple objects, implementing specified interfaces, and automatically injects them into dependent components. This unique feature simplifies PROToS design, making it cleaner and much more robust. But Spring is much more than DI-compliant container, being the leading Java, non-EJB application framework. It offers full support for a wide range of popular technologies as JMX, RMI and Web Services. Thus, extending PROToS with new capabilities comes at virtually no cost. All application's configuration, from components definitions to JMX and Web Services settings is stored in one, XML based file. This feature adds much flexibility to the PROToS deployment process.
2. Jetty Servlet Container [19]
   Jetty is lightweight, open-source implementation of the Java Servlet and JSP specifications, written entirely in Java. Embedded, is used in Storage Peer components for deployment of the eXist instance. Main advantage of this approach is high performance with small memory footprint because only truly required libraries are loaded into memory. What is also important to be noted, Storage Peer built with Jetty is able to operate as standalone Java application. It does not need to be packed into WAR and deployed in a fully-fledged container as Tomcat.
3. Apache Tomcat Web Server [5]
   Tomcat is a fully-fledged Java web server, developed by the Apache Software Foundation. Besides standard Servlet and JSP support, as provided by Jetty, it also offers support for other technologies needed by the PROToS, as JNDI registry. With support from native libraries, Tomcat can scale up to thousands of connections, while preserving very good performance. With all that features, it comes completely free. In the PROToS project, Tomcat is used to deploy critical, core component, requiring all mentioned features.
4. XML storage - eXist [9]
   eXist is an Open Source, XML native database with built-in XQuery support. It also supports XUpdate, an open language for modifying XML data. Being native XML database eXist is also quite good performer. On the Java side, eXist supports XML:DB API that provides a common interface to XML databases. Main reason for this choice was XQuery support, as it's one of the primary assumptions for

PROToS system. Another major reason was support for XML:DB API, which speed up PROToS implementation process.

5. Middleware solutions
   - XFire Web Services [47]
     XFire is a next-generation java SOAP framework, with easy to use API and support for many standards, as JAXB. It is built on the StAX model described previously, inheriting it's performance and low memory requirements. XFire was chosen over Apache Axis2 [2] exactly because of performance reasons. In real-world tests, XFire proved to be 2 to 6 times faster than Axis with 1 - 1/5 of it's latency. In my opinion these numbers are the best justification. At present XFire project has been closed and merged with Celtix as Codehaus CXF. Migration to the new framework was considered, but at current stage this project is too immature.
   - RMI [30], internal middleware.
     RMI stands for Remote Method Invocation and is Java-native object oriented API for remote procedure call (RPC). While being pure Java, it's performance is very good. It also has numerous features typical for industry-standard solutions, as support for naming service. RMI provides also tight integration with Java security and encryption mechanisms, making good use of SSL/TLS. All mentioned reasons are convincing enough to choose RMI over other communication methods, as Web Services or CORBA. Of course performance was the most important reason of this choice.
6. Semantic processing
   - Jena2 Ontology API [18]
     Jena is a Java framework for building Semantic Web applications, with excellent API for OWL language. It includes own SPARQL implementation, called ARQ with query engine and rule-based reasoner. Jena allows other reasoners to be deployed using various interfaces, as DIG, and could perform lot of ontology-related processing as validation of OWL classes and individuals. What is more, Jena supports natively reading and storing OWL in XML format, which is crucial as XML database was chosen for data storage. Even though Jena isn't very good performer, it's the best available framework with many useful features, that would be very difficult to obtain otherwise.
   - Pellet [26], an OWL reasoner.
     Pellet is the leading-edge reasoner for OWL ontologies that provides very good performance and full compliance with OWL-DL dialect. It also provides a set of untypical features as ontology analysis and repair, species validation and datatype reasoning, which could be used internally by PROToS. With bundled Jena-compliant interface, Pellet was easily integrated into ontology processing infrastructure of the project.
7. Utilities
   - Woodstox [44]
     Woodstox is a high-performance XML processor, fully compatible with StAX model. It handles also such tasks as XML validation with namespace support. It was chosen over StAX Reference Implementation exclusively because of performance numbers.

- XStream [48]
  XStream is small and fast library for object to XML mapping. It is used in PROToS for run-time configuration persistence. The library was chosen over other solutions, as JAXB because it's XML scheme is very easy to understand. Changing configuration files managed by the XStream requires no specific knowledge of the mapping itself. Also, XStream JAR is much smaller that JAXB's, thus saving class loader memory space.
- Apache Commons - Beanutils [3]
  Because of adopted Java Bean data model for event, PROToS extensively manipulates bean-like data. The Beanutils library contains many convenience methods, absent in the Java JRE. Usage concentrates on dynamic discovery of beans properties, reflecting ontology models that PROToS use.

Chapter 7

Feasibility study

This chapter presents example usage of provenance tracking in the the ViroLab virtual laboratory. In the main part, full provenance-enabled scenario is presented along with provenance data usage. Two section are devoted to integration particulars, living in PROToS environment. In last section, preliminary performance evaluation is given.

This chapter refers to section 4.3, where provenance tracking environment was depicted. Here external components implemented and deployed in the ViroLab virtual laboratory are presented. Also, sample real-world usage is shown. Provenance usage in the ViroLab virtual laboratory is also presented in [52]. PROToS, QUaTRO and virutal laboratory with full provenance capabilities are breifly described in [54].

7.1. Ontologies for the provenance system

As pointed out in many sections of this thesis, ontologies are foundation of provenance tracking. It could be stated that system is as good as the quality of gathered data. Thus, without good provenance models for data, experiment and various domains, whole PROToS system is useless.
In section 1 example ontology models for PROToS are presented. As depicted in Fig. 5.1, good models should have cross-references by object properties and inheritance relations. Fig. 7.1 presents ontology models deployed in the current VLvl.

Diagram contains only few concepts from each model, for the sake of clarity. Ontologies are depicted using UML Class diagram. Even though this kind of diagram can not present such details of ontology models as relation direction or multiplicity, it is able to model object/datatype properties along with inheritance relation. Thus presenting ontology in familiar and easy to understand way.
As shown, there is one additional ontology model - **Upper ontology**. This specific

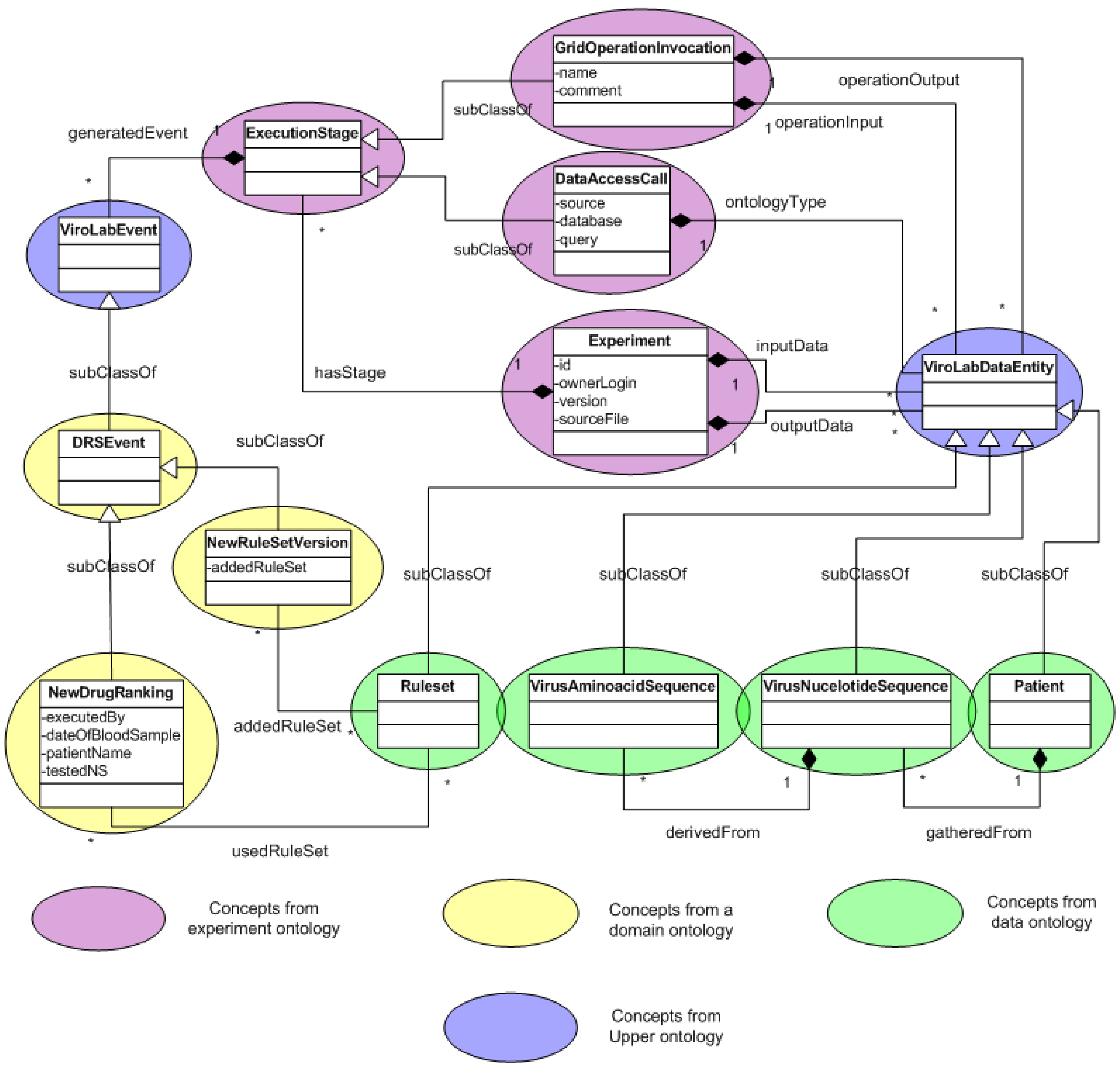


Figure 7.1. Ontology models currently deployed in the ViroLab virtual laboratory. Diagram includes experiment, data and domain specific models, along with special model - upper. Presented models are tightly integrated by multiple inheritance and object property relations.

model contains root concepts defining ViroLab's semantic hierarchies - data (*ViroLab-DataEntity*), event (*ViroLabEvent*) and others. From PROToS point of view, these concepts are treated as part of data models and propagated to all Nodes in the distributed storage. Upper ontology concepts are also common point for cross-domain references. For example, concepts from generic ontology refers only to *ViroLabDataEntity* because inputs and outputs can be of any data type. Also, *ExecutionStage* aggregates domain events, using root concept *ViroLabEvent*. When defining models for particular domain, exact types used by adequate services are known. Therefore, not root but leaf concepts from data ontology could be used. This feature is presented as relation between *NewDrugRanking* and *Ruleset* concepts.

7.2. Provenance usage

This section provides brief description of how provenance data is used in the Vlvl. Description concentrates on user-oriented querying capabilities, as this aspect was emphasized during requirements gathering stage.
Provenance information, gathered during experiment execution in destined primary for user's mining. However, also other virtual laboratory components can make use of this data. Following list summarizes this kind of usage:

1. **Optimization**
   Our Grid application optimizer - GrAppO - is designed to take advantage of gathered provenance data. Such information as service call times, delays in experiment execution and detailed performance of particular technologies involved, passed from monitoring infrastructure to the PROToS constitute great base for optimization purposes. Furthermore, complete performance metrics of many experiments are subject to statistical analysis involving many factors, resulting in even higher level of optimization possible.
2. **Result management**
   Result management at its current stage is designed as combination of persistent storage solution (WebDav [42] and annotations describing such aspects of result as user, type, service and experiment used. Because in the ViroLab semantic data model also results can be treated as data entities, those annotations are stored within PROToS. This results in tight integration of results with regular application data. Also, all capabilities of provenance-managed concepts are preserved.
3. **Experiment replay**
   Currently, experiment replay feature is only a concept. Its implementation within virtual laboratory is yet to be decided. Nevertheless, feature's design makes extensive use of provenance data. As already stated in section 5, such feature can not exist without provenance tracking and storage. What should be noted, Vlvl users expressed their interest in possibilities of experiment replay. It has been pointed out that in some cases, described feature would be more than appreciated.

7.3. QUery TRanslation tOols - QUaTRO

Following section presents description of the QUaTRO - QUery TRanslation Tool, component designed exclusively for mining repositories of provenance and data in the Vlvl. QUaTRO architecture and ideas were developed by author of this work as part of the VLvl. Implementation was carried on by the author and other colleagues.

7.3.1. Introduction

QUaTRO is first attempt to implement Provenance Data Mining component as depicted in Fig. 4.1. It was presented for the first time in [55]. Main concepts behind QUaTRO are as follows:

1. Ontology based. OWL models used in the ViroLab virtual laboratory encapsulate huge amount of knowledge about the system and its applications. Also, properties of ontologies, like *description* allows for user-friendly descriptions and vocabulary

to be used. This way, end user as Virologists is able to fully understand query under construction.

2. Data mappings. Data ontology models allows for representing application data entities as concepts. But only experiment and domain events are to be stored in PROToS. Core idea is to write special RDBMS mappings for data models, encapsulating information needed to fetch data entities from available databases. This way, not only provenance but also data repositories are to be supported by the QUaTRO queries. In current version, such mappings are defined for RegaDB scheme [69]. Using defined cross-domain references even allows for mixed data-provenance queries. Furthermore, QUaTRO GUI can make use of those mappings, allowing user to see available entities while constructing the query.
3. Operators. Wide range of different operators should be provided. This include arithmetic operators (equal, less, greater than), set theory (union, intersection), aggregation (max, min, average). Coupled with ability to enter data domain operators offer possibilities of very powerful queries.

7.3.2. Overview

Fig. 7.2 presents GUI of QUaTRO's current stage.

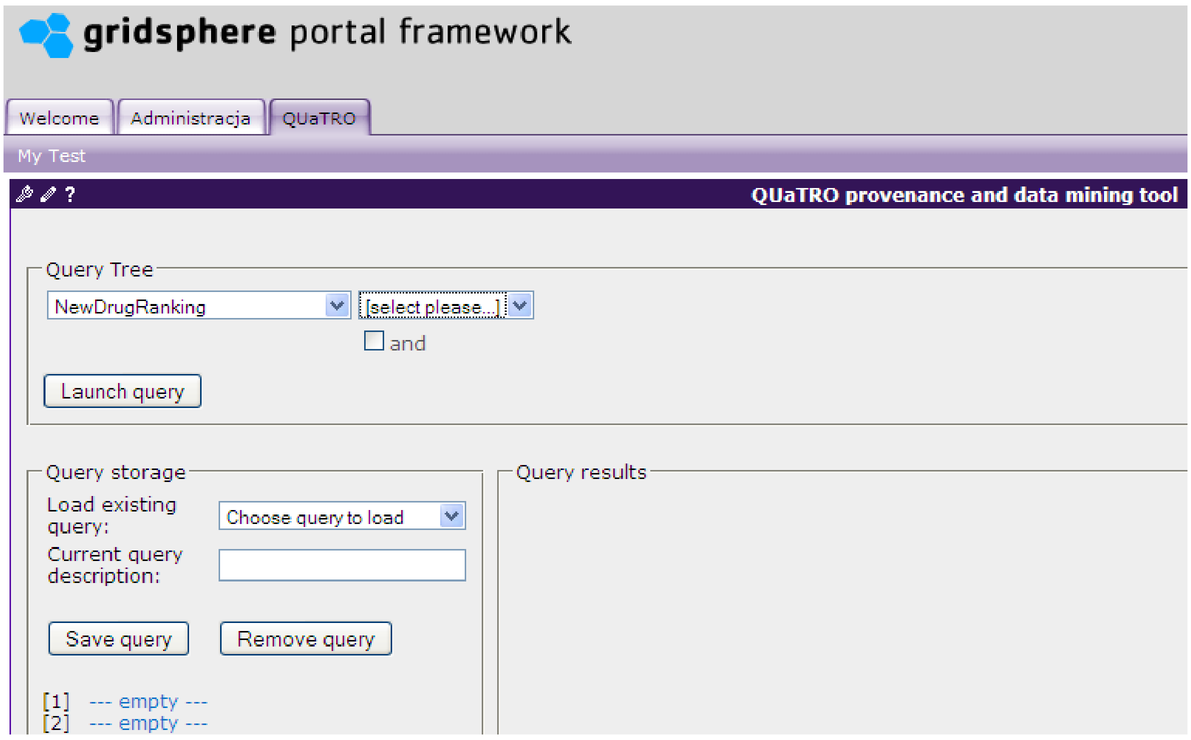


Figure 7.2. Overview of QUaTRO GUI. GUI consists of three main parts: query tree, storage and result view. First panel allows for defining mining queries, using ontologies, data entities and operators. Second one is responsible for managing persistent query storage. Last part presents retrieved query results.

It is composed of three main parts: query tree, query persistence and query result view. First panel is used to defining mixed queries, spanning data and provenance domains. Query definition process is described in section 7.3.3. Third part of the GUI is used for presenting query result. Current stage offers tabular XML view of retrieved provenance or data concepts. Other, more sophisticated result rendering components are under development. Second part is used for query persistence management. It's features are as follows:

1. **Query save**
   This panel part allows persisting currently defined query.Query is to be stored in the internal data base and available for later use. Only user that saved the query is to be able do load it in the future.
   To save current query (that is query present in the editor) user should enter its unique description (*current query description*) and click button *save query*. QUaTRO will display adequate message, depending on the status of saving.
2. **Query load**
   Combo box used for loading queries. The query storage space is separate for each ViroLab's portal user. Credentials for storage are taken automatically from portlet context, so each user is able to see only queries written by himself. User chooses previously saved query using combo box. After selecting a query, QUaTRO automatically loads it into the query tree. Query loaded is immediately ready to be launched.
3. **Query quicsave / quickload**
   Part of the persistence panel, composed of quick slots for queries. Every slot can hold one query, that will be kept in main memory, not persistent storage. This way is much faster than persistent storage, but lasts only as long as portal session of the current user.
   This type of the short-time storage allows user to build few versions of the same query for testing purposes. Actually it acts as a clipboard for QUaTRO users.
   To use this clipboard, users should first input its description in the input box Current query description and click on chosen slot. If slot is already in use, old query will be replaced - and thus lost.

As depicted in Fig. 7.2, technically QUaTRO is a portlet, deployable in the ViroLab portal, running GridSphere [14] container. Moreover, to fit user's requirements, QUaTRO is built with AJAX (Google Web Toolkit [12] used). GWT ensures that GUI response time is minimized. Query persistence module uses Hibernate [15] for ORM mapping. In conclusion implementation uses most modern technologies to enhance user's experience and usability.

7.3.3. Query construction

Fig. 7.3 depicts sample query constructed with usage of QUaTRO GUI.

Depicted query is fairly simple, with three concepts used. Following steps are to be taken in construction process:

- Root concept is selected from combo box. Individuals or data entities reflecting this type are to be returned. Whole query can be viewed as *"Select root concepts fulfilling following criteria..."*. Combo box allows for selecting leaf concept from any model - experiment, data or domain. In this example **Experiment** is chosen.

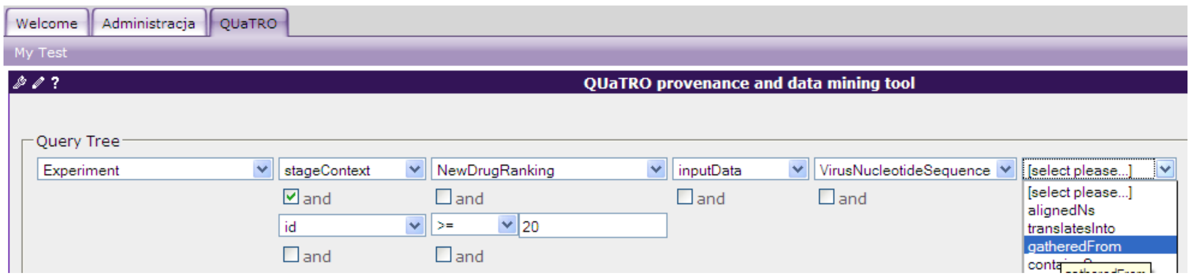


Figure 7.3. Sample query constructed with usage of QUaTRO GUI. Query consists of three concepts, connected by object properties and one datatype property (id) using arithmetic operator.

- Object property is selected from combo box, listing all available properties for concept **Experiment**. In the example, *stageContext* is chosen. This property aggregates domain events sent during experiment's execution.
- Because property chosen is object, whole process of selecting ontology object repeats. However, range of the property reduces scope to child concepts of the **ViroLabEvent**. All other steps are identical as previously. In the example, consequently **NewDrugRanking** and **VirusNucleotideSequence** are selected.
- Next, user chooses to start new criteria branch. It is done by selecting **and** operator. Freshly created combo box allows for selecting Experiment's properties. In the example, datatype property *id* is chosen.
- Because property *id* is datatype, next step involves selecting operator, adequate to property's type. Example query uses arithmetic greater or equal operator.
- In last step user enters required property's value. Thus branch is closed.

To sum up, query construction with QUaTRO is simple and easy to understand. Regardless of query complexity, query is constructed in same way. Currently, test QUaTRO users typically construct queries with as many as 10 branches.

7.4. Sample scenarios

This section presents sample scenarios of provenance usage in the ViroLab virtual laboratory.

7.4.1. Drug Resistance

This scenario makes use of example applications running in the ViroLab virtual laboratory - DRS and REGA tools. The goal of this experiment is to determine best drug combinations for a particular HIV virus taken from the blood sample of a patient. Full scenario contains following stages:

- HIV virus nucleotide sequence is isolated - this stage prepares input for Drug Resistance experiment.
- Alignment of the nucleotide sequence. As a result, HIV genes are identified.

- Obtained gene sequences are compared to reference strains. Mutations are determined.
- Mutations are passed to DRS. Rankings of most effective drugs for particular HIV are obtained.

However presented scenario seems quite simple, provenance tracking of it's execution is quite complex. Multiple events are generated, delivered by the GEMINI [50] monitoring system, collected, semantically aggregated and stored in the PROToS. Collection and semantic aggregation of monitoring events in the ViroLab virtual laboratory is described in [51].
Events generated include:

- generic experiment - start of new experiment instance, calls to services and so on.
- domain specific - enhancing gathered information with knowledge of particular execution's domains. Stages of the scenario generates following domain events: **SequenceAlignment**, **MutationComputation** and **NewDrugRanking**.

Domain-specific events are aggregated and related to events from generic experiment ontology. This way, domain events denotes that particular WS calls are in fact alignment and ranking operations. Moreover, input data entities are to be identified as for example nucleotide sequence.

7.5. Performance evaluation

Aiming to fulfill requirements for provenance tracking system, as defined in section 4.1 basic performance evaluation was performed. However it should be stated that performance improvements for PROToS are yet to be implemented. It is described in details in section 8. Therefore, test configuration involves only one application component of each type. Nevertheless, such test can help evaluating what data sets sizes are acceptable from performance point of view. Also, test involving different queries is helpful in designing query optimization rules for the PROToS.

Fig. 7.4 depicts how PROToS components are deployed in testing environment. Core and Node components are using standard, reference implementations, as configured in appendix B. Peer application component is eXist 2.0.0 instance running in the Tomcat 6.0 container. Physical machines used are as follows:

1. **Machine1** Intel Core Duo, 1.73 Ghz, 1GB RAM, Tomcat 6.0.
2. **Machine2** AMD Opteron 2.4 Ghz dual core, 4GB RAM, Tomcat 5.0.28.

Performance testing was performed for three queries of different complexity. They are as follows:

1. **Query 'simple'**. Select distinct data sources used in calls involving 'Test' bases. Calls are to be part of experiments written in Ruby and calling grid operations with particular substitutions in input.
2. **Query 'advanced'**. Query testing XQuery particulars, as cascade 'for' clause, to retrieve provenance information.

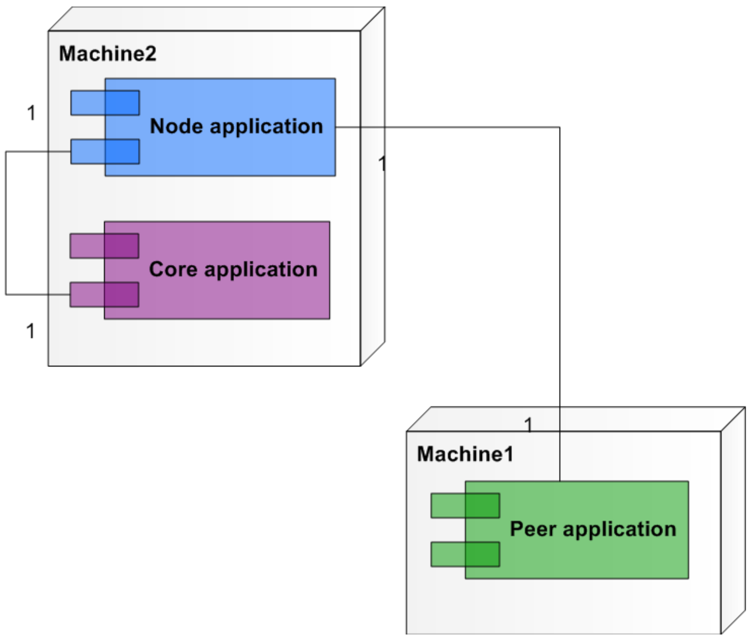


Figure 7.4. Deployment diagram of the PROToS, prepared for preliminary performance testing.

3. **Query 'complex'**. Selects mutations being results of an experiment, calling some grid objects running on Cyfronet's server. Also, experiment's input is set.

Fig. 7.5 presents test results obtained.
As depicted, system scales well for the data sets size under consideration. Response time growth is linear with data amount. However it is clear that achieved response times are acceptable only for simple queries and advance up to 40 mb of data. This should be addressed with implementation of full distributed storage along with other improvements, as described in section 8.

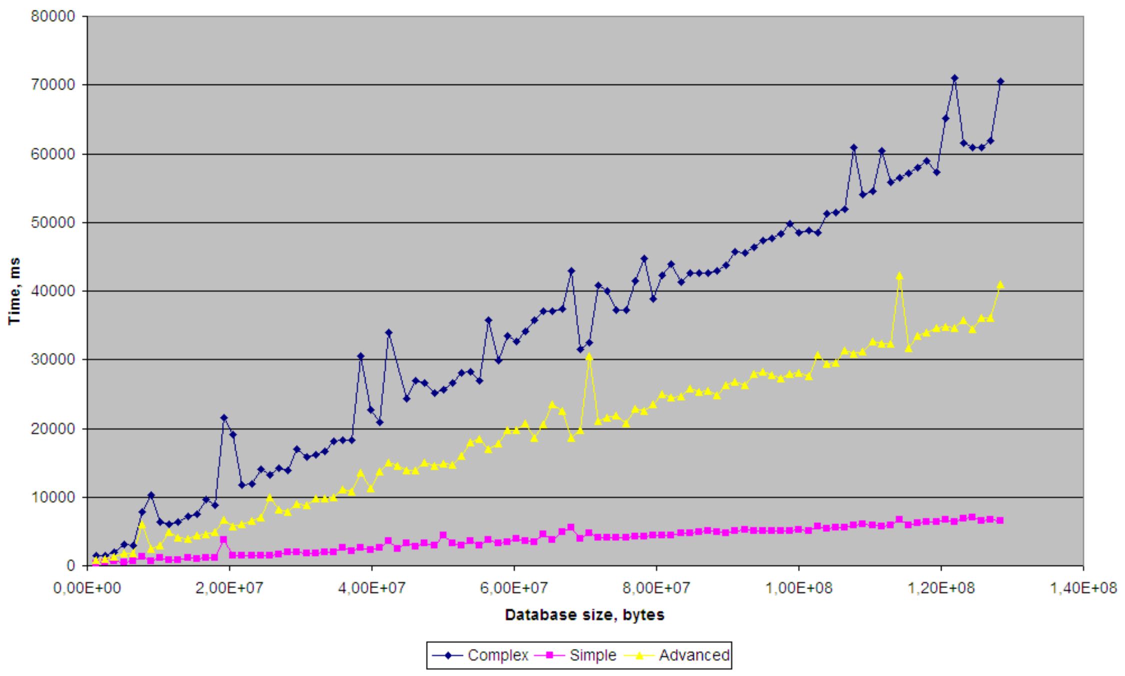


Figure 7.5. Results of the DRE performance testing for three typical queries of different complexity. Tests were performed using PROToS deployment configuration as presented earlier.

Chapter 8

Conclusions and future work

This chapter summarizes work done in this thesis. Main part is devoted to future work that should be done in order to achieve fully-functional provenance tracking system for the ViroLab virtual laboratory.

Principal goal of this work - system for provenance tracking and storage - called PROToS was successfully achieved. This thesis provides precise description of its design and implementation, along with technologies that were used. Furthermore, semantic-rich model for provenance information was presented. It builds on the top of thorough research on virtual laboratories requirements and user's needs. Project identified sweet spot between abstractness and usability of the provenance model. Moreover, also justification in form of possible applications and usage analysis in the ViroLab was shown. Finally, integration in challenging environment of the ViroLab virtual laboratory was described.
What should be pointed out is success of the PROToS, expressed by series of publications. They are listed in appendix D.2. Moreover, as presented in section 7 PROToS is currently fully integrated with VLvl. Provenance information, both generic and domain is tracked in experiments performed in the virtual laboratory and stored in the PROToS distributed storage. Created ontology models reflects applications run by the ViroLab's users and the common data model. Combination of provenance data and those models enables users to perform complex mining, enriching ViroLab's usefulness. What is more, event some widely-used applications, as DRS incorporated provenance usage in their basic use cases. Real-world users from the project are providing feedback for improving both PROToS and QUaTRO.
In conclusion, all this is proof that functional and non-functional requirements defined in chapter 4.1 for the provenance tracking system for the ViroLab virtual laboratory are well met.

PROToS development is somewhat bound to the lifecycle of the **ViroLab** project. Therefore, future work as listed below is bound to last stage of the project, involving performance and reliability improvements.

- Main performance improvement is implementation of fully-functional distributed XQuery support. It is to be based on ontology information, hinting what registries are available on which Nodes. Current prototype supports distributed, multi node architecture, but storage and querying is performed only on one node. Albeit, needed code infrastructure is in place, waiting only for algorithm to be implemented. Also, scattering provenance data based on domain model the data concerns is to be implemented along with distributed queries.
- Future performance enhancements includes also rebalancing ontology data stored on Nodes. First research on proposed architecture shown that with specific usage patterns, some Nodes could be overloaded with data. Current solution is to move some Peers from underutilized Nodes. This however requires user's action and can not be done automatically. New idea assumes that data could be moved between Nodes. Even though moving of the data is proved to be expensive, this could be best solution because in typical systems multiple usage patterns are equally possible. In general, proposed algorithm can also participate in data storage sequence. When delegating provenance event to applicable Node not only domain ontology involved should be taken into account, but also its utilization.
- Reliability improvements concentrates on Nodes backup handling. First enhancement will enable per-Node backup of assigned ontology models and Peer registries. Second enhancement will enable revocation of Nodes and Peers without data loss. When **Core** or **Node** reconfiguration detects that a Node or Peer has been revoked from storage, data from that component will be moved to Node / Peer selected by the algorithm. Possibly, data will be stored on most underutilized Node / Peer. This comes in line with previously described performance improvement.
- System is to be thoroughly tested. High reliability will be assured by code coverage by unit tests at about 90mocks of external components and high code standards secured by conformance to PMD and FindBugs reports.
- Improvements in technologies used - in example, XFire [47] has successor in form of Apache CXF [4]. It addresses most important shortcomings of the XFire. Since Web Services constitute external interface of the PROToS, reimplementation based on CXF could be well justified.

Bibliography

[1] *Apache Ant.*
`http://ant.apache.org`.

[2] *Apache Axis2.*
`http://ws.apache.org/axis2`.

[3] *Apache Commons - Beanutils.*
`http://commons.apache.org/beanutils/`.

[4] *Apache CXF.*
`http://cxf.apache.org/`.

[5] *Apache Tomcat.*
`http://tomcat.apache.org`.

[6] *Chemomentum project.*
`http://www.chemomentum.org/`.

[7] *Cyfronet's GForge server.*
`https://gforge.cyfronet.pl/projects/protos/`.

[8] *Dependency Injection Design Pattern explained.*
`http://www.martinfowler.com/articles/injection.html`.

[9] *eXist Native XML Database.*
`http://exist.sourceforge.net`.

[10] *FindBugs reporting tool.*
`http://findbugs.sourceforge.net/`.

[11] *Globus toolkit.*
`http://www.globus.org/`.

[12] *Google Web Toolkit.*
`http://code.google.com/webtoolkit/`.

[13] *Gridbus.*
`http://www.gridbus.org`.

[14] *GridSphere portal framework.*
`http://www.gridsphere.org/`.

[15] *Hibernate.*
`http://www.hibernate.org/`.

[16] *Java Runtime Environment 6.*
`http://java.sun.com/javase/6/`.

[17] *JAXB - Java Architecture for XML Binding.*
`https://jaxb.dev.java.net/`.

[18] *Jena Semantic Web Framework.*
http://jena.sourceforge.net.

[19] *Jetty6 Servlet container.*
http://www.mortbay.org.

[20] *JMX - Java Management Extensions.*
http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/.

[21] *JRuby language home.*
http://jruby.codehaus.org/.

[22] *Kepler project homepage.*
http://kepler-project.org/.

[23] *Maven 2 Project Management Tool.*
http://maven.apache.org/.

[24] *OGSA - Open Grid Services Architecture.*
http://www.globus.org/ogsa/.

[25] *OGSI - Open Grid Services Infrastructure.*
https://forge.gridforum.org/projects/ogsi-wg.

[26] *Pellet OWL Reasoner.*
http://www.mindswap.org/2003/pellet/.

[27] *PMD reporting tool.*
http://pmd.sourceforge.net.

[28] *Provenance Challenge web site.*
http://twiki.ipaw.info/bin/view/Challenge/.

[29] *Resource Description Framework.*
http://www.w3.org/RDF/.

[30] *RMI - Remote Method Invocation.*
http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

[31] *Spring Application Framework.*
http://springframework.org.

[32] *Spring reference documentation.*
http://static.springframework.org/spring/docs/2.5.x/reference/index.
html.

[33] *StAX - Streaming API for XML.*
http://jcp.org/en/jsr/detail?id=173.

[34] *Taverna project homepage.*
http://taverna.sourceforge.net/.

[35] *TRIANA project homepage.*
http://www.trianacode.org/.

[36] *UNICORE.*
http://www.unicore.eu.

[37] *ViroLab main site.*
http://www.virolab.org.

[38] *ViroLab virtual laboratory.*
http://virolab.cyfronet.pl/trac/vlvl.

[39] *W3C SOAP specification.*
http://www.w3.org/TR/soap/.

[40] *W3C WDL specification.*
http://www.w3.org/TR/wsdl.

[41] *Web Services Architecture.*
`http://www.w3.org/TR/ws-arch/`.

[42] *WebDAV.*
`http://www.webdav.org/`.

[43] *Webster Dictionary.*
`http://www.merriam-webster.com/dictionary/`.

[44] *Woodstox XML Processor.*
`http://woodstox.codehaus.org/`.

[45] *WS-Addressing.*
`http://www.w3.org/Submission/ws-addressing/`.

[46] *WSRF - Web Services Resource Framework.*
`http://www.globus.org/wsrf/`.

[47] *XFire Web Services.*
`http://xfire.codehaus.org`.

[48] *XStream Java-to-XML.*
`http://xstream.codehaus.org/`.

[49] *XStream output reference.*
`http://xstream.codehaus.org/manual-tweaking-output.html`.

[50] Balis B., Bubak M., Dziwisz J., Truong H.-L., Fahringer T.: Integrated Monitoring Framework for Grid Infrastructure and Applications. Cunningham P., Cunningham M., editors, *Innovation and the Knowledge Economy. Issues, Applications, Case Studies*, pages 269–276, Ljubljana, Slovenia, Paz. 2005. IOS Press.

[51] Balis B., Bubak M., Pelczar M.: From Monitoring Data to Experiment Information – Monitoring of Grid Scientific Workflows. *Proc. 3rd IEEE International Conference on e-Science and Grid Computing, e-Science 2007.* IEEE Computer Society, Gru. 2007. In Print.

[52] Balis B., Bubak M., Pelczar M., Wach J.: Provenance querying for end-users: A drug resistance case study. Bubak M., Albada G. D. van, Dongarra J., Sloot P. M. A., editors, *ICCS (3)*, volume 5103 series *Lecture Notes in Computer Science*, pages 80–89. Springer, 2008.

[53] Balis B., Bubak M., Pelczar M., Wach J.: Provenance tracking and quering in virolab. Cracow'07 Grid Workshop, pages 71–76. ACC Cyfronet AGH, 2008.

[54] Balis B., Bubak M., Pelczar M., Wach J.: Provenance tracking and querying in the virolab virtual laboratory. *CCGRID*, pages 675–680. IEEE Computer Society, 2008.

[55] Balis B., Bubak M., Wach J.: User-oriented querying over repositories of data and provenance. *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 187–194, Washington, DC, USA, 2007. IEEE Computer Society.

[56] Balis B., Bubak M., Wach J.: Provenance tracking in the virolab virtual laboratory. Lecture Notes in Computer Science, Parallel Processing and Applied Mathematics : 7th International Conference, PPAM 2007, pages 50–60. Springer, 2008.

[57] Bubak M., Gubala T., Kasztelnik M., Malawski M., Nowakowski P., Sloot P. M. A.: Collaborative virtual laboratory for e-health. *Expanding the Knowledge Economy: Issues, Applications, Case Studies, eChallenges e-2007 Conference Proceedings*, pages 537–544. IOS Press, 2007.

[58] Buyya R., Venugopal S.: A gentle introduction to grid computing and technologies. *Computer Society of India Communications*, 42(1), July 2005.

[59] De Roure D., Jennings N. R., Shadbolt N. R.: The Semantic Grid: A future e-Science

infrastructure. Berman F., Fox G., Hey A. J. G., editors, *Grid Computing – Making the Global Infrastructure a Reality*, pages 437–470. John Wiley and Sons Ltd., 2003.

[60] al. L. M. et: The first provenance challenge. *Concurrencyand Computation: Practice and Experience*, 2007.

[61] Foster I.: What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[62] Foster I., Kesselman C., Tuecke S.: The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–12, 2001.

[63] Goble C., Roure D. D., Shadbolt N. R., Fernandes A. A. A.: Enhancing Services and Applications with Knowledge and Semantics. Foster I., Kesselman C., editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, chapter 23, pages 432–458. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[64] Groth P., Jiang S., Miles S., Munroe S., Tan V., Tsasakou S., Moreau L.: An Architecture for Provenance Systems. Raport instytutowy, University of Southampton, 2006.

[65] Groth P., Luck M., Moreau L.: A protocol for recording provenance in service-oriented grids. *The 8th International Conference on Principles of Distributed Systems (OPODIS'04)*, ["lib/utils:month_11914" not defined] 2004.

[66] Gubala T., Balis B., Malawski M., Kasztelnik M., Nowakowski P., Assel M., Harezlak D., Bartynski T., Kocot J., Ciepiela E., Krol D., Wach J., Pelczar M., Funika W., Bubak M.: Virolab virtual laboratory. Bubak M., Turaa M., Wiatr K., editors, *Proceedings of Cracow Grid Workshop - CGW'07, October 2007*, Krakow, Poland, 2007. ACC-Cyfronet AGH.

[67] Hey T., Trefethen A. E.: The uk e-science core programme and the grid. *Future Gener. Comput. Syst.*, 18(8):1017–1031, 2002.

[68] Kesselman C., Foster I.: The grid: Blueprint for a new computing infrastructure. November 1998.

[69] Libin P., Deforche K., Laethem K. V., Camacho R., Vandamme A.-M.: RegaDB: An Open Source, Community-Driven HIV Data and Analysis Management Environment. *Fifth European HIV Drug Resistance Workshop*, Cascais, Portugal, 2007.

[70] Nemeth Z., Sunderam V.: Virtualization in grids: A semantical approach. Grid Computing: Software Environments and Tools. Springer.

[71] Plale B., Gannon D., Reed D. A., Graves S. J., Droegemeier K., Wilhelmson R., Ramamurthy M.: Towards dynamically adaptive weather analysis and forecasting in lead. Sunderam V. S., Albada G. D. van, Sloot P. M. A., Dongarra J., editors, *International Conference on Computational Science (2)*, volume 3515 series *Lecture Notes in Computer Science*, pages 624–631. Springer, 2005.

[72] Simmhan Y. L., Plale B., Gannon D., Marru S.: Performance evaluation of the karma provenance framework for scientific workflows. Moreau L., Foster I. T., editors, *IPAW*, volume 4145 series *Lecture Notes in Computer Science*, pages 222–236. Springer, 2006.

[73] Sloot P. M., Tirado-Ramos A., Altintas I., Bubak M., Boucher C.: From molecule to man: Decision support in individualized e-health. *Computer*, 39(11):40–46, 2006.

[74] Stevens R. D., Robinson A. J., Goble C. A.: mygrid: personalised bioinformatics on the information grid. *ISMB (Supplement of Bioinformatics)*, pages 302–304, 2003.

[75] Woodruff A., Stonebraker M.: Supporting fine-grained data lineage in a database visualization environment. *ICDE*, pages 91–102, 1997.

[76] Zhao J., Goble C., Stevens R.: Semantically linking and browsing provenance logs for e-science. *Lecture Notes in Computer Science*. Springer, 2004.

[77] Zhao Y., Wilde M., Foster I. T.: Applying the virtual data provenance model.

Moreau L., Foster I. T., editors, *IPAW*, volume 4145 series *Lecture Notes in Computer Science*, pages 148–161. Springer, 2006.

Appendix A

System configuration

Configuration is a major aspect of every system. In such large, fully-distributed and loosely-coupled system as the PROToS it becomes even more important. Careful design of configuration features adds much to the flexibility and overall usability of the application.
PROToS system configuration options fall into one of two types, described in the following section.

A.1. Compile-time

As stated previously, PROToS system makes use of lightweight components, based around Inversion of Control (Dependency Injection) pattern. Current implementation utilizes the Spring Framework [31]. Therefore, configuration of a finished application, that is implementations of required interfaces along with component dependencies is stored in one, declarative file, typically named applicationContext. The file is read on startup by the container and used to bootstrap the application itself with all required components. This configuration can not be changed after container start, so every change requires application restart. Thorough manual on the Spring XML configuration can be found on the Web [32].
Fig. A.1 presents abstract example application based on Dependency Injection container. As depicted, in run-time all component dependencies are resolved by container with regard to application configuration. There is only one instance implementing interface C, injected to two different implementations of interface A. What is more, components type A and C have different implementations of interface B injected. This kind of behavior can be easily configured in the Spring applicationContext file.

Compile-time configuration adds great deal of flexibility to the application. It allows to change almost any part and aspect of the system - from state persistence strategy to the type of the storage used (multi node, single node) by only a few lines of

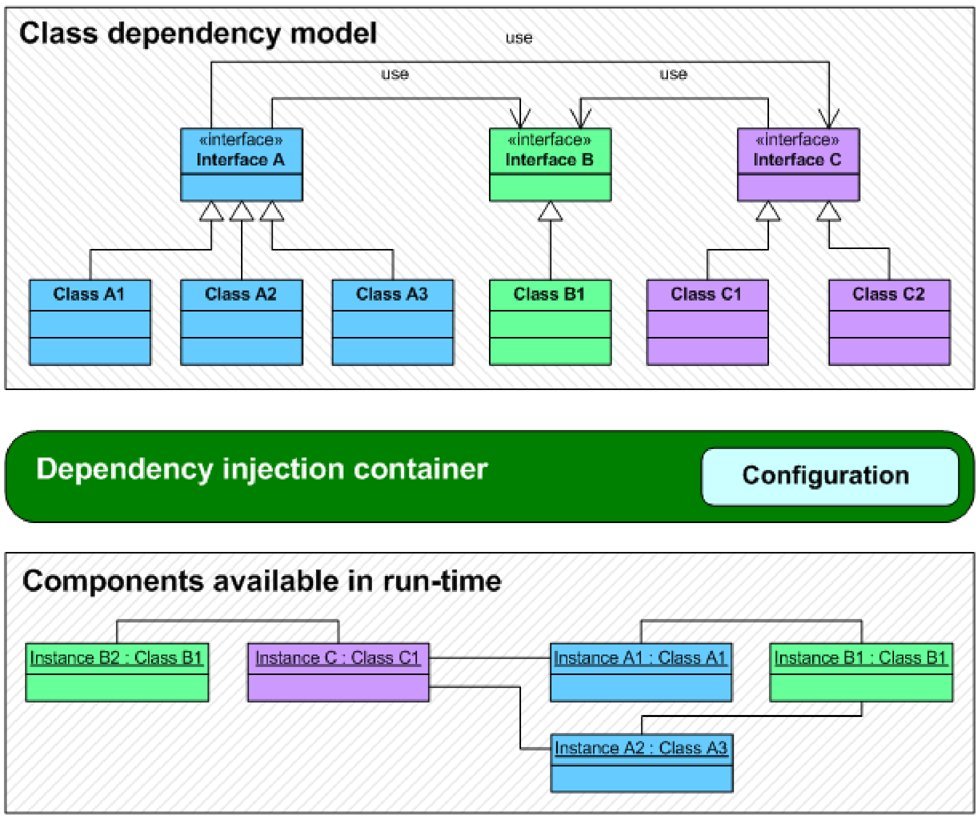


Figure A.1. Example dependency injection container in work. First layer of the diagram presents dependencies between interfaces used in application and available implementations. Middle layer models DI container, using provided configuration to wire application's components. Third layer depicts application in run-time.

XML. What is more, thanks to integration features provided by the Spring, external interfaces of the PROToS are also configured declaratively in the applicationContext file. This includes Web Services, Remote Method Invocation and JMX connectors. So feature simplifies management of the application because such properties as WS endpoint name and interfaces to be managed with JMX are all configured in one file. All these user-friendliness comes without sacrificing separation of concern, as the configuration is still decoupled from the code.

A.2. Run-time

Apart from application configuration that has to be available during boot, as services ports, many components have own properties that could be changed on the fly. Those include in example ontology models used or storage peers belonging to a particular node.
This type of configuration defines actual state of the application. This state, along with configuration defining it should be persistent. It is crucial for usability of the system. To address this requirement, simple model of stateful component was developed.

Fig. A.2 presents stateful model applied to the Storage Super Node component.

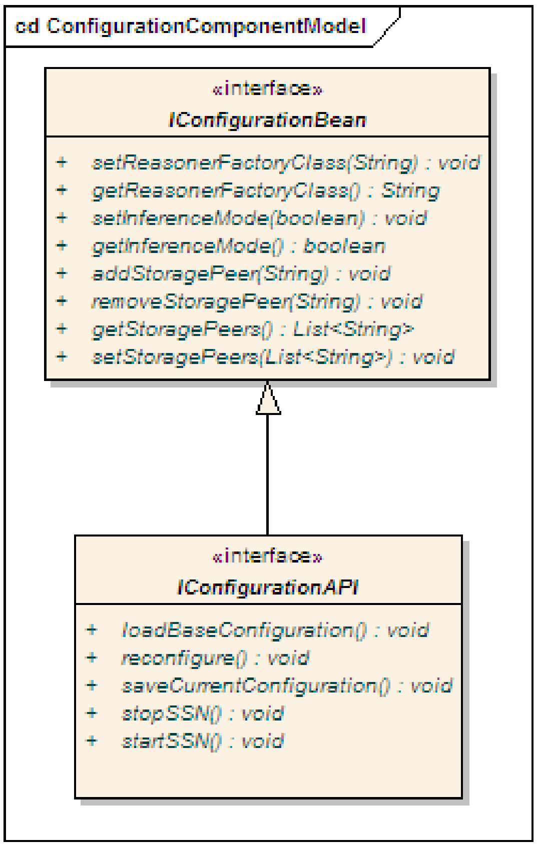

cd ConfigurationComponentModel
«interface»
IConfigurationBean
+ setReasonerFactoryClass(String) : void
+ getReasonerFactoryClass() : String
+ setInferenceMode(boolean) : void
+ getInferenceMode() : boolean
+ addStoragePeer(String) : void
+ removeStoragePeer(String) : void
+ getStoragePeers() : List<String>
+ setStoragePeers(List<String>) : void
«interface»
IConfigurationAPI
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ saveCurrentConfiguration() : void
+ stopSSN() : void
+ startSSN() : void

Figure A.2. Model of the example configuration-enabled stateful component. It consists of one 'Bean' interface, defining configuration attributes and one 'API' interface, defining configuration management behaviour.

It will be used as an example.
Configuration of each component is composed from following interfaces:

- at least one **'Bean'** interface
  These interfaces define configuration properties in some aspect of component's operation, as ontologies or nodes management. These properties are exposed in standard Java bean manner. For property X there should be a read method **getX** and if property is writable, also a write method **setX**. There could be also convenience method for some properties. In the example, for property **storagePeers** apart from get/set methods there exist also methods for adding and removing single entries from a collection.
  Simple convention is to name these interfaces with **'ConfigurationBean'** suffix (in example: IOntologyConfigurationBean, INodesConfigurationBean).
- exactly one **'API'** interface
  This interface extends **'Bean'** interfaces and is implemented by the container to obtain stateful behavior. It defines methods required for stateful component management. These are as follows:
  — loadBaseConfiguration
  Loads configuration beans from persistent storage (called **'base'** one). Any

pending changes to the current configuration will be discarded. This method does not change state of the components, only configuration properties of beans.

— saveCurrentConfiguration
Saves current configuration beans properties to the persistent storage. Thus, current configuration becomes the **'base'** one. Old base configuration becomes obsolete and is not available anymore.

— reconfigure
Reconfigures component, using current configuration. This method actually changes **state** of the component, so should be used with caution. Does not change base configuration, so in case of problems last stable configuration could be loaded and used to reconfigure a component.

— stopX, where X defaults to the component name
This method should cause component to stop it's operations, free resources and persist both configuration and state. State persistence is required because base configuration can still be modified after component is stopped. Therefore, system has to be able to determine after boot whether to reconfigure. The method will be called by a container on application shutdown.

— startX, where X defaults to the component name
This method should initialize component by allocating resources and loading both state and configuration from persistent storage. Also, reconfiguration method should be called. This should be done to assure correct operation when the base configuration was changed after state had been persisted. The method will be called by a container any time component is put into service.

Simple convention is to name this interface 'IConfigurationAPI'.

A.2.1. XML configuration files

First mode of working with a run-time configuration is by usage of XML files. As stated in previous section, configuration-enabled components are persisting configuration as a **'base'** one. The **protos-config** module, described in the implementation section, is provided for this purpose.
Actual resources containing component's configuration are set by container. To modify the configuration, one have to simply edit those files with text or XML editor. Files can be changed while component is both stopped and running. In the latter case, user should call **loadBaseConfiguration** method from respective **API** interface, to make freshly edited configuration the current one.

Setting configuration component

As every component, also configuration one is managed by the Dependency Injection container. Therefore, specific entry in the adequate applicationContext file has to be added.

```
<bean
  class="pl.cyfronet.virolab.protos.sn.ssn.impl.
         XMLConfigurationProvider">
  <property name="beanKey" value="ssnBean"/>
  <property name="classpathResource" value="beansConfig.xml"/>
  <property name="fileSystemResource" value="./beansConfig.xml"/>
  <property name="preferredResource" value="FILE_SYSTEM"/>
  <property name="strategyClassName"
    value="pl.cyfronet.virolab.protos.config.xstream.
            OneFileStrategy"/>
</bean>
```

Listing A.1. Example component configuration. Represents config provider for the SSN component instance.

Listing A.1 shows example configuration of the component for SSN. Management settings applied in the example includes:

- bean key, used by the configuration provider as unique name for storing and retrieval of the bean
- name of the class path resource used for configuration storage. In example it is beansConfig.xml.
- name of the file system resource used for configuration storage. Here it is beansConfig.xml file located in the root application directory.
- definition of the preferred resource to be used. In this example the file system resource is preferred.
- bean storage strategy to be used. Currently single file storage should be used.

Using default configuration implementation

At present, the XStream [48] based implementation is provided for configuration bean (de)serialization. Implementation of the main interface **IConfigurationProvider** is located in the package **pl.cyfronet.virolab.protos.config** as class **GenericXMLProvider**.
The implementation offers one additional setting option - **strategyClassName**. Strategy is used for storing and retrieving XML bean representation from actual resources. Configuration component is distributed with one strategy, namely **OneFileStrategy**. It simply uses one File - type resource, provided in constructor, for storing all beans data. Details can be found in the respective implementation section.
XStream produces XML output that is very easy to understand, modify and use.

```
<pl.cyfronet.virolab.protos.sn.ssn.config.ConfigurationBean>
  <storagePeersAddresses>
    <string>exist::localhost:20080:exist/xmlrpc/db</string>
  </storagePeersAddresses>
  <allModelsUrls>
    <string>http://virolab.cyfronet.pl/onto/test1.owl</string>
  </allModelsUrls>
  <supportedUris>
    <string>http://www.virolab.org/onto</string>
  </supportedUris>
  <inferenceMode>false</inferenceMode>
</pl.cyfronet.virolab.protos.sn.ssn.config.ConfigurationBean>
```

Listing A.2. Example output of the XStream Java to XML streamer. Presents SSN ConfigurationBean instance in XML.

Listing A.2 shows example XStream output - serialized SSN ConfigurationBean instance. Listing demonstrates following XStream syntax features:

- collections and lists content is mapped with element type preserved, enabling usage of non-generic collections.
- in case of built-in types, actual type is omitted, as in case of the boolean **inferenceMode** property.
- user-defined types are simply mapped as full-qualified Java names. This applies also to the bean class.
- properties of beans are written in the standard, Java bean manner.

Complete reference of the XStream output can be found on the web [49].

A.2.2. Remote configuration by JMX

Java Management Extensions [20] is used by the PROToS to enable remote configuration of all components. JMX integration is done with specific support from Spring container. Although this imposes tight code coupling with Spring, it simplified system's implementation and added some interesting features. Besides, with Spring support, every aspect of JMX configuration, as MBean server, is managed from the applicationContext file.

Setting up the JMX

As mentioned previously, all JMX related configuration is managed by Spring. Apart from actual system components, every application needs JMX MBean Server, Connector and Exporter. Therefore applicatonContext have to contain respective entries.

```
<bean id="mbeanServer"
  class="org.springframework.jmx.support.MBeanServerFactoryBean">
  <!-- Will not create new MBeanServer each time -->
  <property name="locateExistingServerIfPossible" value="true"/>
</bean>
```

Listing A.3. Example configuration of the MBean server instance. Sets up factory class to be used along with some configuration properties.

Listing A.3 shows example configuration of the JMX **MBean server**. Besides class serving required functionality, configuration contains only one option - *locateExistingServerIfPossible*. It tells factory, that only one MBeanServer instance should be used within current application. Although not necessary, this options saves some Java memory.

```
<bean id="serverConnector"
  class="org.springframework.jmx.support.
            ConnectorServerFactoryBean">
  <property name="serviceUrl"
   value="service:jmx:rmi://127.0.0.1/jndi/
          rmi://127.0.0.1:3099/Configuration"/>
  <property name="server" ref="mbeanServer"/>
  <property name="threaded" value="true"/>
  <property name="daemon" value="true"/>
</bean>
```

Listing A.4. Configuration of the factory for JMX ConnectorServer's. Sets up adequate class and bunch of required properties.

Listing A.4 depicts example configuration of the JMX **Server Connector Factory**. It's function is creating JSR-160 JMXConnectorServer, registering it with provided MBeanServer instance and putting into service. In the example *serviceUrl* property is configured to use JNDI and expose configuration under name Configuration on local's machine port 3099. Other properties configure JMXServerConnector to be started in a separate thread as daemon.

```
<bean id="JmxExporter"
   class="org.springframework.jmx.export.MBeanExporter"
   lazy-init="false">
  <property name="server" ref="mbeanServer"/>
</bean>
```

Listing A.5. Example configuration of the MBean Exporter bean. Configures class serving functionality and sets up MBean Server that should be used.

Listing A.5 presents example configuration of the JMX **MBean Exporter**. Properties shown only forces Spring container to init the component at once (*lazy-init* property) and configures MBeanServer to be used (*server* property).
Nonetheless, this component plays very important role in the whole configuration, as it exports application components managed through JMX. It is done by two additional properties - *beans* and *assembler*.

```
<bean id="JmxExporter">
  <property name="beans">
    <map>
      <entry key="connector:name=Configuration"
             value-ref="serverConnector"/>
      <entry key="protos.config:name=BeanName1"
             value-ref="Bean1"/>
      <entry key="protos.config:name=BeanName2"
             value-ref="Bean2"/>
    </map>
  </property>
  <property name="assembler">
    <bean ref="MBeanAssembler"/>
  </property>
</bean>
<bean id="MBeanAssembler"
```

```
  class="org.springframework.jmx.export.assembler.
        InterfaceBasedMBeanInfoAssembler">
  <property name="interfaceMappings">
    <props>
      <prop key="protos.config:name=BeanName1">
        pl.cyfronet.virolab.protos.BeanInterfaceA
      </prop>
      <prop key="protos.config:name=BeanName2">
        pl.cyfronet.virolab.protos.BeanInterfaceB;
        pl.cyfronet.virolab.protos.BeanInterfaceC
      </prop>
    </props>
  </property>
</bean>
```

Listing A.6. Configuration of JMX-enabled component's export. Uses previously described Exporter bean and new Assembler component. The latter configures which interfaces of JMX beans shall be externally accessible.

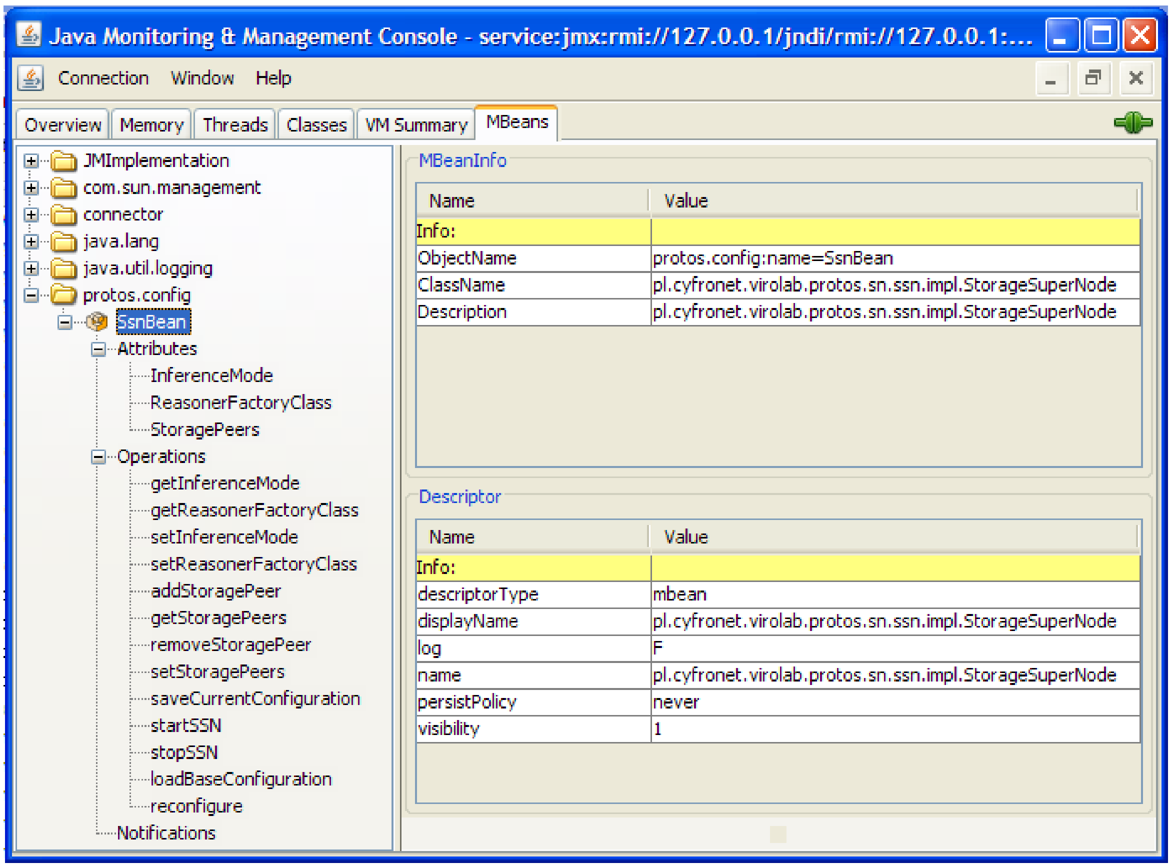
Listing A.6 shows how these properties shall be used to enable JMX remote management for chosen application components.

1. Firstly, each component should be registered under unique name in the JMX Exporter. This part is done by adding entries to the *beans* map. In the example we have two distinct components, namely **Bean1** and **Bean2**. Additionally, the Server Connector should be also added to the map.
2. Secondly, specific MBean Assembler should be configured. In the example an instance of the **InterfaceBasedMBeanInfoAssembler** is used. The exporter allows for choosing which interfaces of the component should be available through JMX. This is very important feature, as it enables to expose only interfaces defined as the Configuration-enabled Component model (Fig. A.2. It is done by configuring property *interfaceMappings* of the Assembler. Each component name is mapped to the list of exportable interfaces. In the example component **Bean1** exports one interface - **BeanInterfaceA** and component **Bean2** exports two interfaces - **BeanInterfaceB** and **BeanInterfaceC**.

All above applicationContext entries constitute JMX configuration for the PROToS applications. No additional code or component is required.

Usage of the JMX

Preferred way of working with JMX configuration is by using the **Java Monitoring and Management Console (JConsole)** tool. It it bundled with Java Runtime Environment, and can be started by typing *jconsole* in the command line. After the tool is started, one should check **Remote Process** box and enter service URL. It is the same as value of the *serviceUrl* property of the **JMX Connector** bean, as shown in listing A.4. In the example it is *service:jmx:rmi://127.0.0.1/jndi /rmi://127.0.0.1:3099/Configuration*. Of course localhost address (127.0.0.1) should be replaced with actual IP address of the machine hosting the application to be configured.


Java Monitoring & Management Console - service:jmx:rmi://127.0.0.1/jndi/rmi://127.0.0.1:...
Connection
Window
Help
Overview
Memory
Threads
Classes
VM Summary
MBeans

```
JMImplementation
com.sun.management
connector
java.lang
java.util.logging
protos.config
    SsnBean
        Attributes
            InferenceMode
            ReasonerFactoryClass
            StoragePeers
        Operations
            getInferenceMode
            getReasonerFactoryClass
            setInferenceMode
            setReasonerFactoryClass
            addStoragePeer
            getStoragePeers
            removeStoragePeer
            setStoragePeers
            saveCurrentConfiguration
            startSSN
            stopSSN
            loadBaseConfiguration
            reconfigure
        Notifications
```

MBeanInfo

Name	Value
Info:	
ObjectName	protos.config:name=SsnBean
ClassName	pl.cyfronet.virolab.protos.sn.ssn.impl.StorageSuperNode
Description	pl.cyfronet.virolab.protos.sn.ssn.impl.StorageSuperNode

Descriptor

Name	Value
Info:	
descriptorType	mbean
displayName	pl.cyfronet.virolab.protos.sn.ssn.impl.StorageSuperNode
log	F
name	pl.cyfronet.virolab.protos.sn.ssn.impl.StorageSuperNode
persistPolicy	never
visibility	1

Figure A.3. Example screen of Console connected to the remote SSN component. As depicted, all configuration attributes and methods are described and available.

Fig. A.3 depicts JConsole connected to the SSN application. Main screen consists of left-sided navigation and right-sided property view. The SSN bean is selected and it's tree is unfolded. All component properties, defined in **'Bean'** interfaces are listed under the *Attributes* branch. Methods, defined in the **'API'** interface along with property getters and setters are listed under the *Operations* branch. This kind of view makes management through JConsole an easy task. What is more, MBean info view provides component's meta data details. Such data as previously configured domain and bean name or actual class of the component can be quickly checked.

A.3. Detailed system configuration

This section presents detailed configuration of all PROToS system applications and components. Where applicable, configuration is divided on run-time and compile-time.

A.3.1. PROToS Core application

The PROToS Core application configuration lies in the **protos-interfaces** module. Main file is default Spring container applicationContext.xml. It consists of beans (components) definitions, which will be described in details later and common application configuration, for such aspects as RMI, JMX and WebServices. JMX was already covered in details in previous section, therefore listing A.7 focuses on other aspects.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

  <bean id="registry"
   class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
    <property name="port" value="5099"/>
    <property name="alwaysCreate" value="true"/>
  </bean>
  <import
   resource="classpath:org/codehaus/xfire/spring/xfire.xml"/>
  <bean id="addressingHandler"
     class="org.codehaus.xfire.addressing.AddressingInHandler"/>
  <bean id="DreWsCommunicatorBean"
     class="pl.cyfronet.virolab.protos.dre.WSCommunicator">
    <property name="implementation" ref="DreBean"/>
  </bean>
  <bean id="DgeWsCommunicatorBean"
     class="pl.cyfronet.virolab.protos.dge.WSCommunicator">
    <property name="implementation" ref="DgeBean"/>
  </bean>
  <bean name="DreService"
     class="org.codehaus.xfire.spring.ServiceBean">
    <property name="serviceBean" ref="DreWsCommunicatorBean"/>
    <property name="serviceClass" value=
    "pl.cyfronet.virolab.protos.dre.interfaces.IDataRetrieval"/>
    <property name="name" value="DreService"/>
    <property name="inHandlers">
      <list>
        <ref bean="addressingHandler"/>
      </list>
    </property>
  </bean>
  <bean name="DgeService"
     class="org.codehaus.xfire.spring.ServiceBean">
    <property name="serviceBean" ref="DgeWsCommunicatorBean"/>
    <property name="serviceClass" value=
    "pl.cyfronet.virolab.protos.dge.interfaces.IDataGathering"/>
    <property name="name" value="DgeService"/>
    <property name="inHandlers">
      <list>
        <ref bean="addressingHandler"/>
      </list>
    </property>
  </bean>
```

```
47
48    <bean id="mbeanServer" />
49    <bean id="serverConnector" />
50
51    <bean id="DreBean" />
52    <bean id="DgeBean" />
53    <bean id="DssBean" />
54  </beans>
```

Listing A.7. Common configuration of the Core application. Contains only application specific beans without logic serving components. Also already described JMX beans are omitted.

First lines of the listing, that is 6-10 configures bean responsible for setting up RMI Registry, required for JMX operations. First property, *port* configures simply TCP server port for incoming RMI connection. The latter, *alwaysCreate*, forces container to create separate Registry for the Core application. It is necessary to avoid potential namespace collisions.
Lines 12-46 represent configuration of the WebServices related beans. First line imports definitions from external file. This is very convenient because configuration common to all WebServices enabled applications don't need to be repeated. In lines 15-22 beans representing communicator for WebServices are declared. Beans are delegating externally exposed methods to internal implementations, provided by the *implementation* property. Next listing lines, namely 23-46 configures actual Web Services for data gathering and retrieval. Each service requires following properties to be set:

- *serviceBean*
  Component serving logic for the service. It is passed as a reference to the Spring bean.
- *serviceClass*
  Fully qualified name of the Java interface that will be exposed as Web Service. This allows for beans with multiple interfaces implemented but only one exposed.
- *name*
  Name of the service, that will be part of the service URL used for Web Service communication.
- *inHandlers*
  List of in handlers required by the service. As the PROToS services are fully configured in the applicationContext, only standard XFire **AddressingInHanlder** is necessary. If system was to use for example JAX-WS services, it would need additional handlers here.

Lines 48-49 contains references to the JMX-specific beans, covered in previous section. Next lines, namely 51-53 defines application beans, described in details in next, respective sections.

A.3.2. PROToS Node application

Configuration for the PROToS Node application lies in the applicationContext.xml file in the **protos-ssn** module. Configuration comprises application beans, which is covered in details in following sections and common RMI/JMX configuration.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
3    "http://www.springframework.org/dtd/spring-beans.dtd">
4
5  <beans>
6
7    <bean id="RmiExporter"
8     class="org.springframework.remoting.rmi.RmiServiceExporter">
9      <property name="serviceName" value="StoragePeer"/>
10     <property name="service" ref="SsnBean"/>
11     <!-- use same Registry as for JMX -->
12     <property name="registry" ref="registry"/>
13   </bean>
14   <bean id="RmiCommunicator"
15    class="pl.cyfronet.virolab.protos.sn.ssn.RMICommunicator">
16     <property name="core">
17       <ref bean="SsnBean"/>
18     </property>
19   </bean>
20
21   <bean id="registry" />
22   <bean id="mbeanServer" />
23   <bean id="JmxExporter" />
24   <bean id="SsnBean" />
25 </beans>
```

Listing A.8. Example configuration of the Node application. Contains only application's configuration. Beans constituting application's logic and JMX configuration are omitted.

Listing A.8 depicts Node application configuration with shortened description of beans covered in other sections.
Lines 7-19 of the appplicationContext contains configuration of the RMI support. Bean **RmiCommunicator** is delegating logic to the actual implementation, provided as the *core* property. Bean **RmiExporter** configures Spring RMI support to export the bean to the RMI registry, provided in property *registry*. Provided component will be served under the name provided in a respective property (*serviceName*). It is all that is required to set up Remote Method Invocation communication for the Node.
Line 21 defines RMI Registry bean, covered in previous section. Lines 22-23 configures JMX as described earlier. Finally line 24 deals with application component, covered in respective following section.

A.3.3. DSS component

Distributed Storage Supervisor component is most important part of the Core application. It manages whole PROToS storage, so careful configuration is key to fulfill desired system requirements.

Compile-time

Compile-time configuration of the DSS component shall be placed in the applicationContext.xml file of the **Core application**, found in the **protos-interfaces** module. Listing A.9 shows typical configuration of the bean.

```xml
<bean id="DssBean"
  class="pl.cyfronet.virolab.protos.dss.impl.StorageSupervisor"
  init-method="start" destroy-method="stop">
  <property name="provider">
    <bean class=
    "pl.cyfronet.virolab.protos.dss.impl.XMLConfigurationProvider">
      <property name="beanKey" value="dssBean"/>
      <property name="classpathResource" value="beansConfig.xml"/>
      <property name="preferredResource" value="CLASS_PATH"/>
      <property name="strategyClassName" value=
      "pl.cyfronet.virolab.protos.config.xstream.OneFileStrategy"/>
    </bean>
  </property>
  <property name="stateProvider">
    <bean class="pl.cyfronet.virolab.protos.pstore.local.
                 LocalSerializableStorage">
      <property name="resources">
        <map>
          <entry>
            <key><value>CLASS_PATH</value></key>
            <value>DssState.xml</value>
          </entry>
        </map>
      </property>
      <property name="preferredResource" value="CLASS_PATH"/>
    </bean>
  </property>
  <property name="nodes">
    <bean
     class="pl.cyfronet.virolab.protos.dss.impl.NodesDataBase"/>
  </property>
  <property name="ontologies">
    <bean
     class="pl.cyfronet.virolab.protos.onts.impl.OntologyDataBase"/>
  </property>
</bean>
```

Listing A.9. Example compile-time configuration of the DSS Bean used in the Core

application. Sets up implementation class along with critical properties as nodes and ontology databases.

First line of listing A.9 contains definition of the bean along with configuration of lifecycle methods. These are *init-method* and *destroy-method* attributes, pointing at methods defined for the stateful component model (Fig. A.2). Consecutive lines configure properties of the bean, conveniently listed below.

- *provider*
  Sets configuration provider, responsible for serializing, persisting and deserializing configuration beans of the component. Configuration as depicted makes use of standard **protos-config** module, which was already described. It is arranged to store data under dssBean name in the file beansConfig.xml available in the classpath.
- *stateProvider*
  This property configures bean responsible for persisting state of the DSS component. Because state is not required to be editable, depicted config uses class **LocalSerializableStorage** from module **protos-pstore** for this functionality. It used default Java serialization mechanisms along with XML headers to manage given objects. Utility requires two properties to be set:
  — *resources*, declaring type and location of resources used to store object in question
  — *preferredResource*, managing which resource should be chosen in case of multiple configured
- *nodes*
  Configures storage nodes database to be used. This is very important because crucial algorithms as data and ontology distribution are implemented in this component. Configuration shown in the example listing uses basic **NodesDataBase** implementation, intended for developer environment and testing.
- *ontologies*
  Sets instance of class managing ontology models. Typically this is simple class, with implementations differing only in applied minor optimizations.

Run-time

Fig. A.4 depicts configuration model for the DSS component. There are two **'bean'** interfaces - *IOntologyConfigurationBean* and *IConfigurationBean* and one **'API'** interface, namely *IConfigurationAPI.*The latter is deeply covered in the respective section about configuration models. 'Bean' interfaces, representing configuration properties of the component are described as follows:

- *IOntologyConfigurationBean*
  The interface contains methods for manipulating ontology properties. Currently only properties are models - domain, data specific and experiment.
  This section uses notion of **storage URL**. This is an specific URL where an ontology model can be found. It's format is: [protocol][://][URI]. Currently the PROToS system supports protocols **http** and **classpath**.
  List of the interface methods:
  — *setAllModelsUrls* - sets list of domain ontology models to be used. List contains storage URLs where these models can be found.

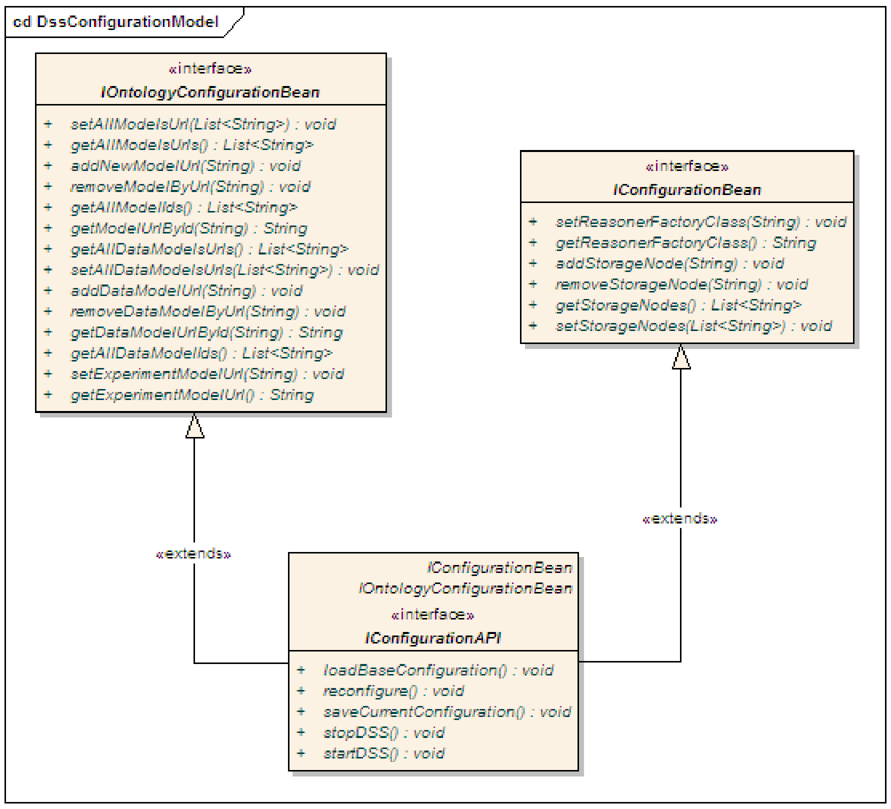

cd DssConfigurationModel
«interface»
IOntologyConfigurationBean
+ setAllModelsUrl(List<String>) : void
+ getAllModelsUrls() : List<String>
+ addNewModelUrl(String) : void
+ removeModelByUrl(String) : void
+ getAllModelIds() : List<String>
+ getModelUrlById(String) : String
+ getAllDataModelsUrls() : List<String>
+ setAllDataModelsUrls(List<String>) : void
+ addDataModelUrl(String) : void
+ removeDataModelByUrl(String) : void
+ getDataModelUrlById(String) : String
+ getAllDataModelIds() : List<String>
+ setExperimentModelUrl(String) : void
+ getExperimentModelUrl() : String
«interface»
IConfigurationBean
+ setReasonerFactoryClass(String) : void
+ getReasonerFactoryClass() : String
+ addStorageNode(String) : void
+ removeStorageNode(String) : void
+ getStorageNodes() : List<String>
+ setStorageNodes(List<String>) : void
«extends»
«extends»
IConfigurationBean
IOntologyConfigurationBean
«interface»
IConfigurationAPI
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ saveCurrentConfiguration() : void
+ stopDSS() : void
+ startDSS() : void

Figure A.4. PROToS DSS component configuration model. Model is composed of two Bean interfaces - for common and ontology configuration along with one standard API interface.

- *getAllModelsUrls* - returns list of domain ontology models. Each model is represented by the URL it was fetched from.
- *addNewModelUrl* - adds new domain ontology model from storage URL passed.
- *removeModelByUrl* - forces component to remove domain model from use. Model is identified by the storage URL.
- *getAllModelIds* - returns list containing unique IDs of all domain models currently in use.
- *getModelUrlById* - returns storage URL of the domain model with passed unique ID
- *getAllDataModelsUrls* - gets list of data ontology models. Each model is represented by the storage URL.
- *setAllDataModelsUrls* - sets list of data ontology models to be used within PROToS system. List contains storage URLs where these models can be fetched from.
- *addDataModelUrl* - adds new data ontology model. Model is represented by the storage URL.
- *removeDataModelByUrl* - removes data model from use. Model is identified by the storage URL.

  — *getDataModelUrlById* - gets storage URL of the data model identified by the passed unique ID
  — *getAllDataModelIds* - returns list containing unique IDs of all data models in use.
  — *setExperimentModelUrl* - sets storage URL of the experiment model.
  — *getExperimentModelUrl* - gets storage URL of the experiment model.
- *IConfigurationBean* This interface defines methods for manipulating generic DSS component functionalities, as storage management.
  This section uses notion of **node address**. Format of the address is: [protocol][dns name—ip = url]:[port]:[node name]. At present, only one protocol is supported - **rmi**. For example, well-formed node address could be:
  'rmi:www.foo.bar.com:1099:StorageNode'.
  Interface methods follow:
  — *setReasonerFactoryClass* - sets class that will act as OWL ontology reasoner. Attribute should be fully-qualified class name available in the run-time.
  — *getReasonerClass* - returns fully-qualified name of the reasoner class.
  — *addStorageNode* - adds new node to the distributed storage. Argument should be well-formed node address.
  — *removeStorageNode* - removes storage node using node address as call argument.
  — *getStorageNodes* - returns list of node addresses available in the storage.
  — *setStorageNodes* - sets storage node list. Collection passed should contain only well formed unique node addresses.

What should be expressly repeated, configuration bean methods do not change actual **state** of the component. For example, after call to the *removeNode* method storage will not change. It would be changed only after call to the *reconfigure* method.

A.3.4. DRE component

This component performs functions related to the data retrieval, as processing incoming queries. It is fully configurable with regard to supported query types, languages and returned results.

Compile-time

Compile-time DRE configuration should be placed in the applicationContext.xml file of a **Core application**. Typically file is located in the **protos-interfaces** module. Listing A.10 shows example configuration of the DRE bean.

```
<bean id="DreBean" class=
  "pl.cyfronet.virolab.protos.dre.impl.DataRetrievalImplementation"
  init-method="start" destroy-method="stop">
  <property name="dss" ref="DssBean"/>
  <property name="provider">
    <bean class=
    "pl.cyfronet.virolab.protos.dre.impl.XMLConfigurationProvider">
      <property name="beanKey" value="dreBean"/>
      <property name="classpathResource" value="beansConfig.xml"/>
      <property name="preferredResource" value="CLASS_PATH"/>
```

```
11          <property name=”strategyClassName” value=
12          ”pl.cyfronet.virolab.protos.config.xstream.OneFileStrategy”/>
13        </bean>
14     </property>
15     <property name=”formatters”>
16        <list>
17          <value>
18             pl.cyfronet.virolab.protos.dre.formatters.
19             BasicQueryResultFormatter
20          </value>
21        </list>
22     </property>
23     <property name=”handlers”>
24        <list>
25          <value>
26          pl.cyfronet.virolab.protos.dre.handlers.SimpleXqueryHandler
27          </value>
28        </list>
29     </property>
30     <property name=”validators”>
31        <list>
32          <value>
33          pl.cyfronet.virolab.protos.dre.validators.
34          XquerySyntaxValidator
35          </value>
36        </list>
37     </property>
38  </bean>
```

Listing A.10. Example DRE Bean's compile-time configuration. Sets up implementation of the IDataGathering interface along with implementation-specific properties.

First line of the listing creates new bean under DreBean name with lifecycle methods (*init-method* and *destroy-method* attributes) bound to the component model ones. Lines 4-31 contains configuration of the DRE properties, as follows:

- Line 2 initializes bean with the DSS instance, which is required as DRE needs access to the distributed storage (property *dss*).
- Lines 5-14 set up configuration provider for the DRE bean. As in previous examples, also this sample uses module **protos-config**
- Lines 15-21 configures result formatters to be used. Value of the *formatters* property should be ordered list containing fully-qualified names of classes implementing the **IQueryResultFormatter** interface. When returning result, DRE will use each applicable formatter, in order as defined in the list. Therefore, ordering of formatters is quite important. In listing A.10 only default, empty formatter is provided.
- In lines 22-28 query handlers are configured. Just as for previous property, also *handlers* takes ordered list. The only difference is that classes passed as values

should implement the **IQueryHandler** interface. Processing behavior is identical. In the example handler for the XQuery language and simple type queries is provided.
- Lines 30-36 initializes the *validators* property. Classes passed as argument should implement the IQueryValidator interface. List is processed in order, although in case of validation this feature does not play important role. In the example listing DRE bean is configured to use XQuery syntax validator only.

Run-time

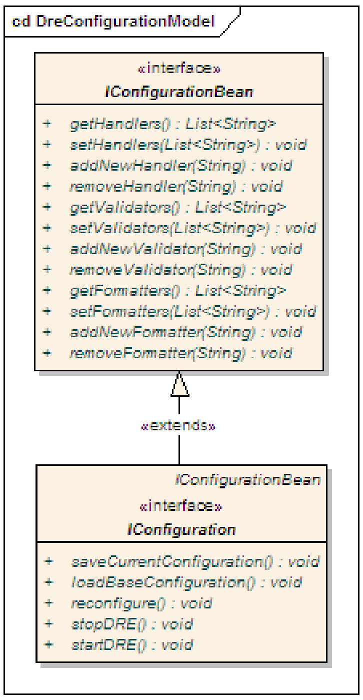

cd DreConfigurationModel
«interface»
IConfigurationBean
+ getHandlers() : List<String>
+ setHandlers(List<String>) : void
+ addNewHandler(String) : void
+ removeHandler(String) : void
+ getValidators() : List<String>
+ setValidators(List<String>) : void
+ addNewValidator(String) : void
+ removeValidator(String) : void
+ getFormatters() : List<String>
+ setFormatters(List<String>) : void
+ addNewFormatter(String) : void
+ removeFormatter(String) : void
«extends»
IConfigurationBean
«interface»
IConfiguration
+ saveCurrentConfiguration() : void
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ stopDRE() : void
+ startDRE() : void

Figure A.5. PROToS DRE component configuration model. Model consists of two interfaces - bean-type and API-type. The latter is common one, shared by all PROToS stateful components.

Fig. A.5 shows configuration model of the DRE component. It contains one **'bean'** interface - namely *IConfigurationBean* and one **'API'** type - *IConfiguration*. The latter is standard API interface as described in a respective section.
Bean interface's methods are as follows:
- *getHandlers* - returns list of fully-qualified class serving as query handlers.
- *setHandlers* - sets query handlers to be used by the DRE component. Argument should be list of fully-qualified class names.
- *addNewHandler* - adds new handler, using fully-qualified class name as argument.
- *removeHandler* - removes handler in use by it's fully-qualified class name.
- *getValidators* - returns validators used by the DRE as list of fully-qualified class names.
- *setValidators* - sets validators to be used. Takes list of fully-qualified class names, available in the run-time.

- *addNewValidator* - adds new validator class to be used.
- *removeValidator* - removes validator that is currently in use from service.
- *getFormatters* - returns list of the formatters in service.
- *setFormatters* - sets list of formatters to be used.
- *addNewFormatter* - adds new formatter to the list of currently available.
- *removeFormatter* - removes formatter from service, using it's fully-qualified class name as key.

As can be seen, run-time configuration properties mirrors some of the compile-time ones. This is because in compile time user can provide default formatters, validators and handlers, which will be available on every time component is started. What is more, these properties are unlikely to change. Still, run-time reconfiguration of for example returned results type is possible.

A.3.5. DGE component

Data Gathering Engine component is responsible for such functionalities as storage of incoming events. Also in case of this component, actual type of events handled depends on configuration, so this should be handled carefully.

Compile-time

This type of configuration of the DGE component should be put in the application-Context file of the **Core application**. File itself resides usually in the **protos-interfaces** module.
Listing A.11 depicts typical configuration of a DRE based bean.

```
<bean id="DgeBean" class="pl.cyfronet.virolab.protos.dge.impl.
                              DataGatheringImplementation"
  init-method="start" destroy-method="stop">
  <property name="dss" ref="DssBean"/>
  <property name="provider">
    <bean class="pl.cyfronet.virolab.protos.dge.impl.
                   XMLConfigurationProvider">
      <property name="beanKey" value="dgeBean"/>
      <property name="classpathResource" value="beansConfig.xml"/>
      <property name="preferredResource" value="CLASS_PATH"/>
      <property name="strategyClassName" value=
     "pl.cyfronet.virolab.protos.config.xstream.OneFileStrategy"/>
    </bean>
  </property>
  <property name="validators">
    <list>
      <value>
        pl.cyfronet.virolab.protos.dge.validators.
        RequiredAnnotatedEventValidator
      </value>
      <value>
        pl.cyfronet.virolab.protos.dge.validators.
        SimpleEventFieldsValidator
```

```
            </value>
        </list>
    </property>
    <property name="handlers">
        <list>
            <value>
                pl.cyfronet.virolab.protos.dge.handlers.
                SimpleAnnotatedEventHandler
            </value>
        </list>
    </property>
</bean>
```

Listing A.11. DGE Bean's compile-time configuration example. Sets up reference implementation of the IDataRetrieval interface along with implementation's specific properties.

In the first line container initializes new bean with *id* DgeBean. Also, lifecycle methods (*init-method* and *destroy-method*) are set to respectively *start* and *stop*, as defined in the stateful component model. Next lines configures following bean properties:

- *dss*
  Initializes DGE bean with an instance of the DSS component. It is required, as DGE needs access to the distributed storage.
- *provider*
  Sets provider to be used for configuration persistence. As usual, the **protos-config** module is used. Also, typical configuration for classpath resource storage is used.
- *validators*
  Configures classes that should be used for validation of incoming events. Argument takes form of list containing fully-qualified names of classes implementing the **IEventValidator** interface. In listing A.11 two validators are configured:
  — **RequiredAnnotatedEventValidator**, that checks if each property annotated as *'Required'* is not null
  — **SimpleEventFieldsValidator**, that checks whether basic event fields are initialized with proper values
- *handlers*
  Configures handlers for incoming event types. Argument passed should be list with fully-qualified class names. Each class should be available in the run-time and implement the **IEventHandler** interface. In the example, one handler is configured - **SimpleAnnotatedEventHandler**.

Run-time

Fig. A.6 shows configuration model for the DGE component. It consists of two interfaces - the *IConfigurationBean* (*bean* type) and the *IConfiguration* (*API* type). First one configures DRE properties as follows:

- *getValidators* - returns list of active event validators.
- *setValidators* - replaces active validators with new list.

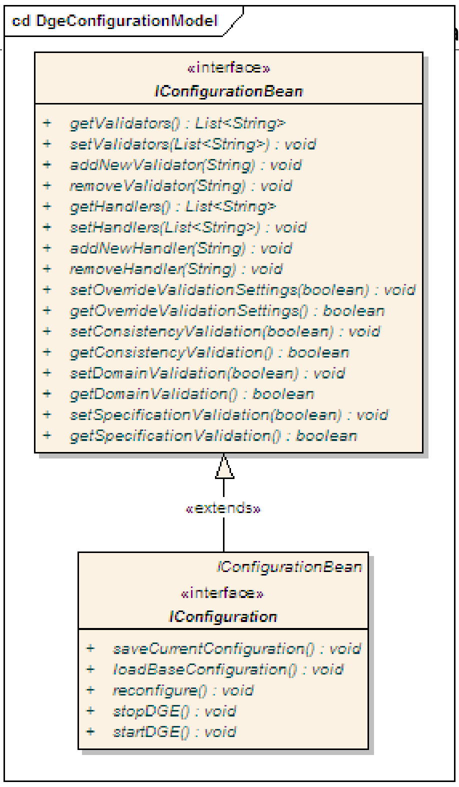

cd DgeConfigurationModel
«interface»
IConfigurationBean
+ getValidators() : List<String>
+ setValidators(List<String>) : void
+ addNewValidator(String) : void
+ removeValidator(String) : void
+ getHandlers() : List<String>
+ setHandlers(List<String>) : void
+ addNewHandler(String) : void
+ removeHandler(String) : void
+ setOverrideValidationSettings(boolean) : void
+ getOverrideValidationSettings() : boolean
+ setConsistencyValidation(boolean) : void
+ getConsistencyValidation() : boolean
+ setDomainValidation(boolean) : void
+ getDomainValidation() : boolean
+ setSpecificationValidation(boolean) : void
+ getSpecificationValidation() : boolean
«extends»
IConfigurationBean
«interface»
IConfiguration
+ saveCurrentConfiguration() : void
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ stopDGE() : void
+ startDGE() : void

Figure A.6. PROToS DGE component configuration model, composed of IConfigurationBean (defining configuration properties) and common ICnfiguration (defining management behaviour).

- *addNewValidator* - adds new validator to the list of active ones.
- *removeValidator* - removes validator from service.
- *getHandlers* - returns list of event handlers in use.
- *setHandlers* - replaces active handlers with new ones.
- *addNewHandler* - puts new handler into service.
- *removeHandler* - removes passed handler from the list of active ones.
- *setOverrideValidationSettings* - if set to true, DRE will override validation settings of incoming events with **general** ones.
- *getOverrideValidationSettings* - checks actual state of validation overriding.
- *setConsistencyValidation* - forces validation of incoming ontology individuals consistency with existing ones.
- *getConsistencyValdiation* - checks state of the consistency validation **general** setting.
- *setDomainValidation* - forces validation of the incoming ontology elements with domain ontology loaded by the system.
- *getDomainValidation* - checks actual state of the domain validation **general** setting.
- *setSpecificationValidation* - forces specification validation of incoming events.
- *getSpecificationValidation* - checks state of the specification validation **general** setting.

Also in case of this component some run-time properties mirror compile-time ones. Behavior is the same as with the DRE component - compile-time settings shall be treated as default ones, available on every start of the component regardless of run-time ones.

A.3.6. SSN component

Storage Super Node is central and therefore most important component of a **node applications**. It manages endpoints (XML databases), so such aspects as data distribution and XML storage details are to be configured here. Careful configuration is highly advised. Bugs in the SSN configuration can possibly render part of the system unusable.

Compile-time

Compile-time configuration of the SSN fits into applicationContext.xml file of a **protos-node** application, residing in the **protos-ssn** module.
Listing A.12 depicts typical configuration of the SSN component.

```
<bean id="SsnBean"
   class="pl.cyfronet.virolab.protos.sn.ssn.impl.StorageSuperNode">
  <property name="provider">
    <bean class="pl.cyfronet.virolab.protos.sn.ssn.impl.
                  XMLConfigurationProvider">
    <property name="beanKey" value="ssnBean"/>
    <property name="classpathResource" value="beansConfig.xml"/>
    <property name="fileSystemResource" value="./beansConfig.xml"/>
    <property name="preferredResource" value="FILE_SYSTEM"/>
    <property name="strategyClassName" value=
    "pl.cyfronet.virolab.protos.config.xstream.OneFileStrategy"/>
    </bean>
  </property>
  <property name="stateProvider">
    <bean class="pl.cyfronet.virolab.protos.pstore.local.
                 LocalSerializableStorage">
      <property name="resources">
        <map>
          <entry>
            <key><value>CLASS_PATH</value></key>
            <value>SsnState.state</value>
          </entry>
          <entry>
            <key><value>FILESYSTEM</value></key>
            <value>./SsnState.state</value>
          </entry>
        </map>
      </property>
      <property name="preferredResource" value="FILESYSTEM"/>
    </bean>
  </property>
  <property name="reconfiguration" >
    <bean class="pl.cyfronet.virolab.protos.sn.ssn.config.
```

```
34                 ReconfigurationContext">
35       <property name="reconfigurator">
36         <bean class=
37           "pl.cyfronet.virolab.protos.sn.ssn.config.basic.
38           BasicReconfigurator">
39           <property name="defaultCollection">
40             <value>provenanceTests</value>
41           </property>
42         </bean>
43       </property>
44     </bean>
45   </property>
46 </bean>
```

Listing A.12. Example compile-time configuration of the SSN Bean coming from the Node application. Configures usage of reference implementation along with required properties. Most important property is reconfiguration algorithm to be used.

Lines 1-2 initializes the bean, giving it an unique *id*. Lifecycle methods do not have to be bound because current SSN implementation directly implements Spring's **Lifecycle** interface.
Following lines configures bean's properties:

- Lines 3-13 configures *provider* property. It is bean managing configuration persistence, derived from the **protos-config** module. Typically new bean is set to use file system resource named beansConfig.xml available in the root directory.
- Lines 14-31 sets *stateProvider* property. Typically used is the **LocalSerializableStorage** class from the module **protos-pstore**. In the example, state provider bean is configured to use file system resource named SsnState.state.
- In lines 32-45, the reconfiguration strategy for SSN bean is initialized. This is the most important property to be configured for SSN. Such aspects as XML databases management and peer revocation are handled by the strategy.
  In the example, the **BasicReconfigurator** strategy class is used.

Run-time

Fig. A.7 depicts run-time configuration model of the SSN component. Apart from standard *API* interface, it consists of one *bean* type interface, namely the *IConfigurationBean*.
This section uses notion of the **peer address**. It is address of a peer (storage endpoint) to be used by the SSN component. It's format is: [type]:[protocol]:[dns name—ip = url]:[port]:[node name]. Supported types:

- **sp** for instances exposing Storage Peer interface
- **exist** for sole eXist XML DB instances

Supported protocols are:

- **rmi** - for peers using RMI protocol
- an empty string for native communication (as in case of the eXist)

An example of well-formed **peer address** is exist::localhost:20080:exist/xmlrpc/db. Bean property methods follow:

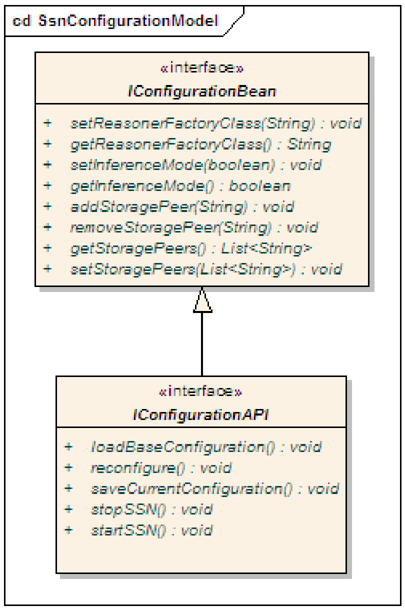

cd SsnConfigurationModel
«interface»
IConfigurationBean
+ setReasonerFactoryClass(String) : void
+ getReasonerFactoryClass() : String
+ setInferenceMode(boolean) : void
+ getInferenceMode() : boolean
+ addStoragePeer(String) : void
+ removeStoragePeer(String) : void
+ getStoragePeers() : List<String>
+ setStoragePeers(List<String>) : void
«interface»
IConfigurationAPI
+ loadBaseConfiguration() : void
+ reconfigure() : void
+ saveCurrentConfiguration() : void
+ stopSSN() : void
+ startSSN() : void

Figure A.7. PROToS SSN component configuration model. Model is composed of IConfigurationBean interface, defining configuration properties of the SSN and common IConfigurationAPI interface.

- *setReasonerFactoryClass* - configures ontology OWL reasoner factory to be used. Takes fully-qualified class name as an argument.
- *getReasonerFactoryClass* - returns current OWL reasoner factory class used.
- *setInferenceMode* - if set to true, SSN component will try to perform ontology reasoning on stored data. Currently this is not recommended, as performance drops dramatically in the inference mode.
- *getInferenceMode* - returns current state of the inference mode flag.
- *addStoragePeer* - adds new storage peer to the current SSN. Argument should be well-formed peer address.
- *removeStoragePeer* - removes storage peer from service, using it's peer address as an argument.
- *getStoragePeers* - returns list containing peer addresses of all storage peers in use.
- *setStoragePeers* - replaces all storage peers with new ones, passing list of peer addresses.

Appendix B

Sample deployment

This chapter presents sample deployment of the PROToS system. It contains complete reference - layout of the components on machines and full configuration.

B.1. Physical component layout

Example deployment involves following PROToS component's configuration:
- one **Core** application
- two **Node** applications
- three **Peer** applications

Those components are deployed on three physical machines:
- **Machine A** at IP address 192.168.1.101
- **Machine B** at IP address 192.168.1.102
- **Machine C** at IP address 192.168.1.103

Fig. B.1 presents how PROToS components are deployed on available physical machines.

Depicted components layout follows philosophy of maximizing resource utilization. This way on **Machine A** there are two components: computational-heavy **Core** and **Peer** using storage resources. The same applies to machines **B** and **C**, where **Node** instances are computational-oriented.
Of course different deployment options are open, but for example installing **Core** and **Node** on the same machine would make idea of distributing computation (see section 5.2 useless.

B.2. Configuration settings

This section presents complete configuration, both run-time and compile-time of the components depicted on the diagram B.1.

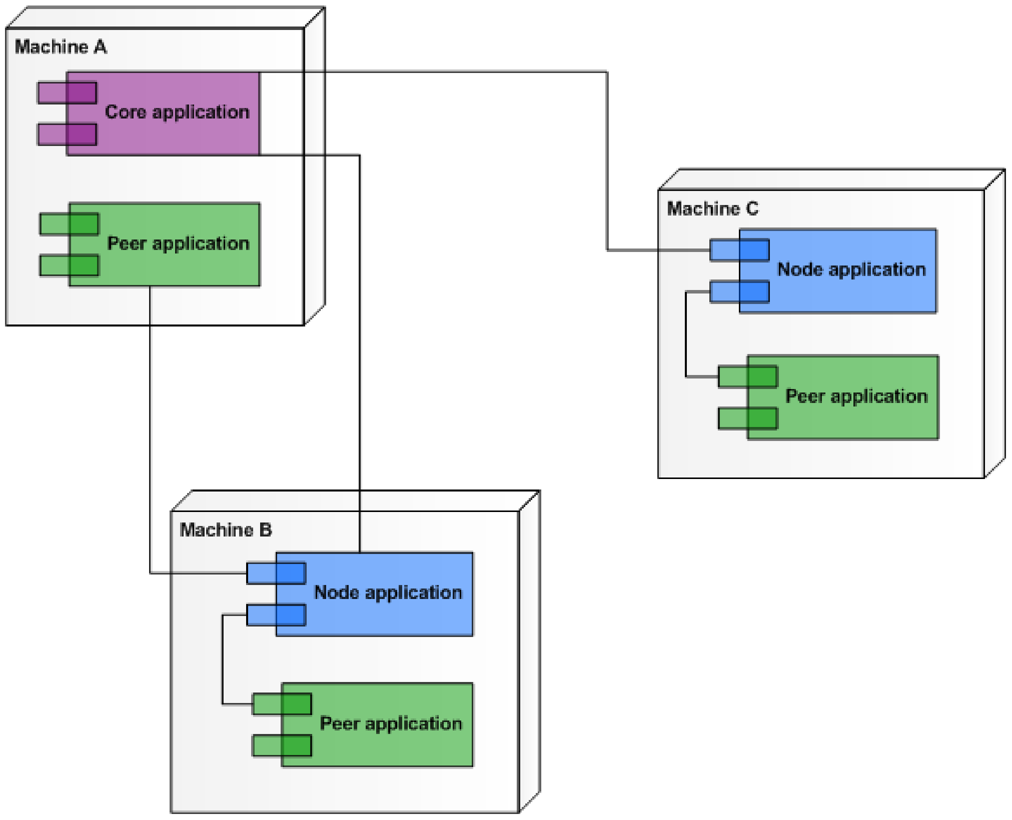


Figure B.1. PROToS example UML deployment diagram. Diagram presents PROToS with distributed storage composed of two Node and three Peer applications. All components are deployed on three physical machines in a way yielding best performance.

For the sake of clarity, compile-time configuration is same for all components of one type. Only run-time configuration differs.

B.2.1. Compile-time configuration

Section presents full listings of component's configurations, with brief description only. Full compile-time configuration reference is to be found in respective sections of the chapter A.

Core application

Listing B.1 presents compile-time configuration of the **Core** application.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

  <import resource=
  "classpath:org/codehaus/xfire/spring/xfire.xml"/>
  <!-- JMX / RMI configuration section -->
  <!-- RMI registry for JMX connector -->
```

```
<bean id="registry" class="org.springframework.remoting.rmi.
                             RmiRegistryFactoryBean">
  <property name="port" value="5099"/>
  <property name="alwaysCreate" value="true"/>
</bean>
<!-- JMX server/connector-->
<bean id="mbeanServer"
  class="org.springframework.jmx.support.MBeanServerFactoryBean">
  <!-- Will create new MBeanServer each time -->
  <property name="locateExistingServerIfPossible" value="false"/>
</bean>
<bean id="serverConnector" class="org.springframework.jmx.
  support.ConnectorServerFactoryBean">
  <property name="serviceUrl"
    value="service:jmx:rmi://127.0.0.1/jndi/
    rmi://127.0.0.1:5099/Configuration"/>
  <property name="server" ref="mbeanServer"/>
</bean>
<!-- BEANS to be exported -->
<bean id="exporter"
  class="org.springframework.jmx.export.MBeanExporter"
  lazy-init="false">
  <property name="beans">
    <map>
      <entry key="protos.config:name=DreBean"
             value-ref="DreBean"/>
      <entry key="protos.config:name=DgeBean"
             value-ref="DgeBean"/>
      <entry key="protos.config:name=DssBean"
             value-ref="DssBean"/>
      <entry key="connector:name=Configuration"
             value-ref="serverConnector"/>
    </map>
  </property>
  <property name="assembler">
    <bean
      class="org.springframework.jmx.export.assembler.
      InterfaceBasedMBeanInfoAssembler">
      <property name="interfaceMappings">
        <props>
          <prop key="protos.config:name=DreBean">
    pl.cyfronet.virolab.protos.dre.interfaces.IConfiguration
          </prop>
          <prop key="protos.config:name=DgeBean">
    pl.cyfronet.virolab.protos.dge.interfaces.IConfiguration
          </prop>
          <prop key="protos.config:name=DssBean">
    pl.cyfronet.virolab.protos.dss.interfaces.IConfigurationAPI
```

```
              </prop>
            </props>
          </property>
        </bean>
      </property>
      <property name="server" ref="mbeanServer"/>
    </bean>
    <!-- Xfire WEBSERVICES section -->
    <bean id="addressingHandler"
      class="org.codehaus.xfire.addressing.AddressingInHandler"/>
    <bean name="DreService"
      class="org.codehaus.xfire.spring.ServiceBean">
      <property name="serviceBean" ref="DreWsCommunicatorBean"/>
      <property name="serviceClass" value=
      "pl.cyfronet.virolab.protos.dre.interfaces.IDataRetrieval"/>
      <property name="name" value="DreService"/>
      <property name="inHandlers">
        <list>
          <ref bean="addressingHandler"/>
        </list>
      </property>
    </bean>
    <bean name="DgeService"
      class="org.codehaus.xfire.spring.ServiceBean">
      <property name="serviceBean" ref="DgeWsCommunicatorBean"/>
      <property name="serviceClass"
        value="pl.cyfronet.virolab.protos.dge.interfaces.
        IDataGathering"/>
      <property name="name" value="DgeService"/>
      <property name="inHandlers">
        <list>
          <ref bean="addressingHandler"/>
        </list>
      </property>
    </bean>
    <!-- Application beans section -->
    <bean id="DreWsCommunicatorBean"
      class="pl.cyfronet.virolab.protos.dre.WSCommunicator">
      <property name="implementation" ref="DreBean"/>
    </bean>
    <bean id="DgeWsCommunicatorBean"
      class="pl.cyfronet.virolab.protos.dge.WSCommunicator">
      <property name="implementation" ref="DgeBean"/>
    </bean>
    <bean id="DreBean" class=
    "pl.cyfronet.virolab.protos.dre.impl.DataRetrievalImplementation"
      init-method="start" destroy-method="stop">
      <property name="dss" ref="DssBean"/>
      <property name="provider">
        <bean class="pl.cyfronet.virolab.protos.dre.impl.
                    XMLConfigurationProvider">
        <property name="beanKey" value="dreBean"/>
        <property name="classpathResource" value="beansConfig.xml"/>
        <property name="preferredResource" value="CLASS_PATH"/>
        <property name="strategyClassName" value="pl.cyfronet.
        virolab.protos.config.xstream.OneFileStrategy"/>
        </bean>
      </property>
      <property name="formatters">
        <list>
          <value>pl.cyfronet.virolab.protos.dre.formatters.
                BasicQueryResultFormatter</value>
        </list>
      </property>
      <property name="handlers">
        <list>
         <value>pl.cyfronet.virolab.protos.dre.handlers.
                SimpleXqueryHandler</value>
        </list>
      </property>
      <property name="validators">
        <list></list>
      </property>
    </bean>
    <bean id="DgeBean" class=
    "pl.cyfronet.virolab.protos.dge.impl.DataGatheringImplementation"
      init-method="start" destroy-method="stop">
      <property name="dss" ref="DssBean"/>
      <property name="provider">
        <bean class="pl.cyfronet.virolab.protos.dge.impl.
        XMLConfigurationProvider">
        <property name="beanKey" value="dgeBean"/>
        <property name="classpathResource" value="beansConfig.xml"/>
        <property name="preferredResource" value="CLASS_PATH"/>
        <property name="strategyClassName" value="pl.cyfronet.
        virolab.protos.config.xstream.OneFileStrategy"/>
        </bean>
      </property>
      <property name="validators">
        <list>
          <value>pl.cyfronet.virolab.protos.dge.validators.
                RequiredAnnotatedEventValidator</value>
          <value>pl.cyfronet.virolab.protos.dge.validators.
                SimpleEventFieldsValidator</value>
        </list>
      </property>
      <property name="handlers">
        <list>
          <value>pl.cyfronet.virolab.protos.dge.handlers.
                SimpleAnnotatedEventHandler</value>
        </list>
      </property>
    </bean>
    <bean id="DssBean"
      class="pl.cyfronet.virolab.protos.dss.impl.StorageSupervisor"
      init-method="start" destroy-method="stop">
      <property name="provider">
        <bean class="pl.cyfronet.virolab.protos.dss.impl.
        XMLConfigurationProvider">
        <property name="beanKey" value="dssBean"/>
        <property name="classpathResource" value="beansConfig.xml"/>
        <property name="preferredResource" value="CLASS_PATH"/>
        <property name="strategyClassName" value="pl.cyfronet.
        virolab.protos.config.xstream.OneFileStrategy"/>
        </bean>
      </property>
      <property name="stateProvider">
        <bean class="pl.cyfronet.virolab.protos.pstore.local.
        LocalSerializableStorage">
          <property name="resources">
            <map>
              <entry>
                <key><value>CLASS_PATH</value></key>
                <value>DssState.xml</value>
              </entry>
            </map>
          </property>
          <property name="preferredResource" value="CLASS_PATH"/>
        </bean>
      </property>
      <property name="nodes">
        <bean
           class="pl.cyfronet.virolab.protos.dss.impl.NodesDataBase"/>
      </property>
      <property name="ontologies">
        <bean class=
        "pl.cyfronet.virolab.protos.onts.impl.OntologyDataBase"/>
      </property>
    </bean>
  </beans>
```

Listing B.1. Example compile-time configuration of the Core application. This configuration is full containing all beans required for application to run. All

implementation used for configuring application are reference ones used for testing purposes.

Configuration shown is pretty standard. Uses reference implementations for all logic components. Details are covered in adequate section of configuration chapter A. What is important from deployment point of view, JMX and RMI servers are configured to listen at port 5099. This is done in lines 12 and 24-25. Full *serviceUrl* as passed to the *serverConnector* shall be used in remote, JMX enabled configuration of the **Core application**.

Node application

Listing B.2 shows compile-time configuration of the **Node** application.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

  <!-- JMX / RMI configuration section -->
  <!-- RMI registry for JMX connector -->
  <bean id="registry" class="org.springframework.remoting.rmi.
                                RmiRegistryFactoryBean">
    <property name="port" value="3099"/>
    <property name="alwaysCreate" value="true"/>
  </bean>
  <!-- JMX server/connector-->
  <bean id="mbeanServer"
    class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <!-- Will not create new MBeanServer each time -->
    <property name="locateExistingServerIfPossible" value="true"/>
  </bean>
  <bean id="serverConnector" class=
  "org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="serviceUrl"
      value="service:jmx:rmi://127.0.0.1/jndi/
      rmi://127.0.0.1:3099/Configuration"/>
    <property name="server" ref="mbeanServer"/>
  </bean>
  <!-- BEANS to be exported -->
  <bean id="JmxExporter" class=
    "org.springframework.jmx.export.MBeanExporter"
    lazy-init="false">
    <property name="beans">
      <map>
        <entry key="protos.config:name=SsnBean"
          value-ref="SsnBean"/>
        <entry key="connector:name=Configuration"
          value-ref="serverConnector"/>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export.assembler.
                     InterfaceBasedMBeanInfoAssembler">
        <property name="interfaceMappings">
          <props>
            <prop key="protos.config:name=SsnBean">
        pl.cyfronet.virolab.protos.sn.interfaces.IConfigurationAPI
              </prop>
          </props>
        </property>
      </bean>
    </property>
    <property name="server" ref="mbeanServer"/>
  </bean>
  <!-- RMI app support section -->
  <bean id="RmiExporter"
class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="serviceName" value="StoragePeer"/>
    <property name="service" ref="SsnBean"/>
    <!-- use same Registry as for JMX -->
    <property name="registry" ref="registry"/>
  </bean>
  <!-- App BEANS section -->
  <bean id="RmiCommunicator"
class="pl.cyfronet.virolab.protos.sn.ssn.RMICommunicator">
    <property name="core">
      <ref bean="SsnBean"/>
    </property>
  </bean>
  <bean id="SsnBean" class="pl.cyfronet.virolab.protos.
                              sn.ssn.impl.StorageSuperNode">
    <property name="provider">
      <bean class=
      "pl.cyfronet.virolab.protos.sn.ssn.impl.
      XMLConfigurationProvider">
      <property name="beanKey" value="ssnBean"/>
      <property name="classpathResource" value="beansConfig.xml"/>
      <property name="fileSystemResource"
        value="./beansConfig.xml"/>
      <property name="preferredResource" value="FILE_SYSTEM"/>
      <property name="strategyClassName" value="pl.cyfronet.
      virolab.protos.config.xstream.OneFileStrategy"/>
      </bean>
    </property>
    <property name="stateProvider">
      <bean class="pl.cyfronet.virolab.protos.pstore.local.
```

```
85                         LocalSerializableStorage">
86          <property name="resources">
87            <map>
88              <entry>
89                <key><value>CLASS_PATH</value></key>
90                <value>SsnState.state</value>
91              </entry>
92              <entry>
93                <key><value>FILESYSTEM</value></key>
94                <value>./SsnState.state</value>
95              </entry>
96            </map>
97          </property>
98          <property name="preferredResource" value="FILESYSTEM"/>
99        </bean>
100     </property>
101     <property name="reconfiguration" >
102       <bean class="pl.cyfronet.virolab.protos.sn.ssn.config.
103        ReconfigurationContext">
104         <property name="reconfigurator">
105           <bean class=
106             "pl.cyfronet.virolab.protos.sn.ssn.config.basic.
107               BasicReconfigurator">
108             <property name="defaultCollection">
109               <value>provenanceTests</value>
110             </property>
111           </bean>
112         </property>
113       </bean>
114     </property>
115   </bean>
116 </beans>
```

Listing B.2. Node's example compile-time configuration. Contains all beans necessary for running full-blown Node application. All implementations used are reference ones designed for testing usage.

Also in the case of **Node** compile-time configuration it is straight-forward, with reference implementation used. RMI and JMX servers are configured to serve on port 3099 (lines 11 and 23-24). Thus, parallel deployment of **Node** and **Core** on one machine is possible. Also, in line 56 name of the service as exported by RMI is configured. This will later be used for **Core** configuration.

B.2.2. Run-time configuration

Section presents full run-time configuration of PROToS components. Description provided is brief, as full reference can be found in respective section A.

Core at machine A

As presented on the listing B.1, run-time configuration of the **Core** application is to be placed in the *beanConfig.xml* file found in application's class path.
Listing B.3 presents run-time configuration of the **Core** running at machine A.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <beans>
 3   <bean name="dssBean">
 4     <pl.cyfronet.virolab.protos.dss.impl.ConfigurationBean>
 5       <reasonerFactoryClass>
 6       com.hp.hpl.jena.reasoner.rulesys.OWLMicroReasonerFactory
 7       </reasonerFactoryClass>
 8       <allModelsUrls>
 9         <string>
10         http://virolab.cyfronet.pl/onto/vlom-upper.owl
11         </string>
12         <string>
13         http://virolab.cyfronet.pl/onto/vlom-geno2drs-protos.owl
14         </string>
15         <string>
16         http://virolab.cyfronet.pl/onto/vlom-drs-protos.owl
17         </string>
18       </allModelsUrls>
19       <experimentModelUrl>
20       http://virolab.cyfronet.pl/onto/vlom-exp-protos.owl
21       </experimentModelUrl>
22       <allDataModelsUrls>
23         <string>
24         http://virolab.cyfronet.pl/onto/vlom-data.owl
25         </string>
26         <string>
27         http://virolab.cyfronet.pl/onto/vlom-data-protos.owl
28         </string>
29       </allDataModelsUrls>
30       <storageNodesAddresses>
31         <string>rmi:192.168.1.102:3099:StoragePeer</string>
32         <string>rmi:192.168.1.103:3099:StoragePeer</string>
33       </storageNodesAddresses>
34     </pl.cyfronet.virolab.protos.dss.impl.ConfigurationBean>
35   </bean>
36 </beans>
```

Listing B.3. Run-time configuration of the Core application running on machine A. Apart from ontology models to be used contains also addresses of Nodes constituting distributed storage of the PROToS.

In lines 5-29 ontology properties are configured. For convenience, presented configuration includes current experiment, data and domain models developed for the ViroLab. Lines 31-32 are most important from deployment point of view. Those lines set up **Nodes** to be used in the PROToS distributed storage. As shown, **Core** configuration is using service URLs as set in **Node** compile-time config.

Node at machine B

As presented on nth listing B.2, run-time configuration of the **Node** is to be stored in the *beansConfig.xml* file from application's folder in the file system.
Listing B.4 presents contents of the **Node** configuration file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans>
3
4    <bean name="ssnBean">
5      <pl.cyfronet.virolab.protos.sn.ssn.config.ConfigurationBean>
6        <storagePeersAddresses>
7          <string>exist::192.168.1.101:20080:exist/xmlrpc/db</string>
8          <string>exist::192.168.1.102:20080:exist/xmlrpc/db</string>
9        </storagePeersAddresses>
10       <allModelsUrls>
11         <string>
12          http://virolab.cyfronet.pl/onto/vlom-drs-protos.owl
13         </string>
14         <string>
15          http://virolab.cyfronet.pl/onto/vlom-exp-protos.owl
16         </string>
17         <string>
18          http://virolab.cyfronet.pl/onto/vlom-data.owl
19         </string>
20         <string>
21          http://virolab.cyfronet.pl/onto/vlom-data-protos.owl
22         </string>
23       </allModelsUrls>
24       <supportedUris>
25         <string>http://www.virolab.org/onto</string>
26       </supportedUris>
27       <inferenceMode>false</inferenceMode>
28     </pl.cyfronet.virolab.protos.sn.ssn.config.ConfigurationBean>
29   </bean>
30 </beans>
```

Listing B.4. Run-time configuration of the Node application running on the machine B. Contains addresses of the Peers running on machines A and B. Also domain models assigned to this Node are configured.

Lines 7-8 configures peers part of the **Node**. Passed addresses contains all information needed by Node's logic - type of peer (pure eXist in this case), transport protocol (exist's native), URL (machines B and A) and exact service (XML:RPC managed db). In lines 10-23 ontology model designated for this **Node** are configured. What should be noted, only one *domain* model in assigned to this node - the *vlom-drs-protos*. Experiment and data models are also present.

Node at machine C

As in the previous example, this **Node** also uses *beansConfig.xml* file for storing run-time configuration. Listing B.5 presents contents of this file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans>
3
4    <bean name="ssnBean">
5      <pl.cyfronet.virolab.protos.sn.ssn.config.ConfigurationBean>
6        <storagePeersAddresses>
7          <string>exist::192.168.1.103:20080:exist/xmlrpc/db</string>
8        </storagePeersAddresses>
9        <allModelsUrls>
10         <string>
11          http://virolab.cyfronet.pl/onto/vlom-exp-protos.owl
12         </string>
13         <string>
14          http://virolab.cyfronet.pl/onto/vlom-geno2drs-protos.owl
15         </string>
16         <string>
17          http://virolab.cyfronet.pl/onto/vlom-data.owl
18         </string>
19         <string>
20          http://virolab.cyfronet.pl/onto/vlom-data-protos.owl
21         </string>
22       </allModelsUrls>
23       <supportedUris>
24         <string>http://www.virolab.org/onto</string>
25       </supportedUris>
26       <inferenceMode>false</inferenceMode>
27     </pl.cyfronet.virolab.protos.sn.ssn.config.ConfigurationBean>
28   </bean>
29 </beans>
```

Listing B.5. Run-time configuration of the Node application running on the machine C. Contains configuration of domain models and Peer designated for this Node.

In line 7 one **Peer** is configured. Lines 9-22 configures ontology models used. Again, only one *domain* model is assigned, but this time it is the *vlom-geno2drs-protos*. This way, all domain models configured for the PROToS in listing B.3 are assigned to adequate **Nodes**.

Peer

In the example, **Peer** component in implemented by the **protos-sp-standalone** module. It wraps pure eXist instance, so service URLs used in previous listings begins with *exist* type prefix.
Configuration of this type **Peer** is only compile-time, residing typically in file *jetty.xml*. It is presented on listing B.6.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <WrapperConfig>
3   <Port>20080</Port>
4   <Port>8091</Port>
5   <WarPath>classpath:exist.war</WarPath>
6   <ContextPath>/exist</ContextPath>
7 </WrapperConfig>
```

Listing B.6. Compile-time configuration of the Peer application based on the sp-standalone implementation.

Lines 3-4 configures ports, where server will be listening for incoming connection. First port - 20080 - is reckoned as primary and second one - 8091 - as backup. All **Node** instances are configured to use primary **Peer** ports. Configuration allows for adding as many ports as needed. Next line (5) selects eXist [9] instance to be run by this **Peer**. In line 6 last important property is configured - context path where eXist will be served. As presented on the listing B.5, context path is part of the service name.

Appendix C

Administrator manual

Following manuals constitute guide for system's Administrators, interested in using PROToS provenance tracking system. Topics covered by manuals are:

- **Prerequisites for PROToS** - section C.1
- **Obtaining PROToS** - section C.2
- **Building PROToS** - section C.3
- **Installing PROToS** - section C.4
- **Configuring PROToS** - section C.5

Some aspects are covered in greater details in other sections. In those cases, references to applicable pages are present.

C.1. Environment prerequisites

Successful build and installation of the PROToS system requires several other components to be present in the target system. They are as follows:

- for the **PROToS Core** application
  - Java Servlet 2.5 compliant container (only one is necessary), such as:
    - Tomcat 6.0.14 - [5]
    - Jetty 6 (lightweight solution) - [19]
- for all components and applications:
  - Java JDK 6.0 - [16]
  - Apache Maven 2.0.7 - [23]
  - Apache Ant 1.7.0 - [1]
- environmental variables
  - **M2_HOME** - Maven 2 installation directory
  - **M2_REPO** - Maven 2 local artifact repository
  - alter PATH variable to include Maven 2 and Ant binaries

Installation instruction for external software components could be found in respective manual pages, usually shipped with the software or available on the web.

C.2. Obtaining: package and source

C.2.1. Binary package

Binary distributions of the PROToS can be fetched from address: http://virolab.-cyfronet.pl/trac/protos/downloads. Each release archive contains following directories:

- Core, with release WAR archive representing **Core** application
- SSN, containing runnable **SSN** JAR archive, run-time files (state persistence file and configuration) and ANT run script
- SP, containing runnable **SP-stand alone** JAR along with ANT run script

C.2.2. Source package

PROToS system source code is managed by the Cyfronet's GForge server [7].
To obtain full sources, one should download it from the server. Currently, only registered users of the Cyfronet's Gforge are able to do this, by using PROToS SVN repository. Using command line, one should enter following:

```
svn checkout --username developername
https://gforge.cyfronet.pl/svn/protosPROToS/trunk/
```

Listing C.1. PROToS SVN checkout

C.3. Building from source

C.3.1. With ANT

PROToS is bundled with set of ANT scripts, helpful in building, deployment and running of system components. This section shows how to build PROToS with usage of those scripts.
Following steps are required:

1. Download current sources of the PROToS.
2. Navigate to the *download folder*/protos-core/protos-interfaces and run following commands:

```
ant clean-all
ant build-all
```

Listing C.2. PROToS build all

This will clean build whole project and build WAR archive of the PROToS Core application.

3. Navigate to the *download folder*/protos-core/protos-ssn folder and run command:

```
ant build
```

Listing C.3. PROToS SSN build

Above command will build runnable JAR of the Node application.

4. Navigate to the download folder/protos-core/protos-sp/protos-sp-standalone folder and run command:

```
ant build
```

Listing C.4. PROToS SP build

This will build runnable JAR of the Peer application, implemented by the SP-stand alone PROToS component.

C.3.2. Without ANT

Following instruction shows how to build / install PROToS components without using ANT scripts.

1. Download current sources of the PROToS.
2. Navigate to the *download folder*/protos-core and run following commands:

```
mvn clean
mvn install
```

Listing C.5. PROToS build

Above command should have built whole project class code. Now we could move to building final archives of the PROToS components.

3. Navigate to the *download folder*/protos-core/protos-interfaces and run following command:

```
mvn install war:war
```

Listing C.6. PROToS Interfaces build

This should have built the PROToS core - WAR archive in the *download folder*-/protos-core/protos-interfaces/target folder. It is named protos-interfaces-current version.war.

4. PROToS Storage Super Node is located in the download folder/protos-core/protos-ssn. Because this component doesn't need servlet container to run, it is distributed as standard Java archive - JAR. It could be found in the target sub-folder, named protos-ssn-current version.jar. Because of the library compatibility issues it is strongly advised to use unpacked form with Java CLASS_PATH set to a folder containing all necessary JARs. Startup class is **Bootstrap** from package **pl.cyfronet.-virolab.protos.sn.ssn.bootstrap**.

Installation tips:

- Maven option *-Dmaven.test.skip=true* could speed-up building process a bit.
- Another option is -*o*, which causes Maven to run in off-line mode is also useful.

C.4. Installation instructions

C.4.1. From binaries with ANT support

Binaries with PROToS components contain also convenience ANT running scripts. Following instructions apply to runnable JAR versions fetched as binaries.

- PROToS Core application
  1. Navigate to the folder with WAR archive
  2. Copy archive to the suitable servlet container's application directory
  3. Restart your servlet container (Tomcat users: shutdown and then startup).
- PROToS SSN application
  1. Navigate to the folder where SSN content was copied
  2. Enter command:

```
ant run
```

Listing C.7. PROToS SSN run

- PROToS SP application
  1. Navigate to the folder where SP content was copied
  2. Enter command:

```
ant run
```

Listing C.8. PROToS SP run

C.4.2. From source with ANT support

Bundled ANT scripts could also help in running PROToS components. Following instructions apply to runnable JAR versions build from sources.

- PROToS Core application
  1. Navigate to the *download folder*/protos-core/protos-interfaces folder
  2. Enter command:

```
ant deploy
```

Listing C.9. PROToS Core deploy

  3. Afterwards restart your servlet container (Tomcat users: shutdown and then startup).
- PROToS SSN application
  1. Navigate to the *download folder*/protos-core/protos-ssn folder
  2. Enter command:

```
ant run
```

Listing C.10. PROToS SSN deploy

Above command will copy all necessary resources and start runnable JAR containing SSN logic.

- PROTOS SP application
  1. Navigate to the download folder/protos-core/protos-sp/protos-sp-standalone folder
  2. Enter command:

```
ant run
```

Listing C.11. PROToS SP deploy

This will start SP instance using runnable JAR.

C.4.3. For source without ANT support

Following instructions apply to non-runnable JARs, build from sources.

- PROToS Core
  1. Copy newly built WAR archive to the webapp folder of machine's servlet container - in the case of the Tomcat it would be webapps folder.
  2. Restart your server (Tomcat users: shutdown and then startup). After restart PROToS core should be up and running.
- PROToS Storage Super Node
  1. Navigate to the folder with unpacked SSN. In case of the default, build folder it would be the download folder/protos-core/protos-ssn/target
  2. Run command:

```
java -cp {folder with dependency jars}
pl.cyfronet.virolab.protos.sn.ssn.bootstrap.Bootstrap
```

Listing C.12. PROToS SSN run

SSN should be up and running.
- PROToS Storage Peer
  — SP-standalone component
  1. Navigate to the download folder/protos-core/protos-sp/protos-sp-standalone folder
  2. Enter command:

```
java -cp {SP jar}
pl.cyfronet.virolab.protos.sn.sp.simple.ExistWrapper
```

Listing C.13. PROToS SP run

SP should be up and running.

  — as stated in the architecture manual role of the SP could be fulfilled by any software implementing the XML:DB interface, such as eXist [9]. Instructions how to run chosen software could be found in its manual.

C.5. Sample configuration

Given distributed nature of the PROToS and it's complexity, configuration is very important and not easy task. Deep thorough guide for configuring running instances of PROToS components is available in section A.

For quick start up with PROToS, full exemplary configuration of PROToS instance can be found in section B.2 and especially section's XML listings.

Appendix D

User manual

PROToS entry-point application - **the Core** presents its external interfaces - *IDataGatering* and *IDataRetrieval* as simple stateless Web Services. Therefore, access to those services is possible with any WS client. Because current PROToS services are built with XFire stack [47], all examples are based of the XFire client API. Additional advantage coming from XFire usage is simplicity and as few technical details (SOAP, WSDL) as possible.
Following external information is needed for provided test clients:

- **application URL** - in the example PROToS is deployed under *protos-interfaces-1.0.0* name on *localhost's* port *20090*
- **Interfaces definitions** - coming from adequate packages and available in Maven artifacts.

D.1. Storing data

Listing D.1 presents example WS client for the PROToS gathering interface.

```
package pl.cyfronet.virolab.protos.example;

import java.util.Arrays;

import org.apache.log4j.Logger;
import org.codehaus.xfire.client.XFireProxyFactory;
import org.codehaus.xfire.service.Service;
import org.codehaus.xfire.service.binding.ObjectServiceFactory;

import pl.cyfronet.virolab.protos.common.Event;
import pl.cyfronet.virolab.protos.common.events.SimpleEvent;
import pl.cyfronet.virolab.protos.dge.interfaces.IDataGathering;

```

```
14 /**
15  * Test client for PROToS WS gathering interface.
16  *
17  * @author qba
18  *
19  */
20 public class DgeWsTest {
21 
22   private static final Logger log =
23     Logger.getLogger(DgeWsTest.class);
24 
25   private static String service =
26   "http://localhost:20090/protos-interfaces-1.0.0/DgeService";
27 
28   public static void main(String[] args)
29     throws Exception {
30 
31     // create WS client
32     Service serviceModel = new
33     ObjectServiceFactory().create(IDataGathering.class);
34     IDataGathering dge = (IDataGathering)new
35     XFireProxyFactory().create(serviceModel, service);
36     Class clazz =
37     cyfronet.virolab.geno2drs.NucleotideSequenceSubtyping.class;
38 
39     // prepare example data
40     Event event = new SimpleEvent();
41     event.getOntologyComponents().add(
42     "http://www.virolab.org/onto/
43     geno2drs-protos/NucleotideSequenceSubtyping");
44     event.setEventClass(clazz.getCanonicalName());
45     event.getDatatypeProperties().put("ID", Arrays.asList(
46     new String[] {"http://individual.id"}));
47 
48     // and execute
49     dge.storeEvent(event);
50     log.info("Event sent without problems.");
51     // no exception shall be thrown
52   }
53 }
```

Listing D.1. Example PROToS data storage client. It uses WebService interface and is based on XFire framework. Stores event from the geno2drs domain ontology.

Client starts by initializing standard, XFire WS client, created from Java interface definition of the *IDataGathering* (lines 31- 34 of the listing). Next, in lines 35-36 actual class of the PROToS event is taken. In lines 39-44 new event is instantiated and initialized. Example use convenience class - *SimpleEvent*, with attributes collections and options already set. Line 47 executes call on remote PROToS interface, finishing the data storage sequence.

D.2. Retrieving data

Listing D.2 presents example WS client used for retrieving data from the PROToS system.

```java
package pl.cyfronet.virolab.protos.example;

import org.apache.log4j.Logger;
import org.codehaus.xfire.client.XFireProxyFactory;
import org.codehaus.xfire.service.Service;
import org.codehaus.xfire.service.binding.ObjectServiceFactory;

import pl.cyfronet.virolab.protos.common.Query;
import pl.cyfronet.virolab.protos.common.QueryResult;
import pl.cyfronet.virolab.protos.common.queries.SimpleXQuery;
import pl.cyfronet.virolab.protos.dre.interfaces.IDataRetrieval;

/**
 * Test client for PROToS WS retrieval interface.
 *
 * @author qba
 *
 */
public class DreWsTest {

  private static final Logger log =
    Logger.getLogger(DreWsTest.class);
  private static String service = "http://virolab.cyfronet.pl"+
    ":20090/protos-interfaces-1.0.0/DreService";

  private static String query =
    "declare namespace j.0=\"http://www.virolab.org/onto/test1/\"+
    ";declare namespace rdf=\"http://www.w3.org/1999/02/22-rdf-"+
    "syntax-ns#\";for $rdf in //rdf:RDF[j.0:test1]"+
    "return $rdf";

  public static void main(String[] args)
    throws Exception {

    // prepare client
    Service serviceModel = new
    ObjectServiceFactory().create(IDataRetrieval.class);
    IDataRetrieval dre = (IDataRetrieval)new
    XFireProxyFactory().create(serviceModel, service);

    // prepare query
42      Query query = new SimpleXQuery();
43      query.setQuery(DreWsTest.query);
44
45      // execute and retrieve result
46      QueryResult result = dre.executeQuery(query);
47      log.info("Got response: ");
48      log.info(result.getHolder().getResult());
49    }
50  }
```

Listing D.2. Example PROToS data retrieval client. Makes use of WebService interfaces and is based on XFire framework. Query send is of type simple and written in XQuery language.

In lines 26-30 of the listing D.2 XQuery string is declared. It is simple query retrieving all individuals named *test1* from adequate namespace. Client execution code starts in lines 36-39, where WS endpoint is instantiated and initialized. Next, in lines 42-43 query holder is prepared with predefined XQuery string. Example uses convenience *SimpleXQuery* class, defining language and type used. In line 46 execution of remote method commences. Finally, lines 47-48 logs obtained query result, extracted from *QueryResult* instance.

List of Figures

Listings

List of Tables

Publications

1. Balis B., Bubak M.,Pelczar M., Wach J.:Provenance querying for end-users: A drug resistance case study. Bubak M.,Albada G.D. van, Dongarra J., Sloot P.M.A., editors, ICCS(3),vol 5103 of Lecture Notes in Computer Science, pages 80-89. Springer,2008.
2. Balis B., Bubak M., Pelczar M., Wach J.: Provenance tracking and quering in ViroLab. Cracow'07 Grid Workshop, pages 71-76. ACC Cyfronet AGH, 2008.
3. Balis B., Bubak M., Pelczar M., Wach J.: Provenance tracking and querying in the ViroLab virtual laboratory. CCGRID, pages 675-680. IEEE Computer Society, 2008.
4. Balis B., Bubak M., Wach J.: User-oriented querying over repositories of data and provenance. E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing, pages 187-194, Washington, DC, USA, 2007. IEEE Computer Society.
5. Balis B., Bubak M., Wach J.:P rovenance tracking in the ViroLab virtual laboratory. Lecture Notes i nComputer Science, Parallel Processing and Applied Mathematics: 7th International Conference, PPAM2007, pages 50-60. Springer,2008.