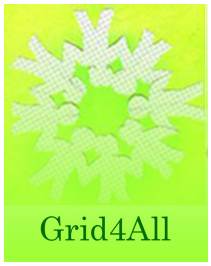


Project no. 034567

Grid4All

Specific Targeted Research Project (STREP)

Thematic Priority 2: Information Society Technologies

D4.8 User Manual

Due date of deliverable: 01-06-2009

Actual submission date: 20-07-2009

Start date of project: 1 June 2006 Duration: 36 months

Contributors : Antares, ICCS, INRIA, FT, KTH, SICS, UPC, UPRC

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Table of Contents

Abbreviations used in this document

Abbreviation / acronym	Description
ACF	Action-Constrain Framework
ADL	Architectural Description Language
AFS	Andrew File System
API	Application Programming Interface
CAS	Combinatorial Auction Service
CFS	Collaborative File Sharing
CIFS	Common Internet File System
CNSE	Collaborative Network Simulation Environment
DAV	Distributed Authoring and Versioning
DHT	Distributed Hash Table
FUSE	Filesystem in User SpacE
GPL	GNU General Public License
GSI	Globus's Grid Security Infrastructure
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
JDK	Java Development Kit
JVM	Java Virtual Machine
LGPL	GNU Lesser General Public License
MIS	Market Information System
NFS	Network File System
OASIS	Organization for Advancement of Structured Information Standards
OWL	Web Ontology Language
P2P	Peer-to-Peer
PIP	Policy Information Points
PAP	Policy Administration Points
PDP	Policy Decision Points
PEP	Policy Enforcement Points
PERMIS	PrivilEge and Role Management Infrastructure Standards
POSIX	Portable Operating System Interface for Unix
SDK	Software Development Kit
SIS	Semantic Information Service
SOA	Service-Oriented Architecture
URL	Universal Resource Locator
VLC	VideoLAN Client
VO	Virtual Organisation
VOFS	VO oriented File System
WebDAV	DAV
WSDL	Web Service Description Language
XACML	eXtensible Access Control Markup Language
YACS	Yet Another Computing Service
YASS	Yet Another Storage Service

Grid4All list of participants

Role	*Participant N°*	*Participant name*	*Participant short name*	*Country*
CO	*1*	*France Telecom*	*FT*	*FR*
CR	*2*	*Institut National de Recherche en Informatique en Automatique*	*INRIA*	*FR*
CR	*3*	*The Royal Institute of technology*	*KTH*	*SWE*
CR	*4*	*Swedish Institute of Computer Science*	*SICS*	*SWE*
CR	*5*	*Institute of Communication and Computer Systems*	*ICCS*	*GR*
CR	*6*	*University of Piraeus Research Center*	*UPRC*	*GR*
CR	*7*	*Universitat Politècnica de Catalunya*	*UPC*	*ES*
CR	*8*	*ANTARES Produccion & Distribution S.L.*	*ANTARES*	*ES*

1 Executive Summary

This document is part of the research project Grid4All (IST-FP6-034567). The document reports on the main software results of Grid4all, serving as a guide to how the Grid4All results can be used by the target stakeholders. It includes a summary table for each result and the user manuals appear as annexes.

2 Introduction

This document is part of the research project Grid4All (IST-FP6-034567). The document reports on the main software results of Grid4all, serving as a guide to how the Grid4All results can be used by the target stakeholders. It includes a summary table for each result and the user manuals appear as annexes.

Project results are grouped in terms of the architectural elements following the architecture of the Grid4All middleware in figure 1.

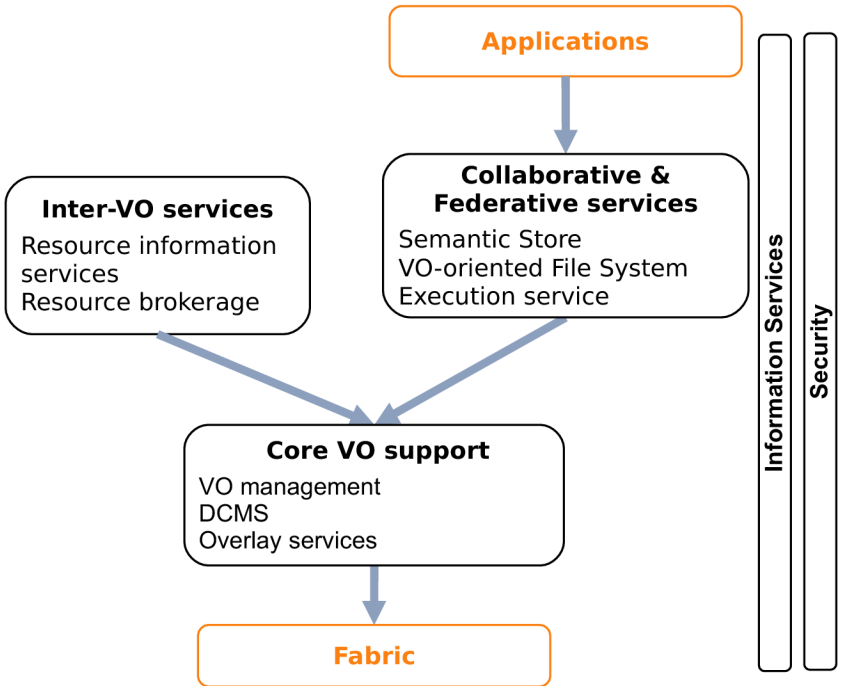


Figure 1: Grid4all architecture.

Core VO services:

- Connectivity (Overlay) and deployment, execution, self-management: Niche, a *Distributed Component Management System* (DCMS).

Inter-VO services:

- Matching service requests: the *Semantic Information System* (SIS)
- Brokering resources in markets: the *Market Information System* (MIS)

Collaborative & federative services:

- Telex, a Principled System Support for Write-Sharing in Collaborative Applications.

- Virtual Organization File System (VOFS), a POSIX-like peer-to-peer file system that adds to traditional file systems the capabilities to federate, share files and storage for collaboration.

- WebDAV: a VO-aware file system based on the WebDAV protocol.

- Niche has been used to develop the following two self-managing services with self-healing and self-configuration capabilities:

    - *Yet Another Storage Service* (YASS), which is a distributed storage system used to store and retrieve files on a network of computers;

    - *Yet Another Computing Service* (YACS) a distributed computing system used to submit and execute jobs containing independent tasks (a bag of tasks).

End-user oriented applications:

- Collaborative Network Simulation Environment (CNSE)

- Collaborative File Sharing (CFS)

- eMeeting, an on-line Multimedia Collaborative Environment

- Sakura, a Shared Calendar

For each result, the description is structured according to a table with the following items:

> Name of the result; What it does; Where it can be obtained; Targeted stakeholders; Scope, "best use" scenarios; Distinguishing features and benefits; Environmental conditions and requirements; Recommendations for deployment (including prerequisites); Experience/lessons learned; Comparison with state of the art.

3 Core VO services

3.1 Niche, a Distributed Component Management System

Table for *Niche* distributed component management system

Name of result: *Niche,* a Distributed Component Management System (DCMS)

What it does (summary):

Niche is a Distributed Component Management System (DCMS) which is used to develop, to deploy and to execute self-managing distributed component-based applications on a structured overlay network of computers. Niche includes (1) a component-based programming model with a set of APIs for the development of self-managing distributed applications; (2) a run-time execution environment for the deployment and execution of the applications together with the appropriate encoded management elements. Niche supports sensing changes in the state of components and environment, and allows individual components to be found and appropriately manipulated. It deploys both functional and management components and sets up the appropriate sensor and actuation support infrastructure.

Where it can be obtained:

http://niche.sics.se/

Targeted stakeholders:

Developers: Niche is a general-purpose system for developers that require that the applications they develop be self-managing. Niche can be used to develop, deploy and provide robust and scalable self-managing services with self-configuration, self-healing and self-tuning capabilities, on a network of computers donated by end-users or/and service service/resource providers.

Scope, “best use” scenarios:

As Niche makes use of structured overlay infrastructure, it has, in particular, good scalability, and is tolerant to churn (high rates of nodes joining, leaving and failing). Examples of services that can be developed using Niche are either general storage and computing services of a Grid environment similar to the storage and computing elements of gLite, or storage and compute clouds, similar to Amazon S3 and ES2, or more specific services such as a parameter-sweep simulation environment, a movie transcoding service (gMovie).

Distinguishing features and benefits:

Niche is a full-fledged programming system for self-managing applications/services in dynamic environments. Niche is used both as a development environment and as an execution run-time

environment. It separates functional and non-functional (management) parts of the application.

The Niche development environment includes a set of APIs for development of functional and management parts of a distributed application. Functional components, component groups and bindings, are first-class entities in Niche that can be monitored and manipulated by the management components (via sensors and actuators) using an extensible monitoring and actuation API provided by Niche. The management part is organized as a network of Management Elements interacting through events. Niche provides basic classes of Management elements, event and actuation APIs. All entities, e.g. components, groups, bindings, and management elements are uniquely identified in Niche; this allows finding and controlling them.

The Niche run-time environment is a set of containers that reside on a structured overlay network of VO computers, and a set of the overlay services (resource discovery, deployment, publish/subscribe, metadata DHTs) provided on each of the containers.

The Niche run-time environment allows initial deployment of a service or an application. The initial deployment code can be either manually written by the developer, or generated by Niche from an ADL (Architecture Description Language) description of the application architecture. The ADL compiler for describing initial configurations of Niche applications is made available together with Niche. In principle, most self-* behaviors could be described in high-level (and often declarative) languages, and compiled to Niche as is done with the ADL, but this is not currently available.

Environmental conditions/requirements:

Niche requires a number of networked computers (depending on the application) to execute. The only restriction is that the computers should be fully connected with real IP addresses. Niche currently does not support Network Address Translation (NAT). For development and testing, Niche can be executed on a single PC.

Recommendations for deployment (including prerequisites):

Niche requires Java SE 6 JDK to execute. Apache Ant (1.7 or later) is needed for reusing build scripts that come with Niche. Any IDE such as Eclipse can be used for development of applications using Niche.

Experience/lessons learned:

Niche has been used to develop the following two self-managing services with self-healing and self-configuration capabilities: (1) YASS: Yet Another Storage Service, which is a distributed storage system used to store and retrieve files on a network of computers; (2) YACS: Yet Another Computing Service, which is a distributed computing system used to submit and execute jobs containing independent tasks (a bag of tasks). These two services are typical services of a Grid environment, e.g. gLite storage and computing elements. The services can be deployed and provided on computers donated by users of the service or by a service provider. The services can operate even if computers join, leave or fail at any time. Each of the services has self-healing and

self-configuration capabilities and can execute on a dynamic overlay network.

When designing and developing the self-managing service YACS using Niche, the developers have experienced that separating functional and self-management parts simplifies design by making responsibilities clearer. It also improves quality of the implementation as individual components and respective source code are focused on one main role, i.e. either functional or self-management functionality. The alternative of mixing those roles would make the logic more complex and harder to program, thereby increasing chances of faulty code. The Niche model of management elements – sensors, watchers, aggregators and managers – is in line with the control loop pattern suggested for self-management. Another positive feature mentioned by the developers is the group abstraction introduced in the extended Fractal model supported by Niche. It allows the developer to masks changes resulting from node failures leaves or joins, which simplifies programming considerably.

In overall, a middleware, such as Niche, clearly reduces burden from an application developer because it enables and supports self-management by leveraging self-organizing properties of structured P2P overlays and by providing useful overlay services such as deployment and name-based communication. However it comes at a cost of self-management overhead, in particular, the cost of monitoring and replication of management; though this cost is necessary for the democratic grid (or cloud) that operates on a dynamic environment and requires self-management. This opens new opportunities for research on efficient monitoring and information gathering/aggregating infrastructures to reduce this overhead. Another research focus is on high-level programming abstractions and a language support that should facilitate development of self-managing applications.

Comparison with state of the art:

Niche is an environment for autonomic computing. There is big interest in autonomic computing in academia and industry; and there exist different approaches and solutions to enabling and achieving self-management in distributed environments and applications. However most of the solutions are application specific.

There also exit a number of component-based environments (e.g. the proActive implementation of the Fractal component model; CoreGRID component model) that allow development of component-based applications and services.

Niche advances the state of art in component-based and autonomic-computing environments as follows. Niche includes the novel use of overlays, where an adapted structured overlay provides a network-transparent sensing and actuation infrastructure that enables and simplifies self-management. The use of overlays makes the system, scalable, tolerant to churn, and provides the framework for assigning (and reassigning upon churn) responsibilities to nodes in the system. For instance, a sensor reports an event such a component failure or overload, to the responsible node(s) for hosting the management element(s) that earlier subscribed to this type of event. We have also adapted known distributed algorithms to dynamic groups. Replication of management elements allows achieving robust self-management. All above features of Niche facilitate development of self-managing applications.

3.2 Security Infrastructure

Name of result: Grid4All Security Infrastructure (security components, API, and tools)

What it does (summary):

The Grid4All policy-based Security Infrastructure has been developed to protect resources (services) from unauthorized access; access can be controlled through Virtual Organization (VO)-wide authorization policies. VO members (resource owners) set security policies, dictating how users may access VO resources. The Grid4All security infrastructure protects resources, enforcing the VO policy.

Security infrastructure provides authentication and authorization to other Grid4All modules with which it is integrated, e.g. VOFS. Authentication establishes the identity of a user; while authorization checks whether the use has rights to access the requested resource (e.g. a file). Authorization guarantees that the user can only access resources she has the right to access, according to VO policies. Authorization is policy-based; policies are expressed in XACML (eXtensible Access Control Markup Language).

The Grid4All security components include Policy Enforcement Points (PEP), Policy Decision Points (PDP), Policy Administration Points (PAP), Policy Information Points (PIP), and VO Membership Service (VOMS). The security components have been developed using Sun's XACML implementation.

Where it can be obtained:

http://www.isk.kth.se/~leifl/vofs/

Targeted stakeholders:

- End users authenticating to the VO.
- Administrators managing the VO Membership Service and VO policy repository.
- Developers of software which needs security infrastructure (authentication and authorization) for access control.

Scope, "best use" scenarios:

Security infrastructure (authentication and authorization) is required to protect course resources (course material, simulation engines, results, etc.) from unauthorized access in the educational usage scenario described in the deliverable D4.6. The scenario assumes that teachers and students will be given different roles, with different access rights to course resources. A teacher will have the right to construct and to modify access policies and assign a student (a student group) the right to

access the material related to the classes in which the student (the group) is enrolled, during specific hours. The security infrastructure enforces mandatory checks whenever a student wants to access the course documents (e.g., reports) or resources, e.g., simulation engine.

Distinguishing features and benefits:

Functional features include:

- Authentication, including login, functionality is used by the end user for login, and by authorization components for identity checks. Using a login application or web browser, the end user logs in to the VO Membership Service (VOMS) in order to obtain an authentication token.
- Authorization functionality is used to check whether the user (her application) is allowed to access a protected resource and to perform an operation on it. This functionality can be used by any resource or service that requires access control.
- Policy administration (CRUD) operations used by VO administrators and resource owners. This functionality allows setting and updating access policies of resources protected by the security infrastructure. A client program used by the administrator and a server updating the policies are available.

Non-functional features include:

- Caching (of recent authorization decisions, of policies and results of authentication checks) in order to improve performance.
- Robust error handling.
- Configurability using configuration files. Neither the end user nor the application developer is required to recompile security components when changing security configuration (settings) in the configuration files.
- Easy to install, all that is needed is to unpack a zip archive. The only extra installation needed is to install a Java Runtime Environment (JRE).
- Easy to use for both application developer and end user. Security infrastructure has a rather simple API and user-friendly tools (e.g. for setting policies); it is well documented, and there are scripts for starting all applications. No need to compile or run ant scripts.
- The services can be called in many ways: through TCP, RMI, a direct method call and standard input.

Environmental conditions/requirements:

Java SDK 6 SE, Sun's XACML Implementation (http://sunxacml.sourceforge.net/)

Recommendations for deployment (including prerequisites):

No specific requirements for deployment but only Java SE 6 JRE.

Experience/lessons learned:

The Grid4All security components (Policy Enforcement Point, Policy Decision Point, Policy Administration Point, etc) have been used to develop a policy-based security infrastructure for VO file systems (see VOFS and WebDAV VOFS).

Some lessons learnt: Access time is sensitive to the security overhead and the overhead should be reduced by caching of security information. There might be also a scalability issue if security actions are performed at a fine-grained level, e.g. at the level of individual users rather than groups, level of files rather than directories. To improve scalability, security control in Grid4All should be done at higher-level of granularity, e.g. at the group (role, community) level. In order to improve access time by reducing the security overhead, we have implemented role-based security; caching of resolved authorization requests at Policy Enforcement Points, caching policies at Policy Decision Points, and caching resolved authentication requests at Policy Information Points.

Comparison with state of the art:

Globus's Grid Security Infrastructure (GSI) defines the state of the art in Grid security. Reasons for not using existing Grid security infrastructures, e.g. Globus GSI or PERMIS, in Grid4All are:

- Globus GSI components are very tightly coupled with GT4 and are heavy-weight (complex, with high administration cost) to be used in Grid4All that assumes dynamic environments and requires autonomic deployment;
- Globus GSI is intended to protect web-services, whereas most of Grid4All components (e.g. VOFS) do not use web-service interfaces in the current prototypes;
- PERMIS has a proprietary policy language.

Grid4All has chosen XACML (the eXtensible Access Control Markup Language, OASIS standard) as a security policy language for its flexibility, which allows, in particular, defining any complex conditions in policy rules, including timing conditions useful for Grid4All. We used Sun's XACML implementation (http://sunxacml.sourceforge.net/) to develop Grid4All security components.

A security infrastructure is required in Grid4All to provide VO-wise security (authentication and authorization). We should indicate that the Grid4All security infrastructure is in line with state of the art in Grid security. It does not significantly improve the state of the art, although it has the following innovative features:

- Support for validity periods (time, date, day of week ranges and combinations of these) in policies;
- A caching mechanism for security data (decisions, identities, policies) which improves performance;
- Extendable generic API that allows extension of generic security components PEP and PAP in order to provide more convenient application-specific interface for setting access rights

(PAP) and application-specific access control (PEP). The considered use case: access control in VOFS – illustrates this approach of extending generic security components in order to provide application specific access control, e.g. an AFS-like access control in VOFS.

- Easy to install and use compared to other similar products.

4 Inter-VO services

4.1 Market Information Service

Name of result: Market Information Service (MIS)

What it does (summary): This MIS a publish/subscribe service based on peer-to-peer (P2P), used by participants at the market place. The main objective is to disseminate and obtain information revealing the market situation. Information is published at the MIS by auctions either during their execution or at their completion. This information is disseminated to clients who may either query or subscribe for notifications. The distributed market information service provides aggregated and summarized and forecast information.

Where it can be obtained: The MIS result prepared for the Grid4All integration can be found at the Gforge Server: *https://scm.gforge.inria.fr/svn/grid4all/wp2/GRIMP* The MIS is basically based on the Distributed Market Information System (DMIS), information and source code of that project can be found at: *https://code.ac.upc.edu/projects/dmis*

Targeted stakeholders: End-users and resources providers (humans or agents).

Scope, "best use" scenarios: Users retrieve information about a certain product such as resources within the market and can check the market for current price to place their product. This allows to get an entry price for a product placement and it allows to a fair trading in a distributed markets.

Distinguishing features and benefits: Avoidance of hot-spots (market instances with excess supply or demand) and stabilization of prices. The MIS should handle increase in number of subscribers to MIS, volumes of queries, subscriptions and publications. The MIS includes a uncertainty management component to reduce the number of sent messages and guarantee a certain level of accurate information.

Environmental conditions/requirements: The MIS needs Java 6 JRE, which can run under Unix and Windows machines. The installed machine needs and open port which is not blocked by a firewall or is behind a NAT.

Recommendations for deployment (including prerequisites): For the deployment it have to been set the used ports within the properties file and the IP direction to the bootstrap node.

Experience/lessons learned: The MIS has been executed on a small-scale test bed of three different machines in a network to test the communication protocol. Scalability tests have been executed on a local machine up to 30000 participants. The MIS has also been evaluated with the CAS component of Grid4All project and evaluated with software bidding agents.

Comparison with state of the art: A few other systems bases on distributed information

acquisition based on P2P to allow a large scalability. However, the MIS differs as it is developed for market information and treats and optimize problems arising from P2P systems such us uncertainty and forecasting in market specific environments.

4.2 Semantic Information System

Name of result: Semantic Information System (SIS)

What it does (summary): The Semantic Information System (SIS) provides a matching and selection service concerning offers and requests (place any of the two kinds of orders i.e. either offers or requests) of resources and services, placed by peers (humans or software agents) within grid environments. This service is used by the Grid4All market place.

Where it can be obtained: http://icsd-ai-lab.aegean.gr:8080/grid4all_sis/downloads.jsp (source + binaries + documents)

Targeted stakeholders: Services developers that advertise markets or application services in Grid environments, peers (humans or software agents) that query such information.

Scope, "best use" scenarios: Application developers in grid and SOA environments need to discover available services based on service characteristics.

Distinguishing features and benefits: The discovery of services/resources using semantics (semantic matchmaking of services' profiles), the support of such matchmaking in Grid Economies (the semantic discovery of traded resources/services according to economic characteristics also), the support of the advertisement, querying and selection functionalities through an API (for human and software agents use), scalability.

Environmental conditions/requirements: SIS is supported by an external tool, namely WSDL-AT for the textual annotation of WSDL profiling information and the automatic computation of semantic annotations (mapping WSDL part names to OWL classes). Also it integrates a Selection module (as a black-box) for the ranking of services resulted from the matchmaking module of SIS.

Recommendations for deployment (including prerequisites): Apache Tomcat, Axis (required libraries also included in the .war file), JDK 1.6, MySql (optional)

Experience/lessons learned: SIS has been evaluated with some test data for performance reasons. It does not scale well for complex queries and large number of offered resources/services (more than 500) due to the non-flat and heavily axiomatized ontology (based on projects' requirements).

Comparison with state of the art: Although SIS has many advantages over state-of-the-art systems (grid economy features considered in the semantic matchmaking process, automatic annotation of services, a selection functionality is provided, an API has been developed, etc.) it is restricted in terms of the provided architecture which is a centralized one (central semantic registry).

The design of a decentralized SIS will be provided in the frame of the project, however such an implementation is out of the projects' scope.

5 Collaborative and federative services

5.1 Telex, a Principled System Support for Write-Sharing in Collaborative Applications

Name of result: Telex

What it does (summary): The Telex middleware facilitates the design of peer-to-peer collaborative applications. It takes care of complex application-independent aspects, such as replication, conflict repair, and ensuring eventual commitment. Telex allows an application to access a local replica without synchronizing with peer sites. The application makes progress, executing uncommitted operations, even while peers are disconnected.

Where it can be obtained: http://telex2.gforge.inria.fr (source + binaries + documents).

Targeted stakeholders: Collaborative application designers.

Scope, "best use" scenarios: Telex targets *peer-to-peer* collaborative applications: no central site is required to store shared data; updates are asynchronous and can be performed even while disconnected. Telex is also helpful when designing decision-support applications: its conflict detection and resolution engine allows users to examine a number of "what-if" scenarios.

Distinguishing features and benefits: The Telex approach enables separation of concerns: application designers focus on semantics and need not worry about distribution or conflict resolution. Telex is based on a principled approach, the Action Constraint Framework, which enables the design of applications with proven safety and liveness guarantees. As applications share the same middleware, novel cross-application scenarios are possible.

Environmental conditions/requirements: Telex requires Java runtime 1.5 and two external libraries: *jgrapht* (graph management), *daisylib/fractal* (communication).

Recommendations for deployment (including prerequisites): Telex is a library. It must be deployed with applications that use it.

Experience/lessons learned: Static constraints defined by the ACF do not cover all needs: some applications require dynamic constraints, others require the "isolation" constraint. The engineering of large dynamic graphs raises performance issues: mechanisms for lock-free updates and lightweight snapshots are needed. Scalability requires the support of partial replication, which is an unexplored problem.

Comparison with state of the art: Compared to previous systems such as Coda, Bayou or Unison, Telex is based on a principled framework (ACF). Furthermore, Telex is completely decentralized and thus can be used in a peer-to-peer environment.

5.2 Virtual Organization File System

Name of result: Virtual Organization File System (VOFS)

What it does (summary): VOFS is a POSIX-like peer-to-peer file system that modernises the traditional file system utility with capabilities to federate and share files and storage for ad hoc or organised collaboration. The features introduced are available both interactively to users and programmatically to applications via extended semantics of POSIX. VOFS users can create shared workspaces and quickly populate them with files from other peers by linking them in. Users may also contribute storage to workspaces where new files will be stored. Files are cached locally and peers allow loss of communication as a normal condition.

Where it can be obtained: http://www.cslab.ece.ntua.gr/vofs/ (source + documents).

Targeted stakeholders: Collaborating Internet users, organisers of collaborative tasks, designers of collaborative applications with generic communication and file-sharing needs.

Scope, "best use" scenarios: VOFS is intended as a personal tool that will replace the cumbersome and nonstandard procedures that users exchange and share files and gather and share collections of documents. While most activities are expected to be ad-hoc, users can develop their own usage patterns and maintain workspaces for long term if they remain useful. VOFS can also be used as a substrate for applications that need simple file-sharing and publish-subscribe messaging.

Distinguishing features and benefits: VOFS extends the traditional file system utility in a non-intrusive way and simplifies common sharing workflows of collaborating users. Ideally, the simple but powerful file sharing primitives introduced by VOFS would become as essential as the rest of file system primitives in modern users' every-day activities.

Environmental conditions/requirements: VOFS requires FUSE 2.7 for Linux, python 2.5, and sqlite3.

Recommendations for deployment (including prerequisites): VOFS is launched as a number of daemon processes with no dependences beyond those specified in requirements.

Experience/lessons learned: Democratising access to the many, elaborate, and highly technical modern computing capabilities is better feasible by introducing simple new concepts and actions to the users' workflows. This way, users are independent from external technical support and free to adapt their tools to their own patterns. Complex services and sophisticated applications can then be available to the users through the same simple environment, requiring no unfamiliar interaction from users.

Comparison with state of the art: The solution of simple new primitives to our everyday collaboration workflows for document sharing is explored by Google Wave, which is currently under development. Google Wave also forces applications and sophisticated services to be treated

through the same environment, requiring no unfamiliar interaction from users. Contrary to VOFS which targets a system-wide utility as the filesystem, Google Wave provides its own environment.

5.3 WebDAV Virtual Organization File System

Name of result: WebDAV VOFS (Virtual Organization File System)

What it does (summary):

The WebDAV VOFS (Virtual Organization File System) is a VO-aware file system based on the WebDAV protocol.

VOFS allows VO members (users) to build a distributed file system on multiple computers donated by the VO members or external resource providers. In order to do that, VOFS allows users to expose their files and directories to a VO file system, i.e. to make them accessible (read, written, renamed, deleted and so on) within the VO. It also allows users to mount the VO file system to their local file systems, so that files in VOFS can be accessed by ordinary applications such as text editors, PDF and picture viewers, etc. In technical terms, files in VOFS can be accessed using the standard POSIX file API.

VOFS includes a policy-based security, which ensures that only VO members can access files in VOFS. Access rights in VOFS confirm to VO policies set by resource (file and storage) owners.

The WebDAV-based VOFS prototype can be used with multiple Operating Systems, such as Linux and Microsoft Windows.

Compared to the FUSE-based VOFS, the WebDAV-based prototype is a simpler solution for VOFS that supports the complete set of standard file operations (defined in POSIX file API) but it does not support the reconciliation functionality achieved by using Telex as implemented in the FUSE-based VOFS.

Where it can be obtained:

http://www.isk.kth.se/~leifl/vofs/

Targeted stakeholders:

End-users: VOFS can be used by any end-users who need a common file system, e.g. for file sharing, common working space for collaboration.

Developers: VOFS can also be used by developers who need a distributed file system to be used as a part of a distributed system or application.

Scope, “best use” scenarios:

Cooperative or collaborative learning, networks of schools engaged in collaborative activities such

as joint projects and courses that require a shared file system as a shared workspace. Jointly-taught courses may involve students and instructors from multiple sites, spanning different departments, schools, countries, time zones or cultures. VOFS can be used in such scenarios as a common distributed file system.

Distinguishing features and benefits:

Functional features include:

- Single shared namespace;
- Computers can join and leave VOFS at any time;
- (Un)Expose files and directories from the local (native) file system to VOFS, i.e. making them accessible to VO members;
- (Un)mount the VOFS to the local file system;
- Access to exposed files using the standard POSIX file API;
- Allows work offline (disconnect operation) with “last-writer-wins” reconciliation;
- VO-wise authentication and policy-based authorization (AFS-like access control);
- Support for validity periods in access rights;
- Allows accessing the exposed files using WebDAV or HTTP.

Non-functional features include:

- Caching to improve performance and for disconnect operation;
- Easy to mount using for example davfs2;
- Can operate under any Operating System that has the WebDAV mount support, e.g. Microsoft Windows, Linux, Solaris.
- The VOFS name space is kept either in a DHT or completely distributed. In the latter case each node stores the entire name space.

Environmental conditions/requirements:

Recommendations for deployment (including prerequisites):

WebDAV requires the Apache Tomcat (included in the WebDAV prototype), Java SE 6 JDK, and some mount utility that supports WebDAV, e.g. davfs2

Experience/lessons learned:

The WebDAV VOFS prototype has been successfully tested under Linux and Windows operating systems. A distributed solution to the name space maintenance using DHTs or name-space views at

each peer, improves scalability and robustness of the name space compared to a centralized solution, e.g. a central database. However it comes at the cost of increased complexity of the consistent name space maintenance that requires replication and a distributed algorithm for name-space maintenance, e.g. a gossip-based algorithm.

To improve file access, the security overhead should be minimized by means of access control at the directory level, caching of access decisions and policies.

Comparison with state of the art:

There are many distributed file systems, including systems specifically developed for Grids, e.g. gLite file catalogs, XtreemOS file system. For detailed state-of-the-art analysis, see Grid4All deliverable D3.1 (Chapters I and III). The major feature of WebDAV VOFS that distinguishes it from ordinary distributed file systems is VO-awareness, i.e. VO-policy-based access control that supports AFS (Andrew File System)-like access rights.

Innovative features of WebDAV VOFS can be summarized as follows.

- Support for single global name space;
- Tolerance to resource churn (computer leaves and failures);
- VOFS can be mounted by the end-user to access exposed files via the standard POSIX API;
- VO awareness: VO-wise authentication and policy-based authorization; support of the AFS-like access control;
- VOFS can operate under any Operating System that has the WebDAV mount support, e.g. Microsoft Windows, Linux, and Solaris.

5.4 Yet Another Storage Service

Name of result: Yet Another Storage Service, YASS

What it does (summary):

YASS is a storage service that allows a client to store, read and delete files on a set of computers. The service replicates files in order to achieve high availability of files and to improve access time.

YASS can be deployed and provided on computers donated by users of the service or by a service provider. YASS operates even if computers join, leave or fail at any time.

The current version of YASS maintains the specified number of file replicas despite of nodes leaving or failing, and it can scale (i.e. increase available storage space) when the total free storage is below a specified threshold.

Where it can be obtained:

http://niche.sics.se/

Targeted stakeholders:

End-users: YASS can be used by any end-users who need a reliable and scalable storage to store and retrieve files, e.g. for backup or file sharing. End-users can access the service by executing and interacting with the YASS client (front-end) on their computers. Executing YASS storage components on their computers, end-users can share their storage.

Developers: YASS can also be used by developers who need a reliable and scalable distributed file storage service to be used as a part of a distributed system or application.

Service provider: YASS can be offered by a service provider.

Scope, "best use" scenarios:

Cooperative or collaborative learning, networks of schools engaged in collaborative activities such as joint projects and courses that require shared file storage as a shared workspace. Jointly-taught courses may involve students and instructors from multiple sites, spanning different departments, schools, countries, time zones or cultures.

Distinguishing features and benefits:

Self-managing capability of YASS allows the users to minimize the human resources required for the service management.

YASS supports transparent replication of files for the sake of robustness (i.e. files are not lost when nodes fail or leave the storage system) and for the sake of performance (multiple replicas of the same file can be accessed in parallel). YASS has the following self-managment capabilities in order to improve file access (availability and performance):

- self-healing: it maintains a specified number of file replicas despite of resource failures and leaves;

- self-configuration: it adds new storage, if any available, when the total free storage space is lower than a specified threshold;

YASS can be extended with self-tuning capability so that the service optimizes the number of replicas depending on access rate, i.e. add new replicas when access rate is high, and remove replicas when the access rate is low.

Environmental conditions/requirements:

YASS requires Java SE 6 JDK and Apache Ant 1.7. YASS client requires swt.jar (Eclipse Standard Widget Toolkit, SWT) compatible with your operating system. YASS deployment requires a Niche

bootstrap node to initiate the deployment process.

Recommendations for deployment (including prerequisites):

YASS requires the Niche environment to be deployed and configured on a number of distributed computers.

Experience/lessons learned:

YASS has been developed as a demonstrator application (service) using Niche to illustrate the development of a self-managing application and to evaluate self-management overhead. Niche provides the basic support to enable self-management, such as an event notification mechanism for monitoring, an actuation API, dynamic (re)binding, and a set of classes to develop self-management code. This simplifies development of the monitoring and actuation parts of the management code, whereas development and implementation of the control algorithm itself is a responsibility of a developer. The control algorithm can be arbitrary complex.

When developing YASS we have learned necessity of coordination and synchronization of activities of different managers with different responsibilities (management objectives). Based on this experience, Niche has been extended to support direct interaction between managers.

Comparison with state of the art:

There are many P2P files storage systems, such as the gLite file catalog, that provide functionalities similar to YASS. Many of the systems do not support self-management (self-healing or self-configuration), however some of them do. YASS is in line with state-of-the-art self-managing storage systems. YASS implements relatively simple self-management algorithms, which can be improved to be more sophisticated; while reusing existing monitoring and actuation code of YASS.

5.5 Yet Another Computing Service

Name of result: Yet Another Computing Service (YACS)

What it does (summary):

YACS is a robust distributed computing service that allows a client to submit and execute jobs (bags of independent tasks) on a network of nodes (computers). To be executed, tasks need to be programmed (or wrapped) by extending the abstract Task class provided by YACS.

YACS can be deployed and provided on computers donated by users of the service or by a service provider. YACS operates even if computers join, leave or fail at any time.

YACS guarantees execution of jobs despite of nodes leaving or failing. Furthermore, YACS scales by itself, i.e. changes the number of execution components, when the number of jobs/tasks changes.

Where it can be obtained:

http://niche.sics.se/

Targeted stakeholders:

Developers: YACS can be used by developers who need a robust and scalable distributed bag-of-tasks execution service to be used as a generic computing service or a job-specific execution service, e.g. a parameter-sweep simulator, a movie transcoder, etc. YACS can be used as a part of another distributed service or application.

Service provider: A job-specific execution service based on YACS can be offered by a service provider. YACS can be used as a part of other services that require a robust bag-of-task execution service.

End-users: A job-specific execution service based on YACS can be used by any end-users who need a robust execution service to execute a bag of tasks, e.g. for parameter-sweep simulation, movie transcoding, etc. End-users can access the job-specific service by executing and interacting with a YACS client (front-end) on their computers. Executing YACS computing components (masters and workers) on their computers, end-users can share their computing resources.

Note that to be executed by YACS, tasks need to be programmed (or wrapped) in Java by extending the Task class provided by YACS.

Scope, "best use" scenarios:

The YACS service can be used to execute different kinds of batch jobs or bag-of-task applications such as parameter-sweep simulation, video transcoding, ray-tracing and other applications that follow the master-worker paradigm. Therefore the service can be deployed and used in cooperative or collaborative learning, networks of schools engaged in collaborative activities such as joint projects and courses that require execution of computational jobs on a set of (shared) computers donated by schools or the users. Jointly-taught courses may involve students and instructors from multiple sites, spanning different departments, schools, countries, time zones or cultures.

Distinguishing features and benefits:

The service automatically distributes tasks among available distributed resources (masters and workers), monitors task execution and restarts failed tasks. YACS guarantees execution of jobs despite of nodes leaving or failing. YACS supports checkpointing that allows restarting execution from the last checkpoint. Furthermore, YACS scales, i.e. changes the number of master and workers, when the number of jobs/tasks changes. In order to achieve high availability, the service always maintains a number of free masters and workers so that new jobs can be accepted without delay.

YACS has the following self-management capabilities in order to improve bag-of-tasks execution:

- Self-healing: When a worker component fails or leaves, YACS transparently assigns a task performed by the failed component to another component and restarts the task from the last checkpoint or from the initial state (if there are no checkpoints);

- Self-configuration: YACS adds new execution components (masters and workers), when the total number of execution components is lower than a specified threshold.

Self-managing capabilities of YACS allows the users to minimize the human resources required for the service management. YACS can be extended with self-tuning capability so that the service optimizes the number of masters and workers depending on job submission rate, i.e. it adds new masters and workers when submission rate is high, and removes masters and workers when the submission rate is low.

Environmental conditions/requirements:

YACS requires Java SE 6 JDK and Apache Ant 1.7. YACS deployment requires a Niche bootstrap node to initiate the deployment process.

The gMovie demonstrator application using YACS requires a common file system and the VLC media player.

Recommendations for deployment (including prerequisites):

YACS requires the Niche environment to be deployed and configured on a number of distributed computers.

Experience/lessons learned:

YACS has been developed as a demonstrator application (service) using the Niche distributed component management system (also developed in Grid4 All) to illustrate the development of a self-managing distributed application and to evaluate self-management overhead of YACS.

YACS can be used as a part of some other service that needs a robust execution service. In order to illustrate this, we have developed a job-specific service called gMovie, which is a movie transcoding service using the VLC media player. The g-Movie application has its front-end (a GUI-controlled client) and uses YACS to execute the trascoding tasks (scripts) in parallel on a set of computers.

The developer of YACS has indicated convenience of the Niche API that allows separating of concerns when developing functional and self-management parts of an application. This separation also improves quality of the implementation as individual components and respective source code are focused on one main role, i.e. either functional or self-management functionality.

Comparison with state of the art:

There is a number of computing (execution) services that defines the state of the art, such as Amazon execution cloud EC2, Apache Hadoop, Grid execution services Globus GRAM, gLite computing element, UNICORE, OGSA-BES. Most of the execution services have self-managing compatibilities similar to those in YACS (self-healing and self-configuration). YACS is in line with state-of-the-art self-managing execution services. The current version of YACS implements relatively simple self-management algorithms; however its self-managing capabilities can be extended with more sophisticated management logic for more complex and effective management objectives, while reusing existing monitoring and actuation code of YACS.

6 End-user oriented Applications

6.1 Collaborative Network Simulation Environment

Name of result: Collaborative Network Simulation Environment (CNSE)

What it does (summary): CNSE provides an environment for the execution of simulations (both single and parameter-sweep) based on *ns-2* as well as the visualization of the results with *nam* and *xgraph*.

Where it can be obtained: http://www.grid4all.eu/index.php?page=cnse (source + binaries + documents+ demo videos).

Targeted stakeholders: Educators (creation of activities that involve network simulations), students (perform the simulations, study the visualizations and understand the results), local administrators (application and server installation).

Scope, "best use" scenarios: CNSE can be use in educational scenarios for collaborative learning and project-based learning. The main scope is the analysis of computer networks, and it supports the comprehension of theoretical concepts by simulating different topologies, varying parameters and visualizing the results.

Distinguishing features and benefits: CNSE leverages the advantages of grid service technology in order to enable the use of computational and storage resources to run parameter-sweep simulations and store the output files. Additionally, it provides collaborative features to support collaborative learning scenarios.

Environmental conditions/requirements: CNSE is supported by three external tools. *Ns-2* to carry out network simulations, *nam* to visualize network animations, and *xgraph* to plot output results.

Recommendations for deployment (including prerequisites): The CNSE servers need Globus Toolkit 4.0 full version, JDK 1.6 and Apache Ant 1.6.5. In addition the machines in which CNSE clients are going to be used need also JDK 1.6.

Experience/lessons learned: CNSE has been evaluated in a real scenario with students of an undergraduate course in Telecommunication Engineering. The results have proven that CNSE supports the performance of parameter-sweep simulations in a computer network scenario. Moreover, students highlighted the easiness of easiness of this tool and its convenience in the realization of the course.

Comparison with state of the art: The *ns-2* is one of the most used simulators in Computer Network education. However, its use is hindered by the number of computational and storage

resources and the lack of collaborative features. To the best of our knowledge there are any simulator that provides collaborative features using the grid service technology.

6.2 Collaborative File Sharing

Name of result: Collaborative File Sharing (CFS)

What it does (summary): The Collaborative File Sharing (CFS) application allows users to collaborate, interact and share information. The collaboration is asynchronous. CFS exports notions of files and forums. Workspaces may contain forums, files and other workspaces thus allowing a hierarchical organization. Users may access an existing item or create a new one. Access to items may be restricted by the creator. In addition, a notion of roles is supported. All users that can access an item have a role that determines what the user can do and this role may differ from item to item. This role may be changed at any time by the creator or the item administrators. Interaction and collaboration between participants may be either through messages posted in a forum or through file versions uploaded in a file.

Where it can be obtained: http://www.grid4all.eu/index.php?page=cfs

Targeted stakeholders: End-users.

Scope, "best use" scenarios: Computer-supported collaboration or learning tasks based on sharing documents and potential concurrent modification of the structure of folders and the content of files.

Distinguishing features and benefits: Management of a shared directory tree (workspace) structure: create, delete, move, change, list; Management of files: create file, create version version, delete, move, rename, list versions, read version; Management of forums (messages): post message, reply, view, modify, delete. These operations can be performed concurrently (uses Telex to automatically resolve conflicts among concurrent actions).

Environmental conditions/requirements: Java 6 JRE, (CFS as a Firefox extension *[does not work with the current version of Firefox]*: Mozilla Firefox v2, Java 6 Plugin for Firefox or above), (CFS as command-line interface: Apache Ant)

Recommendations for deployment (including prerequisites): Needs a shared repository service: can be VOFS or a shared folder (e.g. DAV/NFS/CIFS based, no locking needed as concurrent access is handled by CFS with the Telex library)

Experience/lessons learned: Difficulties with the use of Telex (understanding the API, stability of Telex, due to the concurrent development of Telex and CFS). More info from experiments with real users under way (results by end of May)

Comparison with state of the art: There are a few other client-based applications for document-based file sharing and argumentation. The main advantages of CFS are that it is based on the

Mozilla platform (inheriting some advantages: portability, the application is offered in three forms: command-line, Firefox extension, standalone application), and that it automatically handles conflicts, that otherwise will be very confusing, that can occur in collaborative situations where multiple participants work on the same file workspace.

6.3 eMeeting, an on-line Multimedia Collaborative Environment

Name of result: eMeeting

What it does (summary): eMeeting gives a communication channel between Grid4All users. It allows multiple connected users to share an online experience using a video and audio conference, shared slides, shared documents, and a text tool for collaborative text editing.

Where it can be obtained: eMeeting is not a downloadable application, but it's possible to run an e-meeting session typing the following URL in the web browser: http://emeeting.antares.es/.
In order to get required user and password, please contact Antares at alicia@antares.es

Targeted stakeholders: End users.

Scope, "best use" scenarios: The expected use scenarios are:
On-line learning sessions in which multimedia material is required to support the tutor speech.
Business meetings benefiting from costs reduction and lower transportation time.

Distinguishing features and benefits: It allows making a session reservation, launching it, and once started up users can share documents, audio and video conversation, chat and slide shows. The main benefit is that it shortens distances so it can lower travel costs and allow people to connect in different time frames.

Environmental conditions/requirements: The eMeeting application has the following requirements:

- Internet browser and broadband connection (512kb minimum).
- Client must have JVM (Java Virtual Machine - for future integration, so far, there are no concerns about version)
- Flash Player 9+ installed
- Cookies activated in the browser

Experience/lessons learned: The challenge was to integrate the existing and centralized eMeeting application in a decentralized environment as Grid4All. Having analyzed the characteristics of we saw the possibilities of integration are through VOFS, VOMS, and Shared Calendar.

Since the characteristics of these applications had been already developed in a centralized way eMeeting was modified to allow control through web services. While it has been able to integrate most of the features, the conversion of the video conferencing system means a too high cost for this stage, since the current Grid4All provided services.

Comparison with state of the art: While there are applications allowing p2p on-line audiovisual communication between two single users, the challenge was to get a p2p tool capable of allowing simultaneous communication between multiple users. As stated above, we could not get the distribution of video and audio in a decentralized way even when it is technologically feasible but not possible due to its high cost of development.

6.4 Sakura, a Shared Calendar

Name of result: Sakura a shared calendar application (INRIA)

What it does (summary): The Sakura application is a peer to peer calendaring system that enables a user to share his calendar, to create and invite users to a meeting, to change its time, or to cancel it. Sakura relies on Telex, a platform for decentralized sharing developed in WP3. Sakura maintains the invariant that the user is not double-booked into two different meetings at the same time.

Where it can be obtained: https://gforge.inria.fr/projects/sakura-sc/

Targeted stakeholders: Collaborative application developers who produce new application over Telex, and end-users once a user friendly GUI is developed.

Scope, "best use" scenarios: Optimistic collaboration where users can concurrently organize meetings even if disconnected.

Distinguishing features and benefits: In contrast to common calendar systems such as Doodle, Sakura over Telex ensures consistency even when a user is tentatively engaged in several meetings, i.e., the agreement protocol will not commit conflicting meetings. Additionally, Sakura over Telex correctly supports alternative dates for a meeting and meeting grouping, and automatically propose best solution in presence of conflicts.

Environmental conditions/requirements: Java 5 JRE, and Telex library.

Recommendations for deployment (including prerequisites): Each user needs to download Sakura and install Telex.

Experience/lessons learned: Non-intuitive design of the consistency model due to the reasoning difficulties for optimistic systems and the challenges of partial replication.

Comparison with state of the art: Many collaborative calendars exist. The main contribution of Sakura is a fully decentralized implementation, automatic conflict resolution and eventual

consistency thanks to Telex. Additionally, the classical calendaring systems store all calendars in a single large database. Instead, for scalability and privacy reasons, Sakura maintains a separate calendar document per user, replicated only where some user has the right to access it.

7 Conclusions

The deliverable reports the main software results of the project. Each result is described following the Grid4All architecture under one of the three groups of services (core-VO, inter-VO, collaborative and federative) or end-user oriented applications. The user manuals are presented as annexes.

Level of confidentiality and dissemination

By default, each document created within Grid4All is © Grid4All Consortium Members and should be considered confidential. Corresponding legal mentions are included in the document templates and should not be removed, unless a more restricted copyright applies (e.g. at subproject level, organisation level etc.).

In the Grid4All Description of Work (DoW), and in the future yearly updates of the 18-months implementation plan, all deliverables listed in section 7.7 have a specific dissemination level. This dissemination level shall be mentioned in the document (a specific section for this is included in the template, both on the cover page and in the footer of each page).

The dissemination level can be defined for each document using one of the following codes:

PU = Public

PP = Restricted to other programme participants (including the EC services);

RE = Restricted to a group specified by the Consortium (including the EC services);

CO = Confidential, only for members of the Consortium (including the EC services).

INT = Internal, only for members of the Consortium (excluding the EC services).
This level typically applies to internal working documents, meeting minutes etc., and cannot be used for contractual project deliverables.

It is possible to create later a public version of (part of) a restricted document, under the condition that the owners of the restricted document agree collectively in writing to release this public version. In this case, a new document code should be given so as to distinguish between the different versions.

This page is intentionally left blank

Annex 1. Niche, a Distributed Component Management System

Niche Quick Start Guide

Leif Lindbäck and Vladimir Vlassov

Royal Institute of Technology (KTH), Stockholm, Sweden
{leifl,vladv}@kth.se

Table of Contents

Introduction

Niche is a Distributed Component Management System (DCMS), which is used to develop, deploy and execute self-managing distributed component-based applications on a structured overlay network of computers. Niche includes (1) a set of APIs for the development of self-managing distributed applications; (2) a run-time execution environment for the deployment and execution of the applications together with its management elements. The Niche run-time environment includes a set of containers that reside on a structured overlay network of computers; and a set of the overlay services (resource discovery, deployment, publish-subscribe, metadata DHTs) provided in each of the containers.

The main innovation in Niche is the novel use of overlays, where a structured overlay provides a network-transparent sensing/actuation infrastructure that enables self-management of distributed component-based applications. Functional components, component groups and bindings are first-class entities in Niche that can be monitored and manipulated by management components (via sensors and actuators) using the extensible monitoring and actuation APIs of Niche. Management components are organized as a network of Management Elements interacting through events. Niche supports sensing changes in the state of components and environment, and allows individual components to be found and appropriately manipulated. Niche deploys both functional and management components and sets up the appropriate sensor and actuation support infrastructure. The initial deployment code can be either manually written by the developer, or generated by Niche from an ADL (Architecture Description Language) description of the application architecture. The ADL compiler for describing initial configurations is made available together with Niche.

1 Downloading Niche

Download and unpack the `niche-0.2.zip` archive from `http://niche.sics.se/`

2 Installing Niche

1. In order to be deployed and to operate, Niche requires the Apache Ant build tool to be installed. For Ant download and installation, see `http://ant.apache.org/index.html`.
2. Niche requires Java (`http://java.sun.com`), version 1.6 or higher.
3. In the file `niche-0.2/Jade/etc/dks/dksParam.prop`, substitute `localhost` for your real host (host could be given as either host name or ip address) in the entry `ip`. Note that you can not run Niche without a working network connection.
4. In the file `niche-0.2/Jade/etc/oscar/bundle-jadeboot.properties`, substitute `localhost` for your real host in the entries `jadeboot.registry.host` and `jadeboot.discovery.host`.

5. In the file `niche-0.2/Jade/etc/oscar/bundle-jadenode.properties`, substitute `localhost` for your real host in the entries `jadeboot.registry.host` and `jadeboot.discovery.host`.
6. Make sure your host name is associated with your correct host. On Linux/UNIX machines this is done the following way:
   (a) Find the host name with the command `uname -n`
   ```
   $ uname -n
   myHostName
   ```
   (b) Make sure there is a line in the `/etc/hosts` file that looks like `<ip address> <host name>`, where `<host name>` is the host name returned by the `uname` command in the previous step.
7. Change all occurrences of `<Path to Jade>` to the absolute path to the `niche-0.2/Jade` directory in the following files:
   - `niche-0.2/Jade/etc/execute.properties`
   - `niche-0.2/Jade/etc/oscar/bundle.properties`
   - `niche-0.2/Jade/etc/oscar/bundle-jadeboot.properties`
8. Copy the version of `niche-0.2/Jade/externals/swt-3349-*.jar`, where * corresponds to the name of your operating system, e.g windows, to `niche-0.2/Jade/externals/swt.jar`
9. Niche uses a web server to provide addresses of existing nodes to new joining nodes. The web server must be installed separately since it is not part of the Niche distribution. The only requirement on the web server is that it must support PHP. When you have a working web server, do the following:
   (a) Copy the directory `niche-0.2/Jade/webcache` to the www root of the web server.
   (b) In `niche-0.2/Jade/etc/dks/dksParam.prop`, specify the host and port number of the web server in the entry `publishAddress`. Note that you can not specify `localhost`, you must use a real host.

3 Starting Niche

1. Create run-time archives by running Ant with the target `bundles` specified in the build file `niche-0.2/Jade/build-src.xml`, as follows.
   ```
   cd niche-0.2/Jade/
   ant -f build-src.xml bundles
   ```
2. Start the first node by running Ant with the target `jadeboot` in the build file `niche-0.2/Jade/build.xml`, as follows.
   ```
   cd niche-0.2/Jade/
   ant jadeboot
   ```
3. Start more nodes by by running Ant with the target `jadenode` in the build file `niche-0.2/Jade/build.xml`, as follows. You can start nodes on the same or different machines. The following commands should be executed for each node you want to start.
   ```
   cd niche-0.2/Jade/
   ant jadenode
   ```

4 Defining Niche Components with Fractal

Niche uses the *Fractal Architecture Description Language (ADL)* to define components. For more information about Fractal and ADL see the Niche Programming Guide or the Fractal home page (`http://fractal.ow2.org/`).

Briefly, Fractal components are runtime entities that communicate exclusively through well-defined access points, called *interfaces*. In Niche the interfaces are regular Java interfaces. Interfaces are divided into two kinds: *client interfaces* that emit operation invocations and *server interfaces* that receive them. Interfaces are connected through communication paths, called *bindings*. Each Fractal component has (at least) two basic controllers, (1) *binding controller*, which supports binding and unbinding; (2) *life-cycle controller* supports starting and stopping the execution of the component.

5 The Niche Hello World Program

Fig. 1. The Hello World program's GUI.

The Hello World program has front-end components and service components. The number of components is configurable. By default there are three service components and one front-end component. The service components are placed in a component group. The front-end component displays the GUI showed in Figure 1. There is a one-to-any binding from the front-end to the service component group, which is invoked when the *Hello Any* button in the GUI is clicked. When this happens, *one* of the service components will print the string `Hello World`. There is also a one-to-all binding from the front-end to the service component group, which is invoked when the *Hello All* button in the GUI is clicked. In this case *all* service components will print the string `Hello World`.

The Hello World program is self-healing, which means that if one service component crashes a new component will be instantiated, deployed and started. The self-healing control loop consists of two Management Elements (MEs), the `ServiceSupervisor` aggregator and the `ConfigurationManager` manager. `ServiceSupervisor` is notified whenever a service component leaves or joins the service component group. It notifies `ConfigurationManager` if the number of components in the group drops below 3. When notified, `ConfigurationManager` deploys and starts a new service component.

```
1   <component name="frontend">
2       <interface name="helloAny"
3                  role="client
4                  signature="helloworld.interfaces.HelloAnyInterface"
5                  contingency="optional"
6       />
7       <interface name="helloAll"
8                  role="client"
9                  signature="helloworld.interfaces.HelloAllInterface"
10                 contingency="optional"
11      />
12      <content class="helloworld.frontend.FrontendComponent"/>
13      <virtual-node name="lightweight1" resourceReqs="10"/>
14  </component>
```

Listing 1. ADL definition of front-end component.

The Hello World program also includes `StartManager`, which instantiates and deploys the service component group, `ServiceSupervisor` and `ConfigurationManager`. This is necessary since the current Niche prototype does not allow defining groups or configuring MEs in ADL.

The Hello World program is included in the Niche distribution, in the directory `NicheHelloWorld`; it is explained in detail below.

5.1 Functional Components of Hello World

There are two types of functional components, front-end component and service component.

Front-End Component The front-end component has two tasks: (1) to accept input from the user through a GUI; (2) call service component methods in response to user commands entered through the GUI.

The GUI is in a separate class, `helloworld.frontend.UserInterface`, which is not explained here. For an introduction to writing user interfaces with Swing, see `http://java.sun.com/docs/books/tutorial/uiswing/`. This class is not aware of Niche, its only task is to handle the GUI.

The frontend component is defined in the ADL file, `NicheHelloWorld.fractal`, located in the root directory of the Hello World program (`niche-0.2/NicheHelloWorld`). The component definition is showed in Listing 1. Lines 2-6 define the interface used for the one-to-any binding explained in 5. Note that the role of the interface is `client` (line 3), which means that the front-end component invokes this interface. Line 4 specifies the Java interface that defines the operations in the `helloAny` interface. Lines 7-11 specifies the interface used for the one-to-all binding explained in 5. Line 12 specifies the main Java class of the component. This class must implement server interfaces of the component, and also implement the interfaces of the Fractal controllers, see 4. Line 13 defines (1) a logical name for the Niche node on which the front-end component will be deployed; (2) a requirement for the node. This

requirement has no meaning, it is just a number. It is matched against the available resource specified for each node in the file `niche-0.2/Jade/lines`. Each line in this file contains a node number and a resource. A component specified in `NicheHelloWorld.fractal` will only be deployed on nodes with resource value (in `lines`) at least the value specified in the requirement for that node (i.e greater than or equal to 10 in this case). In a real application this feature could be used to specify for example available memory or CPU speed. This is explained in the Niche Programming Guide.

As specified in the ADL definition in Listing 1, it is the class `helloworld.frontend.FrontendComponent` that implements the Fractal controller interfaces. This part of the class is shown in Listing 2. The `listFc` method (lines 1-3) returns an array with the names of all components to which the front-end component can be bound to. These are the components implementing the front-end's client interfaces and `component`, which is a reference to the front-end component itself. All components should be able to handle such a reference to itself. The `lookupFc` method (lines 5-15) returns the front-end's current reference to the specified component. The `bindFc` method (lines 17-27) stores the given reference to a component. The `unbindFc` method (lines 29-39) removes the reference to the specified component. The `getFcState` method (lines 41-43) tells whether the front-end is active or not, `status` is a boolean instance parameter not showed in the listing. `startFc` (lines 45-50) is called by Niche when the front-end should become active, line 47 instantiates the `UserInterface` class containing the GUI. This is when the GUI is displayed. Finally, `stopFc` (lines 52-54) is called by Niche when the front-end should become inactive.

Listing 3 shows the methods that are invoked when the user clicks one of the GUI buttons (`HelloAny` or `HelloAll`). The component bound to the appropriate client interface is invoked. When the user clicks `HelloAny`, the GUI event handler will invoke the `helloAny` method defined in lines 5-7 in Listing 3. This method in turn invokes the method `helloAny` in the component who's reference is stored in the `helloAny` instance parameter as shown in line 6. The correct reference is stored in this instance parameter by the `StartManager`, as explained in 5.2. `HelloAll` is handled in a similar way as `HelloAny`.

Service Component The service component's task is to print the specified string when any of its server interfaces is invoked.

The service component consists of one class who's source code is in `src/helloworld/service/ServiceComponent.java`. Listing 4 shows its implementation of the server interfaces (`helloAny` and `helloAll`). These methods print the specified string.

The ADL definition is in the same fractal file as the front-end component's definition, `NicheHelloWorld.fractal`, which is located in the root directory of the Hello World program (`niche-0.2/NicheHelloWorld`). Note that there must be one `component` entity for each service component that shall be started. Since there are three such entities there will be three instances of the service component.

```
1     public String[] listFc() {
2         return new String[] { "component", "helloAny", "helloAll" };
3     }
4
5     public Object lookupFc(final String itfName) throws
          NoSuchInterfaceException {
6         if (itfName.equals("helloAny")) {
7             return helloAny;
8         } else if (itfName.equals("helloAll")) {
9             return helloAll;
10        } else if (itfName.equals("component")) {
11            return myself;
12        } else {
13            throw new NoSuchInterfaceException(itfName);
14        }
15    }
16
17    public void bindFc(final String itfName, final Object itfValue)
          throws NoSuchInterfaceException {
18        if (itfName.equals("helloAny")) {
19            helloAny = (HelloAnyInterface) itfValue;
20        } else if (itfName.equals("helloAll")) {
21            helloAll = (HelloAllInterface) itfValue;
22        } else if (itfName.equals("component")) {
23            myself = (Component) itfValue;
24        } else {
25            throw new NoSuchInterfaceException(itfName);
26        }
27    }
28
29    public void unbindFc(final String itfName) throws
          NoSuchInterfaceException {
30        if (itfName.equals("helloAny")) {
31            helloAny = null;
32        } else if (itfName.equals("helloAll")) {
33            helloAll = null;
34        } else if (itfName.equals("component")) {
35            myself = null;
36        } else {
37            throw new NoSuchInterfaceException(itfName);
38        }
39    }
40
41    public String getFcState() {
42        return status ? "STARTED" : "STOPPED";
43    }
44
45    public void startFc() throws IllegalLifeCycleException {
46        // Create the GUI.
47        new UserInterface(this);
48        status = true;
49        System.err.println("Frontend Started.");
50    }
51
52    public void stopFc() throws IllegalLifeCycleException {
53        status = false;
54    }
```

Listing 2. Implementation of Fractal controllers in front-end component.

```
1   // ////////////////////////////////////////////////////////////////
2   // ////////////////// Called from the user interface. /////////////
3   // ////////////////////////////////////////////////////////////////
4
5   public synchronized void helloAny() {
6       helloAny.helloAny("HelloWorld");
7   }
8
9   public synchronized void helloAll() {
10      helloAll.helloAll("HelloWorld");
11  }
```

Listing 3. Calls from GUI in front-end component.

```
1   public void helloAny(String s) {
2       System.out.println(s);
3   }
4
5   public void helloAll(String s) {
6       System.out.println(s);
7   }
```

Listing 4. Implementations of server interfaces in service component.

The component's definition and implementation of Fractal controllers are very similar to the front-end component's and are therefore not shown.

5.2 Management Elements of Hello World

Management elements (ME) can be defined in ADL in a Fractal file that should have the same name as the Java class implementing the element, and be located in the same directory. Such a definition will not cause the deployment of the ME, it must be deployed programmatically.

The MEs are:

- `ServiceSupervisor`, which is notified whenever a service component joins or leaves the service component group. If the number of components in this group drops below three it notifies `ConfigurationManager`
- `ConfigurationManager`, which deploys a new service component, starts it and makes it join the service component group.
- `StartManager`, which instantiates and deploys the service component group, `ServiceSupervisor` and `ConfigurationManager`. This is necessary since the current prototype does not allow defining groups or configuring MEs in ADL.

Management elements are described in detail below.

ServiceSupervisor. The `ServiceSupervisor` is an aggregator, which is defined in `ServiceSupervisor.fractal`, shown is Listing 5. Lines 2-4 define the following server interfaces implemented by this ME.

```
<definition name="helloworld.aggregators.ServiceSupervisor" >
 <interface name="init" role="server"
      signature="dks.niche.fractal.interfaces.InitInterface" />
 <interface name="eventHandler" role="server"
      signature="dks.niche.fractal.interfaces.EventHandlerInterface" />
 <interface name="movable" role="server"
      signature="dks.niche.fractal.interfaces.MovableInterface" />
 <interface name="trigger" role="client"
      signature="dks.niche.fractal.interfaces.TriggerInterface" />
 <content class="helloworld.aggregators.ServiceSupervisor" />
</definition>
```

Listing 5. ServiceSupervisor definition.

```
String idAsString =
    failedEvent.getFailedComponentId().getId().toString();

if (!currentComponents.containsKey(idAsString)) {
    // The failed component was not in our list of
    // active service components.
    return;
}

// Remove failed component from list of active components.
currentComponents.remove(idAsString);
currentAllocatedServiceComponents--;

if (myId.getReplicaNumber() < 1) {
    System.out.print("ServiceSupervisor ");
} else {
    System.out.print("ServiceSupervisor REPLICA ");
}
System.out.println("has received a ComponentFailEvent!\n"
    + "The result is that there are now " +
        currentAllocatedServiceComponents
    + " service components");

if (currentAllocatedServiceComponents <
    MINIMUM_ALLOCATED_SERVICE_COMPONENTS) {
    // There are too few service components.
    if (myId.getReplicaNumber() < 1) {
        System.out.println("ServiceSupervisor triggering!");
    }

    // Cancel eventual running timers since we are telling
    // ConfigurationManager now.
    actuator.cancelTimer(availabilityTimerId);
     // Tell ConfigurationManager there are too
     // few service components.
    eventTrigger.trigger(new ServiceAvailabilityChangeEvent());

    // When the timer goes off we will check if enough service
    // components have become active.
    availabilityTimerId =
        actuator.registerTimer(this,
                    AvailabilityTimerTimeoutEvent.class,
                    CHECK_NR_OF_COMPONENTS_AFTER_THIS_TIME);
}
```

Listing 6. Handling of ComponentFailEvent in ServiceSupervisor.

- `InitInterface`, which contains (1) method for receiving a reference to a component implementing the Niche API; (2) methods for receiving initialization attributes specified at deploy time.
- `EventHandlerInterface`, which lets the ME receive events.
- `MovableInterface`, which enables reading the ME's state when the ME is copied or moved.

Line 5 defines the client interface, `TriggerInterface`, to which this ME should be bound. This interface is used to send events, in this case to `ConfigurationManager`. A reference to a component implementing this interface will automatically be handed to `ServiceSupervisor` when it is deployed. Finally, line 6 defines the implementing class, `ServiceSupervisor`. Listing 6 shows what happens when `ServiceSupervisor` receives a `ComponentFailEvent`: it sends an event to `ConfigurationManager` if there are less than three active service components. Comments in the listing explain how it works.

ConfigurationManager. The `ConfigurationManager` is a manager, which is defined in `ConfigurationManager.fractal`; it is very similar to the definition of `ServiceSupervisor`. Listing 7 shows what happens when `ConfigurationManager` receives a `ServiceAvailabilityChangedEvent`: it deploys and starts a new `ServiceComponent`. Comments in the listing explain how it works.

StartManager. The `StartManager` is a manager which is defined in `NicheHelloWorld.fractal`; the definition is shown in Listing 8. Note that the value of the `definition` attribute is `ManagementType` (line 1). This definition contains a set of client interfaces that correspond to the Niche API. These interfaces are automatically bound after `StartManager` is instantiated. Listing 9 show how the StartManager creates the service component groups; Listing 10 shows how the `ServiceSupervisor` and `ConfigurationManager` are configured and deployed. Both listings are explained by comments in the code.

5.3 Deploying Hello World

1. Place the file `NicheHelloWorld.fractal` in `niche-0.2/Jade/examples` and the file `nichehelloworld.jar` in `niche-0.2/Jade/externals`. This is already done in the downloadable Niche distribution.
2. The files listed below all need the correct properties concerning the Hello World program. In the distribution all needed properties are already defined.
   - `niche-0.2/Jade/build-src.xml`
   - `niche-0.2/Jade/etc/build.properties`
   - `niche-0.2/Jade/META-INF/jadenode/MANIFEST.MF`
   - `niche-0.2/Jade/META-INF/jadeboot/MANIFEST.MF`

```
1   // Find a node that meets the requirements for a service component.
2   NodeRef newNode = null;
3   try {
4       newNode = myManagementInterface.
5                   oneShotDiscoverResource(nodeRequirements);
6   } catch (OperationTimedOutException err) {
7       System.out.println("Discover operation timed out.");
8       continue;
9   }
10  if (newNode == null) {
11      System.out.println("Could not get a new resource.);
12      break;
13  }
14  System.out.println("Got a resource");
15
16  // Allocate resources for a service component at the found node.
17  List allocatedResources = null;
18  try {
19      allocatedResources = myManagementInterface.allocate(newNode,
            null);
20  } catch (OperationTimedOutException err) {
21      System.out.println("Allocate operation timed out.");
22      continue;
23  }
24  ResourceRef allocatedResource = (ResourceRef)
        allocatedResources.get(0);
25
26  // Deploy a new service component instance at the allocated resource.
27  String deploymentParams = null;
28  try {
29      deploymentParams = Serialization.serialize(serviceCompProps);
30  } catch (IOException ioe) {
31      ioe.printStackTrace();
32  }
33  List deployedComponents = null;
34  try {
35      deployedComponents =
36          myManagementInterface.deploy(allocatedResource,
                deploymentParams);
37  } catch (OperationTimedOutException err) {
38      System.out.println("Deploy operation timed out.");
39      continue;
40  }
41
42  ComponentId cid =
43      (ComponentId) ((Object[]) deployedComponents.get(0))[1];
44  System.out.println("ConfigurationManager is adding new component");
45
46  // Add the newly deployed component to the service group
47  // and start it.
48  myManagementInterface.update(componentGroup, cid,
49          NicheComponentSupportInterface.ADD_TO_GROUP_AND_START);
50  System.out.println("ConfigurationManager says: All done!");
```

Listing 7. Handling of ServiceAvailabilityChangeEvent in ConfigurationManager.

```
1   <component name="StartManager"
        definition="org.objectweb.jasmine.jade.ManagementType">
2     <content class="helloworld.managers.StartManager"/>
3   </component>
```

Listing 8. StartManager definition.

```
1         // Get a reference to the Niche API.
2         NicheActuatorInterface myActuatorInterface =
              nicheService.getOverlay().getJadeSupport();
3
4         // Find the front-end component.
5         ComponentId frontendComponent =
6             (ComponentId) nicheIdRegistry.lookup(APPLICATION_PREFIX +
                  FRONTEND_COMPONENT);
7
8         // Find all service components.
9         ArrayList<ComponentId> serviceComponents = new ArrayList();
10        int serviceComponentIndex = 1;
11        ComponentId serviceComponent =
12            (ComponentId) nicheIdRegistry.lookup(APPLICATION_PREFIX +
                  SERVICE_COMPONENT
13                + serviceComponentIndex);
14        while (serviceComponent != null) {
15            serviceComponents.add(serviceComponent);
16            serviceComponentIndex++;
17            serviceComponent =
18                (ComponentId) nicheIdRegistry.lookup(APPLICATION_PREFIX
                      + SERVICE_COMPONENT
19                    + serviceComponentIndex);
20        }
21
22        // Create a component group containing all service components.
23        GroupId serviceGroupTemplate =
              myActuatorInterface.getGroupTemplate();
24        serviceGroupTemplate.addServerBinding("helloAny",
              JadeBindInterface.ONE_TO_ANY);
25        serviceGroupTemplate.addServerBinding("helloAll",
              JadeBindInterface.ONE_TO_MANY);
26        GroupId serviceGroup =
27            myActuatorInterface.createGroup(serviceGroupTemplate,
                  serviceComponents);
28
29        // Create a one-to-any binding from the front-end
30        // to the service group.
31        // This binding uses the helloAny interface.
32        String clientInterfaceName = "helloAny";
33        String serverInterfaceName = "helloAny";
34        myActuatorInterface.bind(frontendComponent,
              clientInterfaceName, serviceGroup,
35                                 serverInterfaceName,
                                       JadeBindInterface.ONE_TO_ANY);
36
37        // Create a one-to-all binding from the front-end
38        // to the service group.
39        // This binding uses the helloAll interface.
40        clientInterfaceName = "helloAll";
41        serverInterfaceName = "helloAll";
42        myActuatorInterface.bind(frontendComponent,
43                                 clientInterfaceName,
44                                 serviceGroup,
45                                 serverInterfaceName,
46                                 JadeBindInterface.ONE_TO_MANY);
```

Listing 9. Creation of service component groups in StartManager.

```
1         // Configure and deploy the ServiceSupervisor aggregator.
2         ManagementDeployParameters params =
3             new ManagementDeployParameters();
4         params.describeAggregator(ServiceSupervisor.class.getName(),
5                                   "SA", null,
6                                   new Serializable[] {
                                        serviceGroup.getId() });
7         params.setReliable(true);
8         NicheId serviceSupervisor =
9             myActuatorInterface.deployManagementElement(params,
                  serviceGroup);
10        myActuatorInterface.subscribe(serviceGroup,
11                                  serviceSupervisor,
12                                  ComponentFailEvent.class.getName());
13        myActuatorInterface.subscribe(serviceGroup,
14                                  serviceSupervisor,
15                                  MemberAddedEvent.class.getName());
16
17        // Grab the service component's properties from a
18        // service component which is already deployed.
19        // The ConfigurationManager needs these when
20        // it deploys a new service component.
21        ComponentId grabParametersFromThis =
22            (ComponentId) nicheIdRegistry.lookup(APPLICATION_PREFIX +
                  SERVICE_COMPONENT + "1");
23        DeploymentParams serviceComponentProperties = null;
24        try {
25            serviceComponentProperties =
26                (DeploymentParams)
                      Serialization.deserialize(grabParametersFromThis.
27                                        getSerializedDeployParameters());
28        } catch (IOException e) {
29            e.printStackTrace();
30        } catch (ClassNotFoundException e) {
31            e.printStackTrace();
32        }
33
34        // Configure and deploy the ConfigurationManager manager.
35        String minimumNodeCapacity = "200";
36        params = new ManagementDeployParameters();
37        params.describeManager(ConfigurationManager.class.getName(),
38                        "CM", null,
39                        new Serializable[] { serviceGroup,
40                                             serviceComponentProperties,
41                                             minimumNodeCapacity });
42        params.setReliable(true);
43        NicheId configurationManager =
44            myActuatorInterface.deployManagementElement(params,
                  serviceGroup);
45        myActuatorInterface.subscribe(serviceSupervisor,
              configurationManager,
46            ServiceAvailabilityChangeEvent.class.getName());
```

Listing 10. Configuration and deployment of management elements in StartManager.

3. The file `niche-0.2/Jade/NicheHelloWorld-Deploy.bsh` is a script needed for deployment of Hello World.
4. Start Niche as described in Section 3. You should start at least four nodes including the first node started with the Ant target `jadeboot`.
5. Deploy the Hello World program by running Ant with the target `testG4A-NicheHelloWorld-Deploy`, as follows.
   ```
   cd niche-0.2/Jade/
   ant testG4A-NicheHelloWorld-Deploy
   ```

5.4 Starting Hello World

1. Start the Hello World program by running Ant with the target `testG4A-NicheHelloWorld-Start`, as follows.
   ```
   cd niche-0.2/Jade/
   ant testG4A-NicheHelloWorld-Start
   ```
   Note that this target needs the script file `niche-0.2/Jade/NicheHelloWorld-Start.bsh`.
2. When this target is executed, the GUI in Figure 1 will be displayed. How the program works is explained in the beginning of Section 5.

This page is intentionally left blank

Niche Programming Guide

(SICS, KTH, INRIA)

July 8, 2009

Abstract

Deploying and managing distributed applications in dynamic Grid environments requires a high degree of autonomous management. Programming autonomous management in turn requires programming environment support and higher level abstractions to become feasible. We present Niche, which is a Distributed Component Management System (DCMS) that provides an API for programming self-managing component-based distributed applications. Application's functional and self-management code are programmed separately. The framework extends the Fractal component model by the component group abstraction and one-to-any and one-to-all communication patterns. Niche can automatically move application components responsible for self-management when necessary due to resource churn. The framework supports a network-transparent view of system architecture simplifying designing application self-* code. The framework provides a concise and expressive API for self-* code. Programming application self-* behaviours with Niche requires just a few dozens lines per application component. The implementation of the framework relies on scalability and robustness of the Niche structured p2p overlay network. We have also developed a distributed file storage service to illustrate and evaluate our framework. We discuss the current status of the Niche implementation and outline future work.

Contents

List of Figures

Chapter 1

Introduction

Niche is a distributed component management system (DCMS), which used to develop, deploy and execute a self-managing distributed application or service on a dynamic overlay network of the Democratic Grid.

Niche includes a component-based programming model with a set of APIs for development of self-managing applications and services, and a run-time execution environment for deployment and execution of the services. The run-time environment provides a set of overlay services, which mostly deal with connectivity of Niche resources and elements, e.g. nodes, components, component groups.

Niche programming model and API separate functional and non-functional (management) parts of the application. The management part includes self-* code that monitors and manages the functional part. Application components, component groups and bindings, are first-class entities in Niche that can be monitored and manipulated by the self-* code using an extensible actuation API provided by Niche. The actuation API defines a number of operations that can be performed to change architecture, configuration and operation of an application, e.g. deploy and (re)bind components, start/stop threads, move and replicate components, change component attributes.

The Niche programming model is partly based on the Fractal component model [7], in which components are bound and interact with each other using two kinds of interfaces: (1) server interfaces offered by the components; (2) and client interfaces used by the components. Components are interconnected by bindings: a client interface of one component is bound to a server interface of another component. Fractal allows nesting of components in composite components and sharing of components. Components have control membranes, introspection and intercession capabilities. This enables developing of manageable components. Niche supports location-transparent bindings and extends the original Fractal model with component groups that support one-to-any and one-to-many bindings.

Both, functional and management parts of an application can be pro-

grammed using the Niche component-based programming model and the corresponding API. However, elements of the management part (watchers, aggregators, managers) normally interact with each other using the event mechanism rather than the bindings used to interconnect functional components.

The management part of an application is constructed as a set of control loops each of which monitors some part of the application and reacts on predefined events (e.g. node failures, leaves or joins) and application-specific events (e.g. high load, low available storage capacity). The predefined events are fired by the run-time environment. When a control loop finds a symptom that requires some changes in the application, it plans and actuates necessary management actions using the Niche actuation API.

The control loops of the management part are built of distributed Management Elements, MEs, interacting through events delivered by the pub/-sub overlay service. MEs can be of three types: Watchers (Wi on Figure X), Aggregators (Aggr) and Managers (Mgr). A watcher monitors a part of the application, and can fire events when it finds some symptoms of the management concern. Aggregates are used to collect, filter and analyse events from watchers. An aggregator can be programmed to analyse symptoms and to issue change requests to managers. Managers do planning and execution of change requests. Niche programming environment provides basic MEs, publish/subscribe and actuation APIs. The Niche run-time environment includes a set of component containers that reside on a structured overlay network of VO computers; and a set of the overlay services (resource discovery, deployment, publish/subscribe, DHT).

The Niche run-time system allows initial deployment of a service or an application on the Democratic Grid. Niche relies on the underlying overlay services to discover and to allocate resources needed for deployment, and to deploy the application. The initial deployment code can be either manually written by the developer, or generated by Niche from the ADL description of the application architecture.

After/upon deployment, the management part of the application can monitor and react on changes in availability of resources by subscribing to resource events fired by Niche containers. In order to achieve robustness of the management part, MEs are transparently replicated in different Niche containers.

All elements of a Niche application - components, bindings, groups, management elements - are identified by unique identifiers (names) that enable location transparency. Niche uses the DHT functionality of the underlying structured overlay network for its lookup service. Furthermore, each container maintains a cache of name-to-location mappings. Once a name of an element is resolve to its location, the element (its hosting container) is accessed directly rather than by routing messages though the overlay network. If the element moves to a new location, the element name is transparently

resolved to the new location.

Self-Managing Services Using Niche

In order to demonstrate Niche, we have developed two self-managing services:

YASS : Yet Another Storage Service;

YACS : Yet Another Computing Service.

The services can be deployed and provided on computers donated by users of the service or by a service provider. The services can operate even if computers join, leave or fail at any time. Each of the services has self-healing and self-configuration capabilities and can execute on a dynamic overlay network. Self-managing capabilities of services allows the users to minimize the human resources required for the service management. Each of services implements relatively simple self-management algorithms, which can be changed to be more sophisticated, while reusing existing monitoring and actuation code of the services.

YASS (Yet Another Storage Service) is a robust storage service that allows a client to store, read and delete files on a set of computers. The service transparently replicates files in order to achieve high availability of files and to improve access time. The current version of YASS maintains the specified number of file replicas despite of nodes leaving or failing, and it can scale (i.e. increase available storage space) when the total free storage is below a specified threshold.

YACS (Yet Another Computing Service) is a robust distributed computing service that allows a client to submit and execute jobs, which are bags of independent tasks, on a network of nodes (computers). YACS guarantees execution of jobs despite of nodes leaving or failing. Furthermore, YACS scales, i.e. changes the number of execution components, when the number of jobs/tasks changes. YACS supports checkpointing that allows restarting execution from the last checkpoint when a worker component fails or leaves.

Chapter 2

Component Programming Primer

This chapter aims to introduce the Fractal component model, to provide a basic primer on component programming, and to outline the main elements of the architecture description language (ADL).

2.1 The Fractal Component Model

Fractal is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications from operating systems and middleware to graphical user interfaces. The Fractal component model has the following main features:

- `hierarchical containment`: components can be nested at arbitrary levels (hence the "Fractal" name).
- `reflection`: components can be endowed with introspection and intercession capabilities.
- `sharing`: a given component instance can be included (or shared) by more than one component.
- `openness`: Fractal does not impose predefined semantics for reflective behaviour, component containment, and component binding.

Fractal components are runtime entities that communicate exclusively through well-defined access points, called *interfaces*. Interfaces can be divided into two kinds: *client interfaces* that emit operation invocations and *server interfaces* that receive them. Interfaces are connected through communication paths, called *bindings*. Fractal distinguishes primitive components from composite components, formed by hierarchically assembling other

components. Each Fractal component is made of two parts: the *membrane*, which embodies control behaviour, and the *content*, which consists of a finite set of sub-components. The membrane is composed of several *controllers*, which can take arbitrary (including user-defined) forms. Nevertheless, Fractal does define a useful set of four basic controllers. The *attribute controller* supports configuring component properties. The *binding controller* supports binding and unbinding client interfaces to server interfaces. The *content controller* supports listing, adding, and removing sub-components. The *life-cycle controller* supports starting and stopping the execution of a component. Finally, Fractal provides an *architecture description language* (ADL) for specifying configurations comprising components, their composition relationships, and their bindings.

Fractal is an ObjectWeb project, and further details are available at: `http://fractal.objectweb.org`

2.2 Implementing Fractal Components in Java

This section explains how to program Fractal components in Java. The example used is the HelloWorld example, found (more or less adapted) in all Fractal distributions. This example is a very simple application made of two primitive components inside a composite component (see the figure below).

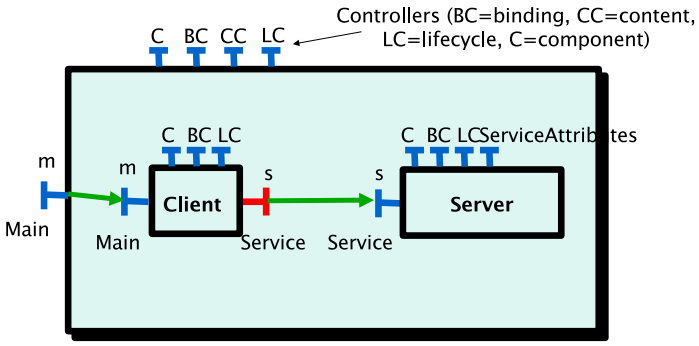


Figure 2.1: HelloWorld Fractal application.

The first primitive component is a "server" component that provides an interface to print messages on the console. It can be parameterized with two attributes: a "header" attribute to configure the header printed in front of each message, and a "count" attribute to configure the number of times each message should be printed. The other primitive component is a "client" component that uses the previous component to print some messages.

The server component provides a server interface named "s" of type Service, which contains a print method. It also has an AttributeController interface of type ServiceAttributes, which contains four methods to get and set the two attributes of the server component.

The client component provides a server interface named "m" of type Main, which contains a main method, called when the application is launched. It also has a client interface named "s" of type Service.

The composite component provides a server interface "m" that exports the corresponding interface of the client component.

The application can be programmed in two steps, by creating the component interfaces first, and then implementing these interfaces in the component classes.

2.2.1 Creating the component interfaces

Three interfaces must be implemented: the two "functional" interfaces Service and Main, and the attribute controller interface ServiceAttributes.

The functional interfaces are programmed "normally", i.e., the Fractal model does not impose any constraints on the Fractal functional component interfaces, except the fact that they must be public. The Service and Main interfaces are therefore very simple to implement:

```
public interface Service {
  void print (String msg);
}

public interface Main {
  void main (String[] args);
}
```

On the other hand, the attribute controller interfaces must extend the AttributeController interface, and must only have getter and setter method pairs (and they must be public too). The ServiceAttributes interface is therefore the following:

```
public interface ServiceAttributes extends AttributeController {
  String getHeader ();
  void setHeader (String header);
  int getCount ();
  void setCount (int count);
}
```

2.2.2 Implementing the component classes

The component classes must implement the previous interfaces, as well as some Fractal control interfaces.

The server component class, called ServerImpl, must implement the Service and ServiceAttributes interfaces. It may also implement the LifeCycleController interface, in order to be notified when it is started and stopped, but this is not mandatory. Since the server component does not have client interfaces, the ServerImpl class does not need to implement the BindingController interface, whose role is to manage the bindings involving the client interfaces of a given component. The ServerImpl class is therefore the following:

```
public class ServerImpl implements Service, ServiceAttributes {

  private String header = "";

  private int count = 0;

  public void print (final String msg) {
    for (int i = 0; i < count; ++i) {
      System.err.println(header + msg);
    }
  }

  public String getHeader () {
    return header;
  }

  public void setHeader (final String header) {
    this.header = header;
  }

  public int getCount () {
    return count;
  }

  public void setCount (final int count) {
    this.count = count;
  }
}
```

The client component class, called ClientImpl, must implement the Main interface. Since the client component has client interfaces, the class must also implement the BindingController interface, a basic Fractal control interface, presented next:

```
public interface BindingController {

  // Returns the names of the client interfaces of the component
```

```
  String[] listFc ();

  // Returns the interface to which the given client interface is bound
  Object lookupFc (String clientItfName) throws NoSuchInterfaceException;

  // Binds the client interface to the server interface
  void bindFc (String clientItfName, Object serverItf)

  // Unbinds the client interface
  void unbindFc (String clientItfName);
}
```

The ClientImpl class is therefore the following:

```
public class ClientImpl implements Main, BindingController {

  private Service service;

  public void main (final String[] args) {
    service.print("hello world");
  }

  public String[] listFc () {
    return new String[] { "s" };
  }

  public Object lookupFc (final String cItf) {
    if (cItf.equals("s")) {
      return service;
    }
    return null;
  }

  public void bindFc (final String cItf, final Object sItf) {
    if (cItf.equals("s")) {
      service = (Service)sItf;
    }
  }

  public void unbindFc (final String cItf) {
    if (cItf.equals("s")) {
      service = null;
    }
  }
}
```

2.3 Deploying Fractal applications

The simplest method to deploy an application is to describe the architecture of the application in the Architecture Description Language (ADL), and to pass this description to the deployment service. The HelloWorld application can be described as follows:

```
<definition name="helloworld.HelloWorld">

  <!-- Interface of the composite component-->
  <!-- NB. myhelloworld.* are the fully-qualified names of interfaces/classes
  <interface name="m" role="server" signature="myhelloworld.Main"/>

  <!-- The client sub-component -->
  <component name="client">
    <interface name="m" role="server" signature="myhelloworld.Main"/>
    <interface name="s" role="client" signature="myhelloworld.Service"/>

    <!-- Implementation of the client component -->
    <content class="myhelloworld.ClientImpl"/>
  </component>

  <!-- The server sub-component -->
  <component name="server">
    <interface name="s" role="server" signature="myhelloworld.Service"/>

    <!-- Implementation of the server component -->
    <content class="myhelloworld.ServerImpl"/>

    <!-- Attributes of the server component -->
    <attributes signature="myhelloworld.ServiceAttributes">
      <attribute name="header" value="-> "/>
      <attribute name="count" value="1"/>
    </attributes>

  </component>

  <!-- The binding between the composite and client -->
  <binding client="this.m" server="client.m" />

  <!-- The binding between client and server -->
  <binding client="client.s" server="server.s" />

</definition>
```

The HelloWorld example demonstrates the main concepts of the standard Fractal ADL; namely, component definitions, components, interfaces,

bindings, and attributes. Full details on the language can be found at `http://fractal.objectweb.org/fractaladl/`.

2.4 Extensions of standard Fractal ADL

Fractal ADL (Architecture Description Language) is an open and extensible language to define component architectures. More precisely, the language is made of a set of modules, each module defining an abstract syntax for a given architectural concern (e.g., component containment). The Fractal implementation includes a standard set of modules that allow describing component types, component implementations, component hierarchies and component bindings. The implementation then allows adding new modules for new architectural concerns and provides a modular toolset for processing the language extensions.

In the context of DCMS, we have extended the standard ADL in the following three areas: deployment, binding, and self-management.

2.4.1 Deployment extensions

The following ADL extract demonstrates the deployment extensions. It refines the client description in the HellWorld example with two new elements: packages and virtual nodes.

```
 ...
<component name="client">
    <interface name="m" role="server" signature="myhelloworld.Main"/>
    <interface name="s" role="client" signature="myhelloworld.Service"/>
    <content class="myhelloworld.ClientImpl"/>
    <packages>
       <package name="ClientPackage v1.3" >
          <property name="local.dir" value="/tmp/j2ee"/>
     </package>
    </packages>
    <virtual-node name="node1" resourceReqs="(&(memory>=1)(CPUSpeed>=1))"/>
</component>
 ...
```

The *packages* element provides information about the software packages necessary for creating run-time components. Packaging currently relies on the OSGI standard, and packages are identified with a unique name in the OSGI bundle respository. The previous ADL extract states that the implementation of the client component (i.e., the class ClientImpl) is part of the OSGI bundle named "ClientPackage v1.3".

The *virtual node* element provides information about the placement of a component. Virtual nodes are logical component containers, automatically mapped to concrete nodes at deployment time. The mapping is based on the

associated resource requirements (*ResourceReqs* attribute) and the actual deployment environment. For example, the previous ADL extract states that the client should be deployed on a node with memory larger than 1 GB and CPU speed larger than 1Ghz. Resource requirements are expressed in the LDAP filter syntax.

2.4.2 Binding extensions

These extensions enable the ADL to represent one-to-any and one-to-all bindings. In the following example, a client component is connected to a group of two server components (server1, server2) using one-to-any invocation semantics (see figure).

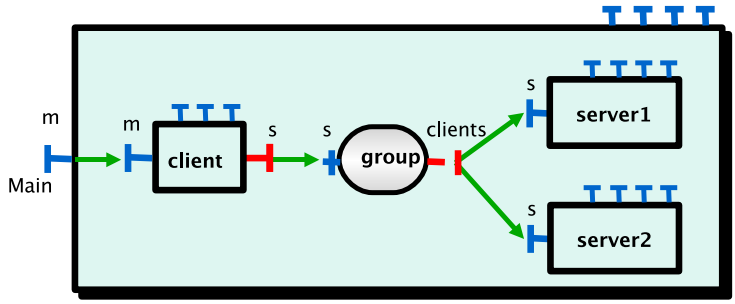


Figure 2.2: Application using group communication

This example can be described in ADL as follows:

```
<definition name="MyHelloGroup">

  <interface name="r" role="server" signature="myhelloworld.Main"/>

  <component name="client">
    <interface name="r" role="server" signature="myhelloworld.Main"/>
    <interface name="s" role="client" signature="myhelloworld.Service"/>
    <content class="myhelloworld.ClientImpl"/>
  </component>

   <component name="group">
    <interface name="s" role="server" signature="myhelloworld.Service"/>
    <interface name="clients" role="client" signature="myhelloworld.Service"
              cardinality="collection"/>
    <content class="GROUP"/>
  </component>
```

```
  <component name="server1">
    <interface name="s" role="server" signature="myhelloworld.Service"/>
    <content class="myhelloworld.ServerImpl"/>
  </component>

  <component name="server2">
    <interface name="s" role="server" signature="myhelloworld.Service"/>
    <content class="myhelloworld.ServerImpl"/>
  </component>

  <binding client="this.r" server="client.r" />
  <binding client="client.s" server="group1.s" bindingType="groupAny"/>
  <binding client="group1.clients1" server="server1.s"/>
  <binding client="group1.clients2" server="server2.s"/>

</definition>
```

As seen in this description, representing one-to-any bindings involves using a special component with content "GROUP". Group membership is then represented as binding the server interfaces of members to the client interfaces of the group (the "collection" value indicates that the group has an arbitrary number of client interfaces). Invoking the group involves creating a binding to its server interface. One-to-any and one-to-all invocation semantics are represented by setting the *bindingType* attribute to "groupAny" or "groupAll".

2.4.3 Self-management extensions

The ADL extract below demonstrates the current self-management extensions. It refines the previous group communication example to add a management component.

```
<definition name="MyHelloGroup">

  <interface name="r" role="server" signature="myhelloworld.Main"/>

  <component name="mgr" definition="org.objectweb.jasmine.jade.ManagementType">
    <content class="org.objectweb.jasmine.jade.examples.managerimpl.ManagerImpl"/>
  </component>

  <!-- As above ... -->
  ...

</definition>
```

Management components have a predefined definition "Management-Type". This definition contains a set of client interfaces that correspond

to the DCMS API (see Chapter 4) that management components require for their operation. These interfaces are implicitly bound by the system after management components are instantiated. In the current prototype, the management components capture only the initial deployment and configuration of the self-management behaviour. In other words, the component implementation (e.g., the ManagerImpl class) contains the code for creating, configuring, and activating the set of DCMS management elements (MEs). More specifically, this code is located in the implementation of the LifeCycleController interface (start operation) of the manager. A future version will enable describing in ADL the deployment and configuration of individual MEs.

Chapter 3

Self-Managing Applications with Niche

The purpose of this chapter is to gradually introduce Niche – the Distributed Component Management System (DCMS) and its concepts without the burden of full details of its API and current limitations of the prototype. Information presented in this chapter should suffice to understand the formal API description in Chapter 4, the YASS example in Chapter 5, and the discussion of features, limitations and future extensions of the current prototype in Chapters 6 and 7. The presentation in this Chapter is informal, and particular syntax of examples can stray from the existing Niche and YASS code for the sake of presentation clarity.

3.1 Functional Code and Self-Management

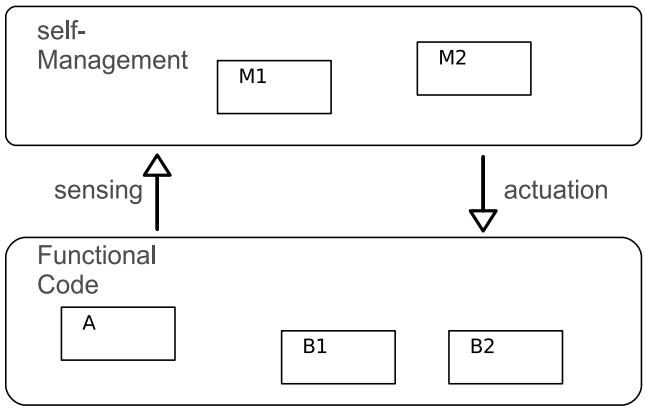


Figure 3.1: Application Architecture with Niche.

Niche is designed to support the development of self-managing behaviors of distributed applications. An application in the Niche framework contains a component-based implementation of the application's functional specification (the lower part of figure 3.1, with components `A`, `B1` and `B2`). During the development of this part of the application – we refer to it as the *functional code* thereafter – designers focus on algorithms, data structures and architectural patterns that fit the application's functional specification. The functional code can fulfill its purpose under a certain range of conditions in the environment such as availability of resources, user load and stability of computers hosting application components.

When the environment changes beyond the assumptions of the functional code, for instance when the user load becomes too high or an important application component fails, the application should either self-heal, self-configure, self-optimize or self-protect. These behaviors are commonly known as *self-* behaviors*, or *self-management*. The component-based implementation of application's self-* behaviors *senses* changes in the environment and adjusts the application architecture accordingly (the upper part of figure 3.1, with components `M1` and `M2`).

Niche provides APIs that allow the application developer to program deployment and management of application components, their interconnection, and also sensing state changes in environment and application components. The APIs provide *network-transparent* services, which means that the effects of an API method invocation are the same regardless where on the network the invocation took place. Niche implements a run-time infrastructure that aggregates computing resources on the network used to host and execute application components, which we discuss in more detail in Section 3.15.

3.2 Component-Based Functional Code

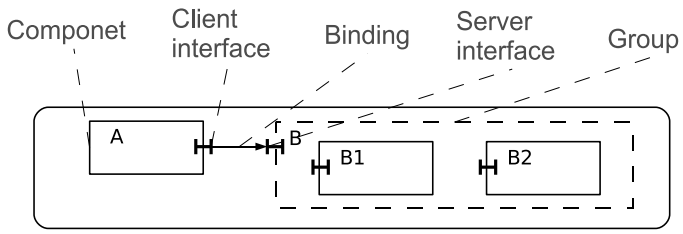


Figure 3.2: Functional Code in a Niche-based Application.

The functional part of application architectures is composed using *components* and *bindings*, see Figure 3.2, which are introduced in the Fractal component model and discussed in detail in Section 2.1. A Fractal com-

ponent contains code or sub-components, and its functionality is accessed through *server interfaces* which separate the component's implementation from other components using the component's functionality. A component's *client interface* manifests external services the component rely upon. A binding interconnects a client interfaces of one component with a server interface of another component. The Niche programming model introduces also *groups*. A Niche group represents a set of similar components and allows the designer to use them as it were one singe component. Niche groups can be exploited to improve application scalability and robustness, as discussed in detail in Section 3.12.

3.3 Application Self-Management by Control Loops

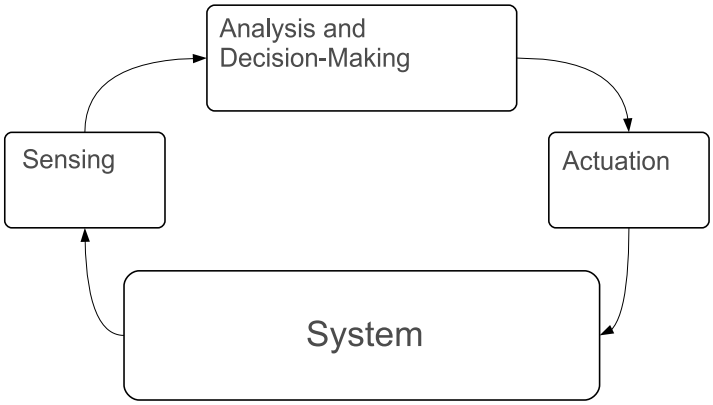


Figure 3.3: Management Control Loops.

*Self-** behaviours in Niche-based applications are implemented by *management control loops* we call just *control loops* when it is clear from the context. Control loops go through the *sensing*, *analysis and decision making*, and finally the *actuation* stages (see figure 3.3). The sensing stage obtains information about changes in the environment and components' state. The analysis and decision-making stage processes this information and decides on necessary actions, if any needed, to adjust the application architecture to the new conditions. During the actuation stage the application architecture is updated according to the decisions of the analysis and decision-making stage.

In a particular application there can be multiple control loops each controlling different kinds of application's self-* behaviors. These loops need in general to coordinate with each other in order to maintain the application's architecture consistently, which we address in Section 3.5.

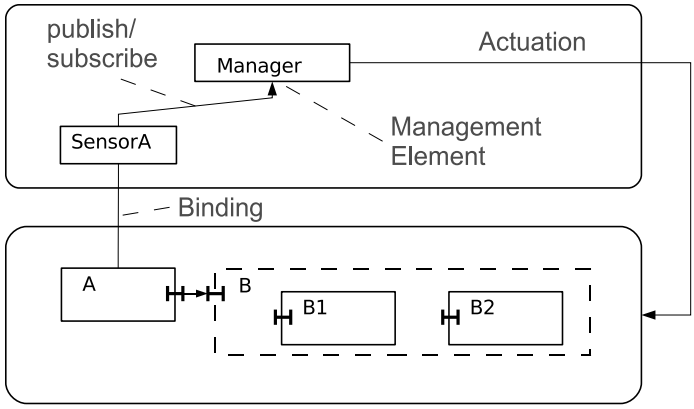


Figure 3.4: Self-Management in a Niche-based Application.

The sensing stages of Niche-based application's control loops are implmented by *sensors*. There are two types of sensors – application-specific sensors that provide information about status of individual components in the application, and sensors provided by the Niche run-time system that deliver information about the environment, such as notifications about component failures. Application-specific sensors are specific to individual component types, and implemented together with those components. At run-time, instances of application-specific sensor types can be dynamically deployed and attached to individual components. On Figure 3.4, `SensorA` is an application-specific sensor that provides necessary status information of component `A`.

The implementation of the analysis and decision-making stages of Niche-based application's control loops consists of *management elements* (MEs). MEs are stateful entities that process input *management events*, or just *events* thereafter (Section 3.8), according to their internal state, and can emit output events and/or manipulate the architecture using the Niche management *actuation* API implemented by Niche and introduced in this document. In general, in a Niche-based application there are multiple MEs that can be organized in different architectural patterns that are discussed in Section 3.5. On Figure 3.4, `Manager` is a management element.

Management events serve for communication between individual MEs and sensors that form application's management control loops. Management events are delivered asynchronously between MEs. MEs are *subscribed* to and receive input events from sensors and other MEs. Subscriptions can be thought of as bindings for unidirectional asynchronous communication for specific event types. Subscriptions are first-class entities that are explicitly manipulated using the Niche API, that is, the programmer designing the architecture of application's self-management explicitly creates subscriptions

between management elements to their sources of input events.

The Niche API provides functions for the actuation stage of application's management control loops. In particular, it provides for deployment of components of functional and self-management parts, and to interconnect those components.

3.4 Watchers, Aggregators and Managers

In a Niche-based application there can be multiple management loops implementing different kinds of self-* behaviors. Different loops should coordinate in order to ensure the coherency of architecture management, for which the coordination schemes discussed in Section 3.5 can be exploited. Individual management elements can participate in several control loops.

We distinguish the following roles of management elements that constitute the body of the analysis and decision-making stages of management control loops: *watchers*, *aggregators* and *managers*. In the simplest case, as on Figure 3.4, all roles can be performed by one single management element, but in a typical Niche application different roles are implemented by different MEs.

Watchers monitor the status of individual components and groups. Watchers are connected to and receive events from sensors that are either implemented by the programmer or provided by the management framework itself. Watchers provide also certain functionality that simplify programming of watching component groups, as discussed in Section 3.12. Watchers are intended to watch components of the same type, or components that are similar in some respect.

An aggregator is subscribed to several watchers and maintains partial information about the application status at a more coarse-grained level. There can be several different aggregators dealing with different types of information within the same control loop. Within an application as a whole there can be different aggregators acting in different management control loops.

Managers use the information received from different watchers and aggregators to decide on and actuate (execute) the changes in the architecture. Managers are meant to possess enough information about the status of the application's architecture as a whole in order to be able to maintain it. In this sense managers are different from watchers and aggregators where the information is though more detailed but limited to some parts, properties and/or aspects of the architecture. For example, in a data storage application a manager needs to know the current capacity, the design capacity and the number of online users in order to meet a decision whether additional storage elements should be allocated, while a storage capacity aggregator knows only the current capacity of the service, and different storage capac-

ity watchers monitor status and capacity of corresponding groups of storage elements.

3.5 Orchestration of Multiple Control Loops

In the Niche framework, application self-management can contain multiple management loops. The decisions made by different loops to change the architecture according to new conditions should be coordinated in order to maintain consistency of the architecture and to avoid its oscillation over time.

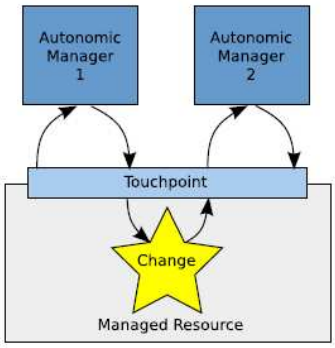


Fig. 1. The stigmergy effect.

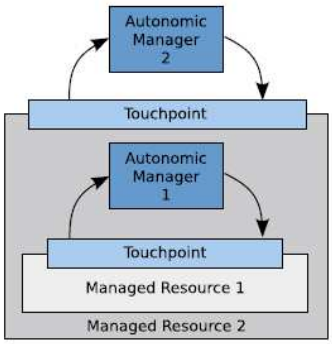


Fig. 2. Hierarchical management.

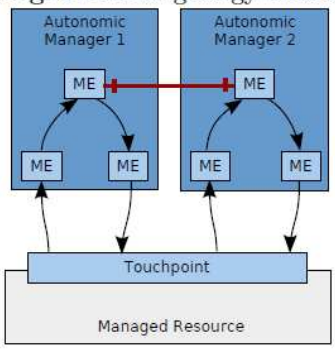


Fig. 3. Direct interaction.

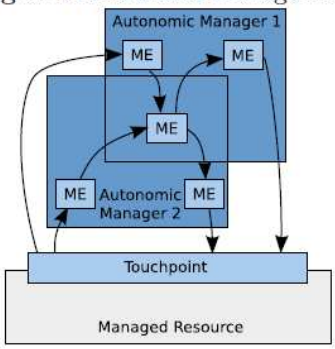


Fig. 4. Shared Management Elements.

Figure 3.5: Orchestration of Multiple Control Loops.

The following four methods can be used to coordinate the operation of several management control loops:

Stigmergy is a way of indirect communication and coordination between agents. Agents make changes in their environment, and these changes are sensed by other agents and cause them to do more actions. Stigmergy was first observed in social insects like ants. In our case agents

are control loops and the environment is the managed application. See Fig. 1 on Figure 3.5.

Hierarchical Management imply that some control loops can monitor and control other autonomic control loops (Fig. 2). The lower level control loops are considered as a managed resource for the higher level control loops. Higher level control loops can sense and affect the lower level ones.

Direct Interaction can technically be achieved by subscribing or binding appropriate management elements (typically managers) from different control loops to one another (Fig. 3). Cross control loop bindings can be used to coordinate them and avoid undesired behaviors such as race conditions and oscillations.

Shared Management Elements is another way of communication and coordination of different control loops (Fig. 4). Shared MEs can be used to share state (knowledge) and to synchronize actions by different control loops.

3.6 Scalability and Fault-Tolerance of Control Loops

Management elements can form hierarchical structures that improve scalability of self-management. Hierarchical structures can also facilitate hiding unnecessary details from higher-level management elements, thus simplifying design and maintainability of self-management code. In particular, lower-level watchers and aggregators can hide fine-grained details about individual components present in the system from higher-level management elements, in particular managers.

Application designers can also improve scalability and fault-tolerance of application self-management by distributing the responsibility of managing the application architecture over a group of sibling managers. The virtual synchrony approach for building scalable and fault-tolerant distributed systems [3, 4] can be used to achieve this: managers in the group can be programmed to receive all input events from all aggregators and watchers and thus be aware of the state of all other sibling managers, but each individual manager would maintain only the part of the architecture that is assigned to it.

It might be possible to improve fault-tolerance of self-management by deploying identical control loops, with some necessary coordination using e.g. the models outlined in Section 3.5. Fault-tolerance of self-management can be also improved by replicating individual management elements, as discussed in Section 3.14.

3.7 Management Elements and Sensors with Niche

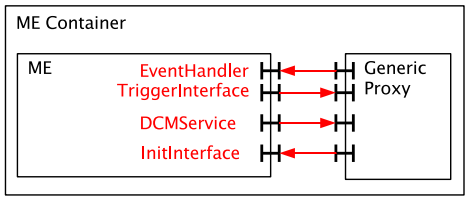


Figure 3.6: MEs in Niche.

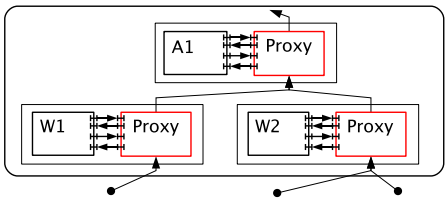


Figure 3.7: Composition of MEs.

Management Elements are programmed by the application developer as regular (centralized) Fractal components (Figure 3.6). MEs need to possess certain client and server interfaces, as explained below. Niche infrastructure provides for ME deployment and inter-ME communication; application developers do not need to explicitly program either of it. Note that MEs can be watched by watchers exactly as components implementing the application's functional specification.

The Niche run-time system hosts each ME in an *ME container* (Figure 3.6). ME containers are elements of the Niche infrastructure that enable operation of application-specific MEs. Specifically, an ME container provides an instance of the application-independent component called *ME Generic Proxy* with interfaces matching the interfaces of the ME.

The concepts of ME container and generic proxies is a part of the Niche implementation model; we present it here solely in order to provide some intuition behind Niche operation concerning MEs. ME containers are in particular important for reliable self-* behaviors achieved by means of replication of management elements, as discussed in Section 3.14.

In our Java-based Niche prototype, MEs are Fractal components implemented as Java classes according to the Java implementation of the Fractal framework. MEs can manipulate the application architecture using the Niche Java-based API provided through server interfaces of ME generic proxies. Niche infrastructure and application-specific ME components interact using certain data structures that identify elements of the application architecture, as discussed in Section 3.9. ME generic proxies provide for communication between MEs, see Figure 3.7. When a ME is deployed, Niche finds a suitable computer among those interconnected by Niche, creates the ME generic proxy and connects application-specific ME component to the proxy. Hosting and executing MEs is accounted to Niche processes executed on individual physical nodes. For example, in a Grid environment the members of a Virtual Organization (VO) would execute Niche processes on their computers. Note that components implementing the application's functional specification are hosted on first-class resources managed by dedicated resource management services, as discussed in Section 3.11. Niche

attempts to evenly balance the load of ME hosting.

Application-specific ME components can have the following client interfaces (see also figure 3.6):

- `TriggerInterface` interface with the `trigger` method used to emit events generated by the management element
- `DCMService` interface that provides Niche API for controlling functional and non-functional application components

Application-specific ME components need to provide the following server interfaces:

- `EventHandler` interface with the `eventHandler` method used when a management event arrives to the ME
- `InitInterface` interface used to (re)configure the management elements

The ME components can have further client and server interfaces which can be bound to functional and management components in the application, under certain restrictions discussed in Niche documentation.

Watchers receive information about status of components by means of component-specific sensors. Sensors are Fractal components. Individual types of sensors are designed to be bound to and receive status information from specific types of components in the application. Application developer designs sensors together with components they can sense. Sensor functionality is not integrated directly into the components because different sensors can be attached to the same component in different situations and at different points in time. Instead, components that need to be sensed implement a minimal interface that can provide enough information to sensors whenever needed. This facility in a component should not draw computing resources when not in use. When an application-specific sensor is deployed, it resides on the same node and interacts with the component through primitive Fractal bindings. In figure 3.8, sensor `Sensor A` is deployed for component `A`, and two instances of `Sensor B` are deployed for each of the components `B1` and `B2` from a group.

Sensors are deployed as a two-part structure similar to MEs, see figure 3.9. Sensors and `Sensor Generic Proxy` components provided by Niche interact through the following two interfaces. Application-specific sensor components can use the following client interface:

- `TriggerInterface` interface with the `trigger` method used to emit new events

and need to provide the following server interfaces :

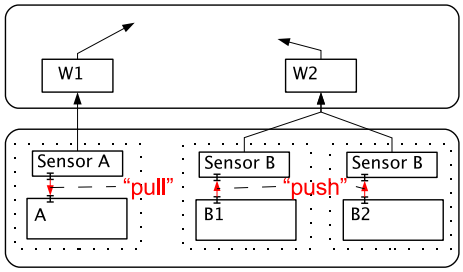


Figure 3.8: Structure of Application-Specific Sensors.

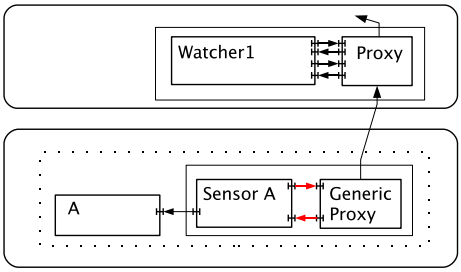


Figure 3.9: Composition of Application-Specific Sensors.

- `SensorInterface` interface used to control sensors

When a sensor is deployed, Niche locates the component for which the sensor is to be deployed, deploys both parts of the sensor appropriately and interconnects them (see figure 3.9). Using the facilities of the Fractal component model [7], application developer also specifies two lists of interfaces – for information "pull" and "push" between the sensor and the component being sensed (see figure 3.8), and Niche uses this information to connect the named application-specific sensor component interfaces to matching interfaces of the component being sensed.

3.8 Management Events

Sensors and management elements communicate asynchronously by means of *management events*, or just *events* thereafter. Events are objects of corresponding management event classes that are defined either by Niche itself, or are application-specific. When a subscription between a pair of sensors or MEs is being created, the subscription's management event type is identified by the event's class name.

Niche-specific events are generated by sensors implemented by Niche. These include in particular the `ComponentFailEvent` that identifies a failed component.

Applications define their own management event classes and generate corresponsing management events. In our current Java-based Niche prototype, events classes must be serializable. The Niche run-time system delivers events according to the established subscriptions.

3.9 Architecture Representation in Self-Management Code

Elements of the application architecture – components, bindings, groups and MEs – are identified by unique identifiers we call *Niche Id:s*, or just *Id:s* when it is clear from the context. Identifiers are unique in the scope of a Niche run-time infrastructure. For example, in a Grid environment the members of a Virtual Organization (VO) will usually run together a single instance of the Niche infrastructure, and different VOs will have separate infrastructures. MEs receive information about status of application architecture elements and manipulates them using the Id:s. Id:s of architecture elements are network-transparent, which allows application developers to design application architecture and its self-* behaviours independently of particular application deployment configurations. In our Java-based prototype of Niche, Id:s are represented in self-* code as Java objects of certain type. We discuss the implementation and performance characteristics of our Niche prototype in Section 3.15.

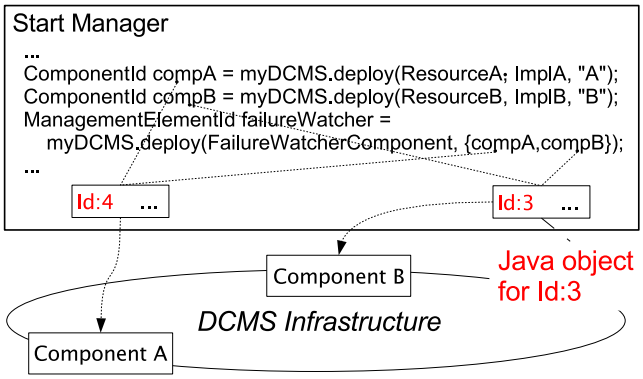


Figure 3.10: Application Architecture Representation in Self-* Code.

In figure 3.9 a snippet of self-* code from a ME called `StartManager` is presented. The code deploys two components and a "failure watcher" that oversees them. Note that this type of code fragments can be generated automatically by high-level tools, in particular by the Grid4All Application Deployment service. There are two components named `A` and `B` represented in self-* code by `compA` and `compB`, respectively. Id:s are introduced in self-* code by Niche API calls that deploy functional components and MEs. In figure 3.9, `compA` and `compB` are results of the `deploy` API calls that deploy components `A` and `B` implemented by Java classes `ImplA` and `ImplB` on resources `ResourceA` and `ResourceB`, respectively. Resource management is discussed in Section 3.11. Id:s are passed to Niche API invocations when

operations are to be performed on corresponding architecture elements, like deallocating a component. In the example, `compA` and `compB` are passed to the `deploy` Niche API call that deploys a watcher implemented by the Java class `FailureWatcherComponent`. `FailureWatcherComponent` watchers expects to receive Id:s of components to watch upon initialization.

Note that Id:s are *network-transparent*: multiple MEs on different nodes can access and manipulate the same architecture element by means of the element's Id. Niche API operations on Id:s have the same effect regardless of the location of the nodes with MEs issuing the operations, and the location of architecture elements identified by Id:s. In the example In figure 3.9, the `failureWatcher` ME will in general be deployed on a different physical node from the one where the `StartManager` is deployed itself, yet both MEs posses references to Niche Id Java objects representing `A` and `B`. Niche Id:s can also be included in application-specific management events passed between MEs, and thus used by the recipient MEs. Different physical nodes necessarily have different Java objects representing the same Niche Id, as discussed in Section 3.15.

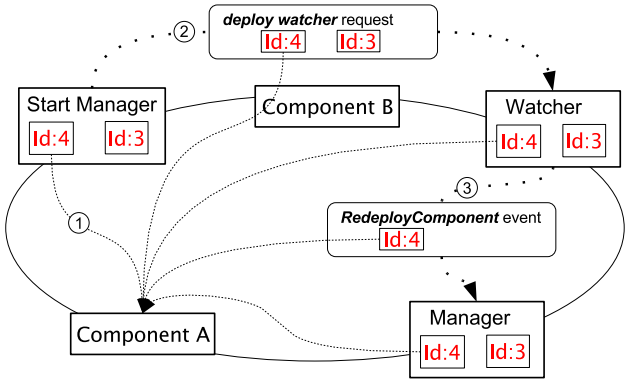


Figure 3.11: Sharing Niche Id:s between distributed MEs.

A possible sequence of actions and events is illustrated on Figure3.9. First, designated by (1), components `A` and `B` are deployed by the `StartManager`. Next (2), `StartManager` issues a request to deploy the `Watcher` ME that upon initialization obtains Id:s of `A` and `B`. Eventually (3), the watcher senses the failure of `A` (using Niche sensing - not illustrated on the Figure), and generates a `RedeployComponent` event that identifies `A` and is delivered to the `Manager` ME (the subscription is established beforehand by e.g. the same `StartManager`). At this point, `Manager` can update its architecture representation and/or perform management actions, like redeploying `A`.

We continue illustrating the manipulation of the application architecture in Section 3.10 using the code for initial deployment as an example.

3.10 Initial Deployment of Applications

```
DCMService myDCMS;
ComponentId compA = myDCMS.deploy(ResourceA, ImplA, "A");
ComponentId compB = myDCMS.deploy(ResourceB, ImplB, "B");
myDCMS.bind(compA, "clientInterfaceA",
            compB, "serverInterfaceB");
ManagementElementId manager =
   myDCMS.deploy(ManagerComponent, {compA, compB});
ManagementElementId aggregator =
   myDCMS.deploy(AggregatorComponent, {compA, compB});
(void) myDCMS.subscribe(aggregator, manager, "status");
(void) myDCMS.subscribe(compA, aggregator, "componentFailure");
(void) myDCMS.subscribe(compB, aggregator, "componentFailure");
```

Figure 3.12: Example of Self-Management Code with Niche.

Initial deployment of applications is performed using the Niche API. In the case when the initial architecture of the application's functional part is specified using an Architecture Description Language (ADL) specification as discussed in Section 2.3, the application deployment service interprets the ADL specification and invokes corresponding Niche API functions. An example sequence of commands executed by Niche is shown in figure 3.12, In this example, `myDCMS` is an object that provides the Niche API. The first `deploy` method deploys a component `A` implemented by `ImplA` on a resource `ResourceA`. We discuss the resource management in Section 3.11. `deploy` invocations contain also symbolic names of components in the application architecture, `A` and `B` in our example – this information is necessary in order to connect the ADL specification of application's initial architecture with the application's self-* architecture, as discussed later in this Section.

Next, the client interface on component `A` named `clientInterfaceA` is bound to the server interface `serverInterfaceB` on component `B`:

```
myDCMS.bind(compA, "clientInterfaceA",
            compB, "serverInterfaceB");
```

Next, the management element `manager` is deployed using the following method:

```
ManagementElementId manager =
   myDCMS.deploy(ManagerComponent, {compA, compB});
```

Here, the new manager `manager` will receive the list {`compA, compB`} as the argument for initialization. This argument is not interpreted by Niche itself. After initialization, the manager will possess the Id:s of `compA` and `compB` and therefore will be able to control these two components.

Finally, both MEs – `manager` and `aggregator` – are interconnected. The manager is subscribed to the aggregator for application-specific `status` events:

```
myDCMS.subscribe(aggregator, manager, "status");
```

The aggregator is subscribed for the predefined by Niche `componentFailure` environment sensing events that are generated by Niche when `compA` or `compB` fail:

```
myDCMS.subscribe(compA, aggregator, "componentFailure");
myDCMS.subscribe(compB, aggregator, "componentFailure");
```

The self-* architecture manipulates application components using their Id:s. If the initial deployment of the application is performed by the application deployment based on an ADL specification, the Id:s of the components are known to the deployment service and are not known initially to the self-* architecture. This problem is solved by means of a component registry that maps symbolic component names to their Id:s.

When the application is deployed by the deployment service, the symbolic names of components are taken from the ADL specification, and used as arguments to `deploy` methods as discussed in Section 3.10. As the side-effect of the deployment, Niche records the mapping, such that later on the management elements can obtain the component Id:s by the component symbolic names:

```
ComponentId compA = myDCMS.lookup("A");
```

3.11 Resource Management and Niche

Resource discovery and management for components implementing the application's functional specification is out of the scope of Niche. Niche is designed to be used together with one or several external resource management services. Deployment and management of resources for MEs is transparent to the application developer and is performed by Niche, as discussed in Section 3.7.

Applications, Niche and resource management services share the `NodeRef` and `ResourceRef` abstract data types. Objects of these types represent physical resources such as memory and CPU cycles.

Applications use the "discovery" request served by resource management services to discover free resources in the Niche infrastructure, see figure 3.13. Resource management services respond with `NodeRef` objects representing free resource(s) on a computing node available to the particular application. Resources discovered this way are not allocated to any application, in particular, a free resource discovered by an application can concurrently

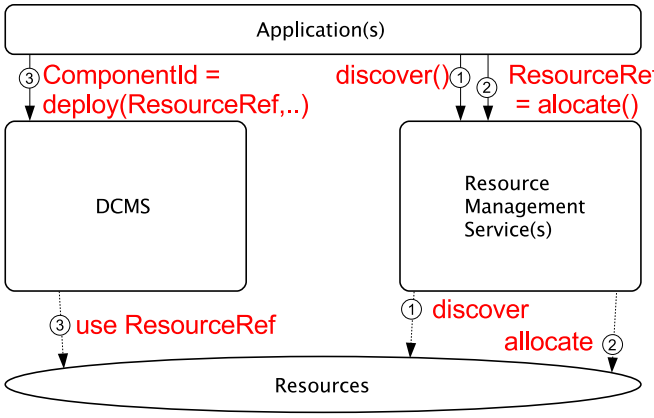


Figure 3.13: Resource Management with Niche.

be discovered and become used by another application. Applications can reserve a part or whole `NodeRef` resource for own usage by means of the "allocate" request to resource management services. If allocation fails due to activity of other applications, the application can try to allocate another previously discovered resource, or restart the discovery from scratch. The result of the "allocate" request is a `ResourceRef` object that represents a resource that is reserved for use by the calling application.

One of the arguments of the "deploy" Niche API function that provides for component deployment (see Section 3.10) is a `ResourceRef` object representing a resource to be used for deploying a particular component. Resources are "consumed" when applications deploy components, i.e. the same resource cannot be used for more than one "deploy" invocation, and for every component Niche verifies its resource usage with respect to properties of `ResourceRef`"s used for deployment of that component.

3.12 Groups and Group Sensing

Niche allows the programmer to group components together forming first-class entities called *Niche groups*, or just *groups* thereafter. Components can be bound to groups through *one-to-any* and *one-to-all* bindings, which is an extension of the Fractal model [7]. For functional code, a group of components acts as a single entity. Group membership management is provided by the self-* architecture and is transparent to the functional code. With a one-to-any binding, a component can communicate with a component randomly chosen at run-time from the given group. With a one-to-all binding, it will communicate with all elements of the group. Irrespective of type of bindings, the content of the group can be changed dynamically affecting

neither the component with binding's client interface (binding source), nor components in the group providing the binding server interfaces.

Niche are created by means of the `createGroup` Niche API call:

```
GroupId groupBId =
   myDCMS.createGroup({compB1, compB2}, {"groupServerInterface"});
```

The first argument is the initial list of components in the group, and the second argument is the list of server interfaces the group will provide. Server interfaces provided by a group can be used when making a binding to the group.

Once a group is created, the `GroupId` object can be used as a destination in binding construction call.

```
myDCMS.bind(compA, "clientInterfaceA",
            groupBId, "groupServerInterface",
            ONE_TO_ANY);
```

Here, the client interface `clientInterfaceA` of the component `compA` is bound to the `groupServerInterface` service interface provided by the group `groupBId`, using the one-to-any communication pattern.

Figure 3.14: Watching Groups in Niche-based Applications.

Groups can be watched by a single watcher. When a watcher for a group is being deployed, application programmer specifies the application-specific part of sensors to be used to generate sensing events for the watcher. Niche automatically deploys and removes sensors as the group membership changes. Subscriptions between watchers and dynamically deployed sensors are implicit and managed automatically by the Niche run-time system. In this sense group watchers are different from other management elements where the subscriptions for input management events are first-class and explicitly maintained by the application. The automatic management of sensors for group members is implemented using the SNR abstraction as described in Section 3.15. In figure 3.14, if the component `B2` is removed from the group `B`, then the corresponding sensor will be automatically removed from `B2`. Conversely, if a new component `B3` is added to `B`, then the sensor specified by the programmer for the watcher `W2` will be deployed on `B3`. Note that watchers can also watch other management elements, thus the self-* architecture can be designed to be self-* on its own.

To deploy a watcher and associate it with a group one needs to specify ManagementDeployParameters as follows:

```
params = new ManagementDeployParameters();
params.describeWatcher(watcherImpl, "watcherW",
```

```
                        initialArguments, groupId)
ManagementElementId watcher =
   myDCMS.deploy(ManagementDeployParameters params);
```

Here, the first line constructs a "parameter container" that is filled by the second line, specifying the Java class implementing the watcher (`watcherImpl`), the symbolic name of the ME for the component registry (`"watcherW"`), watcher's initial arguments, and the Id of the group to be watched (`groupId`). Finally, the watcher is deployed with the `deploy` Niche method.

The watcher, when initialized, must specify the sensor type that Niche will automatically deploy on components in the group:

```
myDeploySensorsInterface.deploySensor(sensorImpl,
          "sensorEvent", sensorParameters,
          clientInterfaces, serverInterfaces);
```

Here, `sensorImpl` is the Java class implementing the sensor, `"sensorEvent"` is the event generated by sensors and processed by the watcher, `sensorParameters` are parameters needed for sensor initialization, and the last two arguments are the lists of client and server interfaces used to by Niche to connect sensors to their components using "push" and/or "pull" sensing methods, as discussed in Section 3.7.

3.13 Controlling Location of Management Elements

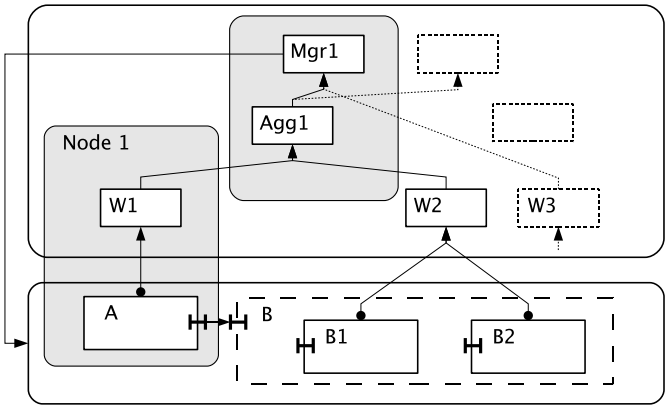


Figure 3.15: Co-location of MEs.

By default, for the sake of load balancing the Niche run-time system attempts to evenly distribute MEs on available computers. In order to reduce communication latency, application developers can control co-location

of MEs, see figure 3.15. ME deployment API calls allow the programmer to specify another architecture element so that the new element will always reside on the same node with the specified one (see also API section for "deploy"). Co-location of MEs can improve the performance of self-management and simplify handling of failures of nodes hosting management elements. However, this facility should be used only when necessary as excessive co-location of MEs can reduce fault-tolerance of application's self-* architecture.

3.14 Reliable Self-* Behaviours By Replication

The developer of a self-* architecture can request Niche to host some MEs such that to a certain degree the failures of nodes forming the Niche infrastructure is transparent to the execution of the selected MEs. This is achieved by means of replication of MEs: Niche creates several ME containers (introduced in Section 3.7) each hosting a replica of the ME, as discussed in more detail in Section 3.15.

In the simplest form, the programmer indicates that an ME should be replicated:

```
ManagementElementId manager =
   myDCMS.deploy(ManagerComponent, {compA}, REPLICATED);
```

After the deployment, the programmer can manage the set of replicas as one single ME. In particular, it can remove the whole set at once, and subscribe and unsubscribe it to sources and sinks of input and output events. Niche restores failed replicas automatically and transparently using the state of one of the alive ones.

In this simplest form, every replica receives input from all alive sources of events it is subscribed to. Individual replicas can learn their index in the replica set, and e.g. trigger output events only if the index is zero. Individial replicas in the set can fail, and it takes some time to restore them. Input events can be delivered in different order to different replicas, so the programmer must *not* assume that all replicas have the same internal state and produce the same output events in the same order. Instead, replicas should be programmed such that their internal states "self-converge" after a while in a fault-free run, by means of e.g. redundancy in input events and/or auxiliary (replicated) MEs acting as shared storage.

The programmer can also request replication of MEs with consistency guarantees using the `SYNCHRONIZED` flag of ME deployment method:

```
ManagementElementId manager =
   myDCMS.deploy(ManagerComponent, {compA}, SYNCHRONIZED);
```

In this case, Niche guarantees the same order of delivery of input events to such synchronized replicas, and exactly one output event is delivered from the set. This property is guaranteed also when a synchronized ME is restored after a node failure.

If the synchronized ME is programmed to be deterministic, i.e. its behaviour is completely determined by the ME's initial state and the sequence of input events, then individual elements in the ME set will pass the same sequence of internal state and produce the same sequence of output events. This approach of providing fault-tolerant services is known as state machine replication [14, 17].

The mechanism of synchronized MEs is a conceptually simple model for programming robust self-management architectures: it gives the programmer the illusion of hosting MEs on failure-free computers.

3.15 The Implementation Model of Niche

Niche implementation relies on structured overlay networking (SON), overlay thereafter. The overlay provides to Niche scalable and self-* address lookup and message delivery services. The overlay is used by Niche to implement bindings between components and message-passing between MEs, storage of architecture representation and also failure sensing.

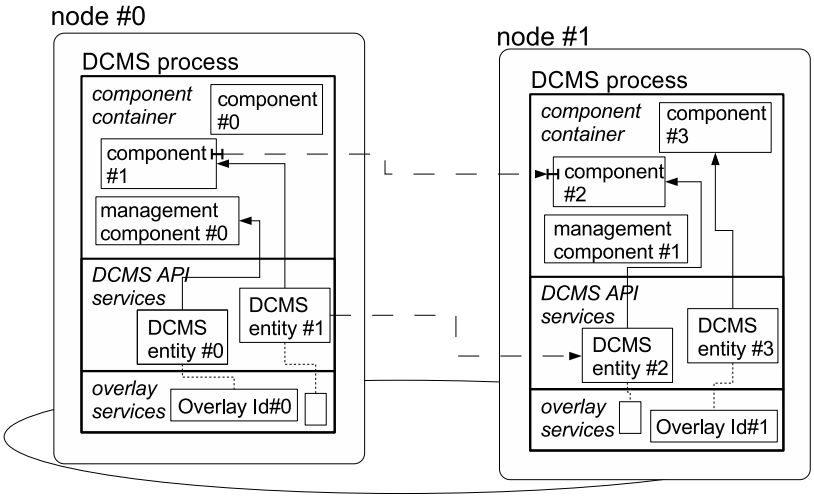


Figure 3.16: Niche infrastructure.

At runtime, Niche infrastructure integrates Niche container processes on several physical nodes using an overlay middleware, DKS [6, 8] in our current prototype. On each physical node there is a local Niche process that provides the Niche API to applications, see figure 3.16. The overlay allows to locate entities stored on nodes of the overlay. On the overlay, entities are

assigned unique overlay identifiers, and for each overlay identifier there is a physical node hosting the identified element. Such a node is usually called a "responsible" node for the identifier. Note that responsible nodes for overlay entities change upon churn. Every physical node on the overlay and thus in Niche also has an overlay identifier, and can be located and contacted using that identifier.

Niche maintains several types of entities we refer to as *Niche* or *DCMS entities*, in particular components of the application architecture and internal DCMS entities maintaining representation of the application's architecture. Niche entities are distributed on the overlay. For example, in figure 3.16 "DCMS entity #0" represents the management component #0. DCMS entities can have references between each other, direct or indirect. For example, in the figure the "DCMS entity #1" can refer to "DCMS entity #2" representing component #2, which is necessary to implement the binding between components #1 and #2. Functional components are situated on specified physical nodes, while MEs and entities representing the architecture might be moved upon churn between physical nodes.

DCMS entities are identified by *DCMS Id:s*. A DCMS Id contains an overlay identifier, "Overlay Id" in the figure 3.16, and a further local identifier. The local identifier allows Niche to distinguish multiple entities assigned to the same overlay identifier. Note that DCMS Id:s act as both unique identifiers and addresses of entities in Niche. In particular, DCMS entities representing components contain the overlay Id of the physical node that the component has been deployed on, and an identifier local to the node. The latter one is mapped by the Niche process to the physical address of the component (in our prototype, a Java object implementing the component). If no co-location constraints are specified for MEs and other DCMS entities, Niche tries to place them on different computers from the Niche infrastructure for the sake of load balancing, by means of assigning random overlay Id:s.

Figure 3.17 illustrates the important types of DCMS entities. There is a management element that deploys another management element with the identifier `Id:3` that we refer to as `ME/Id:3` thereafter. Using the identifier `Id:3`, the Niche process that deployed `ME/Id:3` on a remote node can access it for subsequent operations. `ME/Id:3` deals with a binding `Binding/Id:1`. The `Binding/Id:1` connects the component `Component/Id:6` to a group `Group/Id:5` that contains, among other, component `Component/Id:4`. Components are accessed in general through *references* (`ComponentRef/Id:2` and `ComponentRef/Id:8` in figure), which allows Niche to change the location of the component without affecting other elements of the application self-management.

If, say, `ME/Id:3` wants to perform an operation on the binding `Binding/Id:1`, it can either delegate the execution of the operation to the node where `Binding/Id:1` resides, or obtain and maybe also cache a replica of the bind-

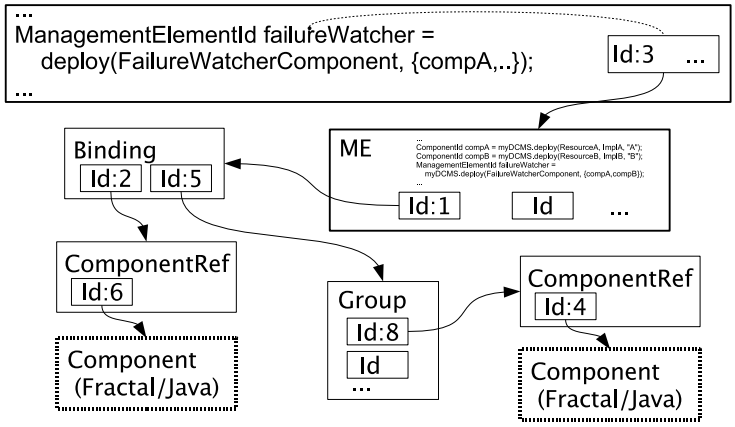


Figure 3.17: Id:s and References in self-* Architecture.

ing entity and execute the operation locally on the replica. For the same entity, some operations on it can be executed using a locally cached replica, while other operations can always require remote execution on the entity itself. Depending on the subset of entity operations that can be executed on a replica, Niche processes cache remote entities partially or entirely. Thus, by "replica" we mean an entity that represents a remote DCMS entity and contains enough information to support local execution of some operations on the entity. Specifically, a replica is not necessarily an verbatim copy of the entity. For instance, in our current Niche prototype a node invoking a binding can attempt to use the cached replica of the binding entity, while binding update is always executed on the binding entity itself. Niche detects invalid identifiers in cached DCMS entities and refreshes the cache contents automatically. Caching policy is determined by the Niche implementation and affects its performance.

Figure 3.18 depicts `ME/Id:3` caching the binding `Binding/Id:1`, the component reference `ComponentRef/Id:2` identifying the component with the binding's client interface, and the component reference `ComponentRef/Id:8` that identifies a member of the group `Group/Id:5`.
Dashed arrows depict the operation of the Niche caching mechanism: when Niche discovers that it possess a cached replica of an entity, then it will use it instead of accessing the remote entity itself.

If MEs or groups are co-located with other DCMS entities, their DCMS Id:s are assigned as follows: the overlay Id is taken from the DCMS Id of the entity to be co-located with, and a fresh local identifier is chosen.

Groups are implemented using *Set of Network References* (SNR) [6, 1] which is a primitive data abstraction that is used to associate a *name* with a set of *references*. SNRs can be thought of as DCMS component reference entities containing multiple references. A "one-to-any" or "one-to-all" bind-

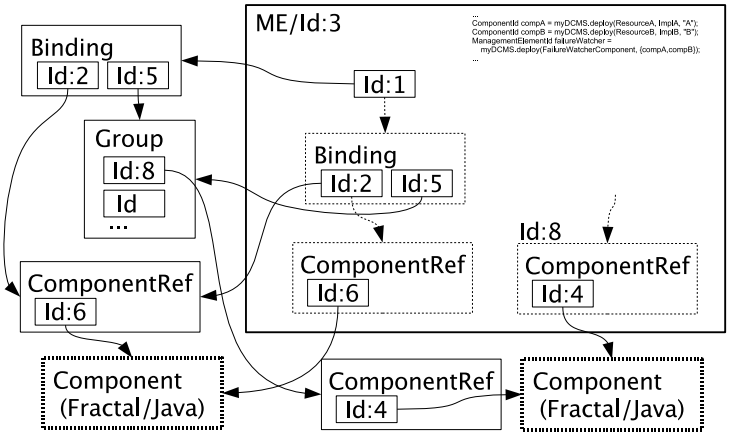


Figure 3.18: Caching of DCMS entities.

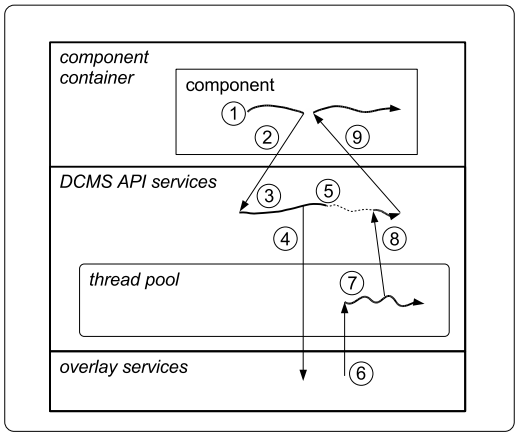


Figure 3.19: Threads of Control in Niche.

ing to a group means that when a message is sent through the binding, the group SNR's location is encapsulated in the group Id, and one or more of the group references from the SNR are used to send the message depending on the type of the binding. A group can grow or shrink transparently from group user's point of view. Finally SNRs support group sensing. Adding a watcher to a group causes deployment of sensors for each element of the group according to the group's SNR. Changing group membership transparently causes deployment/unemployment of sensors for the corresponding elements.

Niche API operations that involve DCMS entities located on remote nodes are implemented using Niche messages, which, in turn, use the underlying overlay network. Operations on a single remote entity without a

return value and without synchronization on completion requires a single Niche message which is delivered asynchronously with respect to the thread initiating the operation. Other types of operations involving a single remote entity require two consequent "request-response" Niche messages. If an operation on an entity involves sub-operations on some further entities, the total number of Niche messages for the operation increases correspondingly. Every Niche message requires an overlay address lookup operation that returns the address of an overlay node with the given overlay Id which is taken from the DCMS Id, and an overlay network message send operation to that address. Overlay address lookup operations usually require the time logarithmic to the size of the overlay, and results of lookup operations are cached by the overlay services layer in Niche processes.

Figure 3.19 illustrates operation of Niche processes. Thread (position 1 in figure) in an application component invokes a Niche operation (2) through the synchronous Niche interface. Niche handles the request (3) which can involve sending messages to remote Niche nodes (4) which, in turn, is handled by the overlay services. If a response message(s) is expected from a remote Niche node in order to complete the request, the thread (1,3) eventually blocks inside Niche (5). Response(s) from remote nodes (6) are handled by threads managed by the Niche thread pool (7). Response handler wakes up (8) the client thread (1,3) blocked inside Niche. Eventually the Niche operation finishes (9) and the thread in the application component (1) resumes the execution. Note that threads from the Niche pool never wait for incoming messages from remote nodes, and thus are never blocked except while waiting in the thread pool for a new request to handle.

The overlay services layer detects failures of other nodes in the Niche infrastructure when it fails to deliver to them pending messages. In this case, messages are handed back to the Niche layer that analyzes and handles the condition. For instance, when a one-to-any binding invocation fails because the destination component picked from the group has failed, Niche will attempt to pick another destination component from the group and repeat the operation. On the other hand, if a node with a resource to be used by a deployment operation has failed, Niche deployment operations fails and this condition is reported to the ME that invoked the operation.

The implementation model for synchronized MEs is presented in figure 3.20. ME generic proxies implement the inter-replica consensus algorithm that totally orders ME input events. One of the better-known algorithms solving this kind of consensus in unreliable environments – the Paxos protocol [13, 16] – has the latency of 3 messages, as opposed to 1 message latency for delivery of input events to non-synchronized and non-replicated MEs. Solutions that allow Paxos to deal with replicas that are restored after node failures [15] do not change the message complexity of Paxos in normal operation. A total-order broadcast algorithm achieving the latency of 2 messages is known [18], but it is unclear whether it can be adopted to

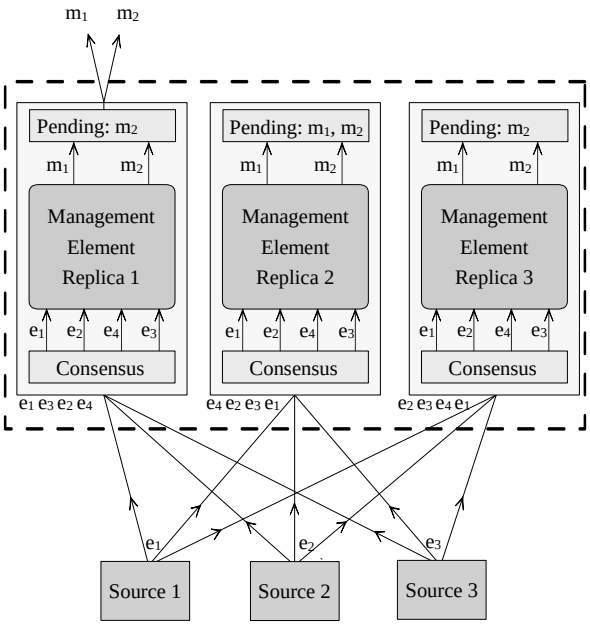


Figure 3.20: Replication of Synchronized MEs in Niche.

deal with replicas that are restored after node failures.

ME generic proxies contain also a queue of pending outgoing events and commands issued by MEs but not yet acknowledged by the recipients. Only the *primary* replica really sends out the events and commands. Acknowledgments are received by all replicas so that a secondary replica can resume exactly where the failure occurred.

In our prototype, the overlay services layer also maintains a pool of threads for managing incoming overlay messages. Threads in Niche API services and overlay services compete for common system resources.

Figure 3.21 illustrates Niche operation with the behaviour of a binding invocation. In figure, component 1 on node 0 is bound to component 2 on node 1. On node 0, Niche with the assistance of the component container created a binding stub – a special type of component with a matching server interface, so that component 1 is actually bound to the stub (position 1 in figure). When component 1 invokes its client interface, the implementation of the server interface in the binding stub calls Niche (2) to deliver the binding invocation to node 1. Niche retrieves the binding destination from the binding entity (3) or uses the cached binding replica, and sends the message to binding destination node (4). At the point, node 1 can associate the incoming request with the binding destination – server interface of component 2, and invokes it (5).

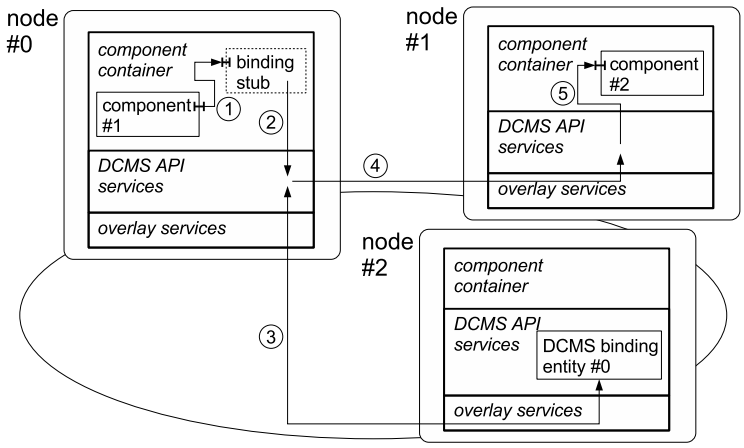


Figure 3.21: Bindings in Niche.

Chapter 4

DCMS API

4.1 Interfaces

4.1.1 Interface **NicheActuatorInterface**

The `NicheActuatorInterface` class

This class fills three purposes: - It acts as the interface class for Niche operations which are available only for management elements, while including the operations available to all components through extending NicheComponentSupportInterface - It acts as the interface class for proxies created by Jade - It gives access to primitive resource management services for systems and applications that do not provide those services themselves

Declaration

public interface NicheActuatorInterface
implements NicheComponentSupportInterface

Methods

- *allocate*
  `public ArrayList` **allocate**`( Serializable` **destinations**`, Object` **descriptions** `)`
  - **Usage**
    - Allocates a (part of a) discovered node, which is needed before deploying components
  - **Parameters**

    - `destinations` - Either a single ResourceId or an ArrayList<ResourceId>for bulk operation
    - `descriptions` - Either a single description or an ArrayList of descriptions for bulk operation. The format of allocate description will depend on the resource management being implemented
  - **Returns** - A list of the allocated resource identifiers, null if the operation could not be completed for the resource

---

- *cancelTimer*
  `public void` **cancelTimer**`( long` **timerId** `)`
  - **Usage**
    - Cancels a timer previously registered with registerTimer
  - **Parameters**
    - `timerId` -

- *deallocate*
  `public void` **deallocate**`( ResourceId` **resourceId** `)`
  - **Usage**
    - Frees a previously allocated resource
  - **Parameters**
    - `resourceId` - The reference to the resource which should be deallocated

---

- *deploy*
  `public ArrayList` **deploy**`( Serializable` **destinations**`, Serializable` **descriptions** `)`
  - **Usage**
    - Deploys one or more fractal components as specified by one or more component descriptions.
      As of now the code of the component to be deployed has to exist on the receiving computer.
  - **Parameters**
    - `destinations` - Either a single (allocated) ResourceId of an ArrayList<ResourceId>for bulk operation
    - `descriptions` - Either a single description or an ArrayList of descriptions for bulk operation. *Insert text from Nikos*
  - **Returns** - A list containing one or more global component ids

---

- *deployManagementElement*
  `public NicheId` **deployManagementElement**`( ManagementDeployParameters` **description**`, IdentifierInterface` **destination** `)`

- **Usage**
  - Deploys a management element as specified by a management element component description.
    As of now the code of the component to be deployed has to exist on the receiving computer.
- **Parameters**
  - `description` - A management element description *Insert text from Nikos*
  - `destination` - The reference to another management element, with which the new element should be collocated
- **Returns** - A management element id

- *discover*
  `public ArrayList` **discover**`( Serializable` **requirements** `)`
  - **Usage**
    - Method to ask the resource manager for currently free nodes matching the requirements
  - **Parameters**
    - `requirements` - The format of requirement description will depend on the resource management being implemented
  - **Returns** - A list of all nodes which can provide resources matching the requirements, null if none could be found

---

- *getComponentType*
  `public ComponentType` **getComponentType**`( String` **adlName** `)`
  - **Usage**
    - Returns the Jade component type corresponding to a given adl name. If the component type was not previously generated on the node where the method call is done, it will be generated on the first invocation. This therefore requires that the given adl name corresponds to an existing adl file.
  - **Parameters**
    - `adlName` -
  - **Returns** - A Jade component type object

- *getGroupTemplate*
  `public GroupId` **getGroupTemplate**`( )`
  - **Usage**

* Returns an 'empty' GroupId to be used as template when specifying which interfaces that should be automatically bound by the system when a component becomes member in a specific group.

- **Returns** - An empty GroupId

- *getId*
  `public NicheId` **getId**`( )`

- *getLogger*
  `public LoggerInterface` **getLogger**`( )`

  - **Usage**
    * Allows the applications/system developer to reuse the logging functionality already present in Niche
  - **Returns** - A reference to the Niche log4j-logger

- *oneShotDiscoverResource*
  `public NodeRef` **oneShotDiscoverResource**`( Serializable` **requirements** `)`

  - **Usage**
    * A shorthand to grab just one node matching the requirements
  - **Parameters**
    * `requirements` - The format of requirement description will depend on the resource management being implemented
  - **Returns** - The first found resource that matched the requirements, null if none could be found

---

- *redeployManagementElement*
  `public void` **redeployManagementElement**`( ManagementDeployParameters` **description**`, IdentifierInterface` **oldId** `)`

  - **Usage**
    * Redeploys a management element which has failed due to no, or insufficient replication
  - **Parameters**
    * `description` - A management element description
    * `destination` - The id of the failed ME to be recreated
  - **Returns** - A management element id

- *registerTimer*
  `public long` **registerTimer**`( EventHandlerInterface` **managementElement**`, Class` **eventClassName**`, int` **timerDelay** `)`

- **Usage**
  - Allows a management element to register a one-off timer. When the time delay has expired the eventHandler method of the management element will be called with a event of class eventClassName
- **Parameters**
  - `managementElement` - The management element which will be called when the timer goes off
  - `eventClassName` - The event which will be generated
  - `timerDelay` - The timer delay in milliseconds
- **Returns** - A timer id which is needed for cancellation

- *sendOnBinding*
  `public Object` **sendOnBinding**`( Object` **localBindId**`, Invocation` **invocation**`, ComponentId` **shortcut** `)`
  - **Usage**
    - Used by Jade-created interface proxies for one-way bindings. Semi-synchronous - propagates a method invocation and waits until the message is on the network.
  - **Parameters**
    - `localBindId` - The id of the proxy
    - `invocation` - The wrapped method invocation
    - `shortcut` - Gives the possibility to specify a specific receiver out of a group
  - **Returns** - Any object as specified by the interface description

---

- *sendWithReply*
  `public Object` **sendWithReply**`( Object` **localBindId**`, Serializable` **invocation** `)`
  - **Usage**
    - Used by Jade-created interface proxies for two-way bindings. Synchronous - propagates a method invocation and waits for a reply.
  - **Parameters**
    - `localBindId` - The id of the proxy
    - `invocation` - The wrapped method invocation
  - **Returns** - Any object as specified by the interface description

---

- *subscribe*
  `public Subscription` **subscribe**`( IdentifierInterface` **source**`, IdentifierInterface` **sink**`, String` **eventName** `)`
  - **Usage**

* Adds a new sink to the event generating management element source

– **Parameters**

* `source` - The id of the management element generating the events of intrest, or the id of the group defining the scope of interest in case of a subscription to an infrastructure event
* `sink` - The id of the management element interested in the event
* `eventName` - The full classname of the event

– **Returns** - A subscription which can be used to later change or stop the subscription

---

- *subscribe*
  `public Subscription` **subscribe**`( IdentifierInterface` **source**`, IdentifierInterface` **sink**`, String` **eventName**`, Serializable` **tag** `)`

  – **Usage**

  * Adds a new sink to the event generating management element source

  – **Parameters**

  * `source` - The id of the management element generating the events of intrest, or the id of the group defining the scope of interest in case of a subscription to an infrastructure event
  * `sink` - The id of the management element interested in the event
  * `eventName` - The full classname of the event
  * `tag` - Can be used to filter events based on the tag

  – **Returns** - A subscription which can be used to later change or stop the subscription

---

- *subscribe*
  `public Subscription` **subscribe**`( IdentifierInterface` **source**`, String` **sink**`, String` **eventName**`, IdentifierInterface` **sinkLocation** `)`

  – **Usage**

  * Adds a new sink to the event generating management element source, given that the sink is present

  – **Parameters**

  * `source` - The id of the management element generating the events of intrest
  * `sink` - The ADL name of the management element interested in the event

    * `eventName` - The full classname of the event
    * `sinkLocation` - The id of any element known to be collocated with the sink
  - **Returns** - A subscription which can be used to later change or stop the subscription, or null if the sink did not exist

---

- *testingOnly*
  `public NicheAsynchronousInterface` **testingOnly**`( )`

- *unsubscribe*
  `public boolean` **unsubscribe**`( Subscription` **subscription** `)`
  - **Usage**
    * Cancels a subscription
  - **Parameters**
    * `subscription` - The subscription specifying source, sink and event to stop listening to

---

- *update*
  `public void` **update**`( Object` **objectToBeUpdated**`, Object` **argument**`, int` **type** `)`
  - **Usage**
    * Generic method to update groups or management elements
  - **Parameters**
    * `objectToBeUpdated` - The id of the management element which should be updated
    * `argument` - The update message, or the item to add/remove, depending on the type
    * `type` - The type specifying the update operation, as given by the constants in *NicheComponentSupportInterface*

4.1.2 Interface **NicheComponentSupportInterface**

The `NicheComponentSupportInterface` class. Gives access to group management, (Niche wide) bind operations and primitive query support the query sytem state.

Declaration

```
public interface NicheComponentSupportInterface
```

FIELDS

- public static final int ADD_TO_GROUP
- public static final int ADD_TO_GROUP_AND_START
- public static final int REMOVE_FROM_GROUP
- public static final int REMOVE_GROUP
- public static final int GET_CURRENT_MEMBERS

METHODS

- *addToGroup*
  `public void` **addToGroup**`( Object` **newItem**`, Object` **groupId** `)`
  - **Usage**
    - Add a new component to an existing group
  - **Parameters**
    - `newItem` - A ComponentId representing the component to be added to the group. The component has to share the same interfaces as the previous group members, as it will be automatically become part of the existing bindings related to the group.
    - `groupId` - The id of the existing group

- *bind*
  `public BindId` **bind**`( Object` **sender**`, String` **senderInterface**`, Object` **receiver**`, String` **receiverInterface**`, int` **type** `)`
  - **Usage**
    - Binds the fractal client interface of 'client' to server interface of 'server'
  - **Parameters**
    - `client` - Normally a ComponentId. Can also be a GroupId, in which case bindings are created between all members of 'client' to the server
    - `clientInterface` - The ADL name of the client interface
    - `server` - Either a single ComponentId or a GroupId where all group members expose 'serverInterface'
    - `serverInterface` - The ADL name of the server interface

        * `type` - The type of the bindId: one-to-one, one-to-any, one-to-many, defined by constants in *currently* JadeBindInterface
    - **Returns** - A bindId id

- *bind*
  `public void` **bind**`( String` **senderInterface**`, Object` **receiver**`, String` **receiverInterface**`, int` **type** `)`
    - **Usage**
        * Binds the fractal client interface of the component calling the method to the server interface of 'server'
    - **Parameters**
        * `clientInterface` - The ADL name of the client interface
        * `server` - Either a single ComponentId or a GroupId where all group members expose 'serverInterface'
        * `serverInterface` - The ADL name of the server interface
        * `type` - The type of the bindId: one-to-one, one-to-any, one-to-many, defined by constants in *currently* JadeBindInterface
    - **Returns** - A bindId id

- *createGroup*
  `public GroupId` **createGroup**`( SNR` **template**`, ArrayList` **items** `)`
    - **Usage**
        * Creates a new group based on the given template and the components in the array list.
    - **Parameters**
        * `template` - A template which defines the interfaces which the group should manage upon membership changes
        * `items` - An array list of all components which should be part of the group. The components must have at least one interface in common.
    - **Returns** - A group id representing the new group

- *createGroup*
  `public GroupId` **createGroup**`( String` **templateName**`, ArrayList` **items** `)`
    - **Usage**
        * Creates a new group based on the template name and the components in the array list. This requires that the template has been previously created and registered with the template name

- **Parameters**
  - `templateName` - A template name which refers to a template which defines the interfaces which the group should manage upon membership changes
  - `items` - An array list of all components which should be part of the group. The components must have at least one interface in common.
- **Returns** - A group id representing the new group

- *getResourceManager*
  `public SimpleResourceManager` **getResourceManager**`( )`
  - **Usage**
    - Gives components access to the local resource manager, which can be used to get the component id based on the ADL name
  - **Returns** - The local resource manager

- *query*
  `public Object` **query**`( IdentifierInterface` **queryObject**`, int` **queryType** `)`
  - **Usage**
    - Generic query method to ask queries about elements in the system. The available query-types are as of now given as constants by this class
  - **Parameters**
    - `queryObject` - The id of the element which the query is concerning
    - `queryType` - The type specifying the query operation, as given by the constants in `NicheComponentSupportInterface`
  - **Returns** - The return type is dependent on the query - it can be a single object/identifierinterface or an arraylist of objects/identifierinterfaces

---

- *registerGroupTemplate*
  `public boolean` **registerGroupTemplate**`( String` **templateName**`, SNR` **template** `)`
  - **Usage**
    - Registers a group template to be used for subsequent group creation, where the user has specified which interfaces that the group should make available to any component bound to that group. Please note the created template name is only valid locally, for one node in the system
  - **Parameters**

    * `templateName` - A String representing the name of the template.
    * `template` - The group template with the interfaces of interest specified
  - **Returns** - True if the template was successfully registered, false if there already existed a template with that name

---

- *removeFromGroup*
  `public void` **removeFromGroup**`( Object` **item**`, Object` **groupId** `)`
  - **Usage**
    * Removes a component from an existing group
  - **Parameters**
    * `item` - A ComponentId representing the component to be removed from the group.
    * `groupId` - The id of the existing group

- *removeGroup*
  `public void` **removeGroup**`( GroupId` **gid** `)`
  - **Usage**
    * Removes an existing group. All watchers subscribed through that group will no longer be notified of changes to the previous group members, although the components themselves will remain unaltered
  - **Parameters**
    * `gid` - The id of the group to remove

- *unbind*
  `public void` **unbind**`( IdentifierInterface` **binding** `)`
  - **Usage**
    * Removes a previously established binding
  - **Parameters**
    * `binding` - The id of the binding to remove.

- *update*
  `public void` **update**`( Object` **objectToBeUpdated**`, Object` **argument**`, int` **type** `)`
  - **Usage**
    * Generic method to update groups or management elements. The available update-types are as of now given as constants by this class, but they might later be moved to the `DCMSInterface`

– **Parameters**
  * `objectToBeUpdated` - The id of the management element which should be updated
  * `argument` - The update message, or the item to add/remove, depending on the type
  * `type` - The type specifying the update operation, as given by the constants in `NicheComponentSupportInterface`

4.2 Interfaces

4.2.1 Interface **EventHandlerInterface**

The `EventHandlerInterface` class This interface must be implemented by all management elements that want to be able to subscribe to events, and have them delivered

Declaration

```
public interface EventHandlerInterface
```

Methods

- *eventHandler*
  `public void` **eventHandler**`( Serializable` **event**`, int` **flag** `)`

  - **Usage**
    - This method is invoked by the system when an event matching a previously done subscription is delivered.
  - **Parameters**
    - `event` - The event coming from one of the sources to which the ME is subscribed
    - `flag` - A flag which indicates whether the event has arrived normally (value zero) or during a period of churn, so that the ME has been moved or restored in between event creation and event delivery

4.2.2 Interface **InitInterface**

The `InitInterface` class
This interface must be implemented by all management elements to be properly initialized by the Niche framework. Please observe that the system gives no guarantees about the order which these methods are called upon ME creation

Declaration

```
public interface InitInterface
```

- *init*
  `public void` **init**`( dks.niche.interfaces.NicheActuatorInterface` **actuator** `)`
  - **Usage**
    - The system will invoke this method on a management element during its creation.
  - **Parameters**
    - `parameters` - An instance of the NicheActuatorInterface to be used by the ME

- *init*
  `public void` **init**`( Serializable [] parameters )`
  - **Usage**
    - The system will invoke this method on a management element during its creation, if it is being created for the first time as a result of managment deployment
  - **Parameters**
    - `parameters` - The initialArguments parameter of the ManagementDeployParameters instance used to deploy the ME
  - **See Also**
    - `ManagementDeployParameters`

- *initId*
  `public void` **initId**`( NicheId` **id** `)`
  - **Usage**
    - The system will invoke this method on a management element during its creation
  - **Parameters**
    - `parameters` - The NicheId of the ME being initialized

- *reinit*
  `public void` **reinit**`( Serializable [] parameters )`
  - **Usage**
    - The system will invoke this method on a management element during its creation, if it is being recreated after a churn event
  - **Parameters**

* `parameters` - The ME-parameters as given by the getAttributes method of the MovableInterface

– **See Also**

* `MovableInterface` (in 4.2.3, page 57)

4.2.3 Interface **MovableInterface**

The `MovableInterface` class Any management interface which wants support from the system to be automatically moved and redeployed upon churn needs to implement this interface.

Declaration

```
public interface MovableInterface
```

Methods

- *getAttributes*
  `public Serializable` **`getAttributes( )`**

  – **Usage**

  * The system will call this method on a management element which is about to be moved or copied.

  – **Returns** - An array of any parameters the management element is dependent on to be properly re-initialized. It is the responsibility of the ME designer to record the state in the way expected by the re-init method of the same ME class.

  – **See Also**

  * `InitInterface` (in 4.2.2, page 55)

4.2.4 Interface **TriggerInterface**

The `TriggerInterface` class This interface is used by management elements that want to be able to trigger events

Declaration

```
public interface TriggerInterface
```

- *removeSink*
  `public void` **removeSink**`( String` **sinkId** `)`

- *trigger*
  `public void` **trigger**`( Serializable` **event** `)`

  - **Usage**
    - Triggers an event
  - **Parameters**
    - `event` - The event, which will be matched against current subscriptions. All management elements which has subscribed to the triggering element for that type of event will get notified

- *trigger*
  `public void` **trigger**`( Serializable` **event**`, Serializable` **tag** `)`

  - **Usage**
    - Triggers an event with a special tag for filtering
  - **Parameters**
    - `event` - The event, which will be matched against current subscriptions. If there are subscriptions matching the event type, they will also be checked against the tag.
    - `tag` -

- *triggerAny*
  `public void` **triggerAny**`( Serializable` **event** `)`

  - **Usage**
    - Triggers an event
  - **Parameters**
    - `event` - The event, which will be matched against current subscriptions. Out of the matching subscriptions, one random subscriber will get notified

4.3 Classes

4.3.1 Class ManagementDeployParameters

The `ManagementDeployParameters` class. An instance of the class is needed as parameter for the call to `deploy` provided by `DCMS`.

To deploy management elements, the element specific methods `describe...` should be used.

Declaration

public class ManagementDeployParameters
extends Object
implements Serializable

Constructors

- *ManagementDeployParameters*
  `public` **ManagementDeployParameters**`( )`
  - **Usage**
    - Standard empty constructor

Methods

- *bind*
  `public void` **bind**`( String` **clientComponentName**`, String` **clientInterfaceName**`, String` **serverComponentName**`, String` **serverInterfaceName** `)`
  - **Usage**
    - Allows the user to specify local bindings which should be initiated at component deploy time
  - **Parameters**
    - `clientComponentName` - Local component ADL name
    - `clientInterfaceName` - Local component client interface name
    - `serverComponentName` - Local component ADL name
    - `serverInterfaceName` - Local component server interface name

- *deployComponent*
  `public void` **deployComponent**`( String` **ADL**`, String` **componentName**`, ComponentType` **componentType**`, .Map` **context** `)`
  - **Parameters**
    - `ADL` - The ADL file name containing the component description
    - `componentName` - The new component name. If null then the name in the ADL will be used
    - `context` - used for example to set attribute=value

- *deployComponent*
  `public void` **deployComponent**`( String` **ADL**`, String` **managementElementName**`, ComponentType` **componentType**`, .Map` **context**`, int` **type**`, Serializable []` **initialArguments**`, boolean` **reliable**`, boolean` **movable**`, boolean` **start**`, NicheId` **managedComponentId** `)`
  - **Parameters**
    - `ADL` - The ADL file name containing the component description
    - `componentName` - The new component name. If null then the name in the ADL will be used
    - `context` - used for example to set attribute=value
    - `type` - used for the framework to automatically bind the management element depending on type

---

- *describeAggregator*
  `public void` **describeAggregator**`( String` **className**`, String` **componentName**`, ComponentType` **componentType**`, Serializable []` **initialArguments** `)`
  - **Usage**
    - Describes an aggregator to be deployed
  - **Parameters**
    - `className` - The class name of the java class file implementing the management element
    - `componentName` - The new component name. If null then the name from the ADL will be used
    - `initialArguments` - An array of initial arguments to be passed to the management element init method

---

- *describeExecutor*
  `public void` **describeExecutor**`( String` **className**`, String` **componentName**`, ComponentType` **componentType**`, Serializable [] initialArguments, NicheId` **actuatedComponentId** `)`

- *describeManager*
  `public void` **describeManager**`( String` **className**`, String` **componentName**`, ComponentType` **componentType**`, Serializable [] initialArguments` `)`
  - **Usage**
    - Describes a manager to be deployed
  - **Parameters**
    - `className` - The class name of the java class file implementing the management element
    - `componentName` - The new component name. If null then the name from the ADL will be used
    - `initialArguments` - An array of initial arguments to be passed to the management element init method

---

- *describeSensor*
  `public void` **describeSensor**`( String` **className**`, String` **componentName**`, Serializable [] initialArguments` `)`
  - **Usage**
    - Describes a sensor to be deployed. This deployment can only be done by the responsible watcher
  - **Parameters**
    - `className` - The class name of the java class file implementing the sensor
    - `componentName` - The new component name.
    - `initialArguments` - An array of initial arguments to be passed to the sensor init method

---

- *describeWatcher*
  `public void` **describeWatcher**`( String` **className**`, String` **componentName**`, ComponentType` **componentType**`, Serializable [] initialArguments, NicheId` **watchedComponentId** `)`
  - **Usage**
    - Describes a watcher to be deployed
  - **Parameters**
    - `className` - The class name of the java class file implementing the management element

        * `componentName` - The new component name. If null then the name from the ADL will be used
        * `initialArguments` - An array of initial arguments to be passed to the management element init method
        * `watchedComponentId` - The new id of the component, or group, with which the watcher is associated

---

- *getReInitParameters*
  `public Serializable` **getReInitParameters**`( )`

- *getType*
  `public int` **getType**`( )`

- *keepAlive*
  `public boolean` **isReliable**`( )`
    - **Returns** - Tells whether the new component is declared to be reliable

---

- *lifeCycle*
  `public void` **lifeCycle**`( String` **componentName**`, boolean` **start** `)`
    - **Usage**
        * The method can be used together with the 'deploy' settings to specify whether the component should be started directly after deployment. It can also be used on its own to remotly start an already deployed component
    - **Parameters**
        * `componentName` - The component name you want to start or stop
        * `start` - true to start it & false to stop it

- *setAttributes*
  `public void` **setAttributes**`( ArrayList` **attributes** `)`

- *setAttributes*
  `public void` **setAttributes**`( String` **componentName**`, String` **controllerName**`, .Map` **attributes** `)`
    - **Usage**
        * Method used to specify initial attribute values of component attributes. Requires the component to implement corresponding attribute controller.
    - **Parameters**
        * `componentName` - Component ADL name

* `controllerName` - Classname of component attribute controller
* `attributes` - A map specifying pairs

- *setReInitParameters*
  `public void` **setReInitParameters**`( Serializable [] param )`

- *setReliable*
  `public void` **setReliable**`( boolean  reliable )`

  - **Usage**
    * Specifies whether the new component should be reliable, that is if the runtime system should keep the element replicated despite churn

Figure 5.1: YASS Functional Part

Chapter 5

DCMS Use Case: YASS

5.1 Architecture

We have designed and developed YASS – "yet another storage service" – as a way to refine the requirements of the management framework, to evaluate itand to illustrate its functionality. YASS stores, reads and deletes files on a set of distributed resources. The service replicates files for the sake of robustness and scalability. We target the service for dynamic Grid environments, where resources can join, gracefully leave or fail at any time. YASS automatically maintains the file replication factor upon resource churn, and scales itself based on the load on the service.

5.1.1 Application functional design

A YASS instance consists out of *front-end components* which are deployed on user machines and *storage components* Figure 5.1. Storage components

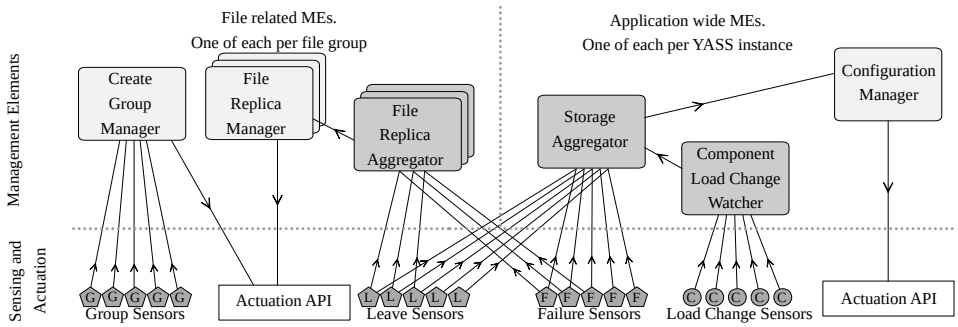


Figure 5.2: YASS Non-Functional Part

are composed of *file components* representing files. The ovals in Figure 5.1 represent resources contributed to a Virtual Organization (VO). Some of the resources are used to deploy storage components, shown as rectangles.

The service contain two types of groups, storage group and file group. The Storage group containing all storage components and file group containing all replicas of a specific file. Each instance of YASS have only one storage group and several file groups depending on the number of stored files (one file group per stored file).

A user store request is sent (using one-to-any binding between the front-end and the storage group) to an arbitrary storage component that will try to find some r different storage components, where r is the file's replication degree, with enough free space to store a file replica. These replicas together will form a *file group* containing the r dynamically created new file components. The user will then use a one-to-all binding to the file group to send the file in parallel to the r replicas in the file group. Read requests can be sent to any of the r file components in the group using the one-to-any binding between the front-end and the file group. Similarly, a delete request is sent to all file components using one-to-all binding between the front-end and the file group.

5.1.2 Application non-functional design

Configuration of application self-management. The Figure 5.2 shows the architecture of the watchers, aggregators and managers used by the application.

Associated with the group of storage components is a system-wide Storage-aggregator created at service deployment time, which is registered to leave- and failure-events which involve any of the storage components. It is also registered to a Load-watcher which triggers events in case of high system load. The Storage-aggregator can trigger StorageAvailabilityChange-events, which the Configuration-manager is subscribed to.

When new file-groups are formed by the functional part of the appli-

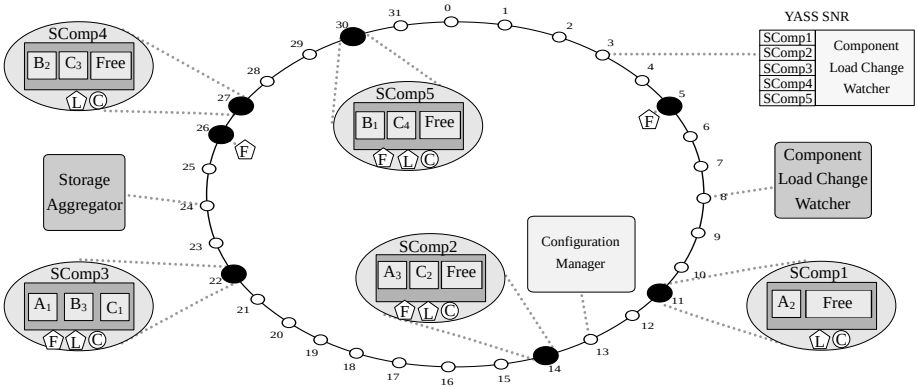


Figure 5.3: Parts of the YASS application deployed on the management infrastructure.

cation, the management infrastructure propagates group-creation events to the CreateGroup-manager which initiates a FileReplica-aggregator and a FileReplica-manager for the new group. The new FileReplica-aggregator is subscribed to resource leave- and resource fail-events of the resources associated with the new file group.

Before explaining these management elements in more details we'll present an example that will help in better understanding.

5.1.3 Test-cases and initial evaluation

The infrastructure has been initially tested by deploying a YASS instance on a set of nodes. Using one front-end a number of files are stored and replicated. Thereafter a node is stopped, generating one fail-event which is propagated to the Storage-aggregator and to the FileReplica-aggregators of all files present on the stopped node. Below is explained in detail how the self-management acts on these events to restore desired system state.

Figure 5.3 shows the management elements associated with the group of storage components. The black circles represent physical nodes in the P2P overlay Id space. Architectural entities (e.g. SNR and MEs) are mapped to ids. Each physical node is responsible for Ids between its predecessor and itself including itself. As there is always a physical node responsible for an id, each entity will be mapped to one of the nodes in the system. For instance the *Configuration Manager* is mapped to id 13, which is the responsibility of the node with id 14 which means it will be executed there.

Application Self-healing. Self-healing is concerned with maintaining the desired replica degree for each stored item. This is achieved as follows for resource leaves and failures:

Resource leave. An infrastructure sensor signals that a resource is about to leave. For each file stored at the leaving resource, the associated FileReplica-aggregator is notified and issues a replicaChange-event which is forwarded

to the FileReplica-manager. The FileReplica-manager uses the one-to-any binding of the file-group to issue a FindNewReplica-event to any of the components in the group.

Resource failure. On a resource failure, the FileGroup-aggregator will check if the failed resource previously signaled a ResourceLeave (but did not wait long enough to let the restore replica operation finish). In that case the aggregator will do nothing, since it has already issued a replicaChange event. Otherwise a failure is handled the same way as a leave.

Application Self-configuration. With self-configuration we mean the ability to adapt the system in the face of dynamism, thereby maintaining its capability to meet functional requirements. This is achieved by monitoring the total amount of allocated storage. The Storage-aggregator is initialized with the amount of available resources at deployment time and updates the state in case of resource leaves or failures. If the total amount of allocated resources drops below given requirements, the Storage-aggregator issues a storageAvailabilityChange-event which is processed by the Configuration-manager. The Configuration-manager will try to find an unused resource (via the external resource management service) to deploy a new storage component, which is added to the group of components. Parts of the Storage-aggregator and Configuration-manager pseudocode is shown in Listing 5.1, demonstrating how the stateful information is kept by the aggregator and updated through sensing events, while the actuation commands are initiated by the manager.

Application Self-optimization. In addition to the two above described test-cases we have also designed but not fully tested application self-optimization. With self-optimization we mean the ability to adapt the system so that it, besides meeting functional requirements, also meets additional non-functional requirements such as efficiency. This is achieved by using the ComponentLoad-watcher to gather information on the total system load, in terms of used storage. The storage components report their load changes, using application specific load sensors. These load-change events are delivered to the Storage-aggregator. The aggregator will be able to determine when the total utilization is critically high, in which case a StorageAvailabilityChange-event is generated and processed by the Configuration-manager in the same way as described in the self-configuration section. If utilization drops below a given threshold, and the amount of allocated resources is above initial requirements, a storageAvailabilityChange-event is generated. In this case the event indicates that the availability is higher than needed, which will cause the Configuration-manager to query the ComponentLoad-watcher for the least loaded storage component, and instruct it to deallocate itself, thereby freeing the resource. Parts of the Configuration-manager pseudocode is shown in Listing 5.2, demonstrating how the number of storage components can be adjusted upon need.

Listing 5.1: Pseudocode for parts of the Storage-aggregator

```
upon event ResourceFailure(resource_id) do
   amount_to_subtract = allocated_resources(resource_id)
   total_storage = total_amount - amount_to_subtract
   current_load = update(current_load, total_storage)
   if total_amount < initial_requirement or current_load > high_limit then
      trigger(availabilityChangeEvent(total_storage, current_load))
   end
```

Listing 5.2: Pseudocode for parts of the Configuration-manager

```
upon event availabilityChangeEvent(total_storage, new_load) do
   if total_storage < initial_requirement or new_load > high_limit then
      new_resource = resource_discover(component_requirements, compare_criteria)
      new_resource = allocate(new_resource, preferences)
      new_component = deploy(storage_component_description, new_resource)
      add_to_group(new_component, component_group)
   elseif total_storage > initial_requirement and new_load < low_limit then
     least_loaded_component = component_load_watcher.get_least_loaded()
     least_loaded_resource = least_loaded_component.get_resource()
     trigger(resourceLeaveEvent(least_loaded_resource))
   end
```

5.2 Implementation

5.2.1 The Component Load Sensor

The component load sensor is implemented in `yass.sensors.LoadSensor`. It is responsible for monitoring the load of a storage component. Sensors can be *push* and/or *pull* style. The load sensor is a push sensor. When a file is added to or removed from a storage component, the new load is *pushed* by the storage component to the associated load sensor. This is done using `yass.sensors.LoadChangeInterface`. The `LoadSensor` triggers a `ComponentStateChangeEvent` when the storage component load changes with a predefined *delta*.

5.2.2 The Component Load Watcher

The component load watcher is implemented in `yass.watchers.LoadWatcher`. It is subscribed to all `yass.sensors.LoadSensor` sensors. Currently the load watcher only forwards the `ComponentStateChangeEvent` triggered by the load sensors.

The watcher must specify the sensor that it requires to be able to watch component(s). For example the component load watcher requires the component load sensor to be deployed to be able to watch storage components. It is the responsibility of the infrastructure to deploy the sensor and bind it with the watched component and do the subscriptions for the events generated by the sensor.

The sensor specification is done when initializing the watcher in `init()`. It includes the following:

- Sensor class name.
- Event class name that is generated by the sensor.
- Any parameters needed to initialise the sensor (delta for load sensor).
- Name of fractal client interface(s) used by pull sensors. A server interface with the same name must exist on the watched component.
- Name of fractal server interface(s) used by push sensors. A client interface with the same name must exist on the watched component.

Here is how the load watcher specifies the load sensor:

```
deploySensor.deploySensor("yass.sensors.LoadSensor",
   "yass.sensors.LoadSensorEvent", sensorParameters,
   null, new String[] { "pushLoadChange" });
```

5.2.3 The Storage Aggregator

The storage aggregator is implemented in `yass.aggregators.StorageAggregator`. It is responsible for keeping the state of the system. When a resource fail or leave it remove the amount of storage that was allocated on the leaving resource and if the remaining storage is less than a predefined threshold it informs the ConfigurationManager through a `StorageAvailabilityChangeEvent`. When a new storage component joins the storage group the storage aggregator adds its storage capacity to the total storage. Currently the Component State Change event is not handled.

5.2.4 The Configuration Manager

The configuration manager is implemented in `yass.managers.ConfigurationManager`. It is registered to `StorageAvailabilityChangeEvent` from the storage aggregator. Upon receiving the event it checks if the load is higher than a threshold or the total capacity is less than the minimum threshold and if so it tries to find new resource to deploy a storage component on it.

5.2.5 The File Replica Aggregator

The file replica aggregator is implemented in `yass.aggregators.FileReplicaAggregator`. It monitors the members of the associated file group for resource leave and failure by subscribing to corresponding resource fail and leave sensors.

When a watched resource fails/leaves, it triggers a `ReplicaChangeEvent` indicating that action must be taken to restore the replication degree.

5.2.6 The File Replica Manager

The file replica manager is implemented in `yass.managers.FileReplicaManager`. When it receive a `ReplicaChangeEvent` it starts to find a new replica by sending a `ReplicaRestoreRequest` to any of the remaining members of the file group.

5.2.7 The Create Group Manager

The create group manager is implemented in `yass.managers.CreateFileGroupManager`. It is subscribed to the `CreateGroupEvent` which is an infrastructure event. The purpose of this manager is to detect when the functional part (storage components) creates a new file group. When a new file group is detected, the create group manager will create a new file replica aggregator and manager for the new file group.

5.2.8 The Start Manager

The start manager is implemented in `yass.managers.StartManager`. This is called at the deployment time. The purpose of this manager is to handle operations that are not yet implemented in the ADL.

5.3 Installation

YASS requires Jade and Niche to be installed and configured properly. In this section we will guide you, step by step, to prepair your environment to run and/or develop applications based on the DCMS.

Download

You will need Jade, Niche, and YASS:

- If you are interested only in running YASS then it is enough for you to get Jade because we already added the JAR files for Niche and YASS in Jade. You can get Jade from the SVN repository at

  `https://gforge.inria.fr/projects/grid4all/` you will find it under `wp1/Jade`.

- You can also checkout the source code of YASS from the same repository under `wp1/YASS`.

- The Niche source code can be found at

  `svn://korsakov.sics.se/dks/`

The Web Cache

The DCMS uses the Niche/DKS [6, 8], an overlay middleware built on the DKS structured overlay network. Any new resource/node that wants to join an existing overlay must first contact a node already inside the overlay to be able to join. The only exception is the first node which created the overlay.

The problem now is how will a new node learn about the reference (ip/-port) of an existing node? The solution used is a simple web page that contains the references to the most recent nodes in the overlay. We assume that the URL of this page is known by the nodes that wants to join the overlay repsented by the page. We call this web page the web cache.

For each overlay you need one web cache. Two different overlays must use different web caches. This web cache is used to cache references to some nodes that are part of the overlay. We assume that all nodes know the URL of the web cache. This is configurable through the `Jade/etc/dks/dksParam-.prop` file as described in the next section. The first node that creates the overlay resets the content of the web cache and puts a reference to itself. Other nodes that join this overlay will use the web cache to get a reference to a node already in the overlay. The new joining node will then use this reference to join the overlay and will also add a reference to itself in the web cache.

To setup the web cache you will need a web server that can run PHP. You can use Apache web server for example. Apache comes with most Linux distributions. For Windows you can use for example www.easyphp.org. After installing your web server copy the `Jade/webcache/` folder to your `www` root folder.

Configuration

The configuration files can be found in the `etc/` folder under the Jade source tree. The `Jade/etc/dks/dksParam.prop` file is used to set the "Arity" K and the number of "Levels" L for DKS routing. The ID space is $N = K^L$. The values for K and L must be the same for all nodes in the same overlay. This file is also used to specify the default IP and port number of each node in the overlay but can be changed in the code. The address of the web cache is also specified in this file. All nodes in a single overlay must share the same web cache. You might need to edit this file and set these values.

Search for `<Path to Jade>` at `Jade/etc` (including sub folders) and replace it with the correct path.

Go to `Jade/etc/oscar/` and edit `bundle-jadeboot.properties` and `bundle-jadenode.properties` to set appropriate values for `jadeboot.registry.host` and `jadeboot.discovery.host`. The JadeBoot is the node that starts Jade and the overlay and currently this node can not fail!

If you are Disconnected from the Internet

In this case you'll have to copy `Jade/etc/www/repo.jasmine/` folder to your `www` root. Then search the `Jade/etc/oscar/` folder and the `Jade/etc/execute.properties` file for the text `repo.jasmine` and replace the remote url with your local one.

For Developers Only

If you want to modify Niche/DKS you must place the new dks.jar file in the `Jade/external` folder. If you modify YASS you must place the new yass.jar file in the `Jade/external` folder and if you modified the fractal architecture then you must place the new `yass.fractal` file in the `jade/examples` folder.

If you want to run a new application you must add the JARs to `Jade/external` and the .fractal file to `Jade/examples`. Then search the Jade source tree for the text "yass.jar" and add similar lines for your application. Finally to run it (described in next section) the simple way is to add ant targets for your application similar to "testG4A-Yass-Deploy" and "testG4A-Yass-Start".

5.4 Running YASS

Compile Jade using the ant file `Jade/build-src.xml` default target. Then to start the Niche/Jade system you will need a boot node and several nodes (depending on the number of nodes you need). You'll need at least one boot and four nodes to run the demo. You can use the `Jade/build.xml` ant file. Use the `jadeboot` target to start a boot node then use the `jadenode` target several times to create as much nodes as you need. Alternatively you can use the `testG4A-init2N` target to start a boot and a node or the `testG4A-init4N` target to start a boot and three nodes.

For testing it is easier to first try to run the nodes on the same machine just to make sure that everything is working.

After starting the Niche/Jade system you can now deploy the YASS application to the overlay using the `testG4A-Yass-Deploy` target in the `Jade/build.xml` at the JadeBoot node. After deployment start the application using the `testG4A-Yass-Start` target.

You can now test the YASS application by storing/retrieving some files, killing some nodes, and joining some other nodes.

Chapter 6

Features and Limitations

6.1 Initial deployment

In the current prototype, only deployment of functional components is done through ADL. The rest, forming the group, establishing bindnings, and creating the self-management architecture is done in the YASS StartManager. The start manager is defined in ADL by adding a "definition= "org.objectweb.jasmine.jade.ManagementType"" tag after the component name.

Currently:

- The start-manager must be the first component to be declared in the ADL file.

- All bindings not established through ADL must declared with "contingency="optional"".

6.2 Demands on stability

In the first version of the prototype, the stable nodes must be present when the application is deployed, and remain present for the duration of the application lifetime.

6.3 Scope of registry

The component registry needed to locate deployed component based on their ADLName is currently only accessible from the boot-node from where the components where initially deployed.

6.4 Resource management

Currently there is no real resource management, parts of corresponding functionality is hardwired in Niche to suit the YASS demo. The following section will demonstrate how to set up the system in a valid way concerning the node id:s.

6.5 Id configuration

To make the prototype work, some settings files should be present and tuned for the specific scenario at hand to achieve desired behaviour, the *lines* file and the *stableNodes* file, two plain-text files in the main Jade folder which list desired node id:s.

In general, a joining node is assigned a random id from the ring id space, ranging from 0 to N. Especially for the prototype, it is desirable to be able to control the assignment of the id:s.

By specifying a *lines* file for each physical computer used, the nodes started on the computer will pick the id:s read form the lines file, in top down order. Since no two nodes should share the same id, the lines files should be non-intersecting. If more nodes are started on a computer than there are id:s in the lines file, the exceeding nodes will be assigned random id:s.

To achieve a very primitive "resource management" functionality, the same *lines* file is used to specify the amount of available storage each logical node offers the system, simply by typing "Id=AmountOfOfferedSpace".

The *stableNodes* file lists the id:s of the nodes which are assumed to be always present in the system for the duration of the test.

By setting these two files with care, nodes can join and leave the system without disrupting the management. The present requirement is that the id:s of any node joining or leaving has to closely follow an id of a stable node. See below for an example.

6.5.1 Id configuration example

The following shows a valid setup to run the YASS demo.
lines-file on:
computer 1: 5000=1, 420000=11
computer 2: 190050=950000
computer 3: 190000=1200000
computer 4: 830000=9500000
computer 5: 830050=9500000
stableNodes-file, on all computers: 190000, 420000, 830000

Listing 6.1: Code to show request-reply behaviour

```
private void fileWrite(String uniqueFileName,
                ComponentId initiator, ...) {

        //Local book-keeping
        ...
        fileWriteAck.fileWriteSucceeded(uniqueFileName, initiator);
}
```

With the above setup, two nodes should be started on the first computer and one on 2, 3 and 4. Thereafter the YASS application can be deployed. To ensure that management is not disrupted, it is the node present at computer 2 that should be stopped to demonstrate the self-* mechanisms. After the node is stopped, a new node can join with id 830050, which will be detected and put to use to replace the lost node.

6.6 Limitations of two-way bindings

Two-way bindings are now implemented for one-to-one and one-to-any bindings. Currently the system cannot aggregate multiple replies on a one-to-many binding. To receive multiple acknowledgements from a set of components, the following work-around can be used: Special attention is given interface method signatures; if the last argument of a method corresponds to a valid ComponentId, this will short-cut the ordinary binding, and be used to send directly to that designated component.

An example from YASS is given in 6.1. In the example the component which should reply is given access to the id of the requestor through the request invocation. The same id is then used to reply through the findReplicasAck interface.

6.7 Caching

In the current prototype functional components are assumed to be static in the sense of non-movable. Therefore the physical address of the components participating in a group or in a binding are cached by the group or the bind-object respectively. On the other hand there is no caching of group content by group users, which means each name based address is resolved on each use.

6.8 Lack of garbage collection

Currently if a group or a management element is removed, not all related bindings and sensors are removed.

6.9 YASS limitations

The current prototype does only store "virtual" files: space is reserved inside the storage component, and file metadata is stored, but the files are not actually transfered over the network.

Chapter 7

Future Extensions

7.1 Initial deployment

Eventually management elements that should be present from initial application deployment time will be deployed through ordinary ADL the same way as functional components. If needed, resource constraints for management elements could then be specified the same way as for functional components.

7.2 Resource Management

Useful improvements include the ability to reallocate resources.

<**TODO**> Nikos should contribute here. </**TODO**>

DCMS currently does not verify resource usage by application components. Moreover, DCMS does not yet define the way to specify required resources for use by specific application management elements, and therefore DCMS cannot verify resource usage by individual MEs even if there were a mechanism to do so. However, if resources that are consumed by MEs are reasonable and distributed fairly between different applications executed by DCMS, accounting of resource usage by individual MEs can be unnecessary.

7.3 Increased Tolerance to Churn: Joins, Leaves and Failures

The next released version of DCM will allow nodes with arbitrary id:s to join without disrupting existing management.

The following version will include ability to gracefully leave any node, which then might cause temporary delays in management response times, but no permanent disruption.

Before the end of the project we hope to implement transparent replication of management elements for increased robustness, which then will allow also nodes hosting management elements to fail. Replication of management elements will involve refining the meaning of bindings between MEs and functional components as such bindings will connect multiple replicas of the same ME to functional components. The set of ME replicas will act as a group, except that MEs and thus ME groups can provide also client interface(s) which will imply "many-to-one" or "many-to-many" communication patterns. We may find it necessary also to restrict the possible interaction types between MEs and functional components in order to make both the programming model and DMCS implementation reasonably simple and efficient.

7.4 Caching

We will implement and test different forms of caching, although the work towards increased efficiency will have lower priority than the work towards increased robustness.

7.5 Improved Garbage Collection

We will improve garbage collection when removing management elements from the system, although the work towards better garbage collection will have lower priority than the work towards increased robustness.

7.6 Replication of Architecture Element Handles

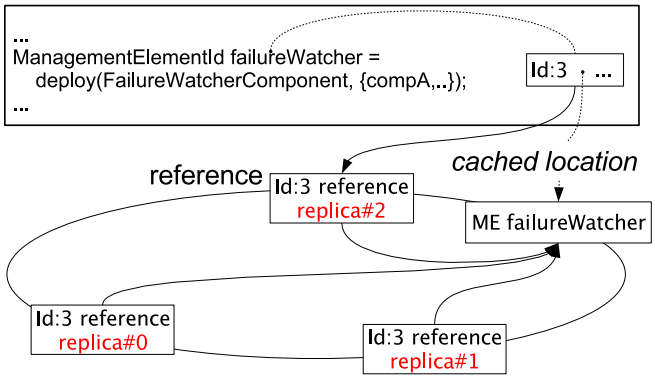


Figure 7.1: Replicated Handles.

The framework will support a network-transparent view of system architecture, which simplifies reasoning about and designing application self-* code. This will be facilitated by replication of internal DCMS entities representing architecture elements, such as references as shown on Figure 3.9. Different MEs access different entity replicas for read accesses. The SNR replication provides eventual consistency of SNR replicas, but transient inconsistencies are allowed. Similarly to handling of SNR caching, the framework recognizes out-of-date SNR references and repeats SNR access whenever necessary.

Chapter 8

Conclusions

In this document we introduce a programming model and API implemented by DCMS – distributed component management service. DCMS facilitates developing self-* applications for community-based Grids, as envisioned by Grid4All use cases. Our framework separates application functional and self-* code, and allows to design robust application self-* behaviours as a network of management elements. DCMS exploits a structured overlay network for naming and lookup, communication, and DHT overlay services. DCMS intends to reduce the cost of deployment and run-time management of applications by allowing to program application self-* behaviours that do not require intervention by a human operator, thus enabling many small and simple applications that in environments like Grid4All's community-based Grids are economically infeasible without self-management.

Glossary

API – Application Programming Interface.

Niche is a Distributed Component Management Service (DCMS), a service supporting developing component-based self-* services. The Niche framework includes the programming model and the API specification.

Niche groups is an abstraction in the Niche programming model that allows the programmer to group together components. One-to-any and one-to-all bindings can be made to a group, and group membership management is independent of bindings to the group.

DKS – Distributed k-ary system [8], a structured overlay network.

Fractal component model [7] is a model for component-based applications. Components have interfaces that are connected by bindings. Fractal provides for nested components and hierarchical composition, and allows component sharing. The novel feature of Fractal is the provisioning for dynamic introspection, reconfiguration and life-cycle management of components.

Groups , see Niche groups.

Management Events are objects that are passed between sensors and MEs. There are event classes pre-defined by Niche, and application-specific event classes defined by the application developer.

ME – Management Elements. A distributed network of MEs constitute the implementation of self-* behaviours in an application. MEs communicate by means of events, and use the Niche API to manage the application architecture.

ME containers encapsulate application-specifc MEs and application-independent ME generic proxies.

ME generic proxies are components provided by Niche that are bound to MEs and implement inter-ME communication and management functions. ME generic proxies allow devlopers of Niche-based application to program MEs as regular Fractal components.

Sensor is a Niche entity that provides information to application self-* code, implemented as a network of MEs, about status of individual components and the environment. The former type of sensors is developed by application programmer together with application components, and the latter one is provided by Niche.

Id,Identifier is a concept in Niche. Id:s uniquely identify implementations and representations of elements of the application's architecture, like components, groups and bindings. Internally in Niche, Id:s are used to address all kinds of entities that are shared by multiple nodes and MEs, in particular – Niche nodes themselves and Niche reference entities.

SNR – Set of Network References, an abstraction provided by Niche that allows to maintain and monitor a set of references to entities on the overlay.

Subscription is an asynchronous communication channel between a pair of sensors or MEs for a specific type of management events.

GCM – Grid Component Model, a refinement of the Fractal component model. Provides for group communication through "collective interfaces", and facilates construction of autonomous component through "behavioural skeletons".

P2P – peer-to-peer networks and systems.

Synchronized MEs – replicated MEs that are maintained in a consistent (synchronized) state, such that such MEs can be seen as hosted on reliable (failure-free) computers.

Index

Bibliography

[1] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand. Enabling self-management of component based distributed applications. In *Proceedings of CoreGRID Symposium*, Las Palmas de Gran Canaria, Canary Island, Spain, August 25-26 2008. Springer. To appear.

[2] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto. Behavioural skeletons in GCM: Autonomic management of grid components. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 54–63. IEEE Computer Society, 2008.

[3] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning Publications Co., Greenwich, CT, USA, 1997.

[4] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications.* Springer, 2005.

[5] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, J.-B. Stefani, N. de Palma, and V. Quema. Architecture-based autonomous repair management: An application to J2EE clusters. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 13–24, Orlando, Florida, October 2005. IEEE.

[6] P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy. The role of overlay services in a self-managing framework for dynamic virtual organizations. In *CoreGRID Workshop, Crete, Greece*, June 2007.

[7] E. Bruneton, T. Coupaye, and J.-B. Stefani. The fractal component model. Technical report, France Telecom R&D and INRIA, February 5 2004.

[8] Distributed k-ary system (dks). http://dks.sics.se/.

[9] Basic features of the Grid component model. CoreGRID Deliverable D.PM.04, CoreGRID, EU NoE project FP6-004265, March 2007.

[10] J. Hanson, I. Whalley, D. Chess, and J. Kephart. An architectural approach to autonomic computing. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 2–9, Washington, DC, USA, 2004. IEEE Computer Society.

[11] P. Horn. Autonomic computing: IBM's perspective on the state of information technology, October 15 2001.

[12] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[13] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[14] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[15] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. *ACM SIGOPS Operating Systems Review*, 40(4):103–115, 2006.

[16] D. Malkhi, F. Oprea, and L. Zhou. *Omega* meets Paxos: Leader election and stability without eventual timely links. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference (DISC' 05*, volume 3724 of *LNCS*, pages 199–213, Cracow, Poland, September 26–29 2005. Springer.

[17] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[18] P. Zielinski. Low-latency atomic broadcast in the presence of contention. *Distributed Computing*, 20(6):435–450, 2008.

This page is intentionally left blank

Annex 2. Security Infrastructure

Grid4All Security User's Manual, Release 0.6

by Leif Lindbäck and Vladimir Vlassov
Royal Institute of Technology (KTH), Stockholm, Sweden
Email {leifl, vladv}@kth.se

FP6 Project Grid4All (IST-2006-034567)

Contents

Abstract

This User's Manual is for developers who intend to use Grid4All security API for authentication and authorization in their distributed applications and systems.

The Grid4All security API allows building a policy-based security infrastructure where policies are expressed in XACML (eXtensible Access Control Markup Language). The Grid4All security API (classes and interfaces) has been developed using Sun's XACML implementation (http://sunxacml.sourceforge.net/).

The Grid4All security distribution (including source and documentation) is available at http://www.isk.kth.se/~leifl/vofs/

This User's Manual is structured as following. First, it describes the Grid4All security architecture; next it gives installation instructions; followed by start-up instructions and user commands; and finally, it presents a programmer's guide that explains how to use Grid4All security API for programming a policy-based security infrastructure in distributed applications and systems.

1 Grid4All Security Architecture

The Grid4All security includes the following security components (Figure 1).

- *Policy Enforcement Point* (PEP) enforces VO policies;
- *Policy Decision Point* (PDP) makes authorization decisions based on VO policies;
- *Policy Information Point* (PIP) collects user security-related data;
- *Policy Administration Point* (PAP) is used to administrate (create, update, delete) policies;
- *Policy Repository* is a persistent store for policies;
- *VO Membership Service* (VOMS) maintains VO membership.

PEP is the only component that resides on the client side and may need to be customized or re-implemented by the developer. Other components remain unchanged across different application domains and use-cases. The components are explained in more detail below.

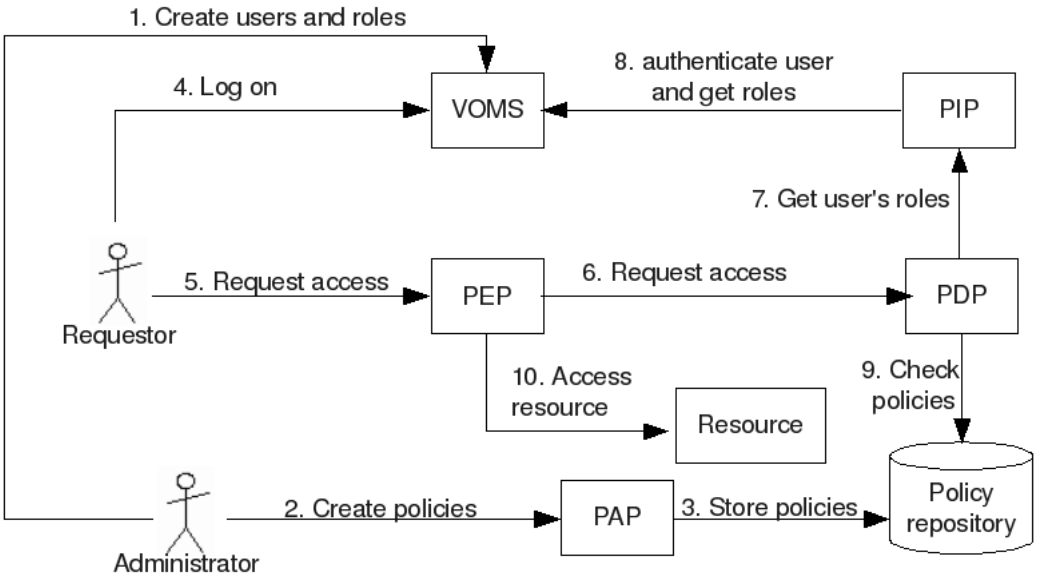


Figure 1. Security architecture

1.1 Policy Enforcement Point (PEP)

PEP protects a resource. It is called whenever access rights shall be checked. An authorization request to PEP includes the following three parameters.

- *Subject* which is the single sign-on identifier of the user accessing the resource. This identifier was returned by the VOMS when the user signed on.
- *Action*, which specifies a name of an action to be performed on the resource.
- *Resource* which is the target resource identifier in the form of a path starting with `@` (e.g. `@/se/kth/ict`).

PEP sends the authorization request to PDP and upon receiving an authorization response from PDP, it enforces the authorization decision. Note that the PEP classes provided in the Grid4All API, do not enforce the authorization decisions but only cache the decisions communicate them to a component requesting the access check. The Grid4All Security API includes a sample PEP class that only caches authorization decisions (see 4.2). This class can be used as a template to develop a PEP enforcing authorization decisions. Caching of PDP responses at PEPs reduces security overhead. The developer can use PEP either with or without cache for PDP responses. The PEP class provided in Grid4All security API supports caching of PDP responses

Note that any object sending authorization requests to PDP and receiving an authorization response could be considered as PEP.

1.2 Policy Decision Point (PDP)

The PDP evaluates requests in the context of a security policy. When the PDP receives an authorization request from PEP, it retrieves the appropriate policy from the Policy Repository or from its policy cache maintained in memory, evaluates the request, makes the authorization decision, and returns it to the requesting PEP as an authorization response. The response includes one of the following possible results:

- *Permit*, this means that access is granted.
- *Deny*, this means that access is rejected.
- *NotApplicable*, this means that there was no matching policy.
- *Indeterminate*, this means that no decision could be taken. For example there might be several contradicting policies.
- *Error*, which means that policies could not be checked because of some exception, for example the communication link might be broken

The PDP maintains a cache of policies which are loaded from the Policy Repository.

1.3 Policy Information Point (PIP)

PIP is responsible for authenticating users and for retrieving their roles from the VOMS. PIP caches VOMS answers.

1.4 Policy Administration Point (PAP)

PAP is used to administer (list or change) policies stored in the Policy Repository (PR). Before accessing the repository, PAP calls its own PEP to check if the policies in PR grant the caller the right to do the requested operation. The provided implementation of PDP assumes that XACML policies are stored as XML files.

1.5 Policy Repository (PR)

The policy repository is a persistent store for policies in the eXtensible Access Control Markup Language (XACML)[1].

2 Installation

1. Download the Grid4All security distribution from `http://www.isk.kth.se/~leifl/vofs/`
2. Unpack the content of the archive
3. Check and edit entries (e.g. host names, port numbers, paths, etc.) in two configuration files, `etc/config/grid4all.secserv.config` and `etc/config/grid4all.pep.config` for PDP/PAP and PEP, respectively. The entries are documented in the files. When running a program, e.g. PDP, PAP or PEP, the paths to configuration files should be either on the class path or specified with the `-fs` option for the PDP/PAP configuration file (`grid4all.secserv.config`) and `-fp` for the PEP configuration file (`grid4all.pep.config`).
4. Make sure that the `policyPath` entry in the `grid4all.secserv.config` configuration file points to a directory that contains the file `etc/policy/default-admin.xml`, otherwise no user will have the right to execute any commands on security components. The `default-admin.xml` file grants the role `admin` all rights to all resources protected by Grid4All security.

3 Running Security Components

PDP (Policy Decision Point), PAP (Policy Administration Point) and VOMS (VO Membership Service) are servers. This section explains how to start PDP, PAP and VOMS, and user commands to control them. It also explains how to run the PEP demo and the PAP client. The PEP demo illustrates interaction of PEP with PDP: it allows the user to send commands to PDP over a TCP connection. The PAP client allows the user to send commands to PAP over a TCP connection.

3.1 Running PDP

Start PDP with the `start-pdp` command in the `bin` directory. Communication with PDP can be done through TCP connection, RMI, local method invocation and standard input/output. The RMI and local methods are described in 4.1.34.1.4 respectively. The TCP-based and standard input/output interfaces accept the following commands.

- **`REQUEST <subject> <action> <resource>`**

  Queries the policy repository. The response is one of `Permit`, `Deny`, `NotApplicable` or `Indeterminate` according to the XACML specification. If the request does not have the correct format or some other exception occurs, then the response is `ERROR`. Note that `subject` should be the single sign-on identifier returned by VOMS when the user signed on. PDP will call VOMS to check that the user is signed on and to get the roles of the user. All roles will be evaluated and the result will be `Permit` if any of the roles has the right to perform the specified action. PDP will answer not only to the specified request, but also to other requests with the same `subject` and `resource` but with all existing actions. The response format is

[1] http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf

<answer to specified request**>** **<**nextAction**>** **<**answer to a request with the originally specified subject and resource, but with action replaced by nextAction**>**

The last two items are repeated for each action that exists in the policy repository.

- **REGISTER <port>**

  Registers the given port number and the IP address from which this command is sent, at this PDP to send cache invalidation requests. <port> defines the port number to which the PDP will send cache invalidation requests. This command can only be used from the TCP interface. On success, the command returns REGISTER_OK. When the cache shall be invalidated PDP sends INVALIDATE to the specified port on the IP address from which the REGISTER command was sent. On receiving this command PEP should discard its entire cache. It should not send any answer to PDP. PEP only gets one chance to discard its cache; if this message is lost it will not be retransmitted.

- **UNREGISTER <port>**

  PEP at the address from which the command was sent will stop receiving information about cache invalidations. <port> should be the same as in the REGISTER command. This command can only be used from the TCP interface. On success, the command returns UNREGISTER_OK.

- **RELOAD**

  Causes PDP to discard its policy cache and reload it from the policy repository. The INVALIDATE command (see above) is sent to all registered PEPs. On success, the command returns RELOAD_OK.

- **STOP**

  Terminates PDP. Note that the entire PDP terminates, not only the command interpreter. PDP is shutdown gracefully. All ongoing work will be finished before shutting down. This command can only be used from the standard input interface.

3.2 Running the PEP Demo (a PDP Client)

The PEP demo illustrates interaction of PEP with PDP: it allows the user to send commands to PDP over a TCP connection. To run the PEP demo use the pep-demo command in the bin directory. The demo uses TCP to communicate with PDP that must be started with the start-pdp command described in 3.1. The PEP demo reads the PDP host name and port number from the client configuration file, etc/config/grid4all.pep.config.

3.3 Running PAP

Start PAP with the start-pap command in the bin directory. Note that PAP will call PDP, so you must start that one also. Communication with PAP can be done through TCP connection, RMI and local method invocation. The RMI and local methods are described in 4.3.2 and 4.3.3 respectively. The TCP interface accepts the following commands.

- **<issuer> setacl -dir <directory> -acl <role> <rwilkad> [-clear] [-time <startTime> <endTime>] [-date <startDate> <endDate>]**
  [-dayofweek <startDayOfWeek> <endDayOfWeek>][-negative]

The setacl command sets the access control list (ACL) entries specified with the -acl argument to the ACL of the directory named by the -dir argument for role <role>. On success, the command returns OK otherwise an error message.
The -clear flag is used to remove the ACLs of the <role> on the directory <directory>. In this case, there is no need to specify the permissions (<rwilkad>) since all the permissions will be erased. The -negative flag is used to impose negative ACLs (i.e. "deny" decision) for the specified actions in <rwilkad> for role <role> on the directory <directory>. Note that issuer should be the single sign-on identifier returned by the VOMS when the user signed on. All roles of the user will be evaluated.
The -time, -date and -dayofweek flags makes the policies, whether positive or negative, valid only the specified periods. Periods can not be open; they must always have both start and end. Time format is hh:mm:ss[TZ]. Time zone is specified as +/- offset from UTC, e.g. 00:00:00+02:00. Time periods are allowed to wrap around midnight. Date format is yyyy-mm-dd. Dates must be valid, i.e. November 31 or February 29 of non leap years are not accepted. All years from 0000-9999 are accepted. Day of week format is strings containing day names in English without abbreviations. Day names are not case sensitive. Day of week periods are allowed to wrap, which means that it is not important which day is considered the first of the week. Time zone is not considered when date and day of week rules are evaluated; this means that midnight always occurs when it occurs on the computer evaluating the time period. The -time, -date and -dayofweek flags can be combined in any way and the rule only applies if all of them match.

Example:
```
123456 setacl -dir @/se/kth/ict -acl teacher rwid -time
01:00:00 13:00:00
```
The specified role, teacher, is given the specified permissions, rwid which is short for read, write, insert and delete, on the specified resource, @/se/kth/ict. The permissions are valid from 1 AM to 1 PM.

Privilege Required:
The issuer must have the a (admin) permission.

Answer:
The answer is OK.

- **<issuer> listacl -dir <dir/file path>**

  The listacl command displays the access control list (ACL) of the <role> associated with the specified file or directory. Note that issuer should be the single sign-on identifier returned by VOMS when the user signed on. All roles of the user will be evaluated.

  Example:
  The response of command: 123456 listacl -dir @ might be:
  ```
  ACL for <@ admin>
       Positive Entry: r valid: 01:00:00-02:00:00
       Positive Entry: w valid: always
  ```

```
Negative Entry: a valid: Monday-Tuesday
```

Privilege Required:
The issuer must have the `r` (read) permission on its ACL and the ACL for every directory that precedes it in the pathname. Otherwise PAP returns an error message. Also, it means that you can not see your own ACL on the directory if you don't have `r` (read) permission on the ancestor directories.

Answer:
See *Example* above.

3.4 Running the PAP client

The PAP client allows the user to send commands to PAP over a TCP connection. To run the the PAP client, use the `pap-client` command in the `bin` directory. The PAP client uses TCP to communicate with PAP that must be started with the `start-pap` command described in 3.3. The client reads the PAP host name and port number from the client configuration file, `etc/config/grid4all.pep.config`.

3.5 Running VOMS

1. Initialize VOMS database with the command `bin/init-voms`.
2. Start VOMS with the command `bin/start-voms`. VOMS can now be accessed with a browser at the url `http://<voms host>:8080/voms`.
3. To create users and roles, log in as the default admin user:
   `Username: admin`
   `Password: g4all`

   It is not necessary to enter anything in the fields `VOFS Host` and `VOFS Port` to be able to create users and roles.
4. The following actions exist in VOMS:
   - `login.jsp` (default) authenticates the user and creates a single sign-on token which is sent to the specified host and port.
   - `GetAllPeerAction.do` returns a list with the host and port of all signed-on users.
   - `CheckIdAction.do?id=<single sign on token>` returns all roles of the user with the specified token. Shows an empty page if there is no such user.

4 Programming Security Components

This section explains how to access Grid4All Security components (PDP, PEP and PAP) programmatically.

4.1 Programming Policy Decision Points (PDP)

PDP is provided as a server application (service) that can be accessed by clients in different ways as explained below. The provided PDP is complete and does not need to be extended in any way. It is defined in the primary class `se.kth.grid4all.security.pdp.PDP` and it is started with the `start-pdp` command in the `bin` directory. When PDP is running,

a client (e.g. PEP) interacts with it by submitting an authorization request and getting in return an authorization response (see Figure 1). The client can communicate with PDP in four different ways: over a TCP socket connection; through the standard input and output streams; using RMI; and by local method invocation on PDP. In the last case, the requesting client must be in the same JVM as the PDP object.

4.1.1 Interaction with PDP over a TCP Connection

The PDP object listens for TCP connections on the port specified in the configuration file, `grid4all.secserv.config`. The accepted commands are specified in 3.2.

4.1.2 Interaction with PDP through Standard Input and Output

Commands can be given through standard input; whereas response can be read from standard output. The accepted commands are specified in 3.1. Note that the command interpreter that reads standard input runs in the same JVM as the PDP itself.

4.1.3 Interaction with PDP Using RMI

The PDP is bound in the RMI registry under the name specified in the configuration file `grid4all.secserv.config`. The remote interface for interaction with PDP is `se.kth.grid4all.security.communication.SecurityServer`. The interface defines the following remote methods.

```
EvaluationResult evaluateRecursively(String subject,
                                     String resource,
                                     String action)
                              throws PolicyException,
                                     RemoteException
```

> Evaluates an authorization request, which includes *subject*, *resource*, and *action*, in the context of security policies, i.e. evaluates whether the given subject has rights to take the given action on the given resource. First checks for a policy for the resource specified by the `resource` parameter. If there is no matching policy, drops the part after the last path delimiter, /, and tries again. This is repeated until a matching policy is found or there is no more path delimiter in resource. The request is evaluated in the context of the found policies.
>
> The method takes the following three parameters. `subject` is the single sign-on identifier of the user wanting to access the resource. The PDP first asks the PIP to call the VOMS to check that this identifier belongs to a signed on user and, if so, gets the roles that this user belongs to. Resolved mapping of identifiers to roles are stored in cache at PIP, where cache entries are valid for the time specified by the value of the key `SSOTimeout` in the configuration file `grid4all.secserv.config`. The parameter `resource` specifies the resource the subject tries to access. The parameter `action` specifies the action the subject wants to take on the resource.
>
> The method returns the PDP's authorization response, which is one of *Permit*, *Deny*, *NotApplicable* or *Indeterminate* (see 1.2). All of the subject's roles are evaluated and the result is *Permit* if any of these roles are permitted to perform

the specified action. The returned `EvaluationResult` object contains the PDP's response for the specified `subject`, `action` and `resource`. It also contains the PDP's response to requests for the same `subject` and `resource`, but with all existing actions. The javadoc for the `EvaluationResult` class describes how to retrieve these responses.

```
String reload() throws PolicyException, RemoteException
```

Causes the PDP and all registered PEPs to discard their caches. Note that the PDP can only invalidate caches of registered PEPs. PEPs inheriting the `CachingPEP` class are automatically registered at the PDP.

4.1.4 Interaction with PDP by Local Method Invocation

This can be done from an object in the same JVM as the PDP. The following methods exist:

```
public String evaluateSSOIdRecursively(String SSOId,
                                       String resource,
                                       String action)
                                throws PolicyException
```

Behaves the same way as the remote method `evaluateRecursively` (see 4.1.3) except that only the answer to the specified subject, resource and action is returned. There is no prefetching of responses to authorization requests with other actions. The method is defined in the class `se.kth.grid4all.security.pdp.PDP`.

```
public void reload() throws PolicyException
```

Discards all policy information cached by the PDP, and then loads all policies described in the XACML files stored in the policy directory. The policy directory path is specified in the `grid4all.secserv.config` configuration file. Files in subdirectories of that policy directory will not be read. Calling this method will not affect PEP caches. This method is defined in the class `se.kth.grid4all.security.pdp.PDP`.

4.2 Programming Policy Enforcement Points (PEP)

All requests accessing resources protected by the Grid4All security infrastructure should pass through PEP. PEP should run in the same process (JVM) as the resource it protects. This way, it is impossible to access the resource without being allowed by PEP. For example, a PEP can be placed as Servlet filter when protecting a Web-application. A distributed application may have several PEPs that could interact with the same or different PDPs. One of the issues to be considered when using the security components is an efficient and effective placement of PEPs (and other components) in order to achieve necessary protection of resources while reducing the security overhead. A resource owner can place its own PEP and PDP in addition to those placed by VO administrator.

The PEP implementation provided in the Grid4All security API performs communication with PDP (sending authorization requests and receiving authorization decisions) and caching of decisions. It does not enforce authorization responses received from PDP; it only caches the responses and passes them to clients. The PEP classes (see below) are provided as examples and as utilities for handling response caching and communication with the PDP. The caching functionality is explained in 4.2.1.

The enforcement of security policies, as expressed in authorization responses, must be implemented in the code calling the PDP (or provided PEP).

4.2.1 The class *CachingPEP*

The `se.kth.grid4all.security.pep.CachingPEP` class is a base class implementing a stateful PEP which caches requests (those already sent to the PDP) and the respective responses (received from PDP). Cache entries are valid for the time specified by the value of the key `SSOTimeout` in the `grid4all.pep.config` file. To ease of use, most of the PEP related functionalities are implemented and put in the `CachingPEP` class. Your application-specific PEP would be a subclass of `CachingPEP` implementing the `doEvaluate` abstract method that should return an authorization response (decision) for the given authorization request. In this method, the programmer has the choice between three possible alternatives to get the authorization decision: (1) using cached decision; (2) asking for new decision from PDP; and (3) do some custom evaluation.

When an authorization request is sent to the PDP, it evaluates and returns decisions not only for the specified request, but also for requests with the same subject and resource but with all existing actions. `CachingPEP` will save all these authorization decisions in the cache.

The source code of the `se.kth.grid4all.security.pep.PEP` class, a subclass of `CachingPEP`, is an example that illustrates use of `CachingPEP` functionality.

4.2.1.1 Cache Invalidation

When a `CachingPEP` object is constructed, it registers itself to the PDP specified in the configuration file `grid4all.pep.config`, in order to subscribe for cache invalidation notifications. The syntax of cache registration and unregistration commands can be found in 3.1. The registration is persistent and remains even if the PDP is restarted.

4.2.1.2 Communication with PDP

`CachingPEP` uses TCP for communication with PDP. The end-point (host and port) of the PDP is specified in the `grid4all.pep.config` file. PEP opens a TCP connection to PDP

for each request, receives the response and then closes the connection (see the `callPDP` method in the source code of the `CachingPEP` class).

4.2.2 Interaction with PEP

A client can interact with PEP in two different ways, by local method invocation or through streams.

4.2.2.1 Interaction with PEP by Local Method Invocation

In order to communicate with PEP by local method calls, the calling thread should reside in the same JVM as the PEP. PEP can be created by instantiating either a concrete subclass of `CachingPEP`, or the `se.kth.grid4all.security.pep.PEP` class which contains the minimum functionality needed by such a subclass. The methods that can be invoked on PEP are as follows.

```
public String evaluate(String subject,
                       String action,
                       String resource)
                throws java.io.IOException
```

Handles the cache and evaluates a given authorization request. The actual evaluation is delegated to the `doEvaluate` method. The `evaluate` method is the method that should be called on the concrete PEP subclass when a request is to be evaluated. String match is used for cache lookups. If `subject`, `action` or `resource` differs in any way, then cache entries will not be found. This method is not thread safe. Note that `subject` is the single sign-on identifier returned by VOMS when the user signed on. It is converted to the user's roles as described in 4.1.3

```
public void register(String pdpHost, int pdpPort)
              throws java.io.IOException
```

Makes PEP listen for cache invalidations from the specified PDP. PEP is allowed to register with multiple PDPs. Invalidation from any PDP will cause the entire cache to be discarded.

```
public void unregister(String pdpHost, int pdpPort
                throws java.io.IOException
```

Makes the PEP stop listen for cache invalidations from the specified PDP. Unregistering with a PDP with which we have not registered will not have any effect at all. Note that calling this method is the only way to unregister since PEP registrations are not discarded by PDP restarts.

```
public void reload() throws java.io.IOException
```

Makes PDP discard all policies in its cache and reload them again. Also, all PEPs registered with that PDP will be notified to discard their caches.

In addition, there is a helper method `callPDP` that subclasses of `CachingPEP` can use to call PDP.

4.2.2.2 Interaction with PEP using Streams

The class `se.kth.grid4all.security.pep.StreamReaderPEP` is a concrete subclass of `CachingPEP` that reads PEP commands from an input stream and sends answers to an output stream. The input and output streams are specified in the constructor when an instance is created. The object starts handling requests when the method `readFromStream` is invoked. There is a command, `PDP_HOME/bin/start-stream-reader-pep`, that starts a `StreamReaderPEP` object (by calling `readFromStream`) which reads commands from the standard input stream and sends answers to the standard output stream. It can handle all commands that the TCP interface of PDP can handle (see 3.1). It can also handle the commands that are accepted by PAP (see 3.2). The host and port of PDP and PAP are specified in the `grid4all.pep.config` file.

4.3 Programming Policy Administration Points (PAP)

PAP is provided as a server application that can be access by clients in different ways as explained below. The provided PAP is complete and does not need to be extended in any way. It is defined in the class `se.kth.grid4all.security.pap.PAP` and is started with the `start-pap` command in the `bin` directory.

A client (e.g. a policy editor or the provided PAP client described in 3.4) interacts with PAP by submitting requests to list or update policies stored in Policy Repository (see Figure 1). The client must have enough rights to issue the policy administration commands (see 3.2).

The client can communicate with PAP in three different ways: over a TCP connection; using RMI; or by local method invocation on PAP. In the last case, the client must be in the same JVM as PAP.

4.3.1 Interaction with PAP over a TCP Connection

PAP listens on the TCP port specified in the config file, `grid4all.secserv.config`. The accepted commands are specified in 3.2.

4.3.2 Interaction with PAP Using RMI

The name that PAP is bound under in the RMI registry is specified in the configuration file `grid4all.secserv.config`. The remote interface is `se.kth.grid4all.security.communication.SecurityServer`. The interface includes the following methods.

```
public void clearACL(String issuer, String resource,
                     String role)
              throws PolicyException, RemoteException
```

Removes an ACL entry from the Policy Repository. There is a one to one mapping between ACL and XACML policy file. So, successful result of this action will be deleting the corresponding file from the repository directory.

```
public void setACL(String issuer, String resource,
                   String role, String permission,
                   boolean positive)
            throws PolicyException, RemoteException
```

Handles the `setacl` (ACL creation) requests. If there exists already a policy for the given `subject`, `resource` pair, then it will be updated; otherwise, a new policy will be generated.

```
public void setACL(String issuer, String resource,
                   String role, String permission,
                   Duration duration, boolean positive)
            throws PolicyException, RemoteException
```

Works exactly as the method above, but also takes a `Duration` object that specifies when the ACL is valid. For a detailed description of durations, see 3.2.

```
public String listACL(String issuer, String resource)
              throws PolicyException, RemoteException
```

Returns the access rights of the given parameters as a string. The required permission is “read” for the specified resource path and all parent paths.

4.3.3 Interaction with PAP by Local Method Invocation

In order to communicate with PAP by local calls, the calling thread must reside in the same JVM as the PAP object. The methods signatures (excluding `RemoteException`) and semantics are exactly the same as the remote methods described above in 4.3.2. The methods are located in the class `se.kth.grid4all.security.pap.PAP`.

5 Limitations

As the current implementation of the Grid4All security infrastructure is a proof-of-concept prototype rather than of production quality, it has the following limitations that could be fixed in more matured releases.

- There is no encrypted communication; all communication is done in plain text.
- PDP can only call one VOMS, which means it can handle only one VO.
- The provided PAP client does not read the single sign-on identifier from the file system but from the command line.

6 Known Bugs

Even though PDP supports timing intervals in policies, it ignores time zones. This seams to be a problem in Sun's XACML implementation.

This page is intentionally left blank

Annex 3. Market Information System

User Manual

Table of Contents

1 Introduction

The main objective of this result is to solve the information provision problem arising from a distributed market environment. The DMIS framework, which has to deal with several requirements, both from distributed environments such as scalability and robustness constraints, and from economic information systems like privacy, time-sensitivity, and historical data availability.

Information about specification and behaviour of participants in electronic markets is essential for sophisticated and efficient negotiation strategies. However, there is currently no completely researched system to provide and consult an overall knowledge of economic information in distributed markets. These markets are implemented for example by Grid applications and gained importance over the last few years. This paper presents the economic information requirements and a high-level architecture overview for a Decentralized Market Information System (DMIS). The proposed system acquires economic data in a distributed environment for providing it to individual traders or other participants in a decentralized manner. First, we outline the economic information requirements which the system needs to achieve. Therefore their properties and a privacy model has to be considered. Then we propose an architecture for the system which combines technologies of distributed information aggregation system and distributed publish-subscribe models, based on a structured overlay network. The architecture has been designed to meet both the economic information requirements and that of scalability and robustness of a large-scale distributed environment. Initial measurements confirm the proof-of-concept implementation in a large-scale of the existing prototype.

2 Deployment Scenario

The DMIS is designed for being deployed in auction-based and bargaining-based distributed marketplaces. This enables resource providers and service providers to sell their products on a market. Obtaining information about the market allows the buyers and sellers to optimize their trading strategy.

Figure 1 shows a scenario for a possible deployment of a DMIS. Coordinated by auctioneers, sellers and buyers are trading on different marketplaces. An auctioneer uses for example an English Auction or a CDA. Among both auctions with different types exist no implicit information exchange. More reasons for such a separation of markets result from different currencies, geographical locations, privacy and trust constrains or political aspects.

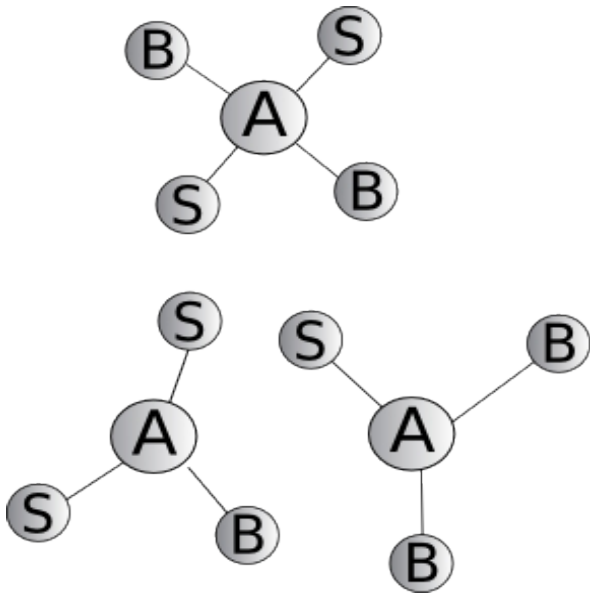


Figure 1: Markets without global information.

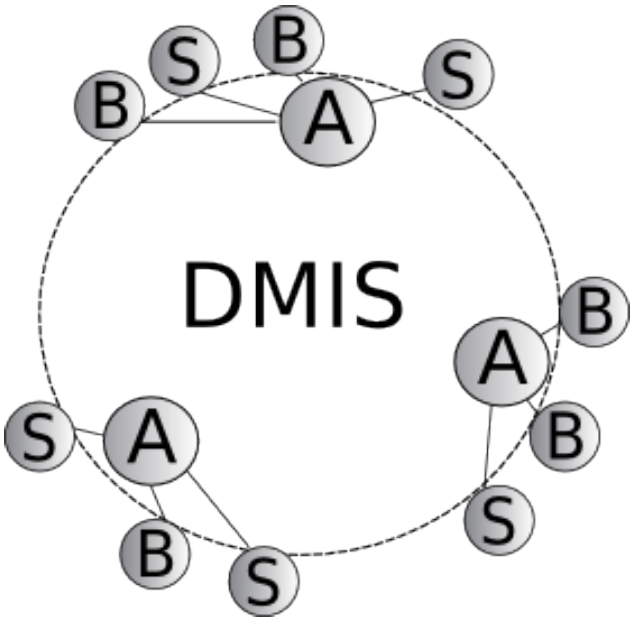


Figure 2: Enhanced with DMIS information provision.

Introducing the DMIS to the introduced scenario of electronic markets (see Figure 2) enables explicitly an information exchange among all participants. Traders can obtain information from other traders or direct from an auctioneer. Alternatively, an auctioneer can be distributed on several nodes, depending on its type and implementation. Interested participants can execute SQL-like queries or can subscribe for new events.

3 Architecture and Use Cases

The technical challenge for a decentralized market information system is to meet the economical

requirements in combination with the technical requirements of distributed systems. The economical side needs the disclosure of aggregated and individual data. Moreover some information has to be accomplished in a high time-sensitivity. But on the other side the technical realization has to cope with a high churn in distributed systems and to scale in regard to traders and products.

Layers

The proposed DMIS architecture consists of the four layers Market Information System (MIS), DMIS, Routing and DHT, which is described in the Figure 3. Each layer of the architecture has to cope with different technical or economical requirements. The MIS layer can provide a communication among a virtual organization (VO) which can allow a separation in other overlays. The Routing layer provides the routing functionalities which used by the DMIS to provide its API. The DHT layer is the main component to cope with the scalability and robustness of the system.

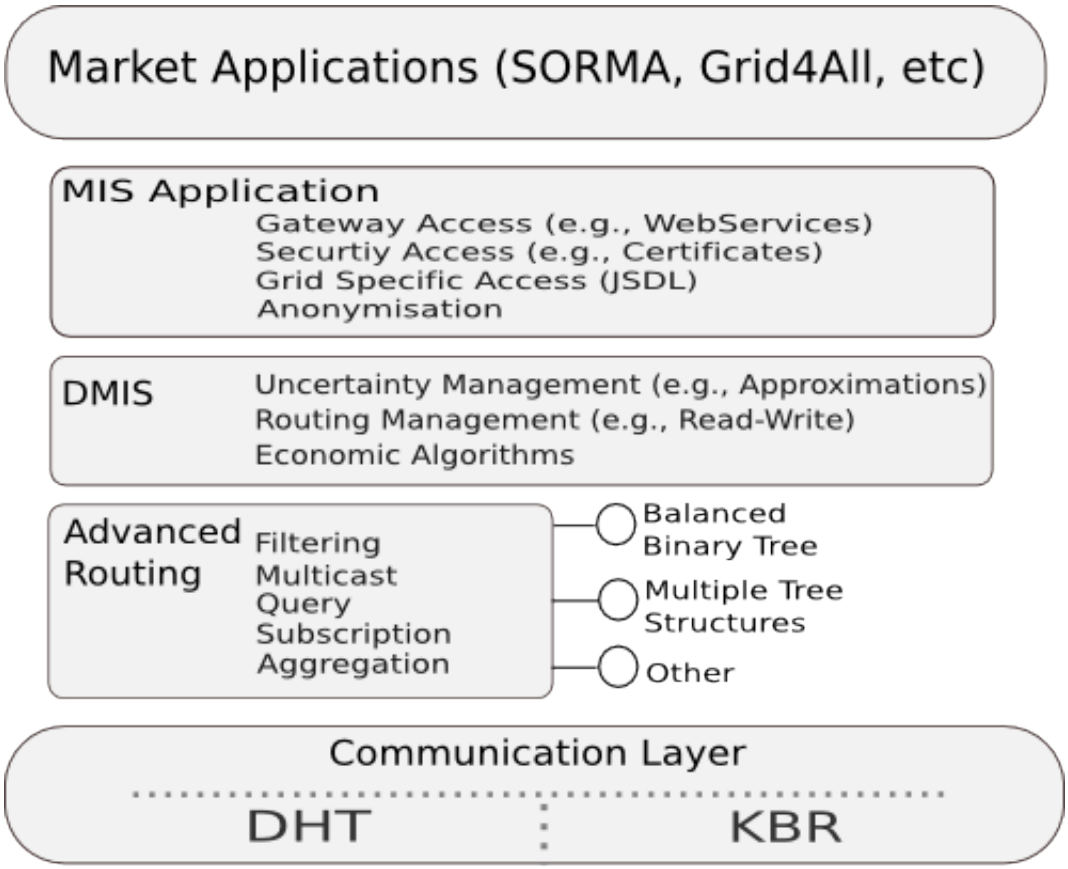


Figure 3: Architecture Layers Building the DMIS.

The DHT layer operates as the communication layer and thereby it builds the basis for the DMIS system. A Kademlia-based DHT [18] is implemented in the actual prototype. But it will be designed to be flexible to switch between different types of DHTs. Some DHT solutions are already implementing publish-subscribe or multicast functionalities. Therefore a trade-off between reusing the existing models and developing new algorithms has to be made. To provide a higher flexibility and the need of unique structures force the DMIS routing structures to base on send, receive, put and get.

The core functionalities of the system are processed in the Routing layer. The current prototype

provides the main components of the following functionalities:

- **Multicast:** sending messages to only a subgroup of nodes. The general multicast is easier to handle when each client knows all members of the same topic, but it is less scalable. Therefore the multicast will be changed to an algorithm where a new node takes a certain place in the tree and knows only the direct parent(s) and children.

- **Query:** This function enables to execute a query for a read-dominated value within the marketplace. Therefore it follows either an epidemic structure, binary-tree structure or multi-tree structure [3]. The current prototype implements a query in a binary-tree in an ordered subset of clients. An objective is to implement multiple-trees, which have a higher robustness but also increase the amount of messages.

- **Subscription:** This is the process to join a certain topic or content, and accordingly to obtain interested information. Actually the prototype follows a subscription to a certain topic, but we are looking to change to a content based subscription to complete fully the requirements of the DMIS.

- **Aggregation:** The gathering of information need an aggregation to decrease the amount of messages and to provide scalability. As some aggregations are simple (maximum, minimum), exists more sophisticated aggregations like the calculation of an average price. Other queries are even more complex, when a combination of more parameters is requested (select price where storage > 100 GB and memory > 3 GHz). This concerns especially the filtering algorithms.

The DMIS layer provides and coordinates the core functionalities for the trader or the client. Therefore it provides several handlers like the SubscriptionHandler, QueryHandler or RequestHandler for messages returning or entering the trader. The trader can invoke methods provided by the API like subscribe, query or publish. But also non-functional methods are provided like bootstrap net to be called.

Additional functionalities are provided by the MIS layer. This can be necessary for the integration into different projects. It will provide one or more gateway nodes connected to the DMIS. For example interconnecting via Web Services to traders in the same VO can fulfil this service. The MIS layer is not concerning the functional behaviours of the DMIS it represents an adapter to the environment. For example it could bes the Web Service adapter for the DMIS.

Query Process

A query is executed by a trader to obtain certain information about the market and other traders. This process is illustrated in Figure 4. First, the trader needs to create a filter which narrows the search for information. Then it has to create a handler for the query which will be informed of the result. In this handler it will be defined how the trader processes with the result. Afterwards these two classes are passed to the DMIS, which executes a query to its routing layer. Within the routing process, nodes in the overlay will be asked for the information which will be aggregated and selected.

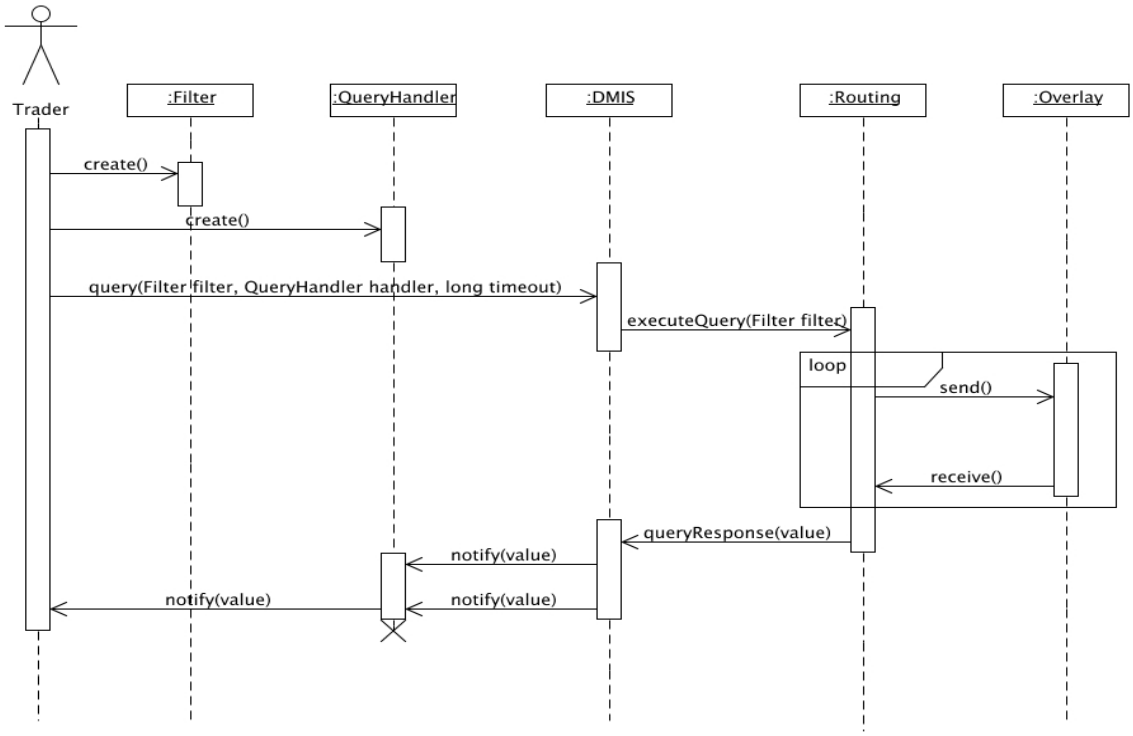


Figure 4: DMIS query process

When the routing layer has a result, it returns it to the DMIS. This will notify the handler assigned by the trader. The DMIS will delete the handler, after a successful query.

Subscription process

Figure 5 shows an example of a trader, subscribing to a topic/content and another trader publishing an event, which will notify the subscriber. First the subscriber has to create a filter, defining the interested events. Then the created Subscriber will be notified when a new event correspond to a set filter. Afterwards the subscriber passes the created classes to the DMIS which will install the subscription with its filter in its routing layer.

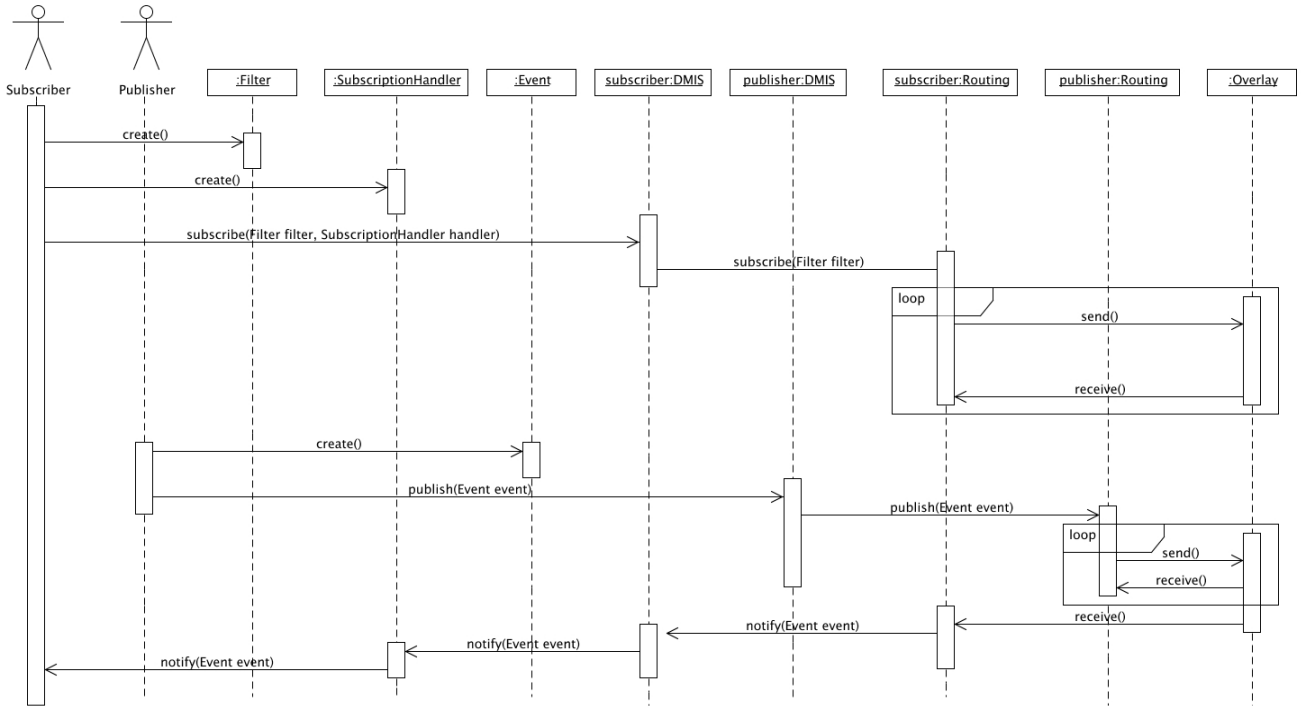


Figure 5: DMIS subscription process

When a publisher has new information, for example an agreement of a seller and buyer, then it creates the corresponding event. This event will be passed to the DMIS, which forwards it to the routing layer. There the event will be send via the overlay to the subscribers, which have set a matching filter. The routing layer of the subscribers will inform their DMIS that a new event is incoming. Afterwards, the DMIS will look for the registered SubscritionHandler and notifies them.

4 Software

License

The DMIS project is under the LPGL license.

GUI

For informations about the GUI for the DMIS follow this link DMIS GUI.

Pre-requisites

- Java 1.6
- Apache ant 1.6 and up to use the component launching scripts.

Provided functionalities

- RDV based subscription mechanism
- multicast for a join/leave group
- topic-based subscription
- delivering the published events to the subscribers of a topic
- binary tree routing
- multi tree routing
- send a query to a subgroup of nodes (subscribed by topic) these return the requested value by aggregating (MIN or MAX) it.
- Sum and Count
- the DMIS prototype is tested with 100 nodes
- the DMIS based on the network simulator is tested with 30000 nodes

5 Getting started

Download

The latest stable version can be found at the subversion repository. The public stable code can be downloaded via svn like the following:

```
svn co https://code.ac.upc.edu/projects/dmis/svn/tags/stable-1.0 dmis_dir
```

Project structure

- src/ source files for the dmis application
- lib/ required libraries
- bin/ compiled classes
- doc/ javadoc documentation for the dmis API
- dist/ ant build directory
- conf/ folder for configuration files
- .project Eclipse project configuration
- .classpath Eclipse project's classpath
- build.num counter file for ant
- README short description guide
- LICENSE license description
- build.xml ant build file

Configuration

The following table describes the configuration parameters for the MIS (see *mis.properties* file within the configuration package of the download section), depending if the MIS is using a database or an overlay. Using the database allows to connect via HTTP to firewall protected machines and allows and easier setup of the system.

Property	Default value	Description	Comments
overlay		Configuration parameters when using an overlay as communication protocol	
overlay.port	20000	is the port on which the instance of the MIS communicates	the port must be open (not behind a firewall or NAT)
overlay.bootport	20000	is the bootport to join the P2P network	If it is the first MIS instance to be created then the own port will be the bootport
overlay.IP	147.83.34.222	this is the IP of the bootstrap node	Please consider that if the bootstrap node is on the local machine "localhost" or "127.0.0.1" does **not** work if the instances should be accessible outside the local machine
overlay.type	Scribe	Defines the type of the used overlay	possible types are 'Kad', 'Scribe' and 'ScribeSimulator'
overlay.simulator	euclidean	Defines the simulation data for the Round Trip Time (RTT) of the peers	possible types are 'generic' or 'euclidean'
overlay.generic_file	conf/GNPINPUT	the absolute path to the the generic simulation input file	This is only important when the overlay.simulator is generic
overlay.simulator.speed	full	regulates the speed for the simulation process	possible types are 'slow' and 'full'
forecast		Configuration parameters to specify the forecasting	
forecast.type	NONE	defines the type of forecasting mechanism.	The following mechanisms are implemented: SMA (Simple Moving Average), WMA (Weighted Moving

			Average), EMA (Exponential Moving Average)
forecast.periods	10	defines the number of periods where the forecasting is based on	
approximation		Configuration of uncertainty management	
approx.uncertainty_management	false	enables or disables the approximations	
approx.max_depth	12	defines the size of the approximation sample through the maximum number of hops	
test		Configuration parameters to test and evaluate the MIS	
test.max_instances	10	defines the number of created MIS instances (nodes)	
test.node	8	defines the instance (node), which executes the testing actions like a query	
test.query	true	Defines if the test node executes a query to all nodes	
test.publish	false	Defines if the test node does a publishing	
log		Defines properties for the logging	
log.properties	log4j.properties	Defines the path to the file with the logging properties	

Test Usage

The ant build.xml file allows an easy compilation and running of the DMIS code. Therefore, the following command lines can be used to execute a compilation and creation of a .jar file for the distribution, which is created in the *dist* folder:

```
ant build
```

To run the example you have to execute the following command line. The *-Darg1=10* defines the number of DMIS instances which are created and used within the example.

```
ant run -Darg1=10
```

This page is intentionally left blank

Annex 4. Semantic Information Service

Semantic Information System for Grid4All

Web Service Guide

1. Introduction

This document is a brief guide to the implementation of the Semantic Information System (SIS) of the Grid4All project. This chapter introduces the reader to the Grid4All project and the purpose of the SIS. The available operations of the SIS as a Web Service are presented in chapter 2, while chapters 3 and 4 describe the key parts of the underlying implementation. Chapters 5, 6 and 7 address some issues related to the use of the SIS Web Service: The creation of remote peers, the installation of the Web service, and local programmatic use.

1.1. Overall description of the Grid4All project

Grid4All embraces the vision of a democratic Grid as a ubiquitous utility whereby domestic users, small organizations and enterprises may draw on resources on the Internet without having to individually invest and manage computing and IT resources.

Internet and its services are central to European life - at work and at home, indoors and outdoors. The next logical step is to provide the possibility to plug-and-upgrade personal computing capacity and to secure collaborative edutainment space on the fly through managed sharing of IT resources over the Internet. Grid4All will enable Grid services to evolve from high performance computing niche markets to a multipurpose service for anybody.

Grid4All - by leveraging the large base of broadband users - is aiming to advance the pervasiveness of Grid computing across society.

The objectives of Grid4All for the Grid community include the following:

- Alleviate administration and management of large scale distributed IT infrastructure - by pioneering the application of component based management architectures to self-organising peer-to-peer overlay services.
- Provide self-management capabilities - improve scalability, resilience to failures and volatility, thus paving the way to mature solutions enabling deployment of Grids on the wide Internet.
- Widen the scope of Grid technologies by enabling on-demand creation and maintenance of dynamically evolving scalable virtual organisations, even short lived.
- Apply advanced application frameworks for collaborative data sharing applications executing in dynamic environments.
- Capitalise on Grids as revenue generating sources to implement utility models of computing.

For further information concerning Grid4All project please visit http://grid4all.eu

1.2. Description of the SIS

The Semantic Information System (SIS) provides a matching and selection service concerning offers and requests (place any of the two kinds of orders i.e. either offers or requests) of resources and services, placed by peers (humans or software agents)

within grid environments . This service is used by the Grid4All market place. In the market place, resource/service consumers and providers negotiate traded entities in auction-based markets, where these markets are spontaneously initiated (instantiated) by different actors, such as resource/services providers, consumers, or 3rd party ones. Markets and their traded resources are accessed as services that are themselves advertised at the Semantic Information System.

The SIS acts as a registry for Market Services and Application Services. The registry may be queried by software agents as well as by human users to select advertised services and resources: Matchmaking is performed by processing various criteria concerning resources and other application specific traded domain entities, as well as services' profiles. The returned query results are ranked according to resources/services matching characteristics and providers'/consumers' features.

The matchmaking process is executed in the following cases:

- A user (consumer) makes a request in provider-initiated markets offering services/resources, and the matched offers are returned,
- A user (provider) makes an offer in consumer-initiated markets, and the matched requests are returned.

Service matchmaking is performed by inspecting the I/O parameter types of registered services, discovering their semantic relationship (similarity) with the query. To identify whether an advertised service matches a service query, it must first be identified whether the I/O types match. Reasoning services are used to identify, in addition to exact, partially matched types (via a subsumption relation).

After the matchmaking is completed, and a set of results is obtained, there is an additional step prior to their presentation: the ranking/selection process. Matchmaking performs a coarse-grained distinction of results, according to the type of matching. The ranking process provides an ordering of results which reflects the preferences of the user (e.g. preference on specific peers). It should be noted that the ranking process is implemented as a component of SIS, namely the Selection component. Finally, the list of results is returned to the user. According to what specifications have been given as input to the system, a list of market or applications service endpoints may be returned.

SIS is provided for public use as a Web Service itself. The SIS Web Service has been implemented as an Enterpise Java application. The Apache Axis SOAP engine (version 1.4) is used to expose the functionality of the service, convert service I/O information to SOAP messages (and vice versa), and exchange such messages with service clients. To support the required functionality, the service provides a set of operations and a set of complex type definitions that abstract ontology data. The exposed operations and the complex types are described in the WSDL description of the service, and form the "external API" of the SIS.

1.3. Main Features

The Semantic Information System of the Grid4All project uses semantic web technologies to facilitate the discovery, matching and selection of services and resources. Semantic descriptions of entities to be discovered are stored into the SIS registry. These descriptions are instances of an ontology that has been developed

especially for the needs of the SIS. Furthermore, semantic technologies are used for querying and retrieving information from the system.

The SIS, as a service discovery mechanism within the Grid4All, provides a registry for performing queries in the purpose of discovering available services/resources that fulfill certain criteria imposed by peers within the Grid4All environment.

The functionality of the SIS is described as a set of features provided by the system. The major features are:

- Advertising of market-related request or offer information related with traded resources and services, and
- Querying in order to obtain a list of relevant services: application services or services that expose market characteristics and information about resources ordered (as a consumer request or as a seller offer) within these markets.

1.3.1. Market advertisement

A Market advertisement is the process of inserting a new offer or request description (by a provider or consumer respectively) into the SIS registry. Such descriptions contain information about the entities that are traded by the associated markets, that is, resources and services, as well as information about the related markets, the participants i.e. providers and prospective consumers of resources and services. Information related to markets is represented as instances of an ontology schema that has been developed in the context of the SIS. A description has the form of object-property-value RDF triples.

Advertisement is supported by the SIS Web Service API. No authoring of formal descriptions of the input information is required from the users in order to create and submit an ontology instance. Advertisement is supported in different ways for different kinds of agents, which are the actors of market advertisement: Consumers and providers of Grid resources. Both these actors have to subscribe into the SIS in order to make advertisements. Only initiated and running services are advertised in the SIS.

Providers advertise registry descriptions of markets that sell resources/services, or the offered resources/services themselves. Concerning markets, these are descriptions of forward markets in which providers sell resources/services to potential consumers. The Provider advertised information is called “Offer”. Providers are able to advertise simple or complex offers, i.e. a list of AND/(X)OR service/resource configurations at different prices, availability times etc (also called a bundle offer). Descriptions of available tradeable resources as well as of markets trading these resources are referred to as *Offers*. Consumers advertise descriptions of reverse markets in which they look for resources/services of intended characteristics, thus forming requests. Descriptions of specific resource characteristics and of reverse markets initiated by consumers who act as provisional buyers of these resources are referred as *Requests*. Both Offers and Requests are specific kinds of *Market Orders*.

A market is to be advertised directly by its initiator, either provider or consumer, or through the Market factory which instantiates this market. Market orders, that is, offers or requests contain the following information:

- Market related information: The description of a market where the resource/service is going to be traded: This includes the location of a market, and its starting and closing time.
- Traded resource/service related information: The description of the technical characteristics of the traded services or resources (i.e. the configuration) in terms of capacity, quality of service, time of availability, etc.
- Offer/request related information: The description of pricing policy (type of related market auction), initial price auction price (minimum price for a forward auction and maximum for a reverse auction).
- Contact information: Information about the provider or consumer.

As a result of advertisement, offer and request descriptions are stored in the SIS registry.

1.3.2. Application Service advertisement

The registration of services involves the submission of services (application or services exposing resources) to be discovered by service providers. A provider submits a service description in WSDL and a respective annotation document (in xml predefined structure). The annotation document provides a mapping between advertised service I/O types and concepts in OWL ontologies (semantic annotations) that are stored in the SIS registry. As a result, an *OWL-S profile* document is automatically generated and inserted in the SIS registry. During service advertisement, the annotation document can be created by inspecting the WSDL structure and it can be filled by the service provider using OWL ontology elements such as resource descriptions, which must already be registered in the SIS and be publicly available. An important part of the registration process is validation. Before actually performing a registration, the provided information needs to be inspected so that it is ensured that no type mismatch in the names of semantic annotations is encountered, and also that consistency of the registered descriptions in the knowledge base is maintained. Similarly, for the definition of queries for registered resources/services, the user can use such dynamic forms to create descriptions of resources and make specifications related to market (or market order) properties.

1.3.3. Querying

In the context of Grid4All, service/resource providers try to find the appropriate (matched) consumers and, respectively, consumers try to find appropriate (matched) providers. The purpose of the query feature of the SIS is to provide to prospective providers/consumers an ordered list of available markets already published into the system, that match certain criteria concerning their own characteristics, as well as the characteristics of the traded goods: performance and QoS characteristics of a service, or the configuration of resources, pricing, market and location of markets, resources and services criteria. Querying answering is based on matching and selection processes.

Matching is the identification of a set of semantic descriptions of markets that satisfy the requirements posed in the query. Selection refers to the ranking of the advertised matched markets according to certain characteristics including the capacity of resources, preferences and intentions of providers and consumers.

A provider or a consumer (human or software) agent submits query data through the SIS API (described in following paragraphs of this document). A particular query contains semantic information, such as references to ontology entries, which are used during semantic matchmaking in order to discover services, resources or markets which are already advertised in the SIS repository. Users are not expected to be familiar with any ontology specific query language. The result of a query is an ordered list of matched market descriptions that the searcher (provider or consumer) can exploit in order to acquire the corresponding resource/service. The result of a query can also be a list of matched providers or consumers. Finally, queries in SIS are authenticated. Only registered agents who have successfully logged into the system can use the query functionality.

2. The SIS Web Service

The SIS utilizes two main components: The matchmaking component and the selection component. These two components, interacting through an internal API, are accessible though a uniform external API presented in this section.

The API provided by the SIS is available as a set of web service operations. The purpose for a web service implementation is:

- To provide access to the system features by agents acting in the context of the Market-based Resource Management System, i.e. consumers and providers,
- To facilitate interoperation with other Grid4All components
- To facilitate the automation of system testing and benchmarking

The external API of the SIS comprises a set of methods (operations) which implement service advertisement and querying within Grid4All, as described in the next section.

2.1. Agent Management operations

Agents interacting with the SIS may take either the role of a provider or a consumer. Providers offer traded resources, while consumers submit requests for offered traded resources. Both the providers and the consumers can initiate markets and advertise them. The SIS Web Service API provides the following methods for the agent management (Package: gr.aegean.icsd.ailab.sis - Interface: IAgentManagement):

2.1.1. registerProvider

This method is used in order to subscribe a new provider in the SIS. Providers need to be subscribed in order to register provider initiated markets or application services. The method's signature is given below:

```
String registerProvider(
  String Username,
  String Password,
  String Location)
  throws AgentAlreadyExistsException,
         InvalidAgentDescriptionException
```

`Username` is a unique identifier of the agent, by which it is known to other agents.

`Location` is a string describing the area where this agent originates from.

`AgentAlreadyExistsException` is thrown if an agent with the given username has already been registered in the SIS.

`InvalidAgentDescriptionException` is thrown if some arguments are null or empty.

The method returns a unique identifier that the agent has to provide in other methods.

2.1.2. registerConsumer

This method is used in order to subscribe a new consumer in the SIS. Consumers need to be subscribed in order to register consumer initiated markets. The method signature is given below:

```
String registerConsumer(
  String Username,
  String Password,
  String Location)
  throws AgentAlreadyExistsException,
         InvalidAgentDescriptionException
```

For the description of parameters see 2.1.1 registerProvider.

The method returns a unique identifier that the agent has to provide in other methods.

2.1.3. updateAgent

This method updates registration information of an agent (provider or consumer). Its signature is the following:

```
void updateAgent(
  String AgentId,
  String Username,
  String Password,
  Location)
  throws NoSuchAgentException,
         InvalidAgentDescriptionException
```

`NoSuchAgentException` is thrown if an unknown agent id is provided.

2.1.4. deleteAgent

This method removes an agent from the SIS registry. As a side effect, all advertisements and queries that the specific agent has performed depending on its type (consumer or provider) are also removed.

```
void deleteAgent(
  String agentId)
  throws NoSuchAgentException
```

2.2. Advertisement operations

Markets are advertised by both providers and consumers. Providers advertise provider-initiated markets, as well as application services, and consumers advertise consumer-initiated markets. The SIS Web Service API defines the following operations for advertisements (Package: gr.aegean.icsd.ailab.sis - Interface: IAdvertisement):

2.2.1. advertiseOffer

This method creates advertisements for resources' offers. An offer is specified in a provider initiated market. In order to insert an advertisement, an agent must have registered to the SIS. The method signature is:

```
String advertiseOffer(
  String providerId,
  Offer offerDescription,
  String marketURL)
  throws InvalidURLException,
         InvalidAgentRoleException,
         InvalidDescriptionException,
         NoSuchAgentException
```

`ProviderId` is a unique identifier for a 'provider' agent (obtained after registration, see agent management).

`OfferDescription` is a description of an Offer (The offer must be an instance of either ClusterOffer or ComputeNodeOffer). `Offer` class is an object-oriented wrapper for data describing an offer that is stored in the SIS. The detailed form of `Offer` objects is described in the Javadoc description of the SIS API (Package: gr.aegean.icsd.ailab.sis.metadata). All `Offer` objects contain information about the offered resources and about markets offering these resources, as described in the SIS ontology.

`MarketURL` is the URL of the market service in which the specific offer is negotiated.

The method returns an identifier of the advertised offer. It throws the following exceptions:

`InvalidDescriptionException` is thrown when the semantic information is malformed.

`InvalidURLException` is thrown when the market URL is null or malformed.

`NoSuchAgentException` is thrown when an invalid agent Id is provided

`InvalidAgentRoleException` is thrown when a consumer Id is provided (instead of a provider Id)

2.2.2. advertiseRequest

This method advertises consumer initiated markets. Consumers advertise these markets providing information about the kinds of resources they want to utilize. They also provide information about the markets. Both kinds of information are stored in the SIS ontology after request advertisement. The form of the `advertiseRequest` method is the following:

```
String advertiseRequest(
  String ConsumerId,
  Request RequestDescription,
  String MarketURL)
  throws InvalidURLException,
         InvalidAgentRoleException,
         InvalidDescriptionException,
         NoSuchAgentException
```

`ConsumerId` is a unique Id for a 'consumer' agent.

`RequestDescription` is a description of a request (Must be an instance of either ClusterRequest or ComputeNodeRequest). Class `Request` is an object-oriented wrapper for information about the intended characteristics of tradeable resources which a customer would like to purchase in a reverse auction.

`MarketURL` is the endpoint reference of the market service.

2.2.3. advertiseAbstractOrder

This method is called by an agent, either a provider or a consumer, in order to advertise a third-party initiated market, that is, a market that does not negotiate any tradeable resources at the time of its creation and advertisement. Markets of these types are discoverable through queries by both providers and consumers whose query

criteria match the description of a particular order. The form of the method is the following:

```
String advertiseAbstractOrder (
  String AgentId,
  AbstractMarketOrder orderDescription,
  String MarketURL)
  throws InvalidURLException,
         InvalidAgentRoleException,
         InvalidDescriptionException,
         NoSuchAgentException
```

`AgentId` is a unique identifier of the agent.

`OrderDescription` is a description of an abstract order request as an instance of class `AbstractMarketOrder`, which serves a similar purpose to class `Request` as an object-oriented wrapper for abstract resource descriptions.

`MarketURL` - The URL of the market service.

2.2.4. advertiseService

Method `advertiseService` makes an advertisement for an application service. The specified service is translated to an OWL-S profile, which is registered in the SIS. To produce the OWL-S profile, this method makes use of an external annotations file, the contents of which are specified in the `Annotations` String.

```
String advertiseService(
  String ProviderId,
  String WsdlFileContent,
  String Annotations)
  throws NoSuchAgentException,
         InvalidAgentRoleException,
         InvalidDescriptionException,
         InvalidDomainException
```

`ProviderId` is a unique identifier of the agent making the advertisement

`WsdlFileContent` is the content of the WSDL document which describes the advertised service

`Annotations` is an XML String with the contents of the annotation file (EAF) for the particular service.

`InvalidDomainException` is thrown if the EAF contains ontology concept URIs that are not identified by SIS (they do not exist in the SIS registry or they are mismatched).

The method returns the service namespace identifier.

2.2.5. updateAdvertisement

This method replaces an existent advertisement description with a new one. The unique id given after the initial advertisement stays the same.

```
void updateAdvertisement(
  String AgentId,
  String DescriptionId,
  MarketOrder MarketOrderDescription,
  String MarketURL)
  throws NoSuchDescriptionException,
```

```
        invalidURLException,
        NoSuchAgentException,
        InvalidDescriptionException,
        InvalidAgentRoleException,
        InvalidURLException
```

The public class **MarketOrder** extends java.lang.Object and implements SISEntity, SPARQLFragment.The market order is the initial specification of what (and by whom) is being traded within a given market. In a provider-initiated market an order becomes "Offer" for a min price or reservation price that a seller is willing to sell. In a consumer initiated market an Order becomes "Request" for the max price that the consumer is willing to pay.

2.2.6. deleteAdvertisement

This method removes an advertisement with the given Id from the SIS.

```
void deleteAdvertisement(
  String AgentId,
  String DescriptionId)
  throws NoSuchDescriptionException
```

`DescriptionId` - The unique Id of the advertised order
`AgentId` - The Id of the agent ordering the deletion
`NoSuchDescriptionException` - If there is no description with given Id advertised

2.3. Querying operations

The SIS supports the following types of queries:

- Queries placed by consumers against provider-initiated markets
- Queries placed by providers against consumer-initiated markets (reverse markets).
- Queries placed by providers and consumers for third-party initiated markets.
- Queries issued by consumers for available application services.

The Querying Interface in SIS API supports operations as described in the following paragraphs (Package: gr.aegean.icsd.ailab.sis - Interface: IQuerying):

2.3.1. queryProviderInitiatedMarkets

This method forms queries in a provider-initiated market in the SIS registry, executes them and returns the results in a QueryResults object. The method signature is given below:

```
QueryResults queryProviderInitiatedMarkets(
  String AgentId,
  Request RequestDescription,
  MarketQuery MarketRelatedConstraints,
  int NumOfResults)
  throws NoSuchAgentException,
         InvalidAgentRoleException,
         InvalidDescriptionException
```

`AgentId` is the unique Id for the agent (consumer)

`RequestDescription` is a description of a request (an instance of class `Request`). *Note*: The request that is passed in this method must not be specified in a consumer-initiated market, so the respective field in the `RequestDescription` object should be null. Otherwise, the SIS will throw an `InvalidDescriptionException`.

`MarketRelatedConstraints` contains restrictions i.e. special characteristics concerning the markets which are being queried. It may also contain restrictions related to the market auctions (auction restrictions are encoded in an `AuctionQuery` object, which is referenced by the `MarketRelatedConstraints` object).

`NumOfResults` is the maximum number of desired results.

`QueryResults` objects are composed of two parts: The actual list of results (A String array), as well as a query identifier, which can be used to re-run the query (see repeatQuery method).

2.3.2. queryConsumerInitiatedMarkets

This method places queries for consumer-initiated markets in the SIS registry, executes them and returns the results in a QueryResults object. The method signature is given below:

```
QueryResults queryConsumerInitiatedMarkets(
  String AgentId,
  Offer OfferDescription,
  MarketQuery MarketRelatedConstraints,
  int NumOfResults)
  throws NoSuchAgentException,
         InvalidAgentRoleException,
         InvalidDescriptionException
```

`AgentId` is the unique Id for the agent (provider)

`OfferDescription` is a description of an offer as an instance of class `Offer`. *Note*: The offer that is passed in this method must not be specified in a provider-initiated market, so the respective field in the `OfferDescription` object should be null. Otherwise, the SIS will throw an InvalidDescriptionException.

`MarketRelatedConstraints` contains restrictions concerning the markets which are being queried. It may also contain restrictions related to the market auctions (auction restrictions are encoded in an `AuctionQuery` object, which is referenced by the `MarketRelatedConstraints` object).

2.3.3. queryProviders

This method places queries for providers in the SIS (issued by agents that have advertised a consumer initiated market), executes them and returns the results in a `QueryResults` object. Its purpose is to help a consumer find potential providers to participate in the former's market. The method signature is given below:

```
QueryResults queryProviders(
  String AgentId,
  String requestId
  int NumOfResults)
  throws NoSuchAgentException,
```

```
        InvalidAgentRoleException,
        InvalidDescriptionException
```

`AgentId` is the unique Id for the agent (consumer)

`requestId` is the Id of a request that has been previously advertised in the SIS .

`NumOfResults` - is the number of expected results.

2.3.4. queryConsumers

This method registers queries for consumers in the SIS (issued by agents that have advertised a provider initiated market), executes them and returns the results in a QueryResults object. Its purpose is to help a provider find potential consumers to participate in the former's market. The method signature is given below:

```
QueryResults queryConsumers(
  String AgentId,
  String offerId
  int NumOfResults)
  throws NoSuchAgentException,
        InvalidAgentRoleException,
        InvalidDescriptionException
```

`AgentId` is the unique id for the agent (consumer)

`offerId` is the id of a resource offer, which has been previously advertised in the SIS. .

`NumOfResults` - is the number of expected of results. It is used in top-N queries.

2.3.5. queryServices

This method places a query for application services, specifying a Service Profile Input list and a Service Profile Output list. These two lists contain ontology concepts' URIs from known ontologies ('known' means that their information is available to the SIS and available to all agents). Also, a domain name needs to be specified, in order to determine the ontology (or set of ontologies) from which the concepts' URIs are drawn. The method signature is given below:

```
QueryResults queryServices(
  String AgentId,
  String domainName,
  String[] inputTypes,
  String[] outputTypes)
  throws InvalidDescriptionException,
        InvalidDomainException,
        InvalidAgentRoleException,
        InvalidAgentDescriptionException
```

`AgentId` is the unique Id for the agent making the query (a consumer)

`DomainName` is a human readable name which is used to determine the ontology(ies) from which the URIs in the `inputTypes` and `outputTypes` lists are drawn to form the query.

`inputTypes` contains a list of ontology concepts' URIs, describing the requirements w.r.t. the Service Profile Inputs of the services being queried.

`outputTypes` contains a list of ontology concepts' URIs, describing the requirements w.r.t. the Service Profile Outputs of the services being queried.

`InvalidDomainException` is thrown if an unknown domain name is specified.

2.3.6. repeatQuery

This method executes a previously registered query. This eliminates the need to produce additional ontology elements in order to perform repetitive executions of the same query. However, it should be noted that queries are automatically removed from the SIS after a specific period of time (this time period can be defined in the SIS service configuration files), so `repeatQuery` must be called within this period. The method signature is given below:

```
QueryResults repeatQuery(
  String AgentId,
  String QueryId,
  int NumOfResults)
  throws InvalidQueryIDException,
  NoSuchAgentException,
  InvalidAgentRoleException,
  NoSuchDescriptionException,
  InvalidDomainException
```

`AgentId` is the unique Id for the agent who has issued the specific query (an agent can only repeat its own queries)

`QueryId` is the Id which is obtained after a successful execution of one of the query methods described above. This id can be retrieved from the `QueryResults` object which is returned.

`InvalidQueryIDException` is thrown if a non-existent query Id is specified. It is also thrown if an agent places a query that has been removed from the SIS.

2.3.7. removeQuery

This method removes a query from SIS, so any ontology elements created during the initial query processing are removed. After a query is removed, `repeatQuery` cannot be executed and a new query needs to be created.

```
void removeQuery(
  String AgentId,
  String QueryId)
  throws InvalidQueryIDException,
  NoSuchAgentException
```

`AgentId` is the unique id for the agent who has issued the specific query (an agent can only remove its own queries)

`QueryId` is the Id which is obtained after a successful execution of one of the query methods described above. This Id can be retrieved from the `QueryResults` object which is returned.

2.4. Selection operations

The SIS Web Service provides several operations through which agents can interact with the Selection Service module. The Selection Service module is used to rank query results based on agent preferences and query load balancing techniques (Package: gr.aegean.icsd.ailab.sis – Interface: ISelection).

2.4.1. selectMarkets

This method informs the Selection Service about a selected ranked subset of the set of Martket results obtained from a query for markets. The method signature is provided below:

```
void selectMarkets(
  String AgentId,
  String QueryId,
  String[] SelectedMarkets)
  throws NoSuchAgentException,
         InvalidAgentRoleException,
         InvalidQueryIDException,
         InvalidMarketException
```

`AgentId` is the Id of the agent performing the selection

`QueryId` is the Id of the query this selection is a response to

`SelectedMarkets` is a list containing a selected ranked subset of the set of results obtained from a query for markets. This list is taken into consideration in future market queries performed by this agent.

`InvalidMarketException` is thrown if a non existent market identifier is provided.

2.4.2. selectConsumers

This method informs the Selection Service about a selected ranked subset of the set of Consumer information results obtained from a query for consumers. The method signature is provided below:

```
void selectConsumers(
  String AgentId,
  String QueryId,
  String[] SelectedConsumersUsernames)
  throws NoSuchAgentException,
         InvalidAgentRoleException,
         InvalidQueryIDException
```

`AgentId` is the Id of the provider performing the selection

`QueryId` is the Id of the query this selection is a response to

`SelectedConsumersUsernames` is a list containing a selected ranked subset of the consumer usernames obtained from a set of query results obtained for consumers. This list is taken into consideration in future queries performed by this agent.

`InvalidAgentRoleMarketException` is thrown if a consumer tries to call this method

2.4.3. selectProviders

This method informs the Selection Service about a selected ranked subset of the set of Provider results obtained from a query for providers. The method signature is provided below:

```
void selectProviders(
  String AgentId,
  String QueryId,
  String[] SelectedProvidersUsernames)
  throws NoSuchAgentException,
         InvalidAgentRoleException,
```

```
        InvalidQueryIDException
```

`AgentId` is the Id of the consumer performing the selection

`QueryId` is the Id of the query this selection is a response to

`SelectedProvidersUsernames` is a list containing a selected ranked subset of the providers usernames obtained from a query for providers. This list is taken into consideration in future queries performed by this agent.

`InvalidAgentRoleMarketException` is thrown if a provider tries to call this method

2.4.4. setPreferencesForProviders

Consumer agents may call this method to express their preference towards specific provider agents. These preferences are encoded as real numbers in the [0..1] space.

```
void setPreferencesForProviders(
  String AgentId,
  String[] PreferredAgentsUsernames,
  double[] Preferences)
  throws InvalidAgentDescriptionException,
         InvalidDescriptionException,
         InvalidAgentRoleException
```

`AgentId` is the Id of the agent expressing its preferences

`PreferredAgentsUsernames` is a list of agent usernames corresponding to the agents which the calling agent expresses its preferences for.

`Preferences` is a list of preference values. This list is the same size as `PreferredAgentsUsernames`. The value at `Preferences[i]` refers to the agent at `PreferredAgentsUsernames[i]`.

2.4.5. setPreferencesForConsumers

Provider agents may call this method to express their preference towards specific consumer agents. These preferences are encoded as preference values (real numbers in the [0..1] space).

```
void setPreferencesForConsumers(
  String AgentId,
  String[] PreferredAgentsUsernames,
  double[] Preferences)
  throws InvalidAgentDescriptionException,
         InvalidDescriptionException,
         InvalidAgentRoleException
```

`AgentId` is the Id of the agent expressing its preferences

`PreferredAgentsUsernames` is a list of agent usernames corresponding to the agents which the calling agent expresses its preferences for.

`Preferences` is a list of preference values. This list is the same size as `PreferredAgentsUsernames`. The value at `Preferences[i]` refers to the agent at `PreferredAgentsUsernames[i]`.

2.4.6. setPreferencesForQueryTypes

This method informs the Selection Service about the preferences of a provider over a list of query types. There are several types of advertisements (queries) that a

provider (consumer) can make, distinguished by the traded entity (tradeable resources or application services). In some situations, it may be the case that a provider is interested in helping consumers searching for a specific type of advertised entities more than it is interested in helping consumers searching for the other types. These preferences can be added to the Selection Service using this method, and used during ranking in query processing. To express the preferences, two arrays are used (`PreferredQueryTypes` and `Preferences`). A preference for the query type in `PreferredQueryTypes[i]` is expressed with a double value in `Preferences[i]`. The list of available query types can be obtained by calling `getQueryTypes()(see next method)`. The preferences are expected to be computed by the agent that calls this method.

```
public void setPreferencesForQueryTypes(
  String AgentId,
  String[] PreferredQueryTypes,
  double[] Preferences)
  throws InvalidAgentDescriptionException,
         InvalidDescriptionException,
         InvalidAgentRoleException
```

2.4.7. getQueryTypes

This method returns a list of query types, i.e., a list of identifiers for the major types of traded entities (compute nodes, clusters, application services). A provider can express its preferences over some of these types by using `setPreferencesForQueryTypes()`.

```
String[] getQueryTypes()
```

2.5. Domain ontology operations

Domain ontologies used in the SIS are associated with unique domain names, which are human readable descriptions of domains. It is expected that agents advertising or querying for application services are familiar with ontologies in general, since such knowledge is required for semantically annotating services (i.e. mapping service operation I/O types to domain ontology concepts) and when making queries (i.e. defining the required service I/O types by submitting lists of domain ontology concept URIs). The supported domain ontologies are stored in a specific directory of the SIS service (under WEB-INF) which is accessible by HTTP to all agents interacting with SIS. The SIS API provides several methods to assist agents in retrieving these ontologies by domain name (Package: gr.aegean.icsd.ailab.sis – Interface: IDomainRegistry).

2.5.1. getDomainNames

This method returns the domain names of all the domain ontologies which are maintained by the SIS.

```
String[] getDomainNames()
```

2.5.2. getOntologyURLFor

This method returns the URL of the ontology which is associated with the provided domain name. This URL is a real resolvable address and it can be used to access the ontology through HTTP.

```
String getOntologyURLFor(String name)
```

2.5.3. getOntologyNamespaceURIFor

This method returns the namespace URI of the ontology which is associated with the provided domain name. A namespace URI may or may not be resolvable, and it used almost exclusively as an identifier.

```
String getOntologyNamespaceURIFor(String name)
```

3. Overview of the SIS Core API

This chapter describes the main characteristics and functionalities of the Semantic Information System for the Grid4All project. It is intended to be used as a guide, not as a detailed reference to the code. Readers are expected to have a basic grasp of the key Grid4All project notions and the functionality offered by the Semantic Information System. Some familiarity with Semantic Web technologies is also expected.

The focus of the chapter is on the description of the matchmaking component of the SIS. The main issues the SIS Core API deals with are:

- The interaction with the Grid4All resource ontology (insertion and deletion of ontology elements)
- Service advertisement (conversion of WSDL descriptions to OWL-S profiles)
- Querying of resources and services using SPARQL
- Ranking of services in service queries

The SIS Core API makes extensive use of the Jena framework and the Pellet reasoner API for the manipulation of ontologies. Potential future developers are required to have at least some basic knowledge of these frameworks. Other libraries that are used include Log4J, Commons Collections, Commons Logging, all of which are projects of the Apache Foundation, and the Semantic Matching Framework (SemMF, obtainable from http://semmf.ag-nbi.de).

3.1. The SIS registry

The Semantic Information System maintains descriptions of resources and services using Semantic Web technologies. However, the representation of resources (as well as markets and market orders) differs from the representation of application services. Grid resources are represented as elements of an ontology which has been designed specifically to support semantic representation, querying and retrieval of resources, markets, orders and peers. The representation concerns a) types and characteristics of the resources, b) properties/constraints related to the specific offers and requests, c) properties of the markets to which the specific orders are placed.

On the other hand, application services which are advertised to the SIS are represented as OWL-S documents. OWL-S is an OWL-based upper level ontology of service concepts for describing the properties and capabilities of web services in a machine interpretable form. It consists of three main parts/classes: the Service Profile, the Process Model, and the Service Grounding. The Service Profile is used for advertising and discovering services, so this is the part of the semantic descriptions that the SIS uses in querying.

Concluding from the above, there are actually two distinct registries maintained by the SIS, both of which make use of Semantic Web technologies: The Resource Registry, which essentially comprises the Grid4All ontology, and the Service Registry which is a directory (in the file-system where the SIS resides) where service descriptions encoded in OWL-S documents are stored.

3.1.1. Resource Registry

The main entities that are described in the Grid4All ontology are the following:

- Resources
- Market orders (offers and requests)
- Markets
- Agents (consumers and providers)

Resources are entities that are offered by resource providers or requested by resource consumers, and are traded in e-markets initiated by consumers, providers and third-parties. Resources are described at the logical level, i.e., their descriptions focus on certain properties, such as performance characteristics, leaving out details concerning, for example, the underlying implementation. This choice is driven by the fact that ordinary Internet users are less concerned about the specific hardware properties of a resource and more about the service rendered by the resource. There are several different categorizations of Resources, such as Atomic, Aggregated, Composite, according to whether they are composed of other resources or not. Information can also be provided regarding their capabilities (Capacity, QoS), location, and the market order where they are described, if they are tradeable. Only Compute Nodes and Clusters are considered as tradeable resources.

Agents can either act as providers or consumers. A provider can advertise an offer of a tradeable resource by creating a provider-initiated market, and make a query for consumers that may be interested in the specific offer, or wait for them to discover it. In order for consumer agents to discover provider-initiated markets, they need to form requests, in which more abstract resource descriptions are provided, and make queries that specify these requests. Once the consumers discover one or more markets that fit their queries' descriptions, they may decide to join them. Like providers, a consumer may also create a market (consumer-initiated markets), in which a request is specified, and either search for providers that may be able to satisfy the consumer's requirements, or wait for providers to discover their market. In this case, the queries of the providers specify descriptions of offered resources that are not part of an advertisement.

	Offer specified in Provider Initiated Market	Offer **not** specified in Provider Initiated Market	Request specified in Consumer Initiated Market	Request **not** specified in Consumer Initiated Market
Offer specified in Provider Initiated Market	X	X	X	V
Offer **not** specified in Provider Initiated Market	X	X	V	X
Request specified in Consumer Initiated Market	X	V	X	X
Request **not** specified in Consumer Initiated Market	V	X	X	X

Table 1. Possible matches between offers and requests

It is important to highlight that a market order may be part of either an advertisement or a query for other advertisements; it cannot be used for both. That is, a provider that has created a marketplace using an offer description, cannot use this description to form a query for consumer initiated markets, but just for consumers which have issued queries for provider initiated markets. The same principle applies to consumers. Table 1 shows the possible matches between markets and market orders. The rows represent the types market order descriptions that may be part of a query, and the columns represent the type of market orders that are returned as query results.

Another thing to note is that the SIS, functioning solely as a registry, goes as far as just providing the endpoints to the market services. It does not take part in, or support in any other way, the process of negotiating that takes place in the markets.

3.1.2. Managing the Resource Registry with the SIS Core API

Interaction with the Resource Registry is performed by using the JENA framework and the Pellet reasoner API. However, these APIs operate at the level of generic RDF/OWL triples. The SIS Core API provides classes and interfaces that are drawn from the need to represent the entities of the Grid4All ontology in particular. The Java package `gr.aegean.icsd.ailab.sis.metadata` of the Core API contains most of the classes that are visible and accessible to client peers in order to form descriptions of SIS entities of any degree of complexity; these classes will be referred to as "*client description classes"*. Given that the reader is familiar with the Grid4Al ontology, it is easy to correspond a Java class (either abstract or concrete) to each of the concepts of the ontology, while the properties have been "translated" to Java class fields (for datatype properties) and references to other Java objects (for object properties). Of course, certain compromises had to be made in class design, because some aspects of the ontology, such as the multiple inheritance of concepts, could not be directly translated to Java. There are also some additional classes that can be used to construct request descriptions. These classes are located in the package `gr.aegean.icsd.ailab.sis.metadata.query`.

The SIS clients provide their offer/request descriptions using objects of the Java classes in the packages mentioned above. In order for those descriptions to be stored in the Grid4All ontology, they need to be converted to RDF triples. The Java package `gr.aegean.icsd.ailab.sis.metadata.ontology` provides classes for this purpose. The class that is responsible for this conversion is `OntologyTranslator`, which contains a static method (`translateDescription()`) for converting client descriptions to ontology elements and storing them in the Grid4All ontology. The ontology is accessible through the `ModelHandle` class as a Jena `Model` (method `getModel()`), or `OntModel` (with or without an attached reasoning engine, method `getOntModel()` and `getPlainOntModel()` respectively).

`ModelHandle` also makes use of some Pellet API classes (like `KnowledgeBase`, `Taxonomy`) to manipulate the ontology in cases where the Jena implementation either fails or is very slow. Inferred statements are also maintained in `ModelHandle` and can be obtained by calling `getRDFTypesModel()`. The RDF Model containing the inferred statements is not updated automatically after each insertion or deletion of ontology elements, so a call to `updateRDFTypes()` must be made first. Another method which is usually called prior to a query execution is `getRDFTypeEnhancedModel()`, which

returns a Model containing both the asserted and inferred statements of the Grid4All ontology.

As mentioned above, the `translateDescription()` method is responsible for the conversion of client descriptions to ontology statements. However, this method does not accept client description class objects as arguments; rather, it accepts objects of the Java class `ObjectDescription`, which are wrapped inside an object of class `ObjectDescriptionHandler`. An `ObjectDescription` object contains a description of a specific ontology element: Its identifier (URI), its RDF type, whether or not it is part of a request description (in which case, the `definedClass` field is true), as well as a list of datatype property values and a list of object property values. Each datatype property value is represented by a `DatatypeProp` object, which contains the following information:

- The URI of the property
- The RDF datatype of the property
- The property value, and/or a valid value range ("From-to" values)

Similarly, each object property value is represented by an `ObjectProp` object, in which the following information is provided:

- The URI of the property
- The range class URI(s) of the property
- The value(s) of the property (Identifiers of other `ObjectDescription` objects)
- The minimum cardinality of the property, if such a restriction exists
- The maximum cardinality of the property, if such a restriction exists

The `ObjectDescription` class can be considered as an intermediate state of client descriptions prior to serialization to the Grid4All ontology.

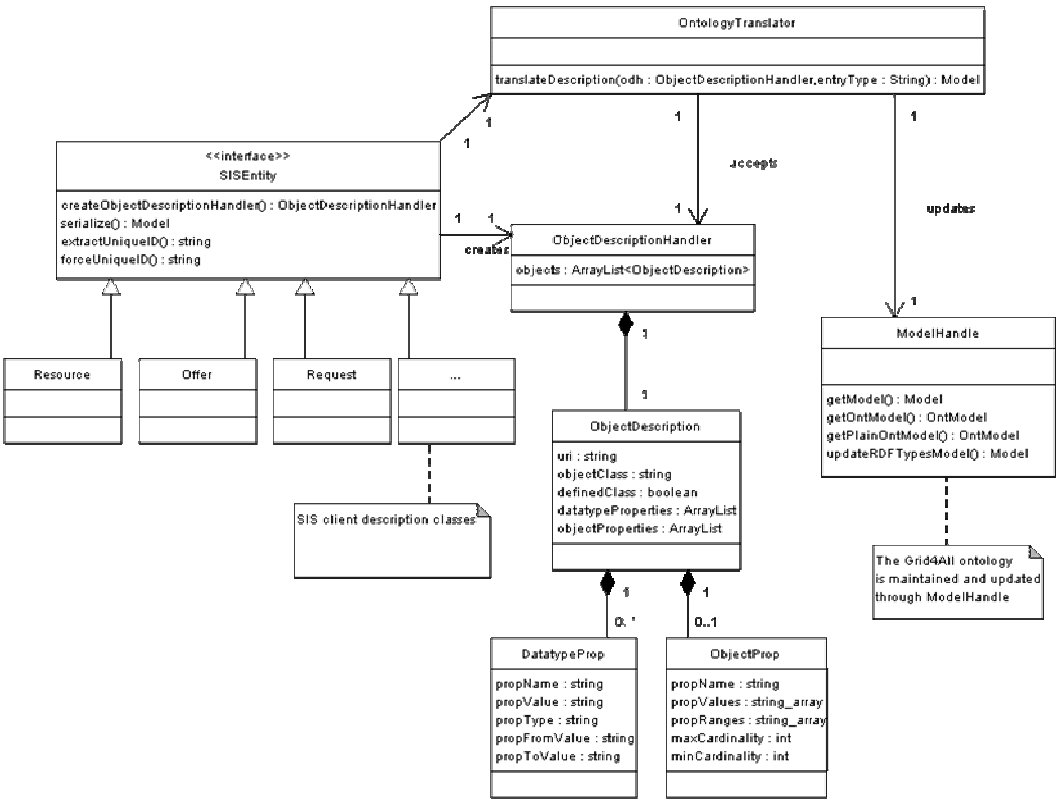


Figure 1. Class diagram for Resource Registry manipulation classes

The main reason that the `ObjectDescription`, `ObjectDescriptionHandler`, `DatatypeProp` and `ObjectProp` classes were originally created was because of the initial design of the SIS, in which the client descriptions were directly sent by HTML forms, and so there was a need to a) hold potentially large numbers of strings, many of which were referencing one another, and b) provide a uniform way of representing client descriptions and then passing them to a Java Servlet to be translated to ontology descriptions (the `OntologyTranslator` implementation emerged from this servlet). In addition, it was not considered practical to incorporate JENA objects and method calls directly in the client description classes mentioned in the beginning of this section, because this would possibly entail refactoring of these classes whenever a new JENA version was used in the SIS (The SPARQL engine API used in JENA, ARQ, is particularly known to present differences across successive JENA versions). Instead, client description classes contain a method to generate `ObjectDescriptionHandler` objects (`createObjectDescriptionHandler()` method) and then translate them through `OntologyTranslator`. The latter is performed within method `serialize()`, which is a method that all client description classes have to implement. This requirement is encoded in the `SISEntity` Java interface, which all client description classes implement. Apart from the `serialize()` and `createObjectDescriptionHandler()` methods, `SISEntity` also has two methods dealing with the unique identifiers of ontology elements (`extractUniqueID()` and `forceUniqueID()`).

The classes used for the representation of the Grid4All ontology elements, as well as their relationships, are shown in figure 1. These classes are also used in query processing, since queries involve providing a request or offer description, as described

previously. Therefore, queries usually entail additional insertions to the Grid4All ontology.

3.1.3. Querying the Resource Registry

The query process involves insertion of new elements to the Grid4All ontology, reasoning with the ontology and performing SPARQL queries. The ontology has been engineered in such a way that the matchmaking process is performed via the reasoning engine. To take advantage of the automatic classification of the reasoning engine and according to the requirements for offers and requests, market orders are represented in the following way: Resources' offers are represented as individuals of class Offer and resources' requests are represented as defined classes (classes described with some necessary and sufficient conditions). Generally, information related to offered resources is represented using individuals and/or primitive classes, while information related to requested resource descriptions is represented using defined classes. The inference engine is used to compute the inferred types of the individuals, or to automatically classify concepts. Potential query results can be obtained this way, but since defined classes cannot represent all the constraints posed by agents, additional SPARQL queries need to be created, in which these constraints can be expressed, and executed on the inferred ontology model.

The following algorithm describes the general steps that take place when a consumer submits a request for resources during a query for resource offers. Similar steps are followed during queries for resource requests, although in this case there is no search for instances of specific defined classes, but rather a search for classes under which specific individuals can be classified.

1. For every Request,
   a. Get the restrictions on object properties.
   b. Create a defined class using the restrictions.
   c. Add the created defined class to the registry.
2. For every Request/,
   a. Get the datatype property restrictions.
   b. Create a SPARQL query to retrieve markets, orders, or agents.
   c. If defined classes have been created in step 1, reference them in the SPARQL query: Only individuals that are recognized as instances of these defined classes will be retuned as valid query results. If no defined class is created (i.e., there are no object property restrictions), reference the default primitive classes of the Grid4All ontology (Request, Resource, Tradeable_Resource, etc) in the SPARQL query.
3. For every datatype property restriction,
   a. Add a corresponding constraint to the SPARQL query.
      i. If the restriction is a single specific value, then add an equality constraint.

ii. If the restriction is a range of values, then add a "greater than" and/or a "less than" constraint.

4. Ensure that all inferred statements are extracted from the Grid4All ontology, by using the inference engine on the ontology model.
5. Execute the SPARQL query, using both the asserted and the deduced knowledge to retrieve results.
6. Present results

The insertion of elements in the Grid4All ontology (step 1) has been covered in the previous section. The `ModelHandle` class is used to get the asserted and deduced knowledge from the ontology (step 4). To create the SPARQL queries that are needed to filter the classification results, it is necessary to be able to encode the descriptions given by SIS clients to SPARQL constraints. Client description classes that may be used to represent constraints are Request, Capacity, QoS and several others that reside in the Java package `gr.aegean.icsd.ailab.sis.metadata` Java package, as well as those that are in `gr.aegean.icsd.ailab.sis.metadata.query`. These classes implement the `SPARQLFragment` interface, in which the following methods are declared: `extractSPARQLVar()`, which is used to obtain an identifier of the constrained element, and `extractSPARQLBody()`, which returns the constraints in the form of SPARQL triples.

SIS entities implementing `SPARQLFragment can be used to create SPARQL` query bodies. Every entity (request, requested resource description, QoS requirements, etc.) that contains some constraints on its properties should be able to provide a SPARQL expression of these constraints. For example, suppose there is an object of a client description class implementing this interface, and in this object the following restrictions are represented:

- "Resource X1 is an instance of class Y"
- "Resource X1 has an integer value for property P, which is less than 5"

These restrictions will be translated to the following SPARQL restrictions:

```
?X1 rdf:type Y.
?X1 P ?valueOfP
FILTER (?valueOfP < "5"^^http://www.w3.org/2001/XMLSchema#int).
```

The `extractSPARQLBody()` method will return the triples in which the restrictions are encoded, and `extractSPARQLVar()` will return the variable name for the resource which is being queried (In the example, the variable name is "X1"). Note that if multiple `SPARQLFragment` objects are connected to each other, `extractSPARQLBody` will be called recursively, thus assembling a large SPARQL body containing restrictions for all these objects.. This happens e.g. when a `SPARQLFragment` contains other `SPARQFragments`, as is the case with a `ClusterQuery` object containing a `ComputeNodeQuery` object, which in turn contains a `HardDisk`, etc - all of which are `SPARQLFragments`. When the `clusterQueryObject.extractSPARQLBody()` method is called, the string that is returned contains restrictions about the Cluster description, the Compute node description, Hard disks, CPUs and so on.

Note that there is no dependency between `SISEntity` and `SPARQLFragment`. `SISEntity` objects contain information that ends up being encoded as ontology elements, while `SPARQLFragment` objects contain information that can only be expressed as a batch of SPARQL query constraints. The instances of most of the client description classes implement both interfaces because they contain both of these types of information, but there are also classes the instances of which have only one type of information. For example, `AuctionQuery` and `MarketQuery` do not contain any information that is inserted in the Grid4All ontology, therefore they only implement `SPARQLFragment`).

As the name implies, `extractSPARQLBody()` does not create a full SPARQL query. SPARQL queries contain the following parts:

- The query header, or prologue
- The query body

The query body contains restrictions expressed as triples with variables, filters etc. The query header contains prefix-URI mappings for the namespaces that are used in the query body, and it also contains the query variables, i.e. the variables to which query results are bound. For example, in the following example, the line between brackets is the query body, while the lines with starting with "PREFIX" and "SELECT <vars> WHERE" constitute the header/prologue of the query.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX myURI: <http://www.mywebsite.com/myNamespaceURI#>
SELECT ?x WHERE {
  ?x rdf:type myURI:SomeClass.
}
```

In order to create complete SPARQL query strings, the `SPARQLUtil` class from the `gr.aegean.icsd.ailab.sis.util` package is used. `SPARQLUtil` is heavily used in the SIS implementation, as it contains static methods for the creation of all the different types of SPARQL queries that are needed, like queries for offers, requests, markets, services, etc. Most of these methods accept SPARQL fragments as arguments and construct full SPARQL strings, which can then be used by the SIS.

`SPARQLUtil` also provides several methods to assist the creation of SPARQL constraints within the implementations of the `extractSPARQLBody()`, to avoid having repeated code bits that deal with string manipulation (there are two methods for that purpose, both called `createSPARQLTriple()` but with different argument lists).

Finally, `SPARQLUtil` provides two methods for query execution. Both of these methods are called `executeSPARQLQuery()` but differ in their argument lists, as one accepts any RDF model and can be used for querying any ontology (this is used in service queries), while the second accepts `ModelHandle` instances, and therefore can only query the RDF model provided by the Grid4All ontology (actually a shorthand for the first method, where the `Model` instance that is passed is obtained from `modelHandle.getModel()`.

Regarding the results returned from queries, these are stored in `HashMap` objects, i.e. hashtables implementing the Java Collections `Map` interface. The keys of the hashtable are the query variables, and each value which is paired with a key is a list of query results for the specific variable.

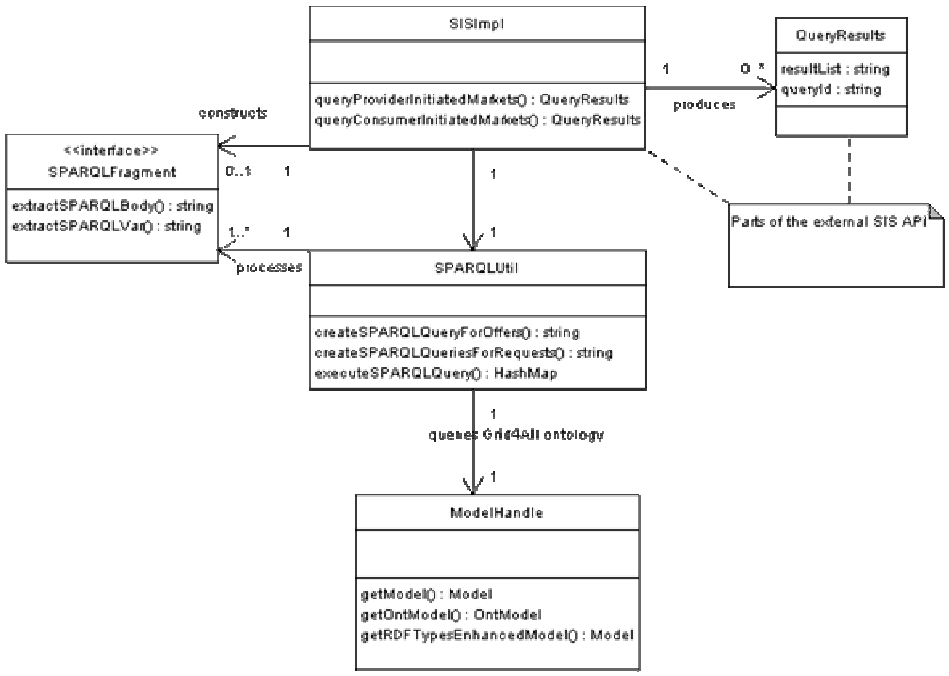


Figure 2. Class diagram for Resource Registry querying classes

Figure 2 shows the main classes involved in SPARQL query creation and execution. Client description classes that implement the `SPARQLFragment` interface are omitted from the diagram.

3.1.4. Service Registry

The SIS stores semantic descriptions of advertised Web Services in the form of OWL-S documents. Generally, we expect that agents advertising services are not familiar with the specifics of OWL-S, and can only provide description of services through WSDL documents. On the other hand, we expect that service creators are at least familiar with ontologies, and are aware of a set of well-known domain ontologies, which they can consult prior to advertising and, in general, they are able to logically associate service I/O types to some of the concepts of these domain ontologies and decide which concept best fits a WSDL I/O type description.

The issue here is to find a way to create semantic service descriptions (OWL-S profiles in particular), taking the following into consideration:

- No intervention should be made to the WSDL service descriptions. Such descriptions, in most cases, are automatically generated from web service frameworks using the source code of the services, so service creators may not deal with them at all.
- Ontology concepts should be available to web service creators.

We deal with this issue by using additional annotation documents, which are related to service descriptions in WSDL documents. Annotation documents are XML documents containing simple annotation blocks, one for every I/O parameter defined in the WSDL description. Information concerning an annotation block can be:

- Comments, written in natural language

- References to domain ontology concepts

Figure 3 shows an example of an annotation document which is created to annotate a WSDL description.

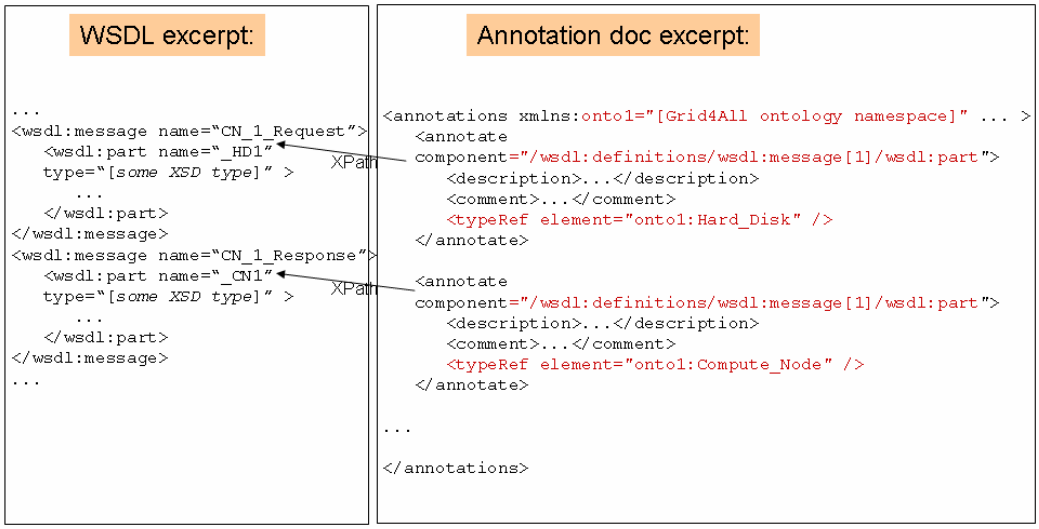


Figure 3. Annotation document.

Note that there are ontology elements specified in the annotation document, which are directly related to the I/O's of the WSDL description, while the latter is not by any way modified. Each annotation block (starting with <annotate> tag) is related to a message part in the WSDL through an XPath expression.

Agents advertising applications services are expected to provide a WSDL description and a complete annotation document for the former. The SIS itself does not provide a facility to help agents with annotation. However, a separate tool has been developed for this purpose (WSDL-AT annotation tool). This tool parses a WSDL file and automatically creates a correspondent annotation file with empty fields, where service providers should add the content. Service providers (or any other agent making the annotations) must fill the empty fields (concerning the WSDL I/O parameters) by consulting the appropriate domain ontology. Information about the domain ontologies supported by the SIS can be obtained using some related external API calls.

After a WSDL and the corresponding annotation document have been submitted to the SIS, they are translated to an OWL-S description. The code implementing the translation is located in the `WSDLToOWLS` class, and more specifically in method `loadWSDL()`. `WSDLToOWLS` makes use of the Mindswap OWL-S API, and some code from a software that was provided with the OWL-S API implementation to illustrate the conversion of WSDL to OWL-S. However, support for WSDL manipulation is very limited in the OWL-S API, so it is necessary to have the WSDL document saved on disk in order to read it (its contents cannot be passed as a string value, which is how they are submitted by clients to the SIS). For this reason, WSDL descriptions and annotation documents are saved temporarily in a location defined in the SIS initialization files. The SIS is then responsible to remove these temporary files from the file-system.

Method `loadWSDL()` combines the WSDL description and the annotations to form an OWL-S document for each of the operations of the service. The annotations are used to add the correct I/O types to the OWL-S profiles. After the OWL-S documents are created, they are stored in a directory in the file system, which is also defined in the SIS configuration files. Figure 4 presents a view of an OWL-S document which is produced by processing the WSDL and annotation documents illustrated in the previous figure.

OWL-S doc excerpt:

```
...
<service:Service rdf:ID="SERVICE_1">
   <service:presents rdf:resource="#SERVICE_1_PROFILE"/>
</service:Service>

<profile:Profile rdf:ID="SERVICE_1_PROFILE">
   <service:isPresentedBy rdf:resource="#SERVICE_1"/>
   <profile:serviceName xml:lang="en">CN service</profile:serviceName>
   <profile:textDescription xml:lang="en">Takes Hard_Disk, returns Compute_Node
   </profile:textDescription>
   <profile:hasInput  rdf:resource="#_Tradeable_Resource"/>
   <profile:hasOutput rdf:resource="#_CPU"/>

<process:Input rdf:ID="_Hard_Disk">
   <process:parameterType rdf:resource="[Grid4All onto NS]#Hard_Disk" />
</process:Input>

<process:Output rdf:ID="_Compute_Node">
   <process:parameterType rdf:resource="[Grid4All onto NS]#CPU" />
</process:Output>
...
```

Figure 4. Generated OWL-S document sample

An example implementation of registering a service using the WSDL and the EAF files for NsSimulation service is given below:

```
import gr.aegean.icsd.ailab.sis.SISImpl;
import gr.aegean.icsd.ailab.sis.SISImplService;
import gr.aegean.icsd.ailab.sis.SISImplServiceLocator;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.Constants;
import javax.xml.namespace.QName;
import javax.xml.rpc.ParameterMode;
import gr.aegean.icsd.ailab.sis.metadata.*;

/**
 * An example of a service advertisement. A WSDL
document and an External Annotations File
 * are read and their contents are submitted to the SIS
so that a semantic description of
 * the service is stored.
```

```
 */
public class ServiceAdvertisement1 {
	// Utility method for the parsing of text files
	public static String parseTextFile(String fileName) {
		String str = new String();
		try {
			java.io.BufferedReader in = new java.io.BufferedReader(new
java.io.FileReader(fileName));
			String line = new String();
			while(true) {
				line = in.readLine();
				if(line == null)
					break;
				str += line + "\n";
			}
		}
		catch(Exception e) {
			e.printStackTrace();
		}
		return str;
	}
	public static void main(String[] args) {
		try {
			SISImplService sisImplService = new
SISImplServiceLocator();
			SISImpl sis = sisImplService.getSIS();
//
// PROVIDER REGISTRATION
//
			// Invoke 'registerProvider'
			System.out.println("Invoking 'registerProvider'");
			String agentId = sis.registerProvider("Agent" +
System.currentTimeMillis(), "", "Athens");
			System.out.println("Agent is registered as provider, ID: " +
agentId);

//
// SERVICE REGISTRATION
//
			// Specify the local path of WSDL and EAF
			String wsdlLocalURI = "d:\\tmp\\NsSimulation.wsdl";
			String annotLocalURI ="d:\\tmp\\NsSimulation.xml";

			// Get content of WSDL and EAF
			String wsdlContent = parseTextFile(wsdlLocalURI);
			String eafContent = parseTextFile(annotLocalURI);
			System.out.println("Invoking 'advertiseService'...");
			String serviceId = sis.advertiseService(agentId, wsdlContent,
eafContent);
```

```
                System.out.println("Advertised service, got service ID: " +
            serviceId);
            } catch (RemoteException ex) {
                log.warn(ex.getMessage());
            } catch (ServiceException ex) {
                log.warn(ex.getMessage());
            }
        }

}
```

3.1.5. Querying the Service Registry

Querying for services involves specifying requirements concerning the service inputs and outputs, by providing lists of domain ontology concepts. Matchmaking is then performed to find advertised services with I/O types that either match the given I/O types, or subsume them.

Three basic types of matching are defined in the context of Grid4All services: Exact match, "Subsumes" match, and Fail. Let T be the terminology (T-Box) of the domain ontology where the service I/O types are specified; CT_T the concept subsumption hierarchy of T. The types of service matching in the context of Grid4All are the following:

- Exact match. Service S exactly matches service query R $\Leftrightarrow \forall\ IN_S \exists\ IN_R$: INS $\doteq$ $IN_R \wedge \forall\ OUT_R \exists\ OUT_S$: $OUT_R \doteq OUT_S$. For every input type of the advertised service one equivalent input type of the required service description is found. Also, for each output type of the required service description one equivalent output type of the advertised service is found. The service I/O signature perfectly matches with the request with respect to logic-based equivalence of their formal semantics.
- "Subsumes" match. Service query R subsumes service S $\Leftrightarrow \forall\ IN_S \exists\ IN_R$: $IN_R \sqsubseteq$ $IN_S \wedge \forall\ OUT_R \exists\ OUT_S$: $OUT_S \sqsubseteq OUT_R$. For each input type of the advertised service exactly one input type of the required service has been found, which is at least subsumed by the input type of the advertised service. This means that the advertised service might be invoked with a more specific input than expected. The output types of the required service subsume the output types of the advertised service or are equivalent to them. This means that the required service might receive a more specific output type than expected.
- Fail. Service S fails to match with service query R in any of the ways described above. This means that one of the following holds: a) at least one input type of the advertised service has not been successfully matched with one input type of the advertised service, and so the service cannot be executed properly, or b) at least one output of the required service has not successfully been matched with an input of the advertised service.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX profile: <http://www.daml.org/services/owl-s/1.0/Profile.owl#>
PREFIX ...

SELECT ?profile WHERE {
            ?profile rdf:type profile:Profile.
            ?profile profile:hasInput ?input1.
            ?input1 process:parameterType I1.
            ?profile profile:hasInput ?input2.
            ?input2 process:parameterType I2.
            ...
            ?profile profile:hasInput ?inputn.
            ?inputn process:parameterType In.

            ?profile profile:hasOutput ?output1.
            ?output1 process:parameterType O1.
            ?profile profile:hasOutput ?output2.
            ?output2 process:parameterType O2.
            ...
            ?profile profile:hasOutput ?outputn.
            ?outputn process:parameterType Om.
}
```

Figure 5. SPARQL query for service matching (Exact match)

Figures 5 and 6 present the SPARQL queries which are created, according to a given list of required I/O types, in order to perform the matchmaking at the service profile level. Granted that the agent making the query has given n input parameters of types I_1, I_2, …, I_n, and m output parameters of types, O_1, O_2, …, O_m, exact matching is performed by finding all services which have input (output) parameters whose types match exactly with a parameter type defined in the service query.

For the second type of service matching, "subsumes" match, the corresponding SPARQL query (figure 6) is created so that a matching service will be recognized if a) each one of its input parameters subsumes an input type which is defined in the service query b) each one of its output parameters is subsumed by an output type which is defined in the service query. To process such a query, which uses the `subclassOf` property of the RDF Schema, it is necessary to use an inference engine, so that inferred subclass/superclass relations may be detected among the parameter types specified in service queries or advertisements.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX profile: <http://www.daml.org/services/owl-s/1.0/Profile.owl#>
PREFIX ...

SELECT ?profile WHERE {
            ?profile rdf:type profile:Profile.

            ?profile profile:hasInput ?input1.
            ?input1 process:parameterType ?inputType1.
            I1 rdfs:subClassOf ?inputType1.
            ?profile profile:hasInput ?input2.
            ?input2 process:parameterType ?inputType2.
            I2 rdfs:subClassOf ?inputType2.
            ...
            ?profile profile:hasInput ?inputn.
            ?inputn process:parameterType ?inputTypen.
            In rdfs:subClassOf ?inputTypen.

            ?profile profile:hasOutput ?output1.
            ?output1 process:parameterType ?outputType1.
            ?outputType1 rdfs:subClassOf O1.
            ?profile profile:hasOutput ?output1.
            ?output2 process:parameterType ?outputType2.
            ?outputType2 rdfs:subClassOf O2.
            ...
            ?profile profile:hasOutput ?outputm.
            ?outputm process:parameterType ?outputTypem.
            ?outputTypem rdfs:subClassOf Om.
}
```

Figure 6. SPARQL query for service matching ("Subsumes" match)

The SPARQL queries described above are executed against a model which is composed of:

- The domain ontology from which the query I/O types are drawn
- The OWL-S descriptions maintained by SIS

The domain ontologies used in the SIS are managed by class `DomainHandle`, found in package `gr.aegean.icsd.ailab.sis.metadata.ontology`. Objects of this class maintain references to multiple RDF models, each of which is uniquely associated with a domain. Also, for each supported domain there is a unique human readable name. Agents that issue queries have to specify the domain to which their query I/O types belong by providing the corresponding domain name. The contents of `DomainHandle` are populated during the SIS service startup, with the help of a configuration file.

Once the two SPARQL queries are executed, and given that results have been retrieved, these results are ranked before being returned to the querying agent. Each query result is given a rank, i.e. a real value in the [0..1] space. This functionality is provided by class `ServiceRanker`. The rank of an advertised service, S, with respect to a service query, R, is equal to $aI_m + bO_m$, where a + b = 1. I_m is the semantic similarity between the set of inputs for S ($I_S = \{I_{S1}, I_{S2} \ldots I_{Sn}\}$), and the set of inputs for R ($I_R = \{I_{R1}, I_{R2} \ldots I_{Rn}\}$). I_m is equal to $\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} nosm(I_{Si}, I_{Rj})$, where nosm() is a function computing the semantic similarity between atomic concepts. O_m is computed in the same way using the output types of the advertised service and the service query. The semantic similarity of atomic concepts is computed by measuring the distance between them in the domain ontology graph, and is then adjusted according to the type of matching (Exact, "subsumes"), so that services that exactly

match a query always rank higher than the others. To compute the semantic similarity of atomic concepts, the Semantic Matching Framework (SemMF) is used.

Figure 7 shows the classes involved in service query processing.

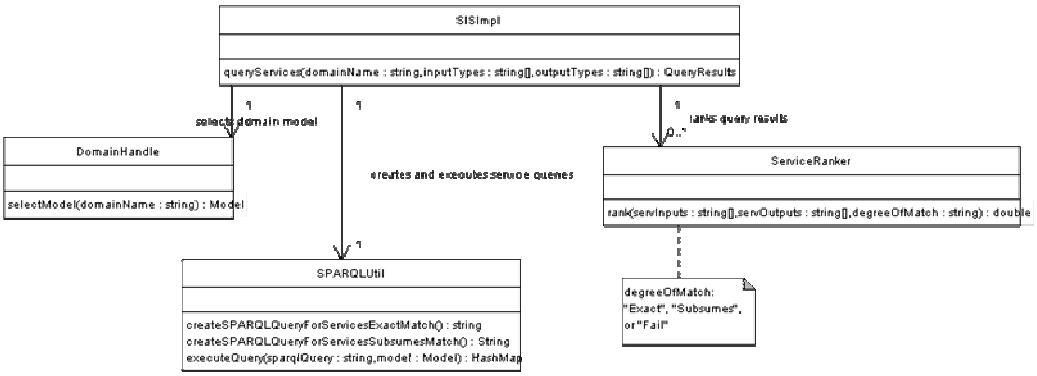


Figure 7. Class diagram of Service Registry query classes

3.2. Utility classes of the Core API

The package `gr.aegean.icsd.ailab.sis.util` contains several classes, all of which provide static methods that are used to support the SIS Core API implementation, as well as the implementation of the external API (Web service). This section provides a brief description of these utility classes, with the exception of `SPARQLUtil`, which has been described earlier.

CollectionUtil

`CollectionUtil` provides methods for the conversion of arrays to `ArrayList` objects. `ArrayList` is a type of object collection which is widely used within the implementation of the SIS. The conversion of arrays to `ArrayList` object is performed in the `arrayToArrayList()` method, which has been overloaded to support arrays of Java objects, as well as arrays of primitive values (arrays of integers, booleans, etc). `CollectionUtil` also provides a method to convert `ArrayLists` to arrays of Java objects (`arrayListToObjectArray()`), a method that performs a search for a specified object inside an array (`arrayContains()`), and methods to convert arrays of String objects to `ArrayLists` and vice versa (`stringArrayToArrayList()`, `arrayListToStringArray()`).

DateUtil

This class provides methods that are related to the representation of dates inside RDF descriptions. Method `createRDFDateString()`, when called with no arguments, returns the current time, encoded as an RDF date/time string. When `createRDFDateString()` is called with a `Calendar` argument (`Calendar` is a well known Java class used for date representation), it returns the given date/time as an RDF date/time string. Finally, `DateUtil` provides a null-safe method that tests equality between two given Calendar objects (`dateEquals()`).

ObjectUtil, StringUtil

`ObjectUtil` provides two methods that are related to the equality of Java objects. The method `objectEquals()` tests two Java Objects for equality, while `allEqual()` tests an entire array of objects. Similarly, `StringUtil` provides the method `stringEquals()`, which tests two String objects for equality.

QNameUtil

Several methods across the SIS implementation make use of objects holding mappings from URIs to short prefixes. These objects are instances of the class `QNameProvider`, provided by the Pellet API. `QNameUtil` provides the method `createPrefix()` that creates a unique prefix for a given URI and inserts it in a passed `QNameProvider` object. The method ensures that the prefix which is created is unique for the specified `QNameProvider`.

UniqueIDGen

`UniqueIDGen` provides a method (`createUniqueID()`) for the creation of random strings that can be used to create unique ontology resource URIs and query variable names.

Randomizer

This class contains a `java.util.Random` object, `rnd`, which can be accessed from everywhere in the SIS implementation and can be used to generate random numbers.

4. Notes on the SIS External API implementation

The implementation of the SIS Web Service and all of the exposed operations is located in the `SISImpl` class of Java package `gr.aegean.icsd.ailab.sis.impl`. `SISImpl` makes use of the facilities offered in the SIS Core API in order to interact with the Resource Registry and the Service Registry. Detailed information about the exposed methods and the specifics of their implementation is provided in the related Javadocs and the code comments. Apart from the objects that interact with the SIS registry, the SIS Web Service has several Java objects the purpose of which is to maintain the state information of the service. The state information is periodically saved in files located in the file-system using Java object serialization, so that it can be retrieved after an unexpected termination of the Web Service. The following section describes the objects that are used to maintain service state information.

4.1. SIS state information

4.1.1. SISSnapshot

The SISSnapshot class contains information organized in hashtables. This information includes:

- Mappings of agent usernames to agent IDs. Each agent is identified by a unique string which is computed during registration and returned to the agent. This ID is required by the operations that are related to advertising, querying and selection.
- Mappings of agent IDs to trader (provider/consumer) IDs. Every agent can act both as a consumer and as a provider using the unique ID return by the operations `registerProvider` and `registerConsumer`. If the agent calls both operations, the same ID will be returned. However, in this case the SIS creates two records in the Grid4All ontology, one representing the Provider role of the agent, and one representing its Consumer role. These records must be discrete, therefore an additional internal identification scheme is required in order to handle the multiple roles of an agent.
- Mappings of trader IDs to order IDs. These mappings are used to hold references to the offers, requests, or application service endpoints that are issued by the agents, thereby eliminating the overhead that would be produced by constructing and executing very short SPARQL queries. Mappings of market order IDs to market service endpoints are created for the same purpose.
- Mappings of query IDs to SPARQL query strings. Due to the existence of the `repeatQuery` operation, the SPARQL query strings that are created during the initial processing of a query are maintained for future use. When a query is removed from the SIS, these mappings are also removed.
- Mappings of service namespace URIs to service profiles. An advertised service may include multiple operations, each of which is translated to a separate OWL-S profile document. SISSnapshot maintains associations of services to their operations and the respective file locations.

The mappings listed above are the primary information pieces stored in `SISSnapshot`. The SIS maintains an object of this class, and serializes it frequently. To perform this serialization, a daemon thread is created during service initialization

that takes on the specific task. This thread is an instance of the class `SISSnapshotUpdateThread` in package `gr.aegean.icsd.ailab.sis.impl`. This thread also serializes all the other objects that constitute the SIS service state.

4.1.2. Selection Service module

The selection module maintains agent related information. Most importantly, this is the object that contains agent preferences to other agents or to certain query types and is used in ranking of query results. Since this information is not encoded in some way in the SIS registry, it is serialized along with the `SISSnapshot` object and the deletion queue.

4.1.3. Deletion Queue

The deletion queue is used by the SIS for cleanup purposes. It is mentioned earlier in this document that queries are automatically removed by the SIS after a specific time period. In order to perform this cleanup, the SIS maintains a list of `DeletionQueueElement` instances. An instance of `DeletionQueueElement` holds a reference to an advertisement or query ID, a string specifying whether the element to be removed is a query or an advertisement, a timestamp showing the time at which (or, after which) this element should be removed, and the ID of the agent that issued the advertisement or query. Whenever an advertisement or a query is submitted to the SIS, a `DeletionQueueElement` object is added to the deletion queue. In addition, upon startup of the SIS service a thread is created that frequently checks this queue for elements that need to be removed, and when it finds such elements, it calls the appropriate methods for advertisement or query removal. This thread is implemented in `DeletionQueueMonitoringThread` class. The deletion queue is also a part of the state of the SIS, therefore its contents are frequently serialized, as well as the contents of `SISSnapshot` and the selection module.

4.2. SIS implementation class (SISImpl)

This section is a brief guide to the implementation of the exposed SIS Web Service operations, which is contained in class `SISImpl`. What is presented here is the logic underlying the operation implementations; future developers should consult the Javadocs and the code comments for a more detailed description of the implementation.

registerConsumer

The method starts by checking the arguments. If any of the arguments is null, an `InvalidAgentDescriptionException` is thrown. Next, it checks whether an agent with the specified username exists, in which case an `AgentAlreadyExistsException` is thrown. If the agent has not also registered as a provider, a unique agent ID is generated. A `Consumer` object is created, populated with the provided information, and serialized in the Grid4All ontology, and the agent is also subscribed in the Selection Service. Finally, the agent ID is returned.

registerProvider

If any of the arguments is null, an `InvalidAgentDescriptionException` is thrown. Next, the method checks whether an agent with the specified username exists, in which case an `AgentAlreadyExistsException` is thrown. If the agent has not also

registered as a consumer, a unique agent ID is generated. A `Provider` object is created, populated with the provided information, and serialized in the Grid4All ontology, and the agent is also subscribed in the Selection Service. Finally, the agent ID is returned.

updateAgent

After checking for null arguments, the method checks the `SISSnapshot` object to determine whether the specified agent ID exists. Then, new RDF triples are created containing the new information and replace the old ones in the Grid4All ontology.

deleteAgent

If the specified agent ID does not exist, a `NoSuchAgentException` is thrown. The agent is unsubscribed from the Selection Service, and the `SISSnapshot` object is used to obtain references to all the market orders that have been advertised by the specific agent. The `deleteAdvertisement` method is used to remove these advertisements.

advertiseOffer, advertiseRequest, advertiseAbstractOrder

The steps that are followed in the implementation of these three methods are similar. First, the provided arguments are validated. The method throws an `InvalidDescriptionException` if:

- The object holding the market order description (`Offer`, `Request`, `AbstractMarketOrder`) is null
- No market related information is provided (the `Market` object inside the `Offer`/`Request`/`AbstractMarketOrder` object is null)
- No information about the traded or requested resource is provided (The `TradeableResource` object inside the `Offer`/`Request`/`AbstractMarketOrder` object is null)
- (Only in `advertiseOffer`) A complex offer is provided, but the operator (or "connective") is not one of "AND", "OR", "XOR", or it is null.

Also, if the market service endpoint is null, an `InvalidURLException` is thrown. Next, the agent ID is validated, the object holding the market order information is serialized in the Grid4All ontology, and the market order is recorded in the `SISSnapshot` object. Finally, the unique ID which is extracted from the `Offer`/`Request`/`AbstractMarketOrder` object is then returned.

advertiseService

The method validates the arguments and checks whether the provided agent ID exists. Two text files are temporarily created and the contents of the provided WSDL description and the annotations are written in these files. A `WSDLToOWLS` object is created and its `loadWSDL()` method is called to produce the semantic service descriptions. For each service operation, an OWL-S document is produced. The new advertisement is recorded in the `SISSnapshot` object, the temporary text files are deleted, and the namespace URI of the service is returned.

deleteAdvertisement

The method checks whether the provided market order ID exists by looking it up in the `SISSnapshot` object. If it does not exist, an `InvalidDescriptionException` is thrown. The method removes all the individuals and classes that are related to this particular advertisement: Classes and instances describing the market order, the traded/requested resource description, the market description etc. The `ModelHandle` class provides two (recursive) methods that are used for mass deletion of ontology elements, `removeClassChain()` and `removeIndividualChain()`. Both of these methods begin with a given ontology element, and then they find and remove related elements using values (for individuals) or restrictions (for classes) of their object properties.

updateAdvertisement

After validating the arguments, this method removes the old advertisement from the SIS registry and creates a new advertisement from the `MarketOrder` object that is provided. However, it maintains the ID of the old advertisement.

queryProviderInitiatedMarkets

After the validation of the arguments, the given `Request` object is serialized in the Grid4All ontology and classification is performed to obtain new inferred statements. Two SPARQL queries are created and executed, one for Atomic Offers matching the request and one for Complex Offers. The `SPARQLUtil` class provides methods which create the appropriate SPARQL strings based on the request description and the given market constraints. These strings are saved in the `SISSnapshot` object for possible repetitions of the query. The query for matching complex offers returns any complex offer that has at least one atomic offer matching the request. These results are then filtered according to the complex offer operator:

- An "XOR" offer is accepted if only one of its atomic offers matches the request.
- An "AND" offer is accepted if all of its atomic offers match the request.
- In cases of "OR" offers there is no additional filtering.

When the list of matching offers is created, the corresponding providers are ranked in the Selection Service Module. The list of offers is reordered according to this ranking. Then, a search for matching abstract market orders is performed. If any matching abstract orders are found, they are appended to the end of the results list. The list is added in a `QueryResults` object. Apart from the list of results, this object contains a unique query ID that can be used to repeat the query. Finally, the query is added to the deletion queue and the `QueryResults` object is returned.

Note: To support repeated execution of queries for provider initiated markets through `repeatQuery()`*, without duplicating code, a large portion of the initial method body (everything after the serialization of ontology elements and classification) is located to method* `prQueryProviderInitiatedMarkets()`*. This private method is called by both* `queryProviderInitiatedMarkets()` *and* `repeatQuery()`.

queryProviders

The method validates the arguments, i.e. the ID of the agent making the query and the ID of a market that this agent has advertised. Classification is performed, and then two SPARQL queries are created and executed: One for atomic offers and one for complex offers. The offers that are returned are not specified in consumer initiated markets. Next, the providers that have issued these offers are retrieved using the `SISSnapshot` object and they are ranked using the Selection Service module. The query is then added to the deletion queue, and the reordered list of agent usernames is added to a `QueryResult` object and returned.

queryConsumerInitiatedMarkets

After the arguments (agent ID, offer description, market related constraints) are validated, the given `Offer` object is serialized in the Grid4All ontology, classification is performed, and the new query is recorded in the `SISSnapshot` object. If the offer is atomic, then a SPARQL query for matching requests is created, otherwise the number of SPARQL queries that are created is equal to the number of atomic offers that are contained in the complex offer description. In the case of a "XOR" complex offer containing N atomic offers, a request is accepted if it matches with only one of the atomic offers. Conversely, for "AND" complex offers, a request is accepted if it matches with all of the N atomic offers. "OR" complex offers and atomic offers are immediately accepted and no further filtering is performed. The Selection Service module ranks the consumers that have advertised the matching requests and reorders the list of requests accordingly. Next, the matching abstract market orders are found by creating and executing an appropriate SPARQL query, and they are appended to the list of results. Finally, the query is added to the deletion queue, and the results are added to a `QueryResults` object and returned.

Note: To support repeated execution of queries for consumer initiated markets through `repeatQuery()`*, without duplicating code, a large portion of the initial method body (everything after the serialization of ontology elements and classification) is located to method* `prQueryConsumerInitiatedMarkets()`*. This private method is called by both* `queryConsumerInitiatedMarkets()` *and* `repeatQuery()`*.*

queryConsumers

The method validates the arguments, i.e. the ID of the agent making the query and the ID of a market that this agent has advertised. Classification of the Grid4All ontology is performed, and a SPARQL query for requests (not specified in provider initiated markets) matching the offer which is specified in the given market is created and executed. Next, the consumers that have issued these request are retrieved using the `SISSnapshot` object and they are ranked using the Selection Service module. The query is then added to the deletion queue, and the reordered list of agent usernames is added to a `QueryResult` object and returned.

queryServices

The arguments of the method, i.e. the agent ID, the domain name, and the lists of input and output types are validated. After the validation, two SPARQL queries are created, one for the exact matches and one for the partial ("subsumes") matches. These queries are executed on an RDF model containing the related domain ontology,

the OWL-S Service Profile ontology, and the OWL-S descriptions of the advertised services. For each matching service, its rank is computed by using a `ServiceRanker` object that is initialized with the I/O types of the query and the related domain ontology. The matching services are then sorted according to their ranks, and they are returned as part of a `QueryResults` object.

repeatQuery

After the validation of the arguments, the `SISSnapshot` object is used to determine the type of the query (Query for Provider Initiated Markets, Consumer Initiated Markets, Consumers, Providers, or Services). According to the type of the query, the appropriate method is called, with the exception of service queries. (`prQueryProviderInitiatedMarkets()`, `prQueryConsumerInitiatedMarkets()`, `queryConsumers()`, or `queryProviders()`). In case that a query for services is repeated, no other method is called. Instead, the SPARQL queries (for exact and "subsumes" matches) that were created when the query was initially submitted are retrieved from the `SISSnapshot` object and are immediately executed.

removeQuery

The method initially validates the agent ID and query ID, and checks whether the specific agent is the one that has issued the query. Then, the query type is determined using the `SISSnapshot` object. In the case of a query for consumer initiated markets or for consumers, the offer description that was serialized in the SIS registry during the initial query execution is removed. Similarly, in the case of a query for provider initiated markets or for providers, the related request is removed from the SIS registry. Finally, the method removes all information related to the query from the `SISSnapshot` object.

selectProviders, selectConsumers

After the methods validate the arguments, they update the Selection Service module with the selected agent usernames by calling the `informFinalSelection()` method of the module.

selectMarkets

This method works similarly with the methods `selectProviders()` and `selectConsumers()`. Since the Selection Service module works with ranking of agents, the method retrieves the usernames of the agents that have advertised the selected markets and calls `informFinalSelection()` using a list of these usernames.

setPreferencesForProviders, setPreferencesForConsumers

The methods check the validity of the arguments, i.e., the agent ID, the list of agent usernames and the list of preference values. The sizes of the two lists are also checked, and if they are not equal, an `InvalidDescriptionException` is thrown. The contents of the lists are then added to a hashtable, which is then passed to the Selection Service module to set the new preferences of the calling agent.

setPreferencesForQueryTypes

This method works in a similar way as do methods `setPreferencesForProviders()` and `setPreferencesForConsumers()`, but

instead of agent usernames, query types are passed. Valid query types are the ones returned by the `getQueryTypes()` operation.

getQueryTypes

This method is used to support `setPreferencesForQueryTypes()`. It returns an array containing the query types used in the Selection Service module: "Compute_Node", "Cluster", and "Service".

getDomainNames, getOntologyURLFor, getOntologyNamespaceURIFor

These methods return information that is stored in the `DomainHandle` object and is related to domain ontologies used for annotation of advertised servives, and for service queries. `getDomainNames()` returns a list containing the human-readable names of all the supported domains, which is retrieved by calling `listDomains()`. `getOntologyURLFor()` and `getOntologyNamespaceURIFor()` redirect to `DomainHandle` methods `getDomainOntologyURL()` and `getDomainOntologyNs()`, respectively.

5. Usage of the SIS Web Service

In order to access the SIS Web Service remotely and use the provided operations, a software client has to be created. The WSDL description of the service, can be used to construct a fully functional SIS client, as it provides all the necessary operation and complex type descriptions in order to successfully call every operation. The following section provides a description of how a client can be generated using the Apache Axis SOAP engine.

5.1.1. Client generation using Apache Axis

The Axis SOAP engine provides the WSDL2Java tool, which can create a set of Java classes by inspecting a given WSDL document. In order to execute the WSDL2Java tool, the Axis libraries must be accessible:

```
java -cp .;%AXIS_CLASSPATH% org.apache.axis.wsdl.WSDL2Java sis.wsdl
```

After the WSDL2Java tool has completed its execution, a source tree will be created in the local file-system, in the path from where WSDL2Java is called. The following packages will be available in the source tree:

```
gr.aegean.icsd.ailab.sis
gr.aegean.icsd.ailab.sis.metadata
gr.aegean.icsd.ailab.sis.metadata.query
```

These packages contain Java representations of all the schema types described in the WSDL description of the SIS service. Also, in the `gr.aegean.icsd.ailab.sis` package, there are several classes which can be used to access the SIS service as a remote object, and use the exposed operations by invoking the methods provided by the remote object. An example of such a Remote Procedure Call is presented below:

```
import gr.aegean.icsd.ailab.sis.SISImpl;
import gr.aegean.icsd.ailab.sis.SISImplService;
import gr.aegean.icsd.ailab.sis.SISImplServiceLocator;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

public class SISClientExample1 {
  public static void main(String[] args) {
    SISImplService sisImplService = new SISImplServiceLocator();

    try {
      // Get reference to SIS service
      SISImpl sis = sisImplService.getSIS();
      // Invoke 'registerProvider' (agent username = ”Agent1”,
      // password = ”pass”, location = ”Location”)
      String id = sis.registerProvider("Agent1", "pass", "Location");
    }
    catch (RemoteException ex) {
      ex.printStackTrace();
    }
    catch (ServiceException ex) {
      ex.printStackTrace();
    }
  }
}
```

To make use of the generated code and create SIS service client applications, the Axis libraries must be included in these applications.

6. Installation of the SIS Web Service

This section describes how to install the Grid4All SIS web Service in your own server. A J2EE application server is required, such as Apache Tomcat, Glassfish, or any other that implements version 2.2 or greater of the Servlet API. Also, a JDK of at least version 1.5 is required.

The Web Archive (.war) file where the Web Service implementation is contained also includes all the necessary libraries required in order for it to function. Among those libraries are the Jena framework (version 2.5.7), the Pellet reasoner (version 2.0.0 RC4), the Axis SOAP engine (version 1.4), the OWL-S API implementation of Mindswap, and others.

To install the service on Tomcat server, place the Web Archive file to the `webapps` folder and start the server. The contents of the .war file will be extracted and a new directory, `grid4all_sis`, will appear under the `webapps` folder. The `grid4all_sis` contains the following folders:

- `META-INF`, which contains Java package related information
- `resources`, where various ontologies are maintained and can be publicly accessed
- `WEB-INF`, in which the actual implementation of the SIS Web Service resides (under `WEB-INF\classes`). Also, in this folder are the libraries used by the service (`WEB-INF\lib`), as well as some files which can be used to either change some parameters concerning the information stored during the lifetime of the service (`WEB-INF\conf`), or undeploy/redeploy the service (`WEB-INF\ws`).

In order for the service to be successfully executed, it has to be able to store information persistently in some hard disk files. To accomplish this, several configuration parameters related to the paths of the required files need to be defined. These parameters are described in the following paragraphs. The requirements with respect to the persistent storage are the following:

The configuration parameters reside in the WEB-INF/conf/sis_conf.properties file. Concerning advertised application services, their OWL-S profiles must be stored in a path, which can be defined in this `.properties` file. Also, during advertisements, information concerning the submitted WSDL documents and annotation information must be temporarily stored to disk, so temporary storage paths must also be defined. In addition to service-related information, the SIS must be able to persistently store information relevant to the Selection Service, as well as other information obtained during SIS service execution, that constitute the *state* of the SIS service. This information is maintained in Java objects, and it is persistently stored using object serialization. Finally, a copy of the Grid4All ontology is periodically serialized, along with state information, as an OWL/XML file. The paths where these Java objects and the Grid4All ontology will be stored (and then retrieved, in case the SIS service is stopped and restarted) must also be defined in the `sis_conf.properties` file. The required paths will be automatically created during service initialization, unless there are not sufficient write permissions in the specified paths. An example of the required path definitions is presented below:

```
sis.repository_path=C:\\apps\\Grid4All\\upload
# Path to the service repository (relative to sis.repository_path)
```

```
sis.owl_s_repository_path=\\services
# WSDL repository path (relative to sis.repository_path)
sis.wsdl_temp_repository_path=\\wsdl
# Annotation repository path (relative to sis.repository_path)
sis.eaf_temp_repository_path=\\annot
# Path to the SIS Snapshot serialization target
sis.snapshot_serialization_file=C:\\apps\\Grid4All\\sis_snapshot\\SIS
Snapshot
# Path to the Selection Service Module serialization target
sis.ss_serialization_file=C:\\apps\\Grid4All\\sis_snapshot\\Selection
Service
# Path to the Deletion Queue serialization target
sis.deletion_queue_serialization_file=C:\\apps\\Grid4All\\sis_snapsho
t\\DeletionQueue
#
sis.db.ontology_cache_file=C:\\apps\\Grid4All\\ontology_cache\\G4AOnt
o_cache.owl
```

After the database connection parameters and the local paths have been defined, the SIS Web Service is ready to be deployed. The `WEB-INF\ws` folder contains a Web Service Deployment Descriptor document, `sis_deploy.wsdd`, which can be used to inform the Axis Engine about the service and its characteristics (e.g. name, exposed operations, etc). The service is deployed with the following command:

```
java -cp .;%AXIS_CLASSPATH% org.apache.axis.client.AdminClient
-lhttp://localhost:8080/grid4all_sis/AxisServlet sis_deploy.wsdd
```

If the Axis libraries are not available in the local file-system, the `AXIS_CLASSPATH` variable may also contain the Axis .jar files maintained in WEB-INF\lib. These files are the following: axis.jar, axis-ant.jar, saaj.jar, wsdl4j.jar, jaxrpc.jar, xml-apis.jar, commons-logging.jar, commons-discovery.jar, log4j.jar. Similarly, for the undeployment of the service, the `sis_undeploy.wsdd` file is used in the same way as above:

```
java -cp .;%AXIS_CLASSPATH% org.apache.axis.client.AdminClient
-lhttp://localhost:8080/grid4all_sis/AxisServlet sis_undeploy.wsdd
```

After the deployment process takes place, the SIS Web Service installation is complete and the service is available to clients in the following address (Service endpoint): `http://<hostname>:8080/grid4all_sis/ws/SIS`, where <hostname> is the domain name of the machine where the service is hosted. A description of the SIS service as a WSDL document can be obtained by requesting the following address: `http://<hostname>:8080/grid4all_sis/ws/SIS?wsdl`. By using this description, SIS clients can be constructed, as described in section 5.1.1.

7. Local usage of the SIS Web Service functionality

The SIS Web Service has been implemented as a set of Java classes not dependent on (i.e., not making use or extending classes of) a specific web service framework (these are usually referred to as POJOs, Plain Old Java Objects), on top of which the Apache Axis SOAP engine is used to create, send and receive SOAP messages. It is possible, however, to also use this functionality locally, as part of another Java project, by creating a local instance of the `SISImpl` class. `SISImpl` offers two constructor methods: one with no arguments and one that accepts two arguments, both of which are objects of the `Properties` class, provided by the Java Development Kit (package `java.util`). The `Properties` class represents a persistent set of properties which can be saved to a stream or loaded from a stream. Typically, such streams are

linked to simple text files, which by convention have the extension ".properties". The SIS needs two sets of properties in order to be initialized and function properly:

- Configuration information: Such information includes database connection parameters, the location of the advertised service descriptions and the temporary files that are created during service advertisement, how frequently the deletion queue should be checked, etc.
- Information about the supported domain ontologies: All the supported domain ontologies must be recorded in this set of properties. Information about the domain ontologies includes their human-readable domain names, their namespace URIs, and the URL from where they can be accessed. This information is used to initialize the `DomainHandle` object used in `SISImpl`.

The SIS Web Service uses such properties sets, and the respective `.properties` files reside in directory `WEB-INF\conf`. The no-argument constructor, which is used by the Axis engine when the Web Service is initialized, actually redirects to the two-argument constructor, using the Properties objects obtained from the `conf` directory. Any other usage of the no-argument constructor is discouraged. The following Java program shows how a `SISImpl` object can be created and used locally:

```
import gr.aegean.icsd.ailab.sis.impl.SISImpl;
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;

public class LocalSISImplObjectExample {
  public static void main(String[] args) {
    Properties sisConf = new Properties();
    Properties domains = new Properties();

    try {
      sisConf.load(new FileInputStream( new File(
        "c:\\tmp\\sis_conf.properties")));
      domains.load(new FileInputStream( new File(
        "c:\\tmp\\domains.properties")));

      SISImpl sis = new SISImpl(sisConf, domains);
      sis.registerProvider("AgentXYZ", "password", "location");
      String[] queryTypes = sis.getQueryTypes();
      ...
    } catch(Exception e) {
      System.exit(1);
    }
  }
}
```

Note that, in order to make use of the SIS functionality locally, virtually all of the Java libraries that are used by the service (in the WEB-INF/lib folder) must be included. The only library that is not needed for local use of the SIS is the Axis library.

The configuration properties, as well as the way they are encoded in the respective properties file, are shown in chapter 6. To form a valid .properties files for the `DomainHandle` initialization, several information pieces need to be provided for every supported domain ontology. These information pieces are:

- A valid QName prefix (used mainly during queries); it must not contain spaces

- A human-readable name for the domain (it may contain spaces)
- The ontology namespace
- A resolvable URL of the ontology

Supposing that there have been 14 entries already in the `.properties` file, the following information is provided for the 15th ontology:

- QName prefix: “ex1”
- Namespace: http://www.example.org/example
- Resolvable URL: http://www.somewhere.com/here
- Human readable name: “Example”

In this case, the following lines must be added in the `.properties` file:

```
domain15=ex1
ex1.name=Example
ex1.ns=http://www.example.org/example
ex1.url=http://www.somewhere.com/here
```

8. Useful URLs

Name	URL	Description	Comments
Apache Tomcat	http://tomcat.apache.org	A Java Servlet container that also acts as a web server.	Version 6 is recommended
MySQL	http://www.mysql.com	Open source RDBMS	
JDK	http://www.java.sun.com		JDK Version 6 is recommended
Grid4All	http://www.grid4all.eu	The Grid4All project home page	
JENA	http://jena.sourceforge.net	An RDF framework for Java	Version 2.5.7 is used by SIS
Pellet	http://www.clarkparsia.com/pellet	Reasoning engine for Java	Version 2.0.0RC4 is used by SIS
OWL-S API	http://www.mindswap.org/2004/owl-s/api/	OWL-S document manipulation library	
SemMF	http://semmf.ag-nbi.de	Library for computing similarity of ontology concepts	
Apache Axis	http://ws.apache.org/axis	Used to expose user created code as Web Service operations and to create, parse and exchange SOAP messages from and to these operations	Version 1.4 is used by the SIS Web Service
SIS Web Service home page	http://icsd-ai-lab:8080/grid4all_sis/	Contains some general information about the Grid4All and SIS, and descriptions of the exposed Web Service operations	
SIS endpoint	http://icsd-ai-lab.aegean.gr:8080/grid4all_sis/ws/SIS	Endpoint address of the SIS Web Service	
SIS WSDL	http://icsd-ai-lab.aegean.gr:8080/grid4all_sis/ws/SIS?wsdl	The WSDL description of the exposed SIS operations	
SIS Web Service copy	http://ai-lab-server.aegean.gr/svn/ai-lab/sis/Service/dist/grid4all_sis.war	A copy of the .war file containing the SIS Web Service implementation, along with the required libraries	
SIS Web Service client copy	http://ai-lab-server.aegean.gr/svn/ai-lab/sis/Service/client	This file contains client code generated by the WSDL2Java tool of Apache Axis. The code can be used to create SIS clients.	JKD 1.4 is required to use this code
SIS Web Service libraries	http://ai-lab-server.aegean.gr/svn/ai-lab/sis/Service/lib	The folder contains all the required libraries for the SIS client and for local SIS usage (without calling a web service)	
SIS JavaDocs	http://ai-lab-server.aegean.gr/svn/ai-lab/sis/Service/Doc/javadoc		
Guide document (this document)	http://ai-lab-server.aegean.gr/svn/ai-lab/sis/Service/Doc/SIS_Web_Service_		

	Guide_final.doc		
WSDL-AT	http://ai-lab-server.aegean.gr/svn/ai-lab/sis/WSDL-AT	WSDL annotation tool	
Grid4All ontology	http://icsd-ai-lab.aegean.gr:8080/grid4all_sis/resources/ontologies/G4Aonto_v8_TBox.owl	The T-Box of the Grid4All resource ontology	

9. Further reading

This chapter provides some pointers for further reading on Semantic Web technologies in general, as well as some more specific aspects of the Semantic Information System, such as the Grid4All ontology, service matching and ranking, and the frameworks that were used in the implementation of the SIS.

Grid4All Ontology

- George A. Vouros, Andreas Papasalouros, Konstantinos Kotis, Alexandros Valarakos, Konstantinos Tzonas, Xavier Vilajosana, Ruby Krishnaswamy, Nejla Amara-Hachmi, "The Grid4All ontology for the retrieval of traded resources in a market-oriented Grid", IJWGS special issue on "Web/Grid Information and Services Discovery and Management", International Journal of Web and Grid Services 2008 - Vol. 4, No.4 pp. 418 – 439.
- G. A. Vouros, A. Papasalouros, K. Kotis, A. Valarakos, K. Tzonas, S. Retalis, and R. Krishnaswamy. Semantic discovery of resources and services in democratized grids. Handbook of Research on Social Dimensions of Semantic Technologies and Web Services, M. Manuela Cunha, Eva Oliveira, Antonio Tavares & Luis Ferreira (Eds), Information Science Reference, ISBN: 978-1-60566-650-1

Semantic Web technologies

- O. Lassila and R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification", W3C Recommendation. February 1999. http://www.w3.org/RDF/
- D. Brickley and R.V. Guha, "RDF Vocabulary Description Language 1.0: RDF Schema", W3C Recommendation, February 2004. http://www.w3.org/TR/rdf-schema/
- E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF", W3C Working Draft, October 2006. http://www.w3.org/TR/rdf-sparql-query/
- D. L. McGuinness and F. van Harmelen, "OWL Web Ontology Language Overview", W3C Recommendation, February 2004. www.w3.org/2004/OWL/
- D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan and K. Sycara, "OWL-S: Semantic Markup for Web Services", W3C Member Submission, Nov. 2004. http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/

Web Services

- Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, Web Services Description Language (WSDL) 1.1, W3C submission, March 2001. http://www.w3.org/TR/wsdl
- Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, Yves Lafon, SOAP Messaging Framework, W3C Recommendation, April 2007. http://www.w3.org/TR/soap12/

Service matching

- M. C. Jaeger, G. Rojec-Goldmann, G. Muhl, C. Liebetruth, and K. Geihs,"Ranked Matching for Service Descriptions using OWL-S", Proceedings of KiVS, p. 91-102 2005
- Alberto Fernandez, Axel Polleres, Sascha Ossowski, Towards Fine-grained Service Matchmaking by Using Concept Similarity, First International Joint Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web (SMR2) at ISWC 2007, p.31--45 (2007)
- M. Klusch, B. Fries, M. Khalid and K. Sycara, "OWLS-MX: Hybrid Semantic Web Service Retrieval", Proceedings of the 1st International AAAI Fall Symposium on Agents and the Semantic Web, Arlington VA, USA, AAAI Press, Technical Report FS-05-01.

Semantic Web libraries/frameworks

- B. McBride, "Jena: Implementing the RDF Model and Syntax Specification", Proceedings of the Second International Workshop on the Semantic Web - SemWeb'2001, pp. 74-83, 2001
- E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur and Y. Katz, "Pellet: A Practical OWL-DL Reasoner", Web Semantics: Science, Services and Agents on the World Wide Web archive, volume 5 , issue 2 (June 2007) pp. 51-53, 2007

This page is intentionally left blank

Annex 5. Telex, a Principled System Support for Write-Sharing in Collaborative Applications

-/-

Application Programming Guide

Table of contents

1- PREAMBLE

This guide describes how to use Telex's API. It complements the javadoc of the API[1].

Before using this guide, the reader should be familiar with Telex's functionality and the design of collaborative applications atop Telex. These topics are covered by the following papers[2]: “Telex: a Platform for Decentralized Sharing” and “Distributed Collaborative Applications Using the Telex Sharing and Reconciliation Platform”.

This guide uses code examples drawn from the `telex.application.sample` package[3]. This package contains a simple Telex application that sketches a collaborative text editor. It includes the application's main class, `SimpleApplication.java`, and two auxiliary classes, `SimpleAction.java` and `SimpleState.java`.

1 see [telex]/docs/javadoc/API/index.html
2 located in [telex]/docs/papers
3 located in [telex]/src/fr/inria/gforge

2- OVERVIEW

A collaborative application involves several *users* located at different *sites*. Through the application, users update a shared *document,* either at the same time or at different times. When connectivity permits, updates are propagated to other sites so that all users can see them. If updates conflict, the shared document have different *states*, each representing a solution to the conflict. A background *reconciliation* protocol ensures that all users eventually see the same document state.

2.1- Application design

The design of a Telex application follows the Model-View-Controller (MVC) pattern. At each site, the application waits for user updates and translates them into *actions* and *constraints* that it passes to Telex. In return, Telex up-calls the application to display the current state of the document whenever the document is updated, either locally or remotely.

These control flows are handled by separate threads. The main thread is created by the JVM when launching the application. It initializes the Telex middleware, listens to user input and calls Telex. Telex creates two additional threads upon initialization. The first thread listens for updates sent by remote sites whereas the second process reconciliation proposals. These threads up-call the application whenever the document is updated.

2.2- Application components

A typical Telex application includes three java files. In our example, `SimpleApplication.java` contains the main code of the application, which interfaces (calls and up-calls) with Telex; `SimpleAction.java` describes the application's actions; `SimpleState.java` represents the state of the application's shared documents.

`SimpleAction` and `SimpleState` classes are opaque to Telex. Telex does not know about the semantics of actions and the internal structure of shared documents: it cannot execute actions and update the state of the document. Rather, Telex computes sound *schedules*, i.e. sequences of actions that comply with constraints, and passes them to the application for execution. In our example, the actual execution of actions takes place in the `SimpleState` class.

2.3- `SimpleApplication` class

The `SimpleApplication` class is the main class of the application. It contains the `main` method of the application:

```
public static void main(String[] args) {

        new SimpleApplication(args).run();

}
```

The `main` method initializes the application by calling the constructor of the class. It then calls the `run()` method which implements the application's main processing loop. These topics are covered is sections 3 and 4 respectively.

The `SimpleApplication` class must also implements the methods that Telex up-calls to interact with the application. These methods are part of the `telex.application.TelexApplication` and `telex.application.ConstraintChecker` interfaces:

```
public class SimpleApplication implements TelexApplication, ConstraintChecker {

}
```

2.4- SimpleAction class

The telex.application.Action class defines attributes common to actions of all applications. These attributes include the action's unique id, the user who issued the action, the site where it was issued, etc. Each application must sub-class the Action class in order to define application-specific attributes such as the operation that the action represents, its parameters, etc. For the sake of simplicity, these attributes are represented as Strings in our example:

```
public class SimpleAction extends Action {

	//
	// CONSTANTS
	//
	private static final long serialVersionUID = 1L;

	//
	// INSTANCE FIELDS
	//
	private String operation;
	private String[] parameters;

	// …

}
```

Note that in the Telex execution model, an action is immutable. This the application's responsibility to ensure this. Currently, the final modifier cannot be used to protect the operation and parameters fields.

Actions are logged to persistent storage. For performance reasons, Telex uses an had-hoc mechanism rather than Java's serialization. This mechanism requires that SimpleAction class (i) provides a public nullary constructor, which Telex calls whenever it reads a new action from the log, (ii) implements ad-hoc methods to read() and write() application-specific fields to/from the log, as follows:

```
	// Mandatory public nullary constructor
	public SimpleAction() {

	}

	// Ad-hoc read and write methods for application's specific fields
	public void write(DataOutput out) throws IOException {

		// mandatory: write Action's fields first
		super.write(out);

		// then write this instance's fields
		FileUtils.writeUTF("operation", operation, out);
		FileUtils.writeObject("parameters", parameters, out);

	}

	public void read(DataInput in) throws IOException {

		// mandatory: read Action's fields first
		super.read(in);

		// then read this instance fields in same order as write
		operation = FileUtils.readUTF("operation", in);
		parameters = (String[]) FileUtils.readObject("parameters", in);

	}
```

Note that read() and write() must first call their counterpart in the telex.application.Action class. The telex.util.FileUtils class provides methods to read and write primitive types, arrays and Object instances.

Actions are also sent to remote sites as part of the reconciliation protocol. For the sake of simplicity, Telex uses Java's serialization in this case. Application-specific actions must therefore be serializable. The `telex.application.Action` class implements the `Serializable` interface. The application must make sure that all non-transient fields of the `SimpleAction` class are also serializable.

Telex will retain a single mechanism for both logging and reconciliation in a future release.

2.5- `SimpleState` class

The class representing the state of a document must implement the `telex.application.DocumentState` interface. It must be serializable so that Telex can store it on persistent storage.

In our example, a shared document is a text file. Its state consists of the contents of the file, represented as a list of lines of type `String`, as follows:

```
public class SimpleState implements DocumentState {

	//
	// CONSTANTS
	//
	private static final long serialVersionUID = 1L;

	//
	// INSTANCE FIELDS
	//
	private List<String> contents;

	//
	// CONSTRUCTORS AND ACCESSORS
	//
	/**
	 * Allocates a new document state.
	 */
	public SimpleState() {
		contents = new ArrayList<String>();
	}

	// …

}
```

The class also implements the static method `display(Schedule)`, which is called by the `SimpleApplication` class to display the schedules returned by Telex (see section 5). The code of the method is as follows:

```
	static void display(Schedule schedule) {

		// execute schedule on base state
		SimpleState state = (SimpleState) schedule.getState();
		for (Action action : schedule.getActions()) {
			state.execute((SimpleAction) action);
		}

		// display state
		state.display();

	}
```

This method computes the state represented by the specified schedule and displays it to the user by calling the `display()` method.

3- INITIALIZATION

The code of the application's constructor is as follows:

```
public SimpleApplication(String[] args) {

      // 1- define documents' processing parameters
      ProcessingParameters parameters;
      try {
            parameters =
                  new ProcessingParameters(new SimpleState(),
                        SimpleAction.class, Constraint.class, this);
      } catch (Exception e) {
            System.err.println("cannot define parameters: " + e);
            System.exit(1);
            return;
      }

      // 2- create a Telex instance for this application
      Telex telex = Telex.getInstance(this, parameters);

      // 3- open the specified document, or "default.doc" if none is specified
      String pathname = args.length > 0 ? args[0] : "textfile.txt";
      try {
            document = telex.openDocument(pathname);
      } catch (Exception e) {
            System.err.println("cannot open " + pathname + ": " + e);
            System.exit(1);
      }

}
```

The application first defines the parameters that Telex will use for processing the application's documents. These parameters consist in (i) the initial (empty) state of the documents, (ii) the classes of actions and constraints used to update them.

The application then creates a specific Telex instance with these parameters. Telex will associate the specified parameters with every document opened through this particular instance.

Finally, the application opens the shared document to process. In our example, the pathname of the document is specified through the command-line. (If not, a default pathname is used.)

4- MAIN PROCESSING LOOP

The code of the application's main processing loop is as follows:

```
private void run() {

    // main processing loop
    while (true) {

        // wait for next user command
        UserCommand command = waitCommand();

        // update document accordingly
        updateDocument(command);

    }

}
```

The application waits for input by calling the `waitCommand()`. The code of this method is application-specific: the application can interact with a user through a Graphical User Interface, or it can read input from a shell script, etc.

The application then calls the `updateDocument(command)` to translates each command into actions and constraint and pass them to Telex to update the document. In our example, the sketch of this method is as follows:

```
private void updateDocument(UserCommand input) {

    // ...

    // translate input into actions and constraints (this is an example)
    Action action1 = new SimpleAction("delete", "l13,o25", "7");
    Action action2 = new SimpleAction("insert", "input text", "l13,o25");
    Constraint constraint1 = new Constraint(action1, ENABLES, action2);
    Constraint constraint2 = new Constraint(action2, ENABLES, action1);

    // group actions and constraints in a fragment
    Fragment fragment = new Fragment();
    fragment.add(action1);
    fragment.add(action2);
    fragment.add(constraint1);
    fragment.add(constraint2);

    // add fragment to document
    try {
        document.addFragment(fragment);
    } catch (Exception e) {
        System.err.println("cannot update document: " + e);
    }

}
```

Actions and constraints that make up a command are grouped into a *fragment*, which is Telex's update unit. The fragment is then added to the document through the `Document.addFragment()` method.

5- DISPLAYING DOCUMENT STATES

The `TelexApplication` interface defines three up-calls: `execute(Document, ScheduleGenerator)`, `bindDocument(Document, Document)` and `execute(Document, Schedule)`.

Telex uses the first call to notify the application that the specified document has been updated, either locally or remotely. Telex passes as the second argument a *schedule generator*, which the application call to dynamically compute the list of sound schedules until satisfied.

```
public void execute(Document document, ScheduleGenerator generator) {

    // in our case, generator is of type IterableScheduleGenerator
    IterableScheduleGenerator iterable =
                (IterableScheduleGenerator) generator;

    // display possible states to user
    for (Schedule schedule : iterable) {
        SimpleState.display(schedule);
    }

}
```

Telex uses the last two up-calls only when processing *bound documents*, i.e. documents linked by one or more cross-document constraints. The `bindDocument()` call notifies the application that the specified documents are now bound. The `execute(Document, Schedule)` call requests the application to execute the specified schedule because its counterpart on a bound document has been executed.

```
public void bindDocument(Document opened, Document bound) {

      // warn user that specified documents are now bound
      System.out.println(opened + " is now bound to " + bound);

}


public void execute(Document document, Schedule schedule) {

      // display the state corresponding to the specified schedule
      SimpleState.display(schedule);

}
```

ANNEX 1. Simple application code

SimpleApplication.java file

```
package fr.inria.gforge.telex.application.sample;

import fr.inria.gforge.telex.*;
import fr.inria.gforge.telex.application.*;
import fr.inria.gforge.telex.extensions.*;

import static fr.inria.gforge.telex.application.Constraint.Type.*;

/**
 * The skeleton of a simple Telex application. It sketches a basic collaborative
 * text editor. In this example, a <i>document</i> is a shared text file; an
 * <i>action</i> is an edition operation (insert, delete, etc.).
 * <p>
 * For more information, see Telex's Application Programmer Guide:
 * [telex]/docs/guides/applicationPragrammerGuide.pdf.
 *
 * @author J-M. Busca INRIA/Regal
 */
public class SimpleApplication implements TelexApplication, ConstraintChecker {

      //
      // INSTANCE FIELDS
      //
      private final Document document;

      //
      // CONSTRUCTOR
      //
      /**
       * Creates a new instance of the application with the specified parameters.
       *
       * @param args
       *            the command-line parameters.
       */
      public SimpleApplication(String[] args) {

            // define documents' processing parameters
            ProcessingParameters parameters;
            try {
                  parameters =
                        new ProcessingParameters(new SimpleState(),
                              SimpleAction.class, Constraint.class, this);
            } catch (Exception e) {
                  System.err.println("cannot define parameters: " + e);
                  System.exit(1);
                  return;
            }

            // create a Telex instance for this application
            Telex telex = Telex.getInstance(this, parameters);

            // open the specified document, or "default.doc" if none is specified
            String pathname = args.length > 0 ? args[0] : "textfile.txt";
            try {
                  document = telex.openDocument(pathname);
            } catch (Exception e) {
                  System.err.println("cannot open " + pathname + ": " + e);
                  System.exit(1);
            }

      }
```

```
//
// MAIN PROGRAM
//
/**
 * Main method of the application.
 *
 * @param args
 *            command-line parameters.
 */
public static void main(String[] args) {
      new SimpleApplication(args).run();
}

//
// UPDATING DOCUMENT
//
/**
 * Main program of the application.
 */
private void run() {

      // main processing loop
      while (true) {

            // wait for next user command
            UserCommand command = waitCommand();

            // update document accordingly
            updateDocument(command);

      }

}

/**
 * Waits for next user's command.
 *
 * @return the current user's command.
 */
private UserCommand waitCommand() {
      // to be defined: GUI, shell script, etc.
      return new UserCommand();
}

private void updateDocument(UserCommand input) {

      // translate input into actions and constraints (this is an example)
      Action action1 = new SimpleAction("delete", "l13,o25", "7");
      Action action2 = new SimpleAction("insert", "input text", "l13,o25");
      Constraint constraint1 = new Constraint(action1, ENABLES, action2);
      Constraint constraint2 = new Constraint(action2, ENABLES, action1);

      // group actions and constraints in a fragment
      Fragment fragment = new Fragment();
      fragment.add(action1);
      fragment.add(action2);
      fragment.add(constraint1);
      fragment.add(constraint2);

      // add fragment to document
      try {
            document.addFragment(fragment);
      } catch (Exception e) {
            System.err.println("cannot update document: " + e);
      }

}
```

```
	//
	// COMPUTING CONSTRAINTS
	//
	/**
	 * @inheritDoc ConstraintChecker interface.
	 */
	public Fragment getConstraints(Action added, Action existing) {

		Fragment fragment = new Fragment();

		// to be defined

		return fragment;

	}

	//
	// DISPLAYING DOCUMENT STATES
	//
	/**
	 * @inheritDoc TelexApplication interface.
	 */
	public void bindDocument(Document opened, Document bound) {

		// warn user that specified documents are now bound
		System.out.println(opened + " is now bound to " + bound);

	}



	/**
	 * @inheritDoc TelexApplication interface.
	 */
	public void execute(Document document, ScheduleGenerator generator) {

		// in our case, generator is of type IterableScheduleGenerator
		IterableScheduleGenerator iterable =
			(IterableScheduleGenerator) generator;

		// display possible states to user
		for (Schedule schedule : iterable) {
			SimpleState.display(schedule);
		}

	}

	/**
	 * @inheritDoc TelexApplication interface.
	 */
	public void execute(Document document, Schedule schedule) {

		// display the state corresponding to the specified schedule
		SimpleState.display(schedule);

	}

}
```

SimpleAction.java file

```
package fr.inria.gforge.telex.application.sample;

import java.io.*;
import java.util.*;

import fr.inria.gforge.telex.application.*;
import fr.inria.gforge.telex.util.*;

/**
 * The representation of a {@link SimpleApplication}'s action, i.e. an edition
 * operation. For convenience, the operation and its parameters are of type
 * {@code String}.
 *
 * @author J-M. Busca INRIA/Regal
 */
public class SimpleAction extends Action {

	//
	// CONSTANTS
	//
	private static final long serialVersionUID = 1L;

	//
	// INSTANCE FIELDS
	//
	private String operation;
	private String[] parameters;

	//
	// CONSTRUCTORS AND ACCESSORS
	//
	/**
	 * Allocates a new action. This <b>public nullary</b> constructor is
	 * mandatory.
	 */
	public SimpleAction() {

	}

	/**
	 * Allocates a new action representing the specified operation with the
	 * specified parameters.
	 */
	SimpleAction(String operation, String... parameters) {
		this.operation = operation;
		this.parameters = parameters;
	}

	/**
	 * Returns the operation this action represents.
	 *
	 * @return the operation this action represents.
	 */
	public String getOperation() {
		return operation;
	}

	/**
	 * Returns the parameters of this action.
	 *
	 * @return the parameters of this action.
	 */
	public String[] getParameters() {
		return parameters;
	}
```

```
	/**
	 * @inheritDoc
	 */
	public String toString() {
		return operation + Arrays.asList(parameters);
	}

	//
	// LOGGING
	//
	/**
	 * @inheritDoc
	 */
	public void write(DataOutput out) throws IOException {

		// mandatory: write Action's fields first
		super.write(out);

		// then write this instance's fields
		FileUtils.writeUTF("operation", operation, out);
		FileUtils.writeObject("parameters", parameters, out);

	}

	/**
	 * @inheritDoc
	 */
	public void read(DataInput in) throws IOException {

		// mandatory: read Action's fields first
		super.read(in);

		// then read this instance fields in same order as write
		operation = FileUtils.readUTF("operation", in);
		parameters = (String[]) FileUtils.readObject("parameters", in);

	}

}
```

SimpleState.java file

```
package fr.inria.gforge.telex.application.sample;

import java.io.*;

/**
 * The state of a {@link SimpleApplication}'s document, i.e. a text file. For
 * convenience, the state is represented as a set of lines of type {@code
 * String}.
 *
 * @author J-M. Busca INRIA/Regal
 */
public class SimpleState implements DocumentState {

	//
	// CONSTANTS
	//
	private static final long serialVersionUID = 1L;

	//
	// INSTANCE FIELDS
	//
	private List<String> contents;

	//
	// CONSTRUCTORS AND ACCESSORS
	//
	/**
	 * Allocates a new document state.
	 */
	public SimpleState() {
		contents = new ArrayList<String>();
	}

	/**
	 * @inheritDoc
	 */
	public String toString() {
		String result = "";
		for (String line : contents) {
			result += line + "\n";
		}
		return result;
	}

	//
	// DISPLAYING STATE
	//
	/**
	 * Displays the document state corresponding to the specified schedule.
	 *
	 * @param schedule
	 *            the schedule to display.
	 */
	static void display(Schedule schedule) {

		// execute schedule on base state
		SimpleState state = (SimpleState) schedule.getState();
		for (Action action : schedule.getActions()) {
			state.execute((SimpleAction) action);
		}

		// display state
		state.display();

	}
```

```
	/**
	 * Executes the specified action on this state.
	 *
	 * @param action
	 *            the action to execute.
	 */
	private void execute(SimpleAction action) {

		// to be defined

	}

	/**
	 * Displays this state to user.
	 *
	 */
	void display() {

		// to be defined

	}

	//
	// DocumentState INTERFACE
	//
	/**
	 * {@inheritDoc}
	 * <p>
	 * For the sake of simplicity, this implementation splits the text in lines.
	 * This maximizes fragment re-use between successive snapshots, but this
	 * creates numerous files on persistent storage. Splitting the text in
	 * paragraphs would probably be a better trade-off.
	 */
	public Serializable[] split() {
		return contents.toArray(new String[0]);
	}

	/**
	 * @inheritDoc
	 */
	public void assemble(Serializable[] fragments) {
		contents = new ArrayList<String>();
		for (Serializable line : fragments) {
			contents.add((String) line);
		}
	}

}
```

This page is intentionally left blank

Annex 6. Virtual Organization File System

VOFS Manual

Georgios Tsoukalas, ICCS
gtsouk@cslab.ece.ntua.gr

June 28, 2009

1 Requirements

- Python 2.5
- Fuse-2.7 and development files
- SQLite 3
- Libc development files

2 Installation & Launch

- Unpack the archive.
- Run setup.sh

`setup.sh` will ask for a name and an IP address for your peer, a member list and a provider list with which to initialize your workspace, a location for the VOFS-launching script and a location for the mountpoint. The script will write a configuration file (`src/scripts/config`) and create a launching script that will read this configuration file.

2.1 Launch a typical VOFS instance

The launching script is named `launch-vofs` and its location is configured by the `setup.sh` script.

To launch the configured (by `setup.sh`) VOFS workspace, run this script.

`launch-vofs` will read the configuration file and then launch a VOFS server and a VOFS client. The client will mount the VOFS server onto the mountpoint configured. The files served by the VOFS server will be available under the mountpoint.

The script then will initialize the VOFS filesystem with a members directory with symbolic links to the configured members of the workspace, and initialize the storage pool of workspace (the top-level directory of the filesystem) with the providers configured during set-up.

By default, the VOFS server will be configured to also be the storage provider.

The launcher will create 2 log files in the current directory: one for the client peer and one for the service (filenames start with a '.'). The log files contain debugging output.

The service and client can be individually started with ./runservice.sh in src/, provided PYTHONPATH is set-up correctly. See the output of setup.sh for PYTHONPATH settings.

2.2 Launching file servers and providers independently

It is possible to launch VOFS servers and clients manually.
`./launch-vofs runpyfuse` will launch a VOFS client:

```
runpyfuse <mountpoint> <client_addr> <server_addr>
options:
    mountpoint:   where to mount server
   client_addr:   local client address to bind to
   server_addr:   remote server address to mount
```

`./launch-vofs runservice <name:service>` will launch a VOFS server. *name* must be a valid hostname or a virtual hostname configured in `tractal/lib/hostnames.py`. *service* must be a port number, a service configured in the file mentioned above. If the *service* character string cannot be resolved to a port number, it will be converted to a port number by in a random but consistent way (by hashing it). Therefore, one can use any string as a service, with a minimal risk of collision due to hashing (which maps to 8192 different ports).

VOFS servers act as both file servers and storage providers at the same time.

3 Introduction

VOFS is fundamentally a network of peers which are identified by a unique cryptographic identity that can be used in authentication and authorisation of transactions among them. A peer's name can be either its identity or a name through which the identity can be retrieved. Each peer may assume one of three roles all supported by a unified protocol: file server, storage provider and client.

The file server is the authority to serve a user's files to the network. File servers keep the file hierarchy and metadata for each file. Each file is identified by the peer's identity followed by a path unique to each file server. Clients may explore a file server's hierarchy. There are symbolic links that can be followed accross the network, effectively linking the filesystems together in a WWW-like fashion.

File data are not normally kept in the file server. Instead, a list of storage chunks is kept with the file metadata. These storage chunks are kept and served by storage providers, in similarly to file metadata. VOFS can be extended through storage plug-ins to handle other protocols for storage than its own (FTP, for example).

Clients create files by first allocating data chunks with their data and then writing creating or updating the file in the file server with the new storage chunk list. Clients are responsible for the allocation of data chunks on their own. However, the file server may refuse to accept a file if there has been implemented a policy that refuses the file's storage chunk list.

A set of of key-value pairs, the *attributes* is associated with each file. Attributes are inherited from parent directories so that large hierarchies can be conveniently managed. POSIX attributes (like access times) are encapsulated

in those attributes. These attributes may be used privately by clients or they may be given special semantics by extensions to VOFS.

An example of a standardized attribute is the "pool" attribute. This attribute holds a list of storage providers available so that clients know where to allocate storage chunks for file data for this filesystem (or a subset of it, since "pool" as an attribute can be independently be set for any file).

Clients keep a local cache with copies of files and storage chunks. User requests are served from this cache to reduce latency and overcome connectivity problems. Clients may cache the attributes of files and keep an additional local set of attributes. The local attributes can be used by users or applications to configure the behaviour of the client.

For example, the contents of the *forward* local attribute for each file is automatically attached by the client to every request that is made to the file's server. Client- and server-side extensions may use this mechanism to exchange credentials without any client modifications.

Traditionally, the file system interface is used by both users and applications. This has been preserved in VOFS. In addition to all that is available to users, applications may subscribe to file servers for events happening to files. Applications may also trigger a message-passing `PUBLISH` event that can be subscribed to and used for group communication based on a file as a publishing point. The list of subscribers to this publishing point file is a special attribute and is cached by clients, limiting the disruption from the disconnection of its file server.

Disconnection is expected by VOFS and users can actually force it. Disconnectedness is implied by communications timing out and the state is maintained per host. Users and applications may force files in and out of the state of disconnectedness.

Local changes to a client are immediately stored in the local cache. The local cache is synchronised periodically or when required by the user or a system operation. VOFS does not guarrantee any consistency as the last request to be served by the file server will supersede all previous ones.

Access control in VOFS is based on filtering access requests. Each request has an origin and a recipient peer, a target resource and a specific access that is requested. Based on this design, access control may be implemented as simple as setting a password for files (or whole hierarchies), or as complex as filtering the requests through a sophisticated external security infrastructure. Whatever the access control method employed, users just have to place their credentials in their `forward` virtual files, as described earlier.

To make VOFS features available over POSIX calls, VOFS implements *virtual files*. Each path in VOFS may have an additional path component separated by a `@` character:

```
/normal/path@virtual
```

The virtual path refers to a virtual file that is specific the normal path. For example, `file@data` displays what storage provider hosts the data for the specific file.

The most important virtual file (actually a directory) is `directory/@/`, which represents the whole VOFS network as a directory. This is a global virtual file and can be accessed from any directory. Users can chdir in a peer address there to browse that peer's files:

```
ls -l @/peer:fs/
```

or link documents across filesystems:

```
ln -s @/ICCS:fs/docs/profile.pdf profiles/ICCS.pdf
```

4 Configuration

4.1 Timeouts

In the file `src/tractal/client/config/base.py` there are two timeout assignments:

```
network_timeout = 3
stale_timeout = 5
```

If communication is expected from a peer for more than `network_timeout` seconds, then that peer is considered disconnected.

When a file is accessed, it is checked how much time has passed since it was last retrieved from the network. If more than `stale_timeout` seconds have passed, then the file is re-requested to discover if there is a new version available.

Both timeouts deliberately default to small values, to aid the demonstration of VOFS features despite creating heavier load.

5 Usage

5.1 The Mountpoint

The mounted filesystem is a FUSE volume that is served in userspace with the credentials of the user launching VOFS. Other users won't normally have access to the mount point.

Beware that the mountpoint must be empty before launching and that there are no applications with their current working directories under the mountpoint or it will not be able to unmount it.

5.2 Virtual Files

Virtual files are special files that do not represent a VOFS file, but their contents and metadata are dynamically computed. They are artifacts that have been introduced to support VOFS filesystems in a POSIX-compliant way.

In order to preserve compatibility, virtual files are normally invisible to directory listings. Nevertheless, they remain accessible, as if someone creates them the moment before the access and deletes them the moment afterwards (this is valid behaviour in conventional filesystem).

Paths to virtual files contain the reserved character '`@`'. This character cannot be used in normal filenames. The character '`@`' splits the path into two distinct paths:

`<real path>@<virtual path>`.

This way, every virtual path is associated with a real one.

Due to the way FUSE works, it is not possible to write input and read output from the same virtual file. Therefore, pairs of virtual files are introduced, one for writing and one for reading. For example, one such pair is `@setconfig` and `@config`.

5.2.1 List of Virtual Files

`@/` Effectively this is the virtual file with an empty virtual path component. It is a global virtual file, accessible in every directory, but canonically accessed in the root of the mountpoint. It is a directory that represents the whole VOFS network. Entering paths in the form `host:service/` in that directory will cause the client to contact the VOFS peer with that address and retrieve its files.

This is the mechanism to create symbolic links across two different peers and thus, filesystems, in the network.

`@config` contains a list with the attributes of a file. Attributes are key-value pairs that are inherited from parents and can be set through `@setconfig`.

`@setconfig` is used to create and delete attributes for a file. There are two types of attributes, *plain* key-value pairs and *dictionaries*. *Plain* attributes have a string value, while *dictionaries* contain their own list of plain attributes. Dictionaries can be used as sets of if they contain only keys with empty values.

To assign a plain attribute:

```
echo -n "key = value" > file@setconfig
```

To delete a plain attribute:

```
echo -n "-key" > file@setconfig
```

To assign a plain attribute to a new dictionary attribute:

```
echo -n "dict = key=value" > file@setconfig
```

To add a plain attribute to an existing dictionary attribute:

```
echo -n "dict += key=value" > file@setconfig
```

To delete a plain attribute from an existing dictionary attribute:

```
echo -n "dict -= key=value" > file@setconfig
```

To delete a dictionary attribute:

```
echo -n "-dict" > file@setconfig
```

`@offline` provides access to the connectivity status of a file. Attempting to read a connected file will result in an error:

```
EISCONN (Transport endpoint is already connected)
```

Attempting to read an offline file will result in an error:

```
ENOTCONN (Transport endpoint is not connected)
```

Attempting to write will force the file to be offline and return an error:

```
ENOTCONN (Transport endpoint is not connected)
```

Attempting to remove (unlink) it will discard the offline status of the file:

```
EISCONN (Transport endpoint is already connected)
```

`@forward` contains the credentials stored for forwarding to the server for the specific file.

`@setforward` is where the credentials to be stored for forwarding are written.

`@pool` contains a list of the `pool` dictionary attribute, for convenience.

`@status` reports the cache and connectivity status of a file. According to local modifications a file can be `CLEAN` or `DIRTY`. According to last network access a file can be `FRESH` or `STALE`. If a file is offline `OFFLINE` is also returned.

`@node` dumps the internal VOFS structure for the metadata of the specific file.

`@data` displays where the specific file's data are stored.

5.3 Browsing and Symbolic Linking Across the Network

Accessing other peers (and their filesystems) can be done directly:

```
cd @/name:service/
```

or a permanent link to the peer can be created as a symbolic link:

```
ln -s @/name:service Georgios; cd Georgios/docs
```

name must be a valid hostname or a virtual hostname configured in `tractal/lib/hostnames.py`. *service* must be a port number, a service configured in the file mentioned above. If the *service* character string cannot be resolved to a port number, it will be converted to a port number by in a random but consistent way (by hashing it). Therefore, one can use any string as a service, with a minimal risk of collision due to hashing (which maps to 8192 different ports).

5.4 Storage Pooling

5.5 Disconnected Operation

When a file is available in cache, it is immediately served. If the file is stale, then an attempt to fetch a new version will be made. This ensures that if a file is in cache, then being connected or offline makes no difference.

If a file is not in cache and the attempt to retrieve it times out, the file is placed in offline status and any subsequent attempt to retrieve it will immediately fail. After some time, the offline status expires and communication is attempted anew. The offline status can be manipulated by the user.

5.5.1 Check a file's connectedness status

see `@forward` or `@status` virtual file.

5.5.2 Force a file's connectedness status

see `@forward` virtual file.

5.6 Access Control

VOFS uses the *forward* mechanism to forward credentials from clients to servers, so access control is entirely an external module to be plugged in VOFS. By default, the authorisation mode (attribute `authmode`) is `PRIVATE`. In this simple mode, every file has a `password` attribute. Clients are required to provide this password in order to access the file. By default, both password in the server and forward in the client is the empty string, so access is actually public.

Setting `authmode` to other values triggers other authorisation modes (such as filtering through a PEP), provided that the appropriate plugins are available at the serverside.

5.7 Tips

- Search logfiles for exceptions (string "Traceback") to see if something is going wrong
- The mountpoint must be empty, and chdir out of it when stopping the client so that it can be unmount
- `pkill -f python` will probably kill everything when needed
- if you run runpyfuse.sh or runservice.sh, stop them with Ctrl-

This page is intentionally left blank

Annex 7. WebDAV VOFS (Virtual Organization File System) User Manual

by Leif Lindbäck and Vladimir Vlassov
Royal Institute of Technology (KTH), Stockholm, Sweden
Email {leifl, vladv}@kth.se

FP6 Project Grid4All (IST-2006-034567)

1. Installation

1. Download and install the security infrastructure, `http://www.isk.kth.se/~leifl/vofs/grid4all-sec-0.6.zip`. For instructions, see the Security Infrastructure, User's Guide. The security infrastructure will only run on one host.
2. Download and unpack the vofs core, `http://www.isk.kth.se/~leifl/vofs/vofs-core-0.2.2.zip`.
3. In the config file `vofs-core-0.2.2/apache-tomcat-6.0.14/webapps/webdav/WEB-INF/grid4all.pep.config`
   change the entry `PDPHost` to the address of the host where the PDP will run.

2. Start up

1. Initialize and start the voms according to the documentation in the security infrastructure.
2. Start the PDP according to the documentation in the security infrastructure.
3. Set the environment variable `VOFS_HOME` to the absolute path of the installation directory of the vofs core, i.e. `vofs-core-0.2.2`.
4. Start the VOFS with the command `bin/vofs`.
5. Log in to the VO by opening the URL below in a web browser.
   `http://<host where voms is running>:8080/voms/`
   In the fields `VOFS Host` and `VOFS Port`, enter the ip address and port of your local VOFS peer. The ip address should be accessible from other vofs peers, i.e. it should not be `localhost`. The default port number is `8082`. Note that the VOFS peer must be started before this is done.
6. Mount the VOFS with a mount utility supporting WebDAV, for example davfs2 in Linux or NetDrive in Windows. The address of the mounted peer should be `http://localhost:8082/webdav`

3. Usage

The vofs has both a GUI and a command line interpreter. Both are displayed when the vofs is started.

3.1 Command line interpreter

The following commands exist.

- **`expose LocalResource VOFSPath`**
  Exposes the file or directory specified by `LocalResource` to the directory specified by `VOFSPath`. Note that an exposed file retains its local name, `VOFSPath` only specifies the directory in which it is placed. Exposed directories on the other hand, are renamed to the specified `VOFSPath`. Directories specified in `VOFSPath` need not exist, they are created if needed.

  Example: If `LocalResource` is the file `c.txt` and `VOFSPath` is `/a/b`, the file will be accessible in vofs as `/a/b/c.txt`. If `LocalResource` is the directory `c` and `VOFSPath` is `/a/b`, the directory will be accessible in vofs as `/a/b`.
- **`unexpose VOFSPath`**
  The resource at the specified `VOFSPath` is removed from the vofs name space. However, cached copies are not removed.
- **`list`**
  Lists all exposed resources
- **`add PeerName`**
  Adds `PeerName` as a neighbor. `PeerName` should have the format `host:port`.
- **`rm PeerName`**
  Removes `PeerName` from the neighbor set. `PeerName` should have the format `host:port`.
- **`peers`**
  Lists all neighbors.
- **`help`**
  Lists all commands.
- **`exit`**
  The console, but not the vofs itself, is terminated.

3.2 GUI

When VOFS is started the control panel, see figure 1, is showed.

Figure 1. VOFS control panel

- The `Expose` button is used to expose files. The expose semantics are explained in section 3.1 above.
- The `Unexpose` button is used to unexpose files. The unexpose semantics are explained in section 3.1 above.
- The `Peers Management` button displays the `Peers Management` window, see figure 2. This window shows a list containing all neighbor peers, and allows to remove one or all neighbors and to add new neighbors.

Figure 2. Peers Management window.

- The `Settings` button in the control panel (see figure 1) displays the `Advanced Settings` window, see figure 3. This window allows to change cache size and debug level. Debug level zero means no log messages, higher levels mean more messages. Debug level three is the highest, this means all available messages are printed. Debug messages are printed to standard out, in future VOFS versions this will be changed to log files. More settings will be configurable from the `Advanced Settings` window in future versions of VOFS.

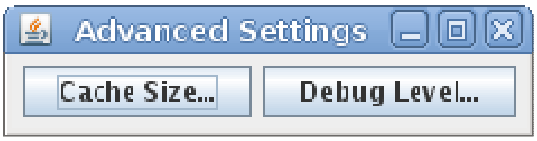


Figure 3. Advanced Settings window

- The Exit button in the control panel (see figure 1) stops the entire VOFS peer, not just the GUI.

This page is intentionally left blank

YASS (Yet Another Storage Service) User's Guide

Leif Lindbäck and Vladimir Vlassov

Royal Institute of Technology (KTH), Stockholm, Sweden
{leifl,vladv}@kth.se

Table of Contents

1 Introduction

YASS, Yet Another Storage Service, is a storage service that allows users to store, read and delete files on a set of computers. The service transparently replicates files in order to achieve high availability of files and to improve access time. YASS can be deployed and provided on computers donated by users of the service or on computers of a service provider. YASS operates even if computers join, leave or fail at any time.

The current version of YASS transparently maintains the specified number of file replicas despite of nodes leaving or failing, and it can scale (i.e. increase available storage space) when the total free storage is below a specified threshold.

YASS can be used by any users who need a reliable and scalable storage to store and retrieve files, e.g. for backup or file sharing. Users can access the service by executing and interacting with the YASS client (front-end) on their computers. Executing YASS storage components on their computers, users can share their storage with each other.

2 Download

YASS is included in the Niche distribution. See *Niche Quick Start Guide* for Niche download instructions. Niche and YASS are available at `http://niche.sics.se/`

3 Installation and startup

In order to be deployed and to operate, YASS requires Niche and the Apache Ant build tool to be installed. For Niche installation and startup, see *Niche Quick Start Guide*. For Ant download and installation, see `http://ant.apache.org/index.html`. As YASS is a part of the Niche distribution, it is installed when Niche is installed.

4 Configuration

YASS is configured by editing values of the following Java runtime properties in the file `niche-0.2/Jade/etc/execute.properties`. Be sure that the properties listed below have the same values in the two sections, `jvm.parameters.oscar.jadenode` and `jvm.parameters.oscar.jadeboot`, of the file.

yass.storage.fileTransferPort The port used for file transfer. If this port is occupied, YASS increases the port number by one until a free port is found.

yass.storage.root The directory in the file system used by YASS for storing files. This is only used on computers running storage components.

yass.storage.bufsize The size of the buffer used for file transfer in YASS.

yass.test.defaultReplicationDegree The number of replicas of each file.

5 Deploying and Starting YASS

1. Deploy by running Ant with the target `testG4A-Yass-Deploy` specified in the build file `niche-0.2/Jade/build.xml`, as follows
   ```
   cd niche-0.2/Jade/
   ant testG4A-Yass-Deploy
   ```
2. Start by running Ant with the target `testG4A-Yass-Start` specified in the build file `niche-0.2/Jade/build.xml`, as follows
   ```
   cd niche-0.2/Jade/
   ant testG4A-Yass-Start
   ```

6 Usage

6.1 User Interface

The user interface of the YASS frontend is shown in Figure 1 and Figure 2. The frontend allows the user to store/retrieve files to/from YASS as explained below. Figure 1 shows how the local directory tree is displayed. When a directory is chosen its contents are shown in the local file view, which is marked in Figure 2.

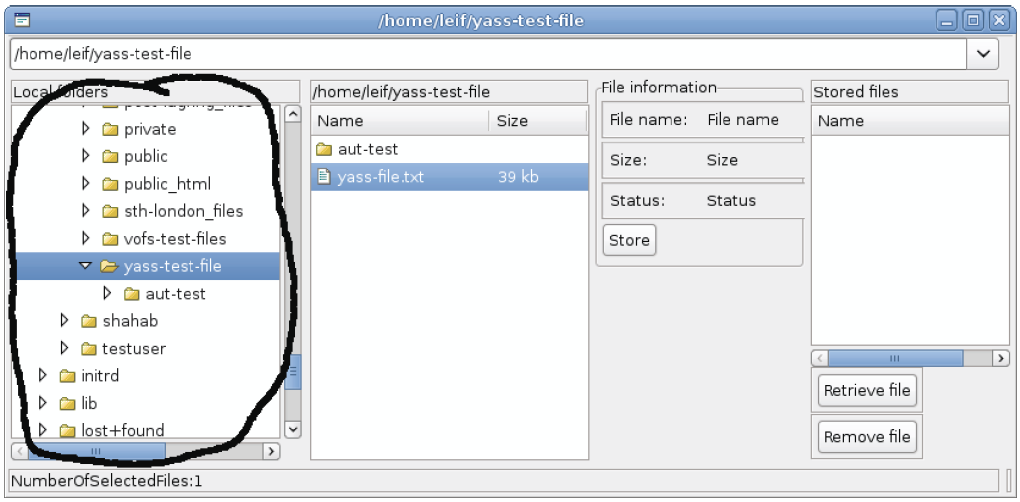


Fig. 1. The local directory tree.

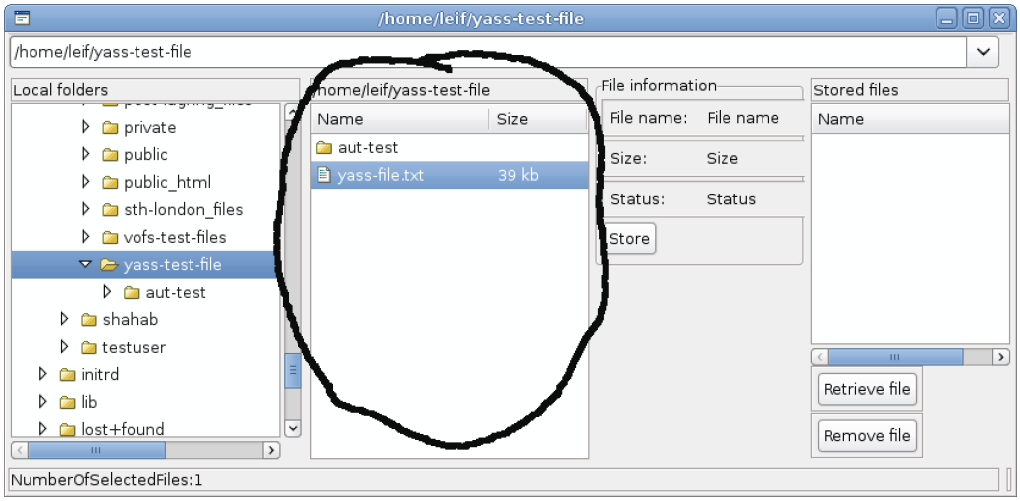


Fig. 2. The files in the chosen local directory.

6.2 Store File

1. Select the local directory containing the file that shall be stored.
2. Double click the file to store.
3. Click the Store button.
4. When successfully stored, the file will appear in the list of stored files, as shown in Figure 3.

6.3 Retrieve File

1. Select the file to retrieve in the list of stored files. This list is marked in Figure 3.
2. Click the Retrieve File button.
3. The file is copied from the remote storage to the local file system path from which it was read when stored.

6.4 Remove File

1. Select the file to remove in the list of stored files. This list is marked in Figure 3.
2. Click the Remove File button.
3. The file is removed from the remote storage.

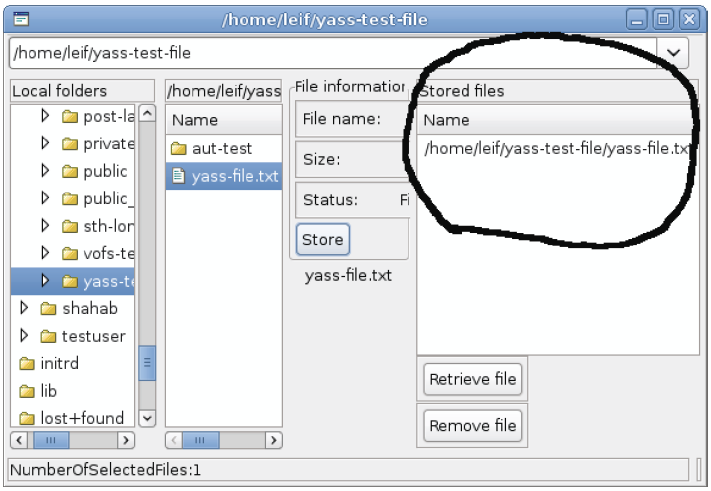


Fig. 3. The list of stored files.

This page is intentionally left blank

ANNEX 9. YACS: YET ANOTHER COMPUTING SERVICE

PROGRAMMER'S AND USER'S MANUAL

by Atli Thor Hannesson and Vladimir Vlassov
Royal Institute of Technology (KTH), Stockholm, Sweden
Email {athan, vladv}@kth.se

FP6 PROJECT GRID4ALL (IST-2006-034567)

Contents

1 Introduction

YACS (Yet Another Computing Service) is a robust distributed computing service that allows a user to submit and execute jobs, which are bags of independent tasks, on a network of computers (nodes). The service can be used to perform different kinds of batch jobs or bag-of-task applications such as parameter-sweep simulation, video transcoding, ray-tracing or other applications that follow the master-worker paradigm. YACS can be deployed and provided on computers donated by users of the service or on computers of a service provider.

YACS executes jobs, which are collections of tasks, where a task represents a particular type and instance of work that needs to be done. For example, in order to transcode a movie, the movie file can be split on several parts (tasks) to be transcoded independently and in parallel.

Tasks are programmed by the user and can be programmed to do just about anything. Tasks can be programmed in any programming language using any programming environment, and placed in a YACS jobs (bag of tasks) using the YACS API.

YACS uses distributed masters and workers to execute jobs (see Figure 1). A user submits jobs through the YACS frontend, which assigns jobs to masters (one job per master). A master finds workers to execute tasks in the job. YACS monitors execution and restarts failed jobs and tasks. When all tasks complete, results of execution are returned to the user.

YACS guarantees execution of jobs despite of nodes leaving or failing. YACS supports check-pointing that allows restarting execution from the last checkpoint. Furthermore, YACS scales, i.e. changes the number of masters and workers, when the number of jobs/tasks changes. In order to achieve high availability, the service always maintains a number of free masters and workers so that new jobs can be accepted without delay.

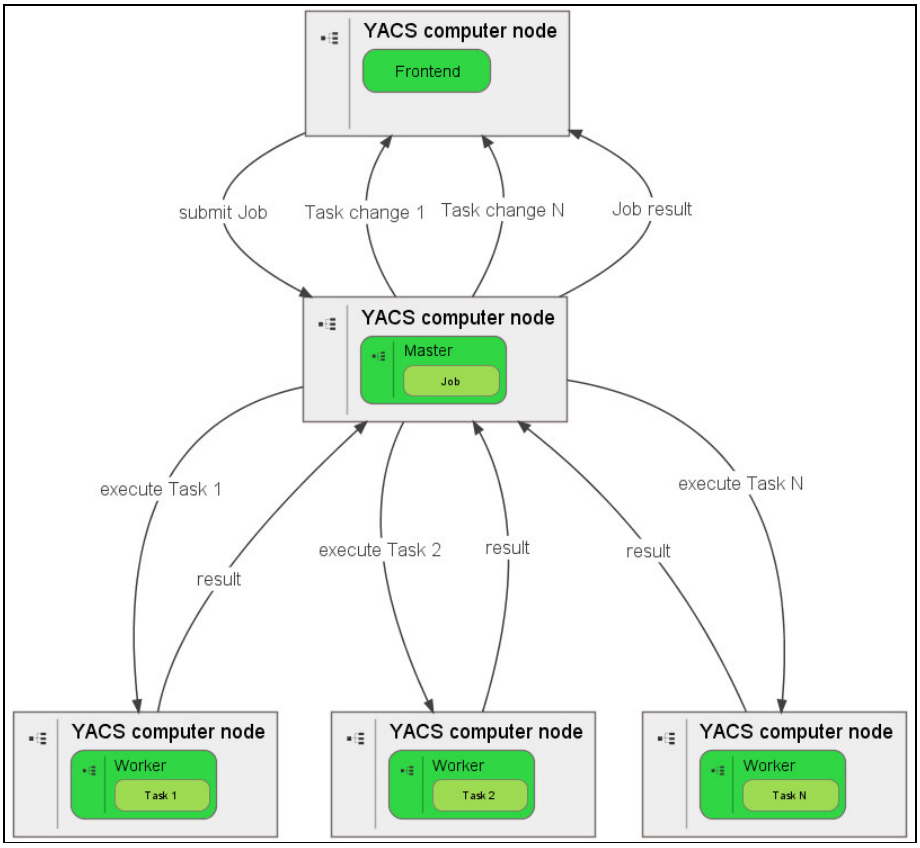


Figure 1. Architecture of the functional part of YACS: frontend, masters and workers.

Next sections describe how to program tasks to be executed by YACS, encapsulate the tasks in a job, submit the job and receive results of execution. Assume the following three roles for YACS stakeholders:

- *Task Programmer* who is responsible for programming tasks;
- *Client Developer* who is responsible for developing a YACS client application used to create and submit jobs to YACS and to get results of execution;
- *End-User* of the service, who submits jobs via the YACS client application.

2 Programming Tasks

This section is for the Task Programmer who is responsible for programming tasks. YACS is implemented in Java, and therefore tasks to be executed by YACS are programmed in Java by extending the abstract class `yacs.job.tasks.Task`. The Task Programmer should at least implement the abstract method `execute` and optionally override the methods `initFromState`, `reinitFromState` and `prepareMetacheckpoint`.

The `execute` method should define the code of tasks represented by this `Task` subclass. The Task Programmer should program the appropriate task logic in this method. The method can be programmed to perform any arbitrary tasks, including calling external programs or scripts. The `execute` method is invoked on the `Task` object by a Worker component in order to perform the task. When the `execute` method returns, the Worker assuming that the task is finished sends to its Master an object of the `yacs.job.TaskContainer` class that holds results and status of execution. The Task Programmer should not access fields of the `TaskContainer` object directly but rather using corresponding setters and getters of the `Task` class.

The `initFromState` and `reinitFromState` methods are optional and they can be overridden by the Task Programmer to include initialization code to be executed by the worker before the `execute` method is invoked. The expected semantics of `initFromState` and `reinitFromState` methods is to get and interpret initialization parameters (in the `initParams` array of `TaskContainer`) in order to initialize the task to its initial state, or to the state of its last checkpoint, respectively. The `initFromState` or `reinitFromState` method is called by the Worker responsible for the given task before it calls the `execute` method on the `Task`.

The `prepareMetacheckpoint` method should contain the code to prepare checkpoint of the `Task` to be stored by the checkpoint service described below.

In order to productively use the YACS task API, it is strongly recommended for Task Programmer to study fields of the `TaskContainer` class, lifecycle of a task execution, and methods of the `Task` class.

2.1 Important Fields of the `TaskContainer` Class

For each task, YACS creates a serializable object of the `yacs.job.TaskContainer` class that is used to hold a current lifetime status, id, and some other information for the associated `Task`. This information can be used by Task Programmer when programming tasks. The Programmer should not directly use the `TaskContainer` object, but rather access its fields by corresponding getters and setters of the `yacs.job.tasks.Task` class. Some important fields of `TaskContainer` are described in the table below.

`int resultCode`	The result code of execution that can be set to some value to be interpreted by the client application.
`int tid`	A unique identifier of the associated `Task` in a particular instance of `Job`.
`int status`	Indicates current lifecycle status of the associated `Task` that can be in one of the following four statuses – `TASK_NOT_INITIALIZED` – the `Task` has been assigned and deployed but has not be initialized and has not been started yet, i.e. the `execute` method has not been invoked yet; – `TASK_IS_PROCESSING` – the Worker thread executes the `execute` method of the `Task`; – `TASK_FAILED` – execution of the `Task` has failed; – `TASK_COMPLETED` – the `execute` method of the task has returned The value of `status` is set by YACS and can be interpreted by both, YACS and a client application.

`boolean redeployable`	Indicates whether the associated `Task` can be restarted (redeployed) at another Worker if the current Worker fails for some reason. This field should be set to `false` if the `Task` defines work which is non-repeatable.
`Serializable[] initParams`[1]	An array of values that used to initialize this `Task` to its initial state or to the state of its last checkpoint. In order to initialize the `Task`, the worker invokes `initFromState` or `reinitFromState` method on this `Task`, respectively, before calling the `execute` method of this `Task`. The array should be used for the following three purposes: 1. To provide initialization parameters for the very first initialization of the associated `Task` to its initial state; 2. To provide initialization parameters for initialization of `Task` from its last checkpoint (for example, an URL of a file with the latest checkpoint) 3. To hold results of execution.

2.2 Task Lifecycle

Figure 2 depicts lifecycle of a task. Remind that the task is represented by the `Task` object containing the code of the task, and by `TaskContainer` object containing state of the task. When the `Task` object is instantiated, the task is in the `TASK_NOT_INITIALIZED` state. The task passes to the `TASK_IS_PROCESSING` state, when the worker invokes the `initFromState` (or `reinitFromState`) method and then calls `execute` on the `Task` object. The task is initialized either to its initial state by the `initFromState` method, or to its last checkpoint by the `reinitFromState` method. The initialization state is stored in the `initParams` array of `TaskContainer`. Which of the initialization methods is invoked depends on whether the task is executed first time, or it is restarted on a new worker as the previous worker has failed to execute the task. In the former case (initial execution), the worker calls `initFromState`; in the latter case (restart) it calls `reinitFromState`. If execution or (re-)deployment (i.e. starting on a new worker) of the task fails, it passes to the `TASK_FAILED` state. When the `execute` method returns, the task passes to the `TASK_COMPLETED` state.

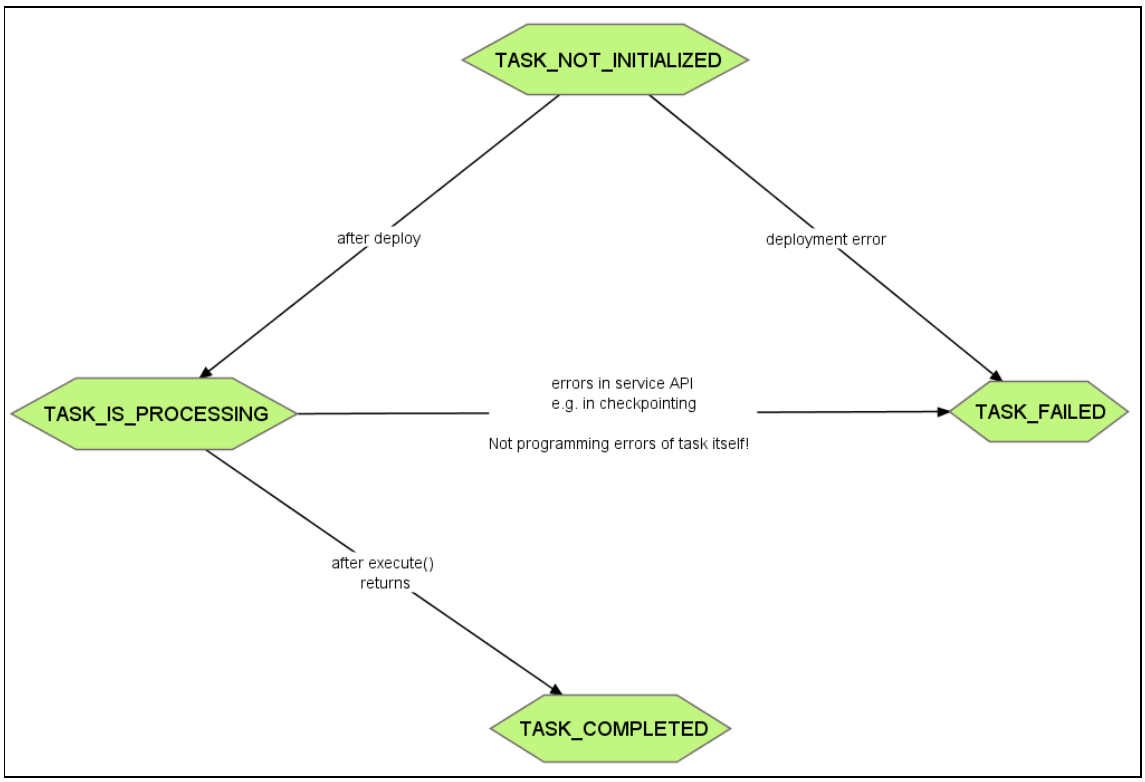


Figure 2. Task lifecycle

[1] Note that the `initParams` field might be subject to change in future version of YACS in order to distinguish initialization parameters, results of execution, and checkpoints, stored in this field of `TaskContainer`.

2.3 Important Methods of the `Task` Class

The following table describes some methods of the `yacs.job.tasks.Task` class used to program tasks. Remind that the Task Programmer should at least implement the `execute` method and, optionally, override the `initFromState`, `reinitFromState` and `prepareMetacheckpoint` methods. The `execute` method should contain the code of the tasks to be performed by YACS workers. The `initFromState` and `reinitFromState` methods should contain initialization code to initialize the task to its initial state or to the state of its last checkpoint, respectively. One of these methods is executed before the task is started, i.e. before the `execute` method is invoked. The `prepareMetacheckpoint` method should contain the code to prepare checkpoint of the `Task` to be stored by the checkpoint service described below.

`execute()`	Start execution of this `Task`. The Task Programmer should program the appropriate task logic in this method. When a worker instantiates the `Task` object, it invokes the `execute` method on that object. Once the method returns, the Worker assumes that the task is finished, and it sends a `TaskContainer` object (that contains results) to its Master.
`initFromState()`	Initialize this `Task` to its initial state using `initParams` of `TaskContainer`. This method is invoked by the worker before it calls the `execute` method on this `Task`. This method should be overridden by the Task Programmer to include a code that initializes `Task` to its initial state represented by `initParams` of the `TaskContainer`. If not overridden, this method does nothing.[2]
`reinitFromState()`	This method is similar to `initFromState` except that it initializes this `Task` to the state of its last checkpoint. Invocation of this method indicates that a prior worker responsible for this `Task` has failed, and the task (together with its checkpoint) has been handed over to another worker. This method should be overridden by the Task Programmer to include a code that initializes the `Task` object to its last checkpoint represented by `initParams` of `TaskContainer`. If not overridden, this method does nothing.
`TaskExecutionContext getExecutionContext()`	Get the Task Execution Context of this `Task`. This method returns a handle to an interface which provides access to the checkpoint service provided by YACS. The service is used for storing and retrieving checkpoints during task execution. The checkpoint service provides ability to restart a task from its last checkpoint in the case of worker failure.
`prepareMetacheckpoint()`	Prepare a (meta-)checkpoint[3] of this `Task`. This method should be overridden by the Task Programmer to include code that creates a checkpoint to be stored in the checkpoint service. The checkpoint data are to be prepared in the `initParams` array of `TaskContainer`. It can be an array of values, or an URL of a checkpoint file. It is up to the Task Programmer to define date, the format, and the amount of state to be saved in a checkpoint. When using checkpointing, the Programmer should also implement the `reinitFromState` method to provide code that initializes the task to the state of its last checkpoint.
`log( String msg )`	This method will write the log-message to the same logfile which the Worker component uses.
`initTaskInfo()`	This method can be overridden by the Task Programmer to set a member variable called `logName`. This `logName` will be used along with the log function to make those log entries more easily identifiable in the logfile. For example for error analysis or debugging.

[2] Note that the method signature is a subject to change that a `Serializable[]` array that represents the initial state is given as an input parameter to the method instead of using `initParams` of `TaskContainer`.
[3] We call a checkpoint *metachackpoint* to indicate that it may contain information on location of the checkpoint rather than the checkpoint itself.

2.4 Task Execution Context and Checkpoint Service

Apart from distributed execution, the Task Programmer can use some additional services provided by YACS through the task execution context (`TaskExecutionContext`). The current YACS prototype supports only one additional service – the checkpoint service that allows Task Programmer to perform task checkpointing whenever needed. YACS can be extended to provide more services, for example, a uniform storage interface.

`metacheckpoint()`[4]	Request the checkpoint service to store a checkpoint of this `Task`. This method can be called by the worker. The Task Programmer can program to call this method whenever appropriate during task execution. The checkpoint service allows restarting the `Task` from its last checkpoint in the case of worker failure. This method invokes the `prepareMetacheckpoint` method (to be overridden by the Task Programmer, see 2.3) on the `Task` object before the checkpoint service stores a copy of `initParams` as task checkpoint.

2.5 The `TaskContainer` Class

The Task themselves are not instantiated at the client side but only at the Worker component that the task is assigned to. For this reason a special `yacs.job.TaskContainer` class is used to contain the `Task` subclass definition and associated data, such as the `resultCode`, the `status`, the `redeployable` flag and the `initParams Serializable[]` array (see 2.1). This class is used by the `Job` class for storing task information and state in the service, and for communication of tasks between the client and the service. The `TaskContainer` contains the static method `contain` for convenient containment of tasks.

`TaskContainer`
`contain(`
`int tid,`
`boolean redeployable,`
`String className,`
`String classfileLocation,`
`Serializable[] initParams )`
	Creates a `TaskContainer` which contains all data which are necessary to instantiate and initialize an instance of the `Task` subclass with the given class name at the worker, which is assigned this task to execute. `tid`, `redeployable` and `initParams` parameters were explained earlier (see 2.1); `className` is the fully qualified name of the `Task` subclass; `classfileLocation` is the path to the class file.

The following fragment of Java code illustrates usage of the `contain` method to create a `TaskContainer` object that represents the "sleep" task of the `DirectedSleepTask` class.

```
Serializable[] initParams = new Serializable[]{
            "/taskfiles/commands.txt",
            new Long(10000)
      };
TaskContainer tc = TaskContainer.contain( 1, true,
      "yacs.job.tasks.DirectedSleepTask",
      "/yacs/job/tasks/DirectedSleepTask.class",
      initParams) );
```

The `TaskContainer` class includes setters and getters to access task data such as task id, status, and any data (init parameters, execution results, or checkpoints) stored in the `initParams` array. See 2.1 for `TaskContainer` fields used to store the task data. The `TaskContainer` also includes some setup and initialization code to be performed at the Worker, such as loading the task class so that the Worker can instantiate and execute the assigned task whose class is not known at compile time. Note that in the current implementation, `TaskContainer` supports only one class file for the corresponding task. As a consequence, the entire task implementation must be contained within one class file.

The `yacs.job.tasks` package contains a number of sample task classes, e.g. which are can serve as examples that illustrate how a task can be programmed.

[4] Note that this method is subject to change so that it takes a checkpoint (a `Serializable[]` array) as an input parameter rather than relying on the `prepareMetacheckpoint` call and the `initParams` field.

3 Programming and Submitting Jobs

This section is for the Client Programmer who is responsible for developing a YACS client application used to submit jobs to YACS and to get results of execution.

3.1 Creating Jobs Using the `Yacs.Job.Job` Class

The class `yacs.job.Job` represents a job, which is a task container ("bag of tasks") where each task is represented by a object of the `yacs.job.TaskContainer` class containing task data, e.g. task id, class name, status, results, initialization parameters or checkpoint data, etc, as described in 2.1 and 2.5.

In order to create a job as a bag of tasks, a YACS client (developed by the Client Programmer) should instantiate a `Job` object, create `TaskContainer` objects using the static method `contain` of the `TaskContainer` class, and add those tasks to the job. The following example illustrates creation of a job containing 5 tasks `DirectedSleepTask` numbered 1, 2, … 5:

```
Job job = new Job("EJ"); // creates a Job instance
job.setCreator( myGlobalId );
Serializable[] initParams = new Serializable[]{
            "/taskfiles/commands.txt", new Long(10000)
      }; // init parameters for tasks
for (int i = 1; i < 6; i++) {
   TaskContainer tc = TaskContainer.contain( i, true,
      "yacs.job.tasks.DirectedSleepTask",
      "/yacs/job/tasks/DirectedSleepTask.class",
      initParams) ); // Create a task container for each "sleep" task

     job.getRemaining().add( tc );
}
…
SubmissionReply jr = jobManagement.performJob( job, false );
```

The `Yacs.Job.Job` class defines the following fields that hold different list of tasks. These variables can be accessed using corresponding setter and getters.

`String name`	A name of this `Job`, defined by user.
`Vector<TaskContainer> remaining`	A vector of tasks (`TaskContainer` objects) to be executed. A YACS client (developed by the Client Programmer) should fill this list before submitting the job.
`Vector<TaskContainer> pending`	List used internally by the service while the Job is in progress. This should always be empty at the client site.
`Vector<TaskContainer> done`	List of Tasks which the service was able to execute.
`Vector<TaskContainer> failed`	List of Tasks which the service has failed to execute. For example, this list may include those tasks which were on failed workers and are non-repeatable.

3.2 Submitting Jobs and Getting Results

YACS API, namely the `yacs.frontend` package, includes some basic support for developing YACS client applications that can be used to create and submit jobs and to present results of execution to the End User. The Client Programmer can develop a GUI-controlled YACS client, and use the generic frontend class `yacs.frontend.FrontendImpl` for submitting jobs to YACS. The `FrontendImpl` class implements the `yacs.frontend.FrontendInterface` interface that includes two methods `submit` and `deleteJob`. See the YACS API specification at http://niche.sics.se for more details. Implementation of the `submit` method in is the `FrontendImpl` class is briefly described below.

`String submit( Job job )`	Submit a given Job to a YACS master. As implemented in the `FrontendImpl` class, the method finds a Master available to take on the job, binds to it, and hands the job over to the Master.

For the YACS service, the generic frontend `FrontendImpl` will appear as a client, and, during Job lifetime, it will be sent notifications on task changes, and (when all tasks complete) a Job result. If Client Programmer develops its own frontend rather than (re)using the `FrontendImpl` class, the external frontend wanting to be notified on Job and Task related events, can choose to implement the interface `yacs.frontend.FrontentClientInterface` which includes the following two methods:

`taskChange( TaskContainer task )`	This method is invoked on a Task change event. The `TaskContainer` object passed as a parameter to this method, contains task data in its fields, `resultCode, status, initParams` and so on (see 2.1)
`jobResult( Job result )`	This method is invoked when the service has finished with the Job and returned results of the Job execution. The Job object passed as a parameter to this method contains `TaskContainer` objects representing completed tasks with results stored in the fields `resultCode` (resulting code) and `initParams` (resulting state). These results can be examined by the client to determine how the job went, and can be presented to End User.

If Client Programmer prefers to launch a developed external frontend, then the `startFc` method of the `FrontendImpl` class is the recommended method to place the code for creation and launch of an external frontend, as well as set the `FrontendClientInterface` handle which will be called when the Frontend learns of `taskChange` or `jobResult`.

4 Emulators

In order to test and debug tasks (by Task Programmer) and YACS clients (by Client Programmer) without large-scale deployment of YACS, the YACS distribution includes an emulation package, `yacs.zemulation`, that allows the developer to submit and execute jobs locally without having to submit jobs into a deployed service.

The most notable class is `WorkerEmulator`, which emulates the behavior of a worker, i.e. it accepts a task, instantiates, deploys (initializes), and executes it. In particular, the `WorkerEmulator` class includes the following method to give the worker a task (represented by a `TaskContainer` object) to perform.

`performTask( TaskContainer task, boolean redeploy )`	Perform the given task. This method accepts a `TaskContainer` object representing the task and executes the given task.

Another notable class is `FrontendEmulator`, which emulates a frontend component. In the current implementation, this class contains code to launch the gMovie GUI, and an implementation of the `FrontendInterface`. This was used during development of the gMovie demonstrator application. The code can be used as an example of a frontend. The `Emulate` class has been used during YACS development as an entry point to other emulation classes and also contains some useful examples.

5 Use Case: The gMovie Demonstrator Application

The gMovie demonstrator application is an application on top of YACS that has been used to demonstrate the functionality of YACS, i.e. execution of bag of tasks written in an arbitrary language. The gMovie application performs transcoding of a given movie from one format to another. Transcoding is a CPU intensive and time consuming job, therefore YACS was use to perform this job in parallel by several distributed workers transcoding different parts of the movie in order to improve the total transcoding time.

The gMovie application is included the YACS distribution, and it consists of two parts:

1. The GUI, `yacs.frontend.gmovie.GMGui`, shown in Figure 3. The GUI allows the user to select a movie to be transcoded, define transcoding options, and submit the movie to the gMovie frontend.
2. The Task subclass `yacs.job.tasks.MovieTranscodingDirectedTask` which invokes the VLC program to perform the actual transcoding.

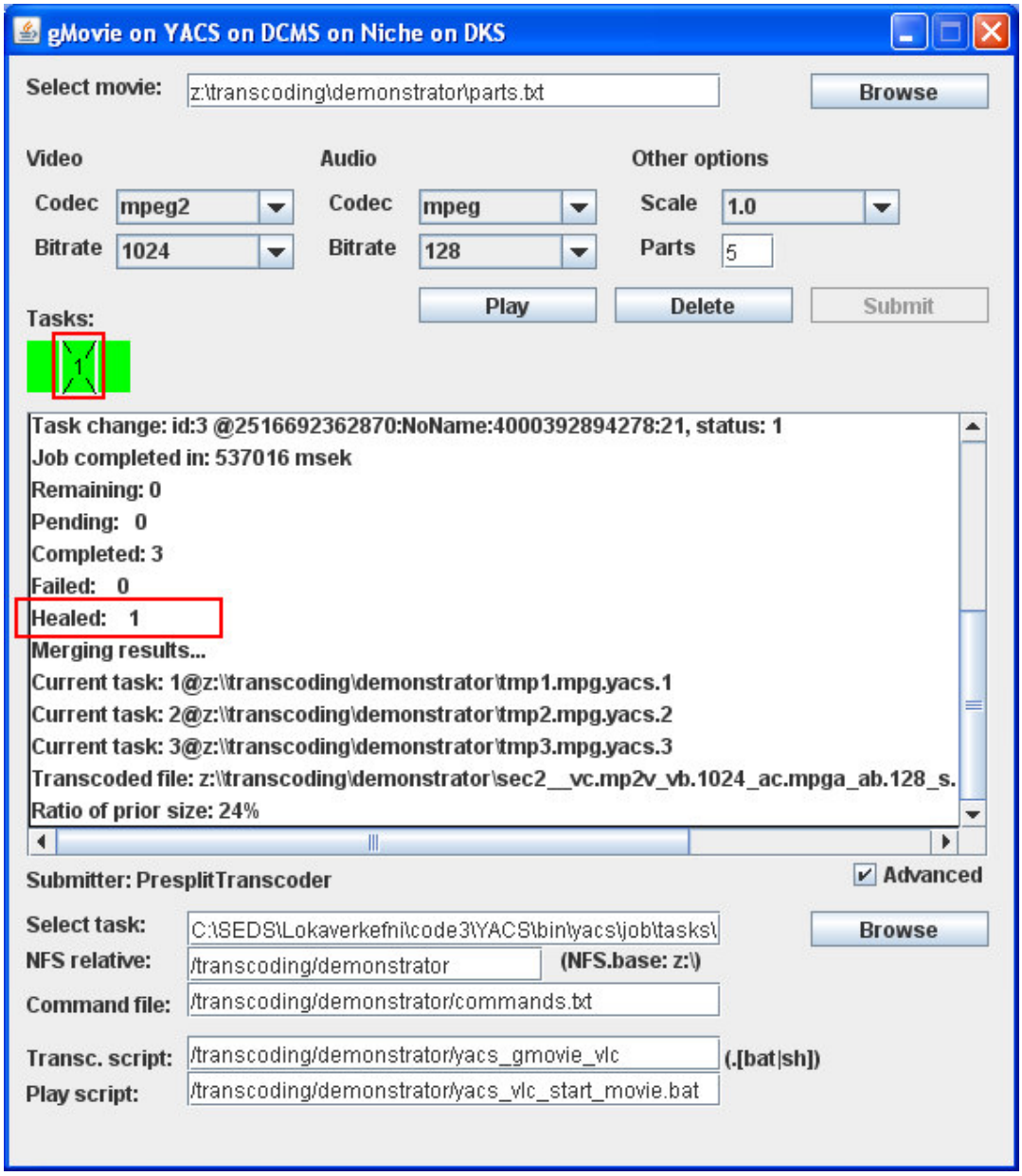


Figure 3. The GUI of the gMovie application

When the application starts, the client selects a file using the file chooser in the gMovie GUI (see Figure 3) which lists pre-split movie parts, chooses transcoding options, and submits the movie to the YACS frontend by clicking on the Submit button. On submit, the frontend creates a Job that contains a number of tasks (one for each movie part) of the `MovieTranscodingDirectedTask` class, the `initParam` array, which holds locations of movie parts (the application assumes a global file system accessible to all workers), transcoding options and the path where the transcoded result should be placed. This job is submitted to the YACS frontend, which, in its turn, submits it to a Master component. During the lifetime of the Job, taskChange notifications are sent to the Frontend which forwards them to the gMovie GUI. The same applies for the end result except than the gMovie GUI combines all the transcoded parts into one file and shows it to the user.

This page is intentionally left blank

Annex 10. Collaborative Network Simulation Environment

1. - Main Screen

The CNSE client includes a Graphical User Interface to interact with the services and provide the functionality. Once loaded, the client shows the *main screen* (Figure 1) with the following items:

***Menu**:*

- *File*. It manages the input files that can be loaded in the CNSE. The input file must be a .zip file with an XML manifest indicating the structure of single and parameter-sweep simulations as well as the expected output filenames, the *.tcl* file for the *ns-2* and some additional files as an option. There are some examples of input packages and XML manifests in the CNSE installation package. The *menu items* are: "*Open simulation package*", "*Save simulation package*" and "*Exit*".
- *Edit*. It manages the simulations allowing their creation and elimination. The CNSE supports two kinds of simulations: *single simulations* and *parameter-sweep simulations*. So, the *menu items* are: *"Add single simulation", "Add parameter-sweep simulation"*, and *"remove simulation"*
- *Tools.* It contains different tools that can be useful for the users. There is currently just one tool, the *visualization manager* that manages the visualization that has been already launched. So, there is one *menu item:* "*Visualization manager*".

Text areas:

- *Simulation Package Information.* Once a package is loaded, its main information appears in this text area (Figure 2), such as a description of the package, the *.tcl* and additional filenames, a description of the input parameters, and the expected output files. The output files can be *multi-line* (with the evolution of the output parameters through time) or *single-line* (with the average output values).
- *Single simulations.* It shows a list with the single simulations that has been created for this package.
- *Parameter-sweep simulations*. It shows a list with the parameter-sweep simulations that has been created for this package.
- *Simulation information*. It shows a description of the selected simulation, including its title, identifier, status (completed or not) and input parameters.

Buttons

- *Launch simulations.* It launches a single or parameter-sweep simulation. This button is available if the selected simulation has not been launched before.
- *Show output values*. It shows the output values for a set of input values. This button is available if the selected simulation has been launched and completed.
- *Perform statistical analysis*. It shows a statistical analysis with the influence of the input parameters on the output results. This button is available only for completed parameter-sweep simulations whose input parameters are described as sets of two values (*2krfactorial*).

- *Launch nam visualization.* It launches a *nam* visualization. This button is available only for single simulations that generate an output *nam* file.
- *Launch x-y visualization*. It launches an *xgraph* visualization. This button is available for every completed simulation (single or parameter-sweep). For a single simulation it represents the evolution of an output parameter through time (a *multi-line output file*). For a parameter-sweep simulation it represents the values of an output parameter while an input parameter is swept (a set of *single-line files* is needed).

2. - Single Simulations

A single simulation can be created with the *menu item* "*add a single simulation*". The following information must be provided (Figure 3):

- *Identifier*. It represents the single simulation in the *main screen*.
- *Title.* It can be used to describe the single simulation
- *Trace file expected*. A *nam* output file is expected when ticked.
- *Tabbed file expected.* An output file with information for the x-y visualization is expected when ticked.
- *Parameter values.* Input parameter values for the single simulation.

Once created, the single simulation identifier is shown in the *main screen*. Additionally, when this simulation is selected, its information appears in the *simulation information* text area and the *launch* button is available (Figure 4). The identifier changes the way it looks with the status of the simulation: *normal* (not launched), *italics* (launched but not completed) and *bold* (completed).

After launching and completing the single simulation some buttons become available while the *launch simulation* button appears unavailable. They are the *show output values* button, the *launch nam visualization* button and the *launch x-y visualization* button.

- *Show output values*. It is available when the option *tabbed file expected* was ticked and only for single-line files. So, the single-line file must be selected first (Figure 5), and then the output values obtained from the simulation results are shown in a pop-up text area (Figure 6).
- *Launch nam visualization*. It is available when the option *trace file expected* was ticked. The user must select a *trace file* (Figure 7), and then a VNC containing the *nam* visualization is popped-up (Figure 8).
- *Launch x*-y visualization. It is available when the option *trace file expected* was ticked and only for multi-line files in a single simulation. The user can select the output parameter that wants to represent in the *x-graph* visualization. Once again a VNC containing the *x-graph* is popped-up (Figure 9).

3. - Parameter-Sweep Simulations

A parameter-sweep simulation can be created with the *menu item* "*add a parameter-sweep simulation*". The following information must be provided (Figure 10):

- *Identifier*. It represents the parameter-sweep simulation in the *main screen*.
- *Title*. It can be used to describe the parameter-sweep simulation
- *Repetitions*. It defines how many times the parameter-sweep simulation must be done. This field is useful to detect variation in the output results, and is supported for the statistical analysis.
- *Design type*. A "*2krfactorial"* design type must include a set of two values for every input parameter and allows the statistical analysis. An "*other"* design type can include any number of values for each input parameter.
- *Parameter values*. Input parameter values for a parameter-sweep simulation. These values can be added as a set of values (*[value 1 value 2 value n]*) or as a range of values (*[init_value:step:end_value]*).

Once created, the parameter-sweep simulation identifier is shown in the *main screen*. When this simulation is selected its information appears in the *simulation information* text area and the *launch* button becomes available. After launching a parameter-sweep simulation there is some awareness information to report the user about the completion status (Figure 11).

After completing the parameter-sweep simulation the *launch simulation* button becomes unavailable, while the *show output values* button, the *launch x-y visualization* button and the *perform statistical analysis* button may appear available.

- *Show output values*. It is always available for completed parameter-sweep simulations but only for single-line files. So, a single-line file must be selected first, and then the user must fix the values of the input parameters (Figure 12) to get the output values in a pop-up text area (Figure 13).
- *Launch x*-y visualization. It is always available for completed parameter-sweep simulations but only for single-line files. The user must select the single-line file as well as the input parameter and the output parameter. Then, a VNC containing the *x-graph* visualization is popped-up (Figure 14).
- *Perform statistical analysis*. It is available for *2krfactorial* parameter-sweep simulations (Figure 15), but only for single-line files. The user must select the single-line file and the output parameter for the statistical analysis, and a popped-up text area will show the results (Figure 16).

4. - Visualization Manager

The visualization manager can be used to manage the launched visualizations. It shows all the single and parameter-sweep visualizations (Figure 17). The name of the visualizations is a combination of the simulation identifier, the output file that was selected and additionally the output parameters. The user can *join* to a visualization that has been launched, or *stop* it. These visualizations are stopped automatically when a new package is loaded.

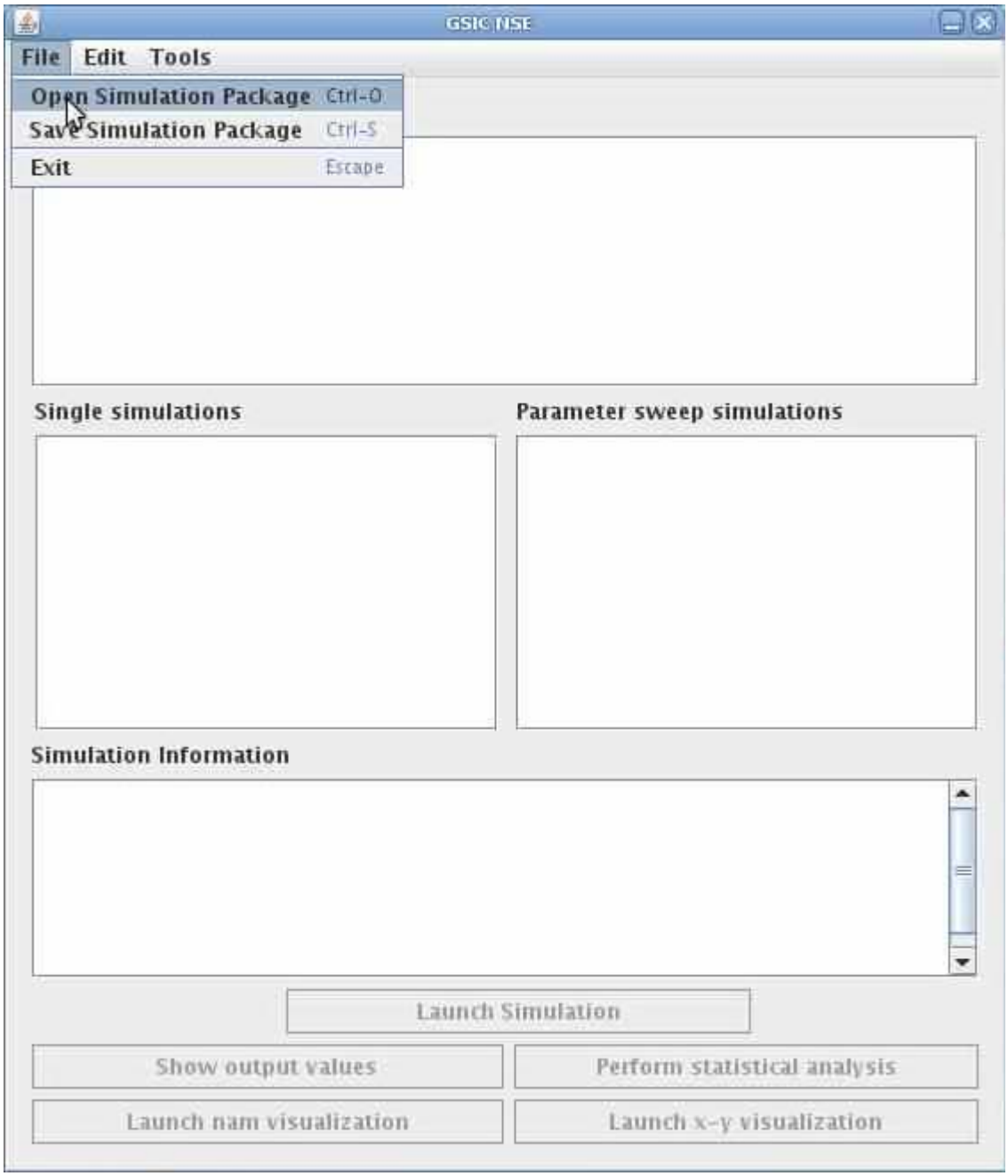


Figure 1 Main Screen. There is no package loaded yet.

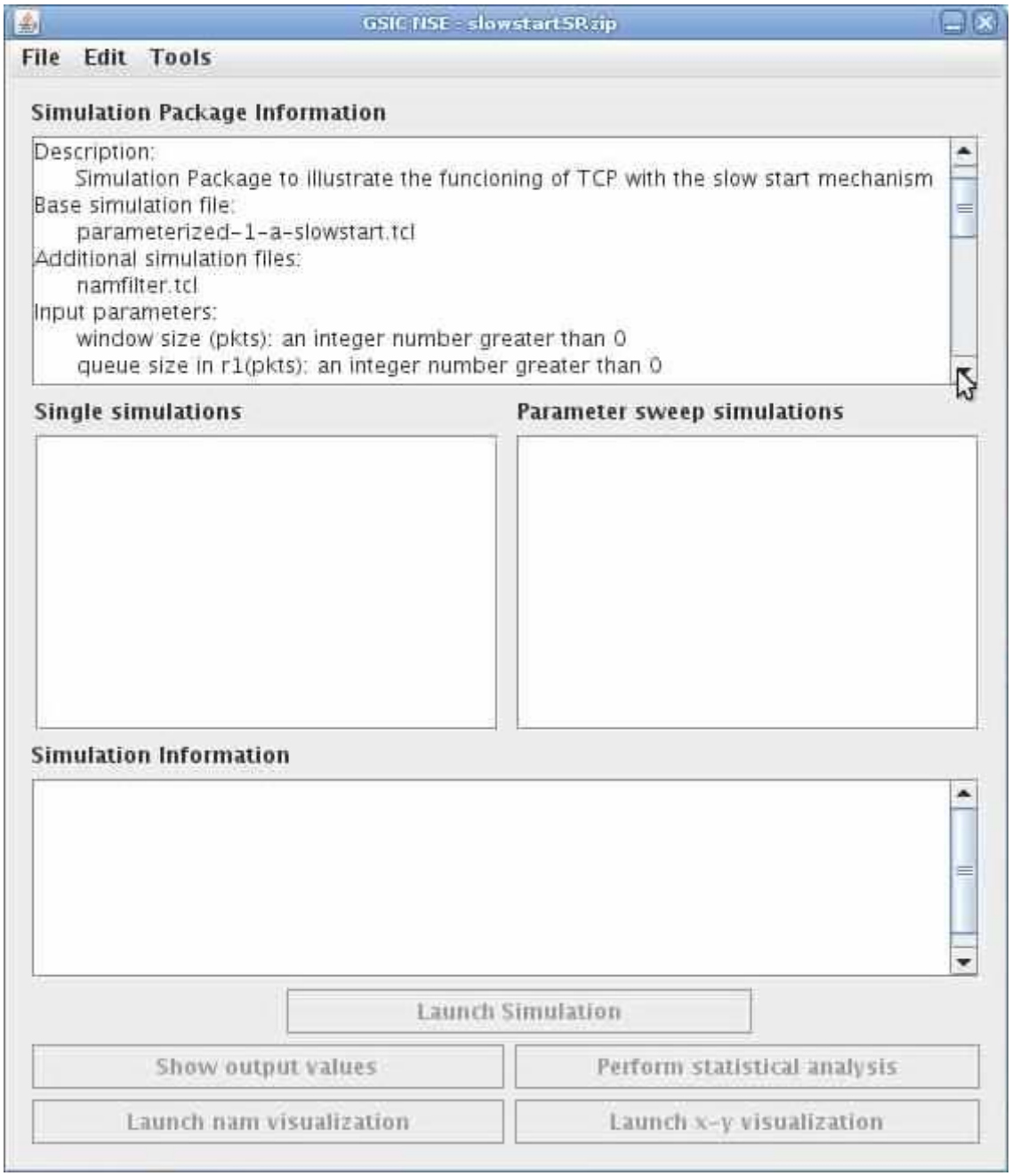


Figure 2 Package loaded. Package description. There are no simulations created yet.

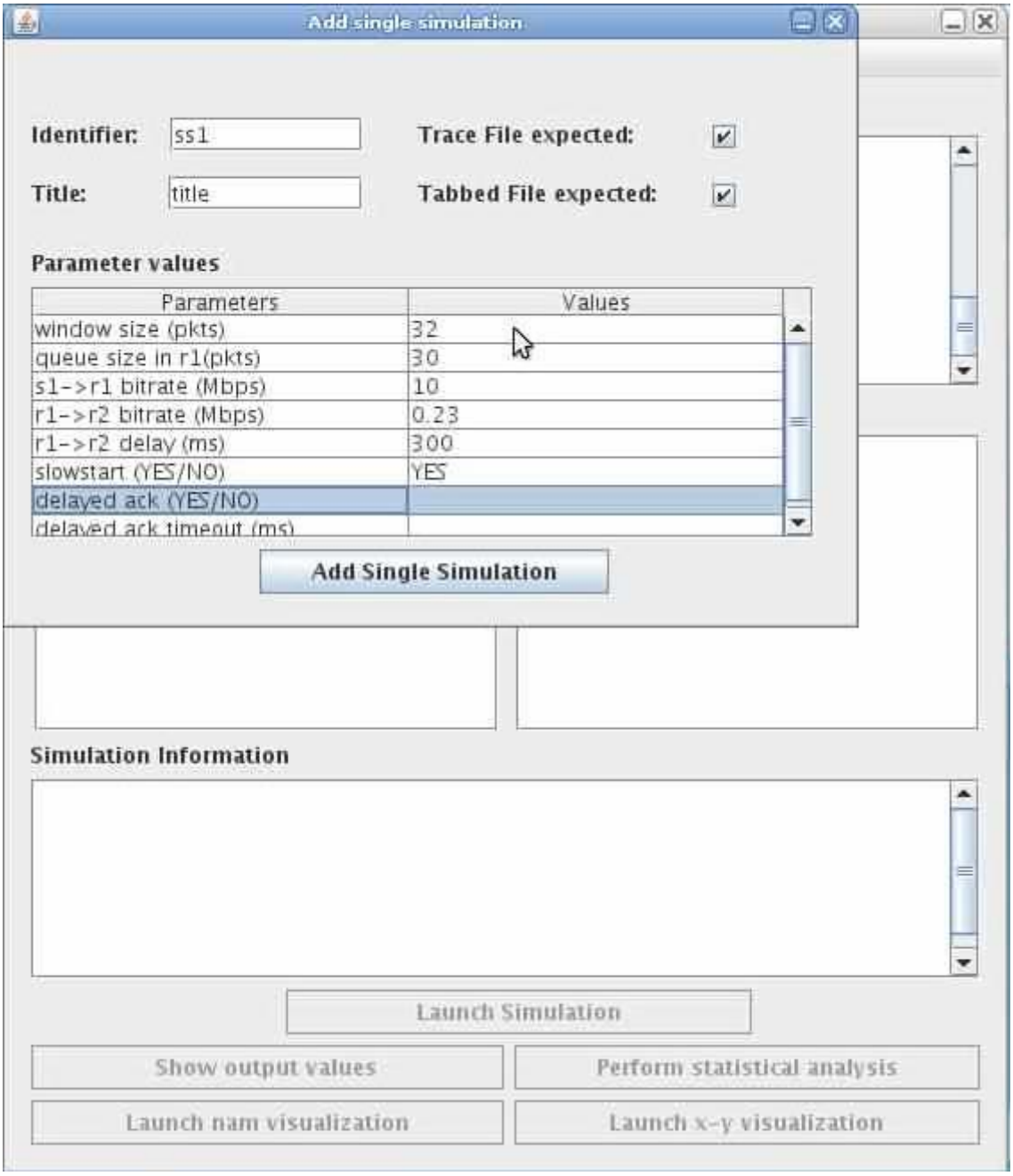


Figure 3 Add single simulation. Input values

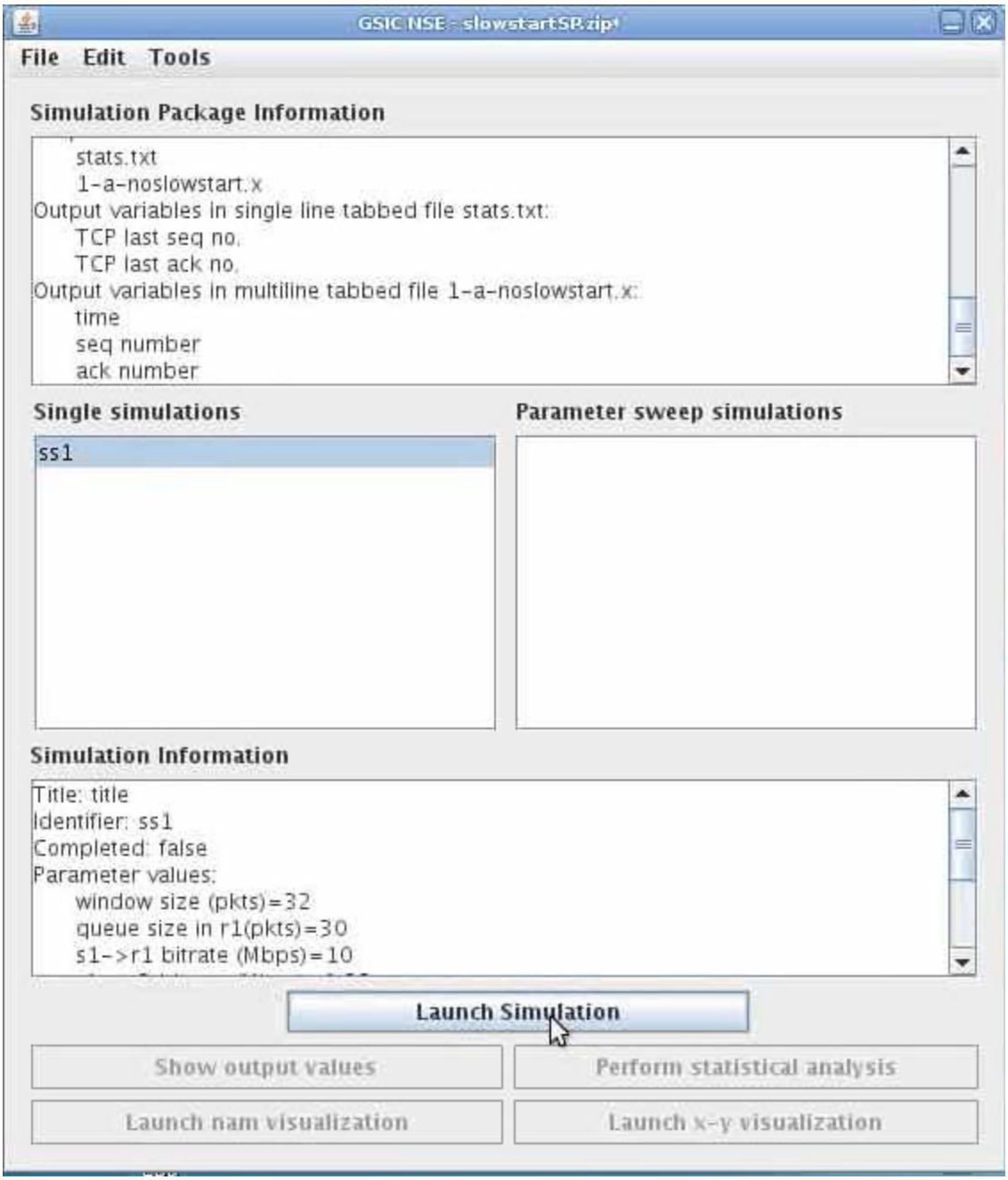


Figure 4 Single simulation. Description

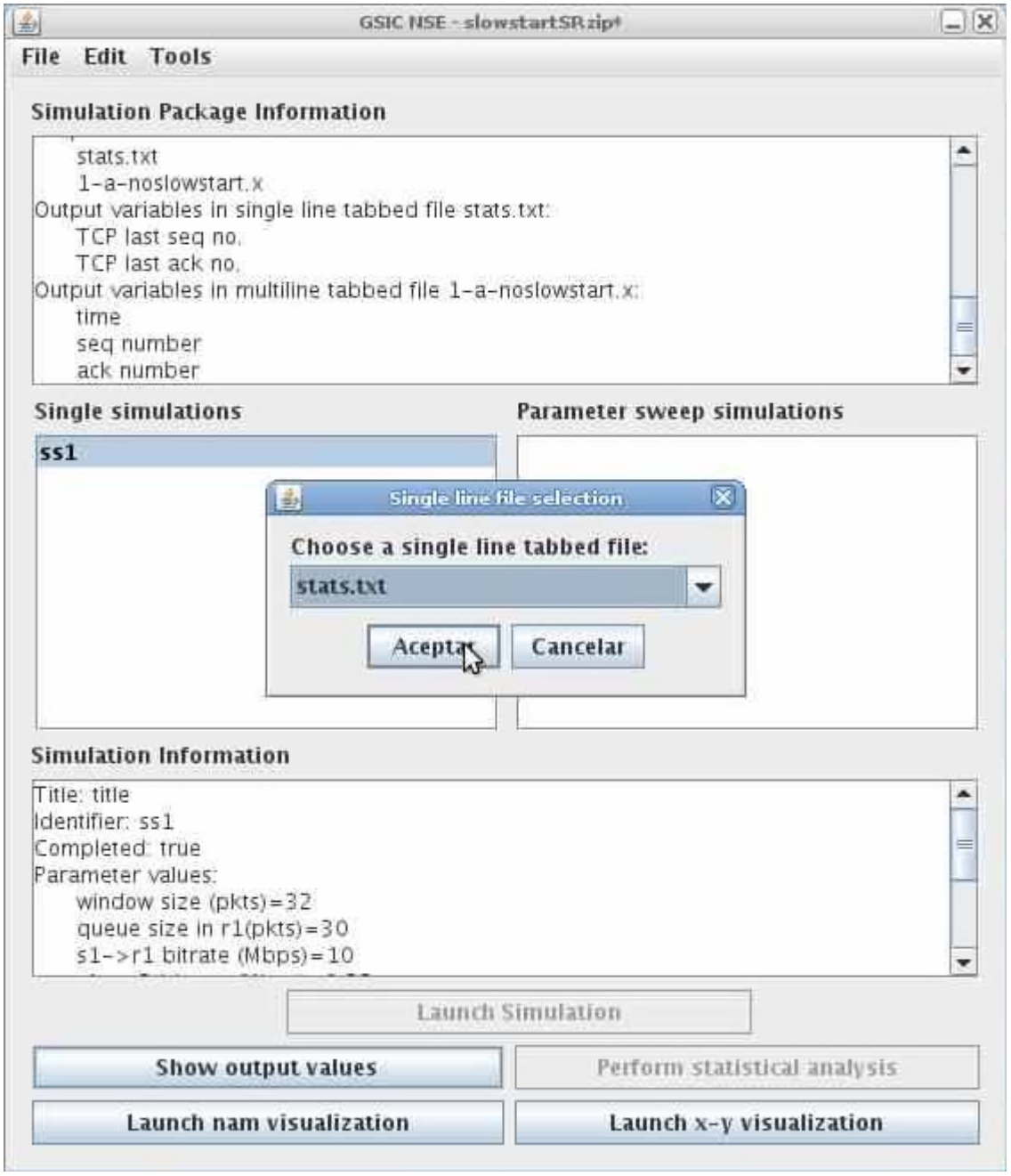


Figure 5 Single simulation completed. Show output values

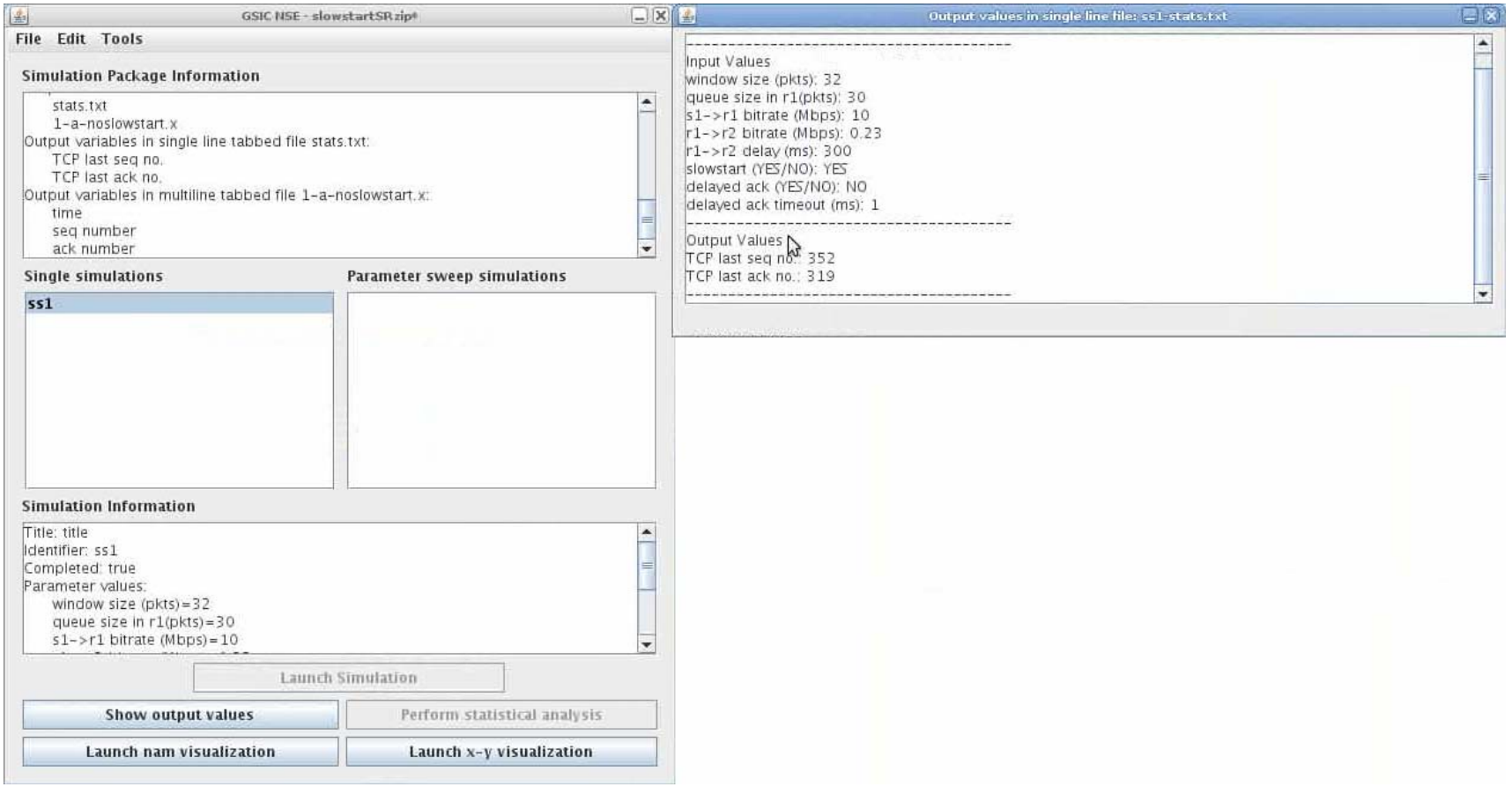


Figure 6 Single simulation completed. Output values

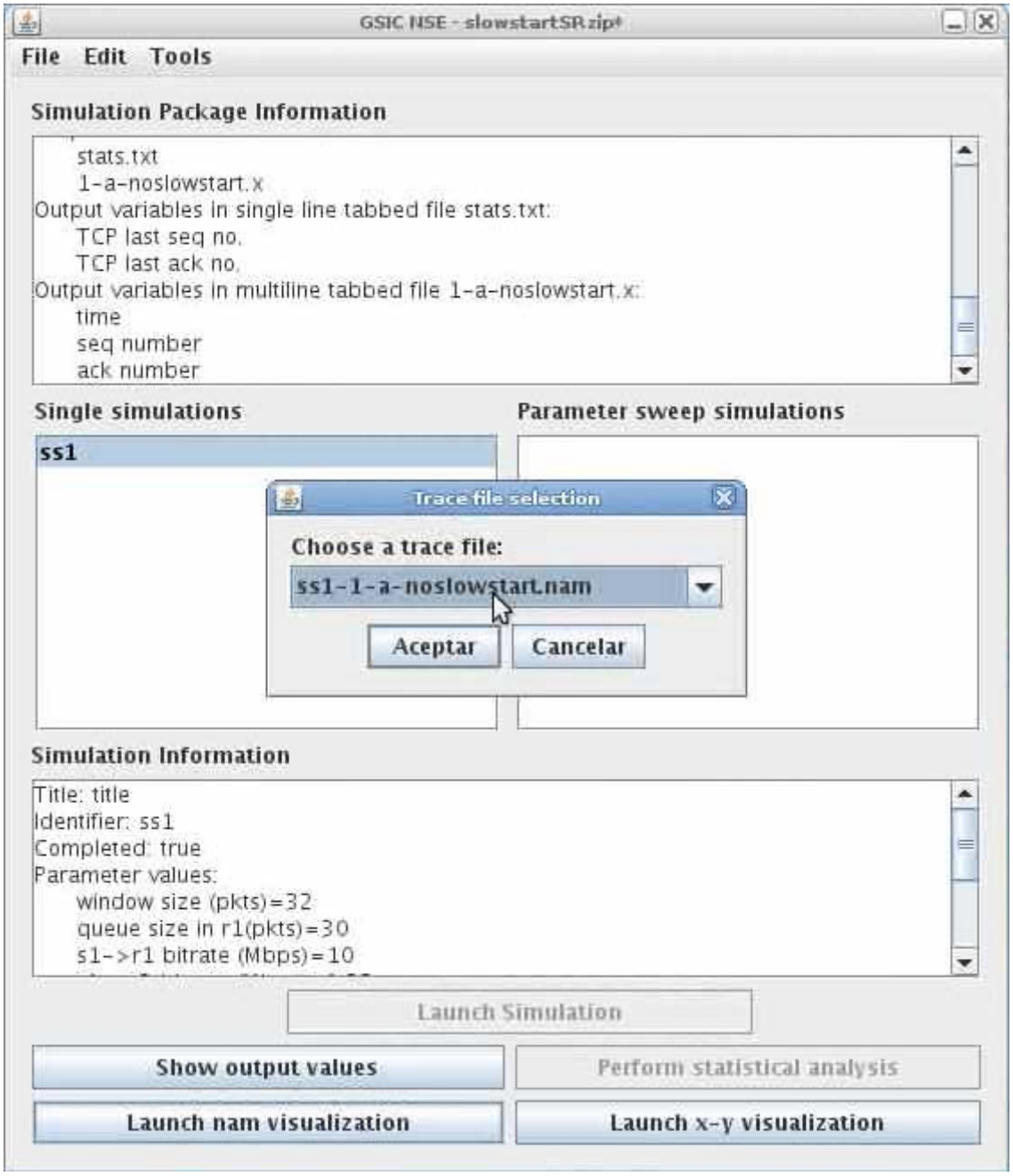


Figure 7 Single simulation completed. Show Nam visualization

Figure 8 Single simulation completed. Nam visualization

Figure 9 Single simulation completed. x-y visualization

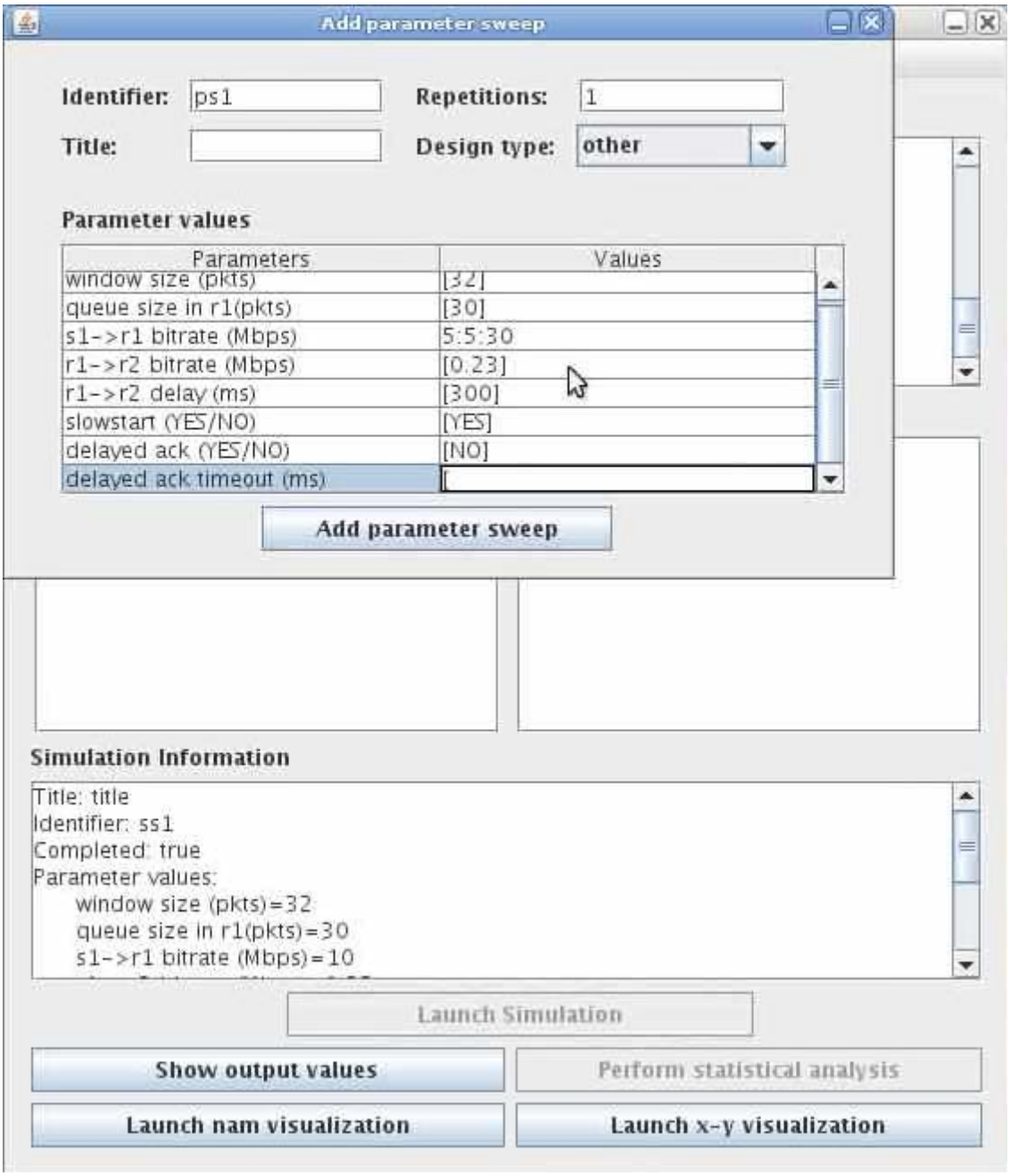


Figure 10 Add parameter-sweep simulation. Input values

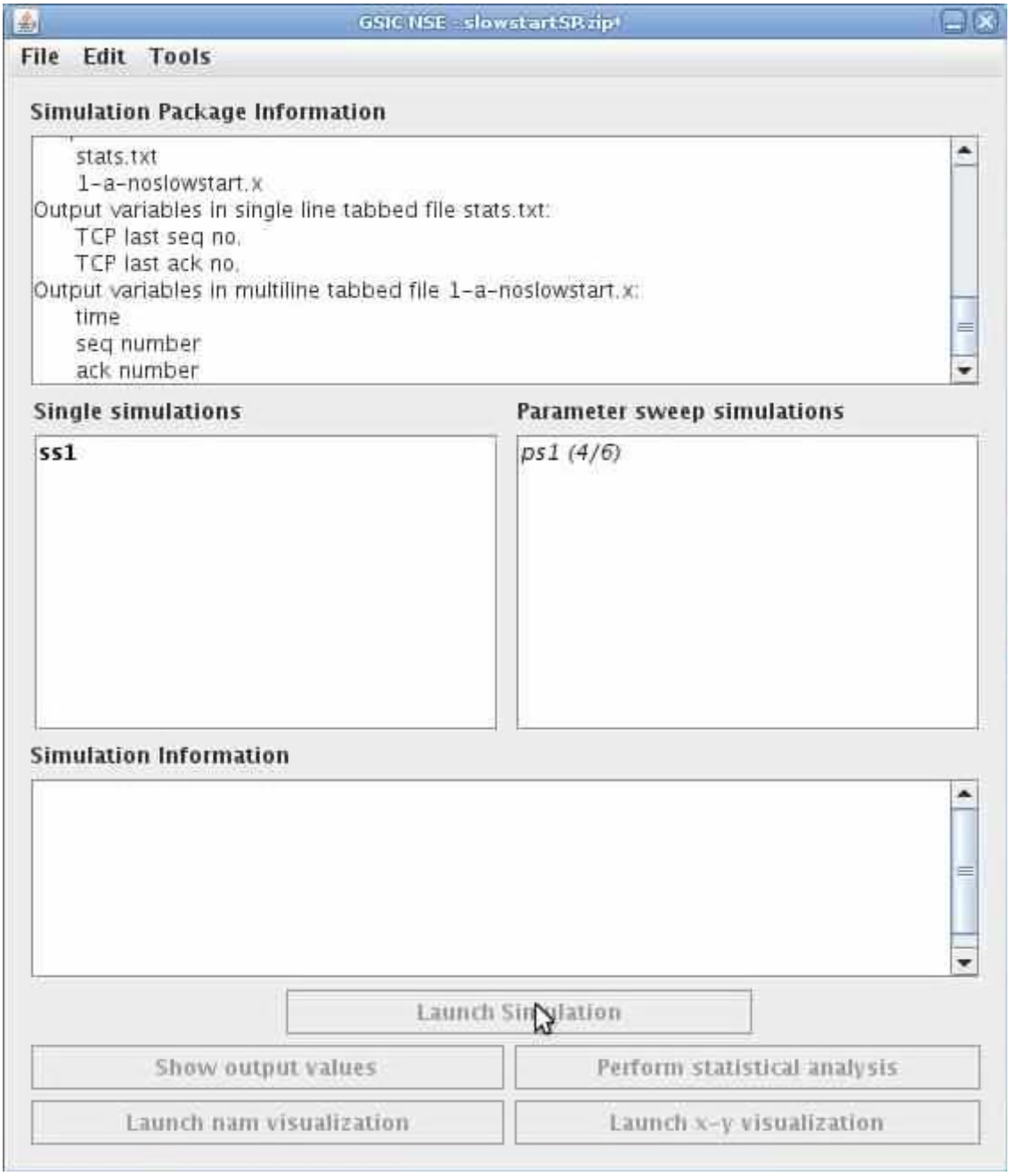


Figure 11 Parameter-sweep simulation launched

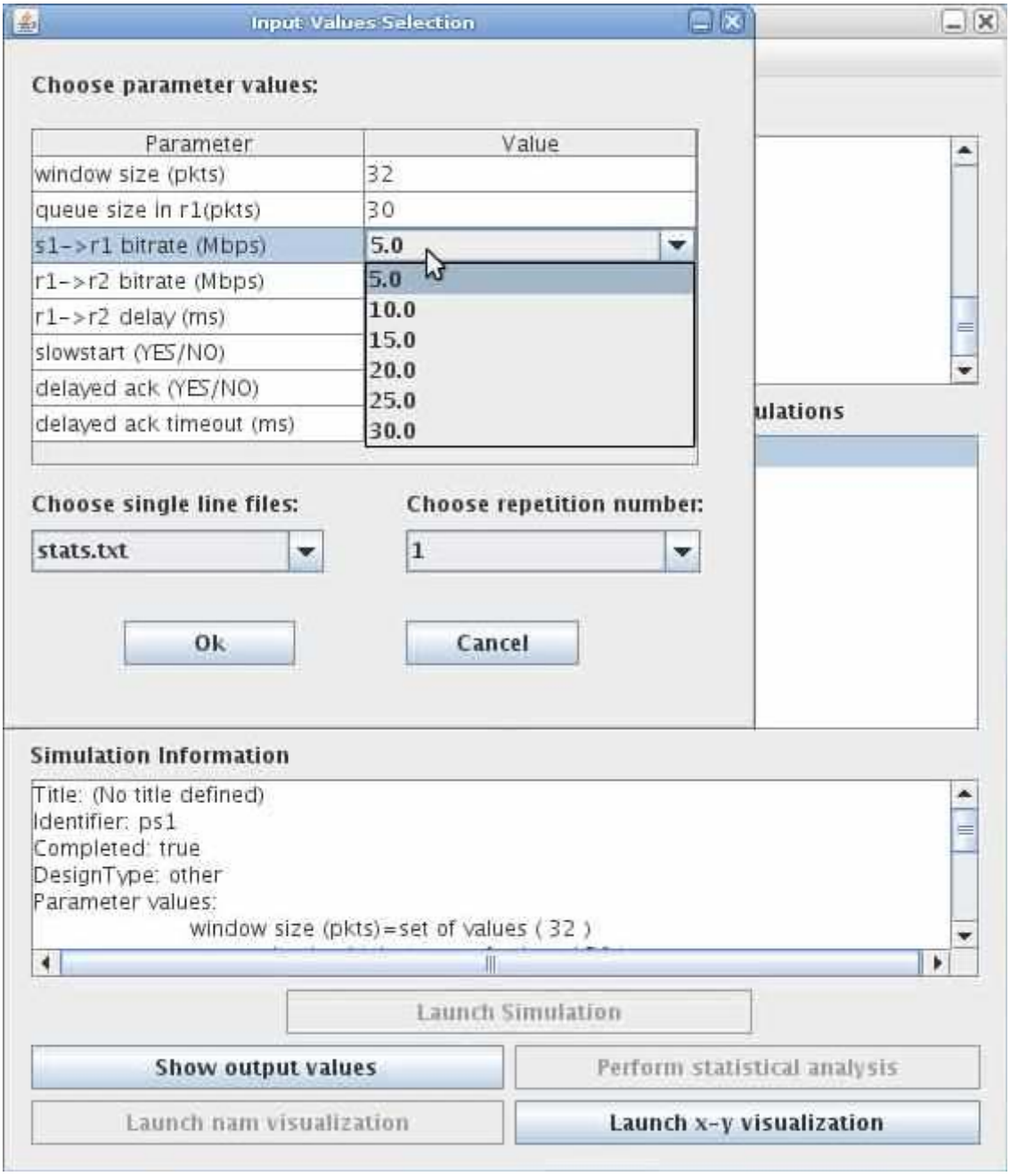


Figure 12 Parameter-sweep simulation completed. Show output values

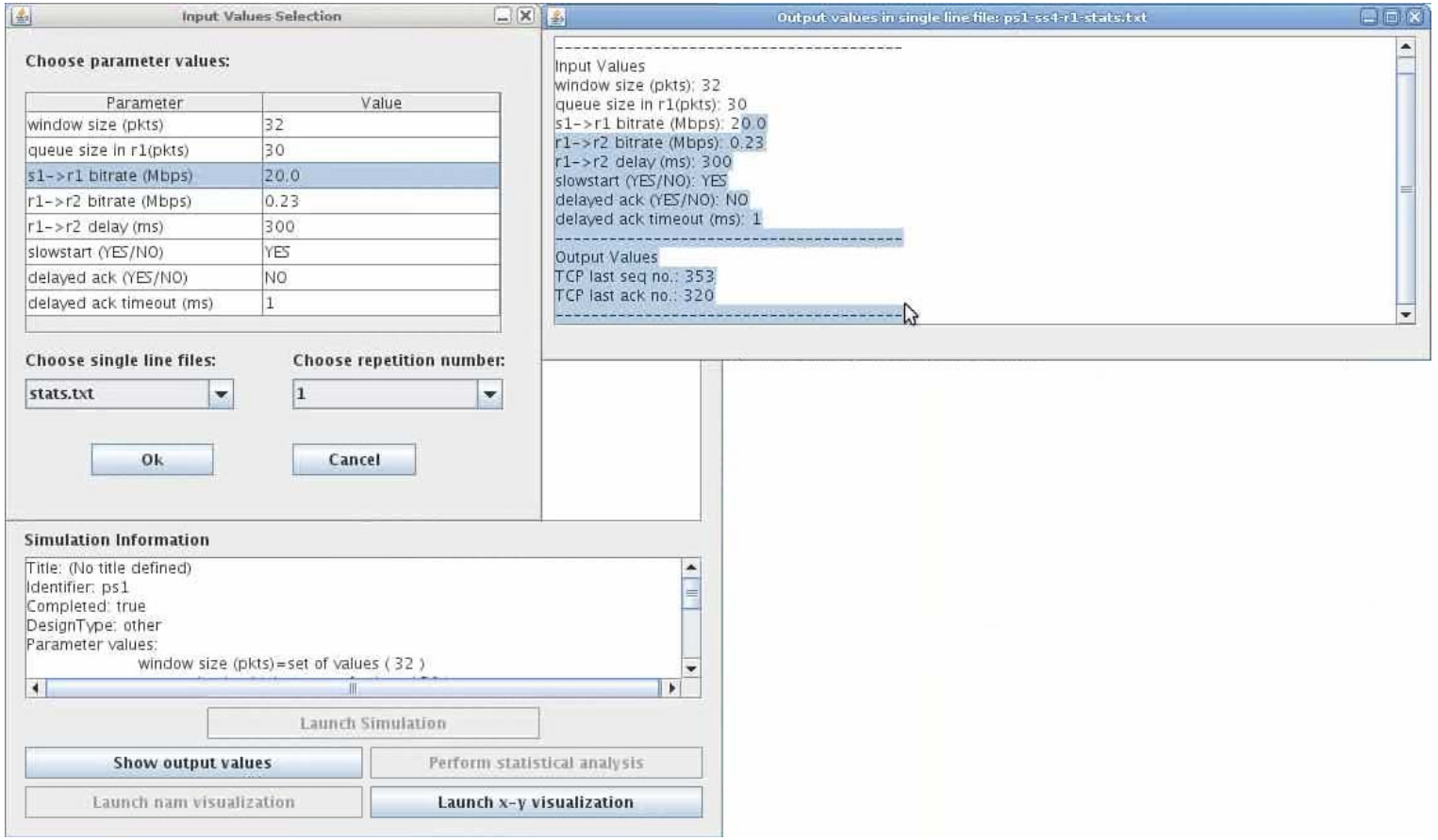


Figure 13 Parameter-sweep simulation completed. Output values

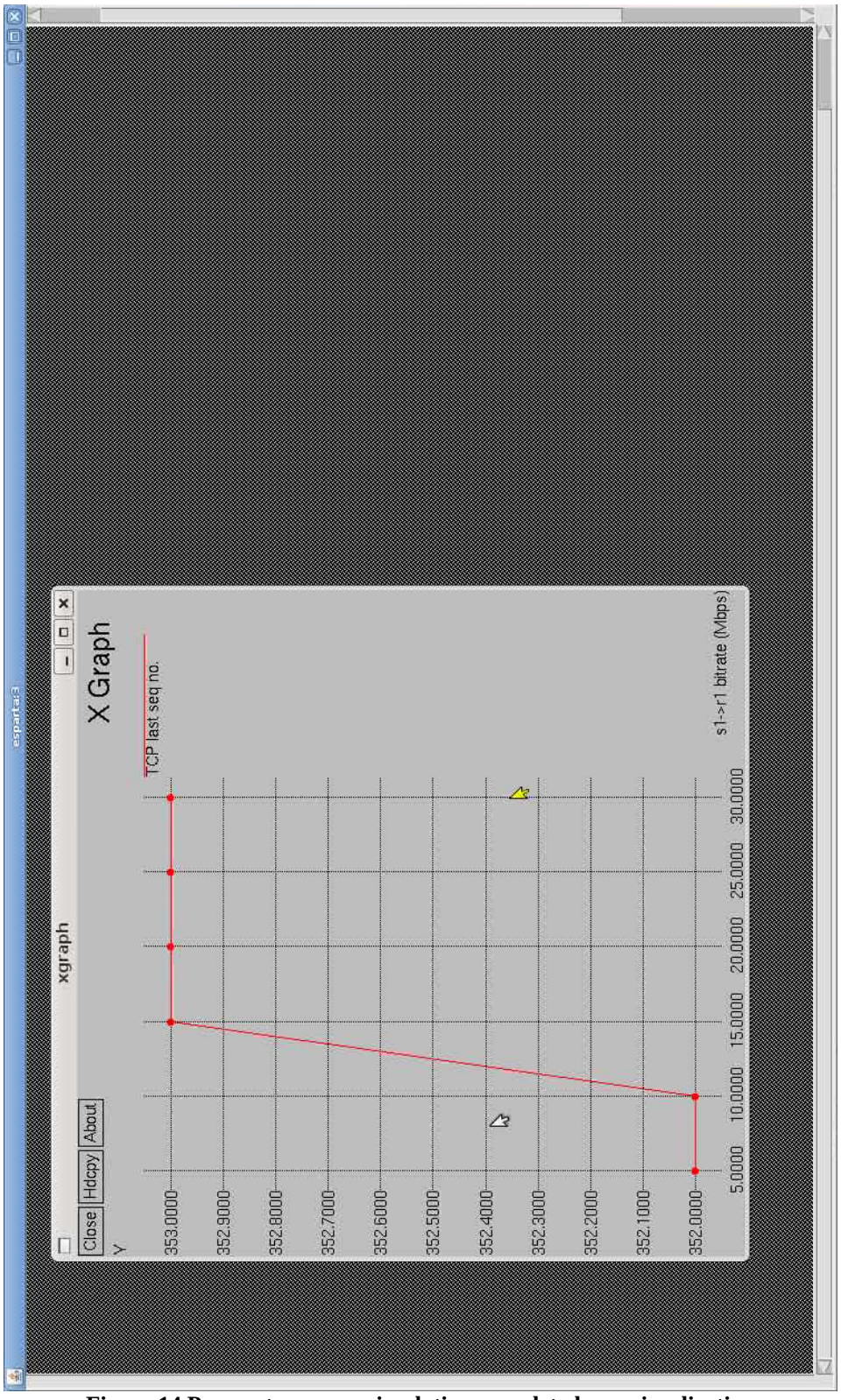


Figure 14 Parameter-sweep simulation completed. x-y visualization

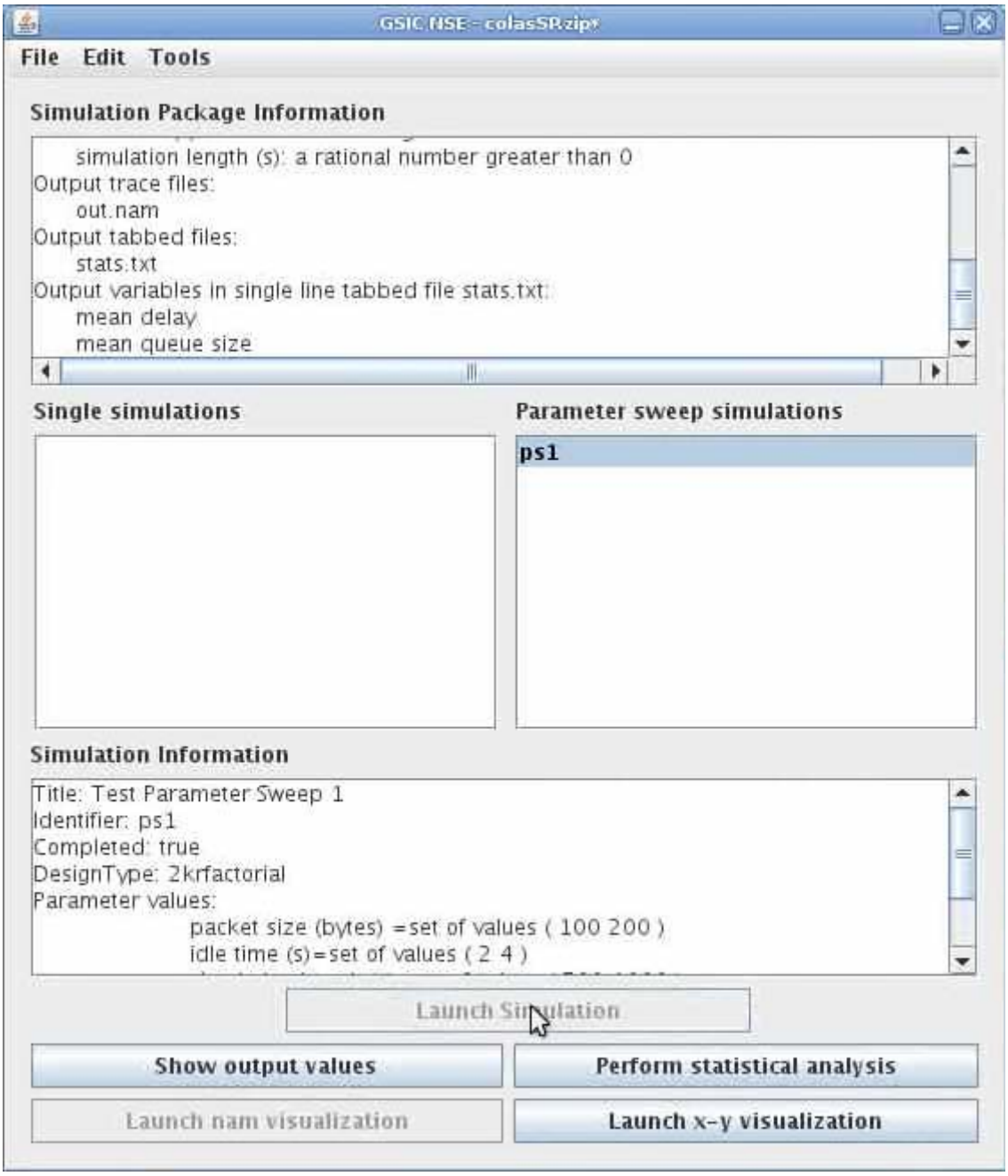


Figure 15 Parameter-sweep simulation completed (2krfactorial) .

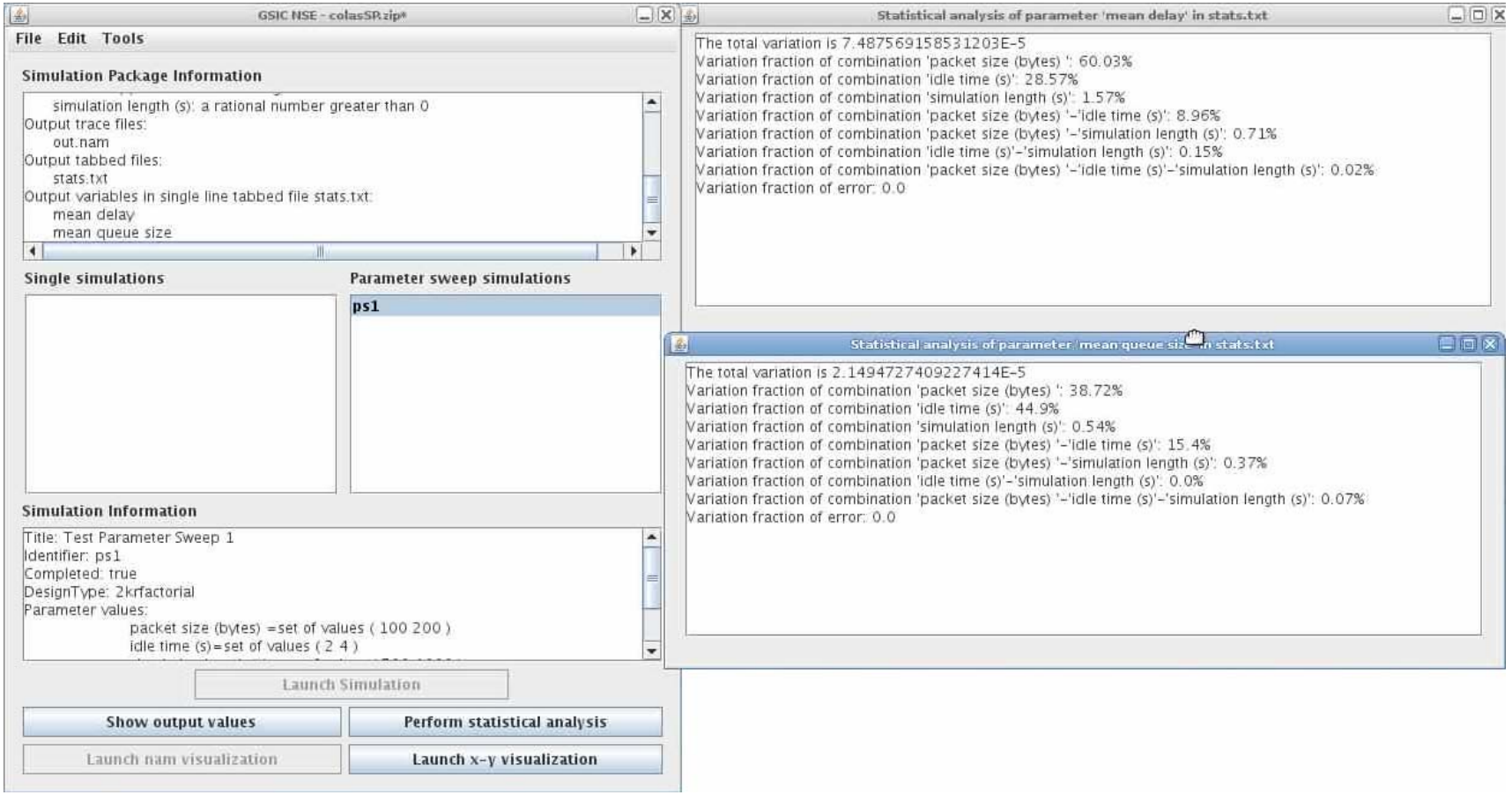


Figure 16 Parameter-sweep simulation completed (2krfactorial). Statistical analysis

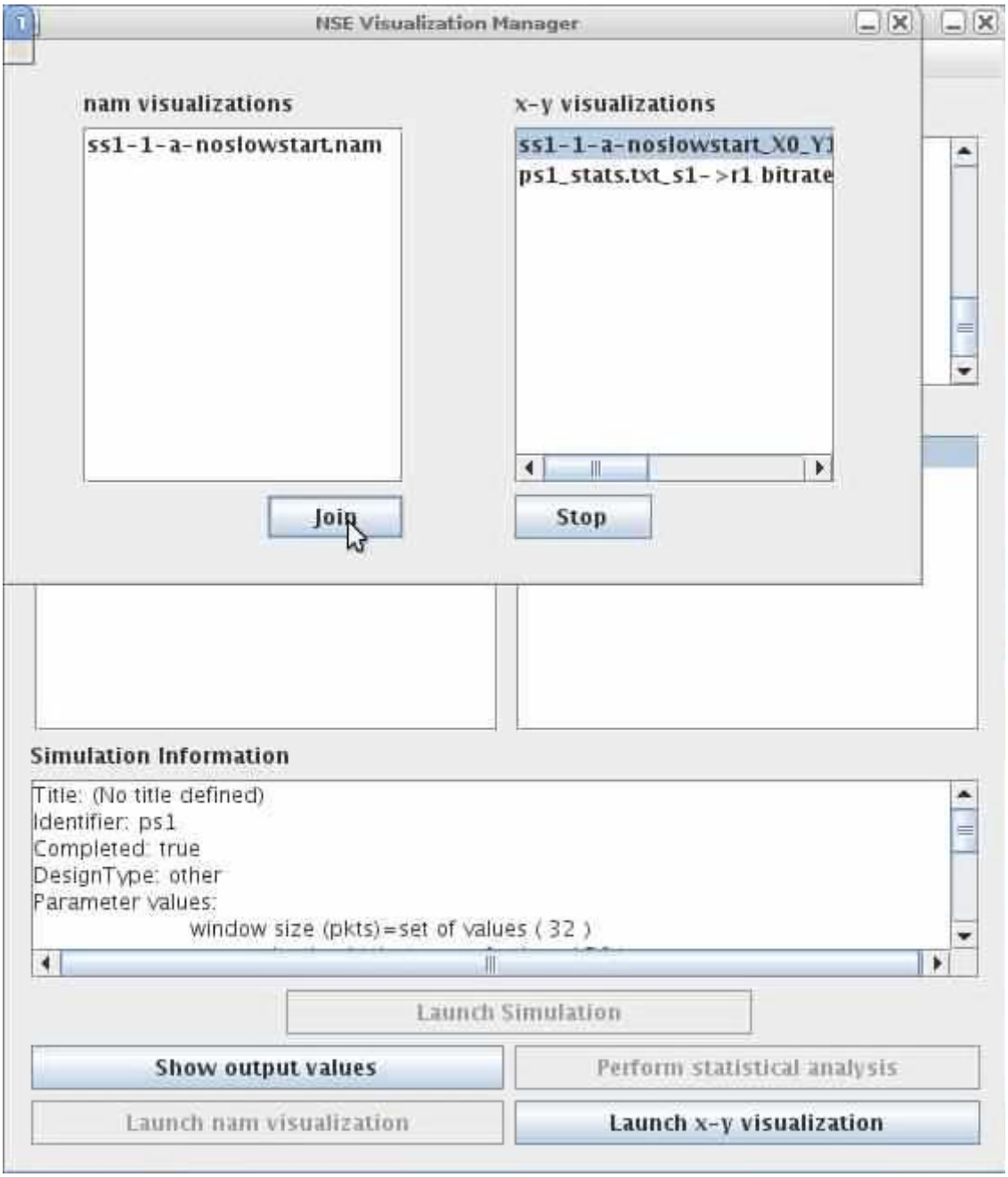


Figure 17 Visualization Manager

This page is intentionally left blank

Annex 11: Collaborative File Sharing

USER MANUAL: CFS

CFS allows a group of users share files and forums organized in folders, following the workspaces metaphor. Figure 0 shows a typical view of a CFS workspace. Left frame presents the structure of workspaces. Right frame presents the content of current folder, file or forum (in this example, the content of root folder).

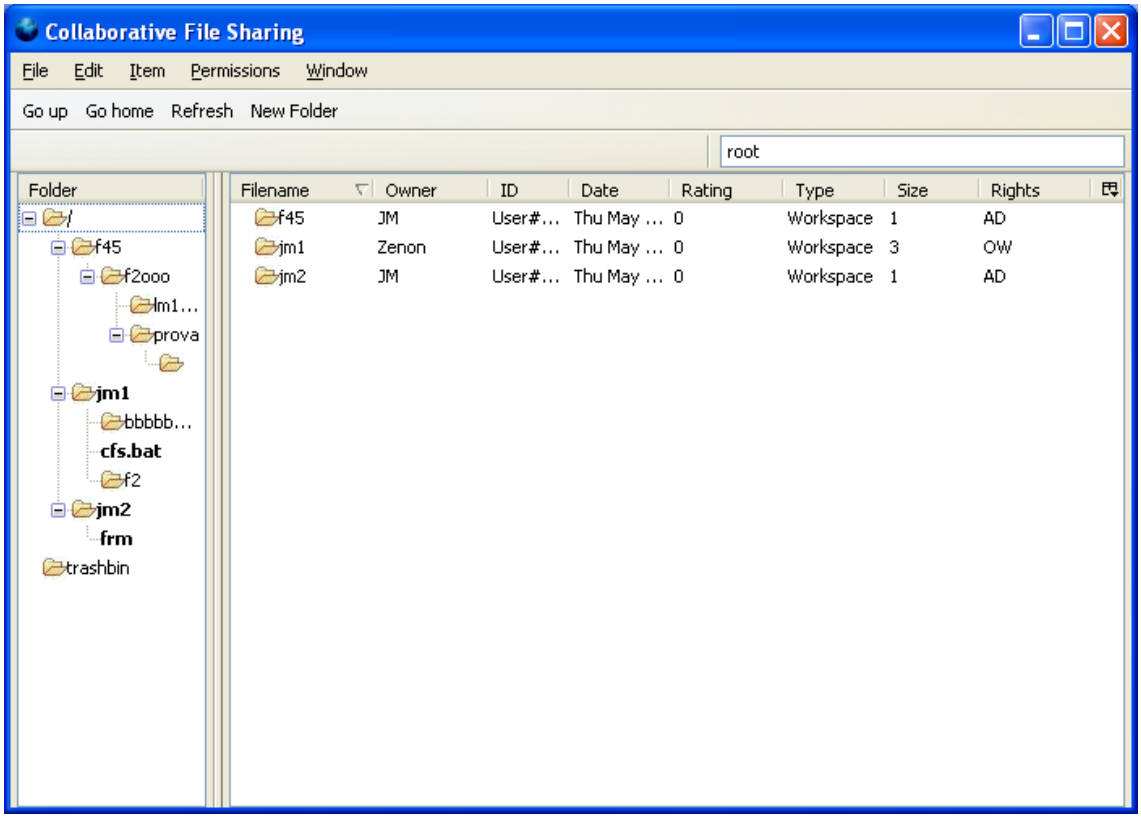
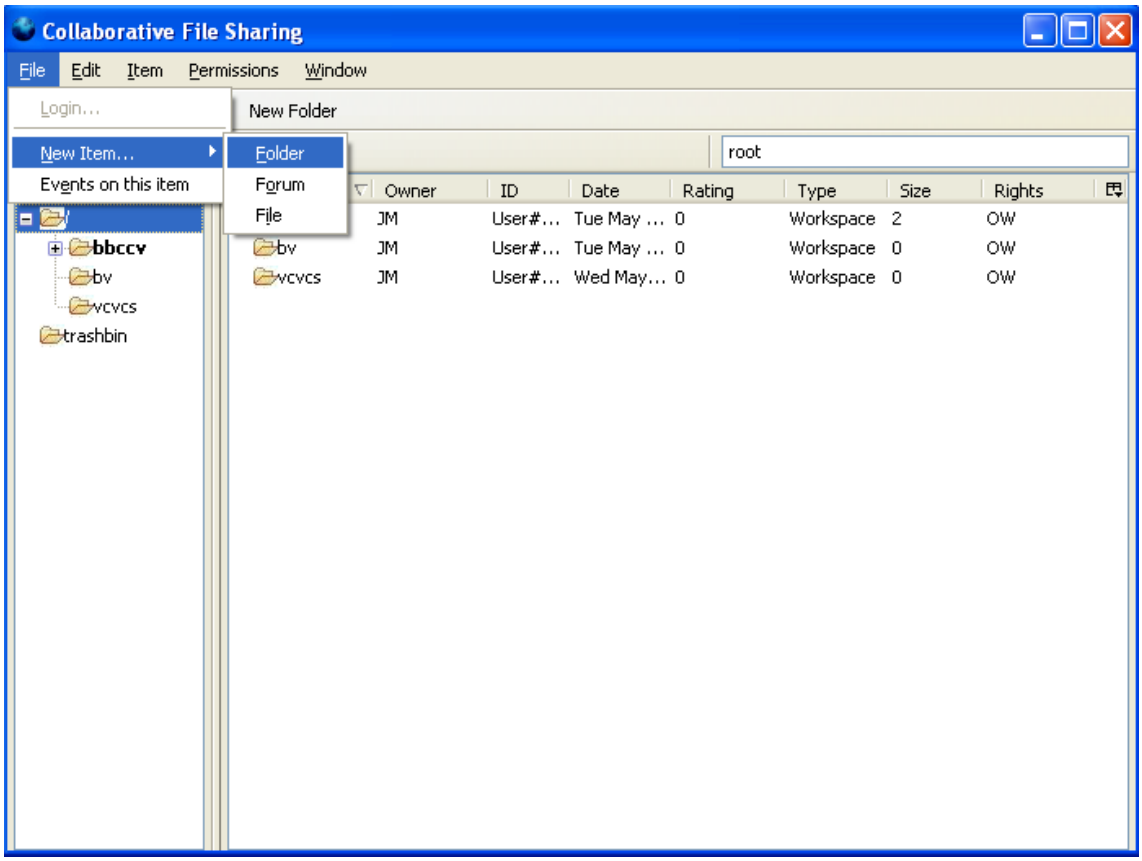


Figure 0. Example of a CFS group

1. Creation of Items: Folders, files and forums

Choose **File -- New Item …** option in application menu to create an item. Items may be: Folder, Forum or File.



New Items are created in current window and with Full access for all workspace members. Users can select a more restrictive access in **Predefined Level** selectable list (figure 1).

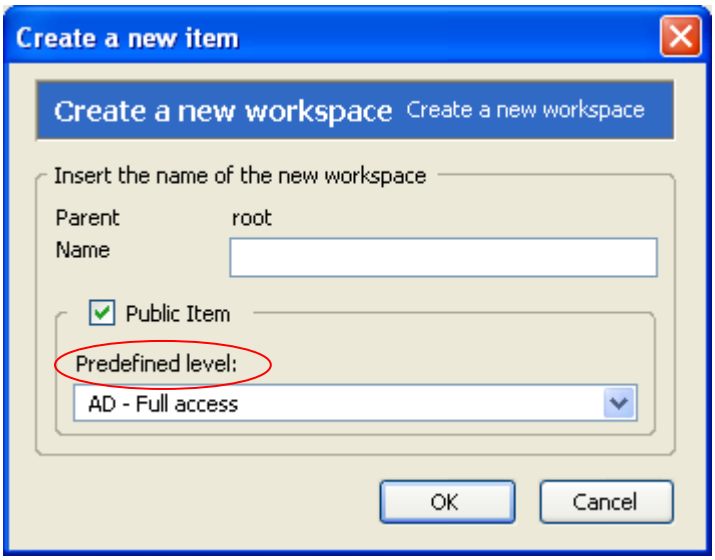


Figure 1. Creating an item. Example of creation of a workspace.

2. Forums

Figure 2.1 shows the content of forum **frm**. It contains two threads. Each thread has several threads.

New Thread button creates a new thread.

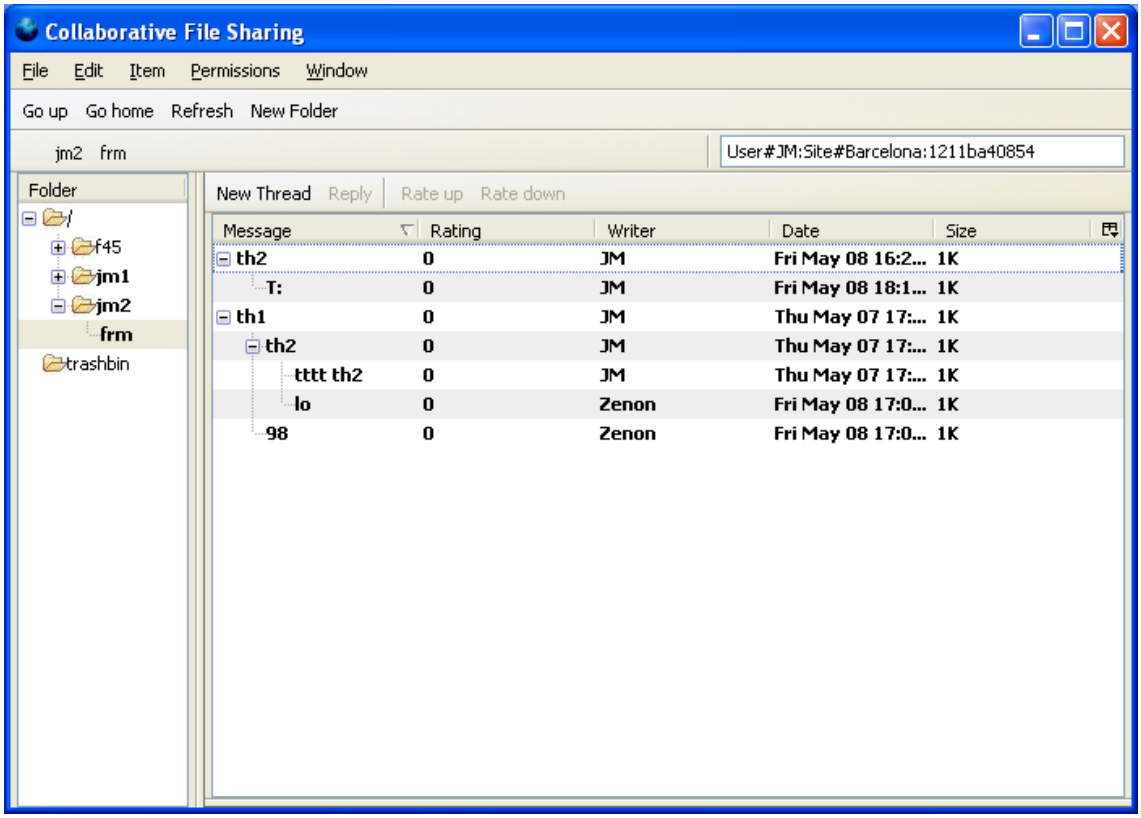


Figure 2.1. Forum frm.

When reading a forum entry, the message is highlighted in yellow in the threads frame (figure 2.2). **Reply** allows replying the message. **New Thread** creates a new thread.

Messages cannot be deleted.

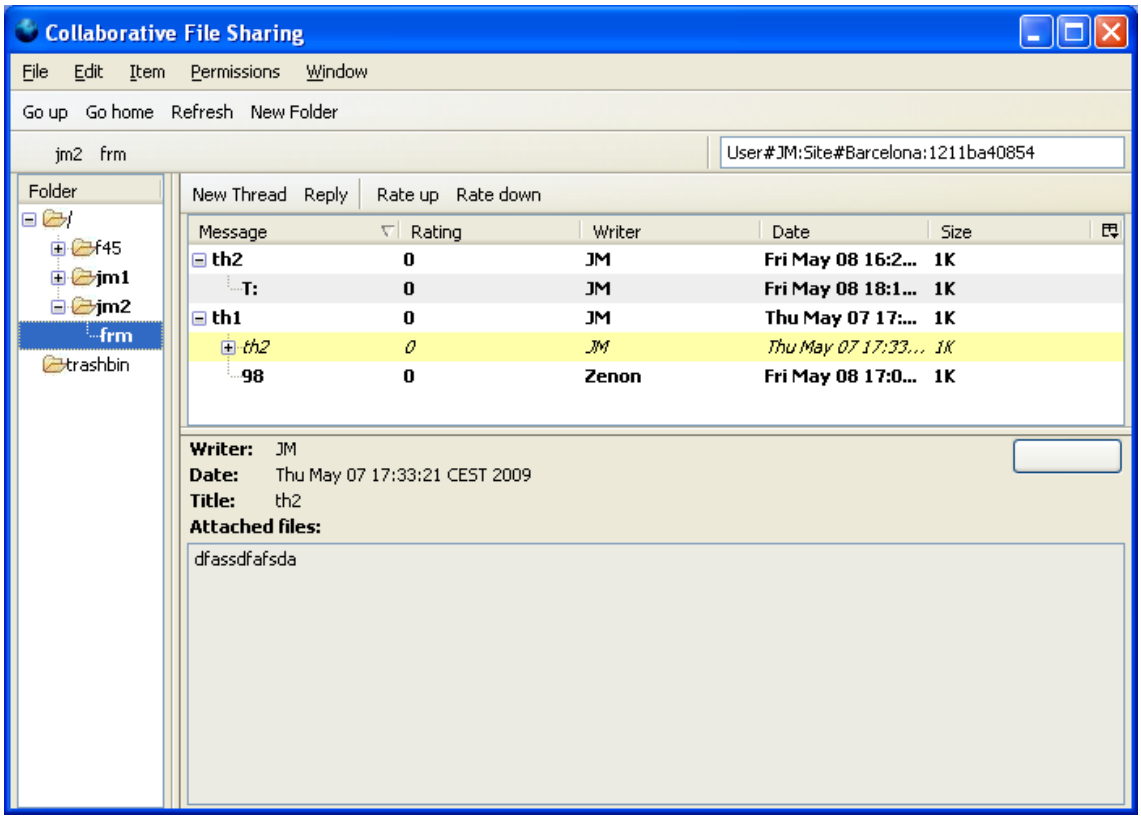


Figure 2.2. th2 message.

When creating a new thread or replying a message, one or more files can be attached (figures 2.3 and 2.4).

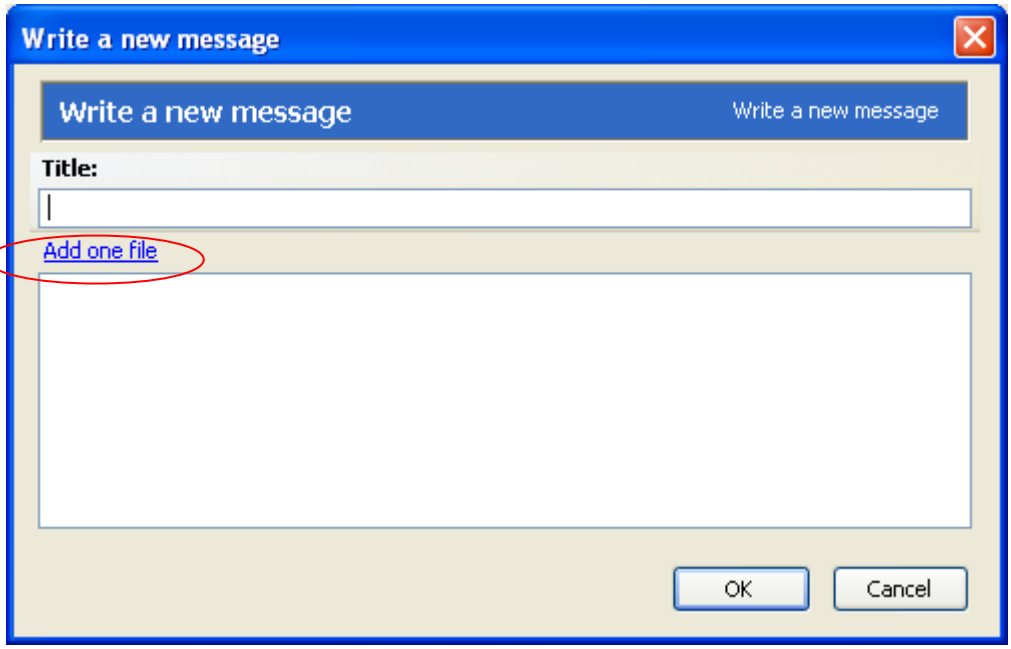


Figure 2.3. **Add one file** option when writing a message.

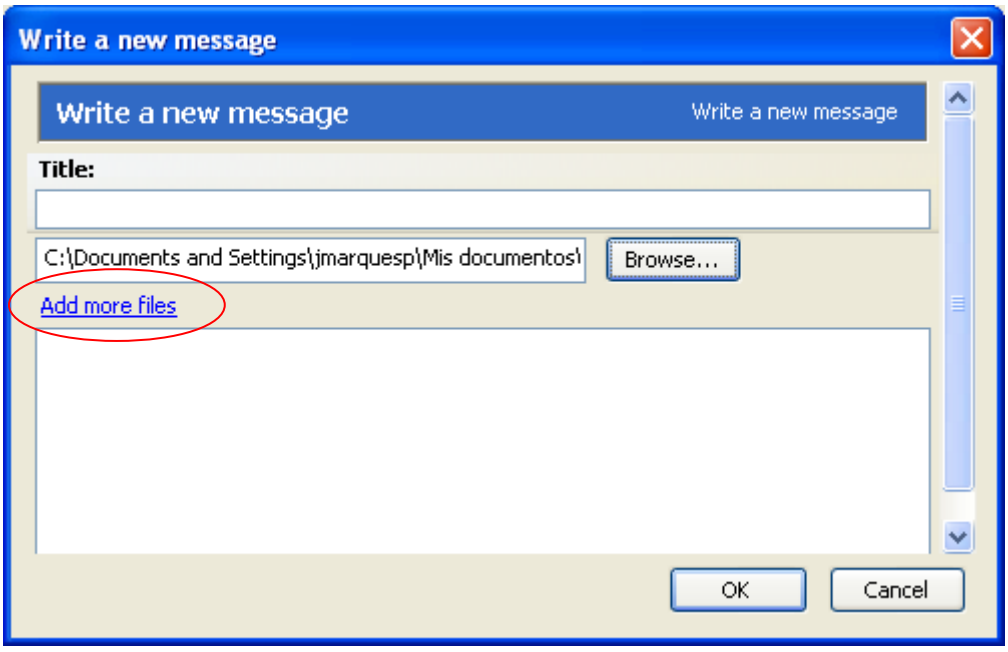


Figure 2.4. Adding files when writing a message.

3. Files

Figure 3.1 present the example of a file in CFS. File name is **cfs.bat**. Download button allows the download of the file.

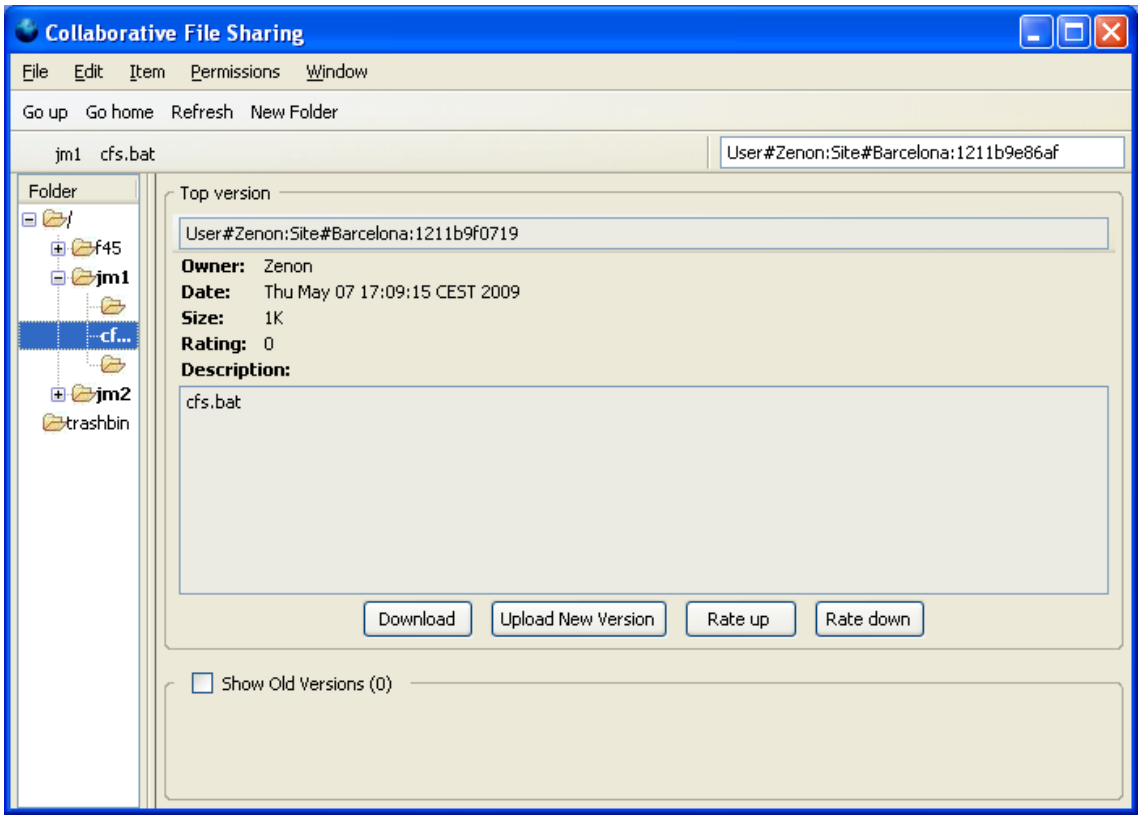


Figure 3.1. File cfs.bat

Each upload creates a new file. Previous versions can be viewed activating **Show Old Versions** checkbox (figure 3.2). The number of previous versions is indicated between

brackets. Contextual menu allows the download of desired version as well as rating it up or down.

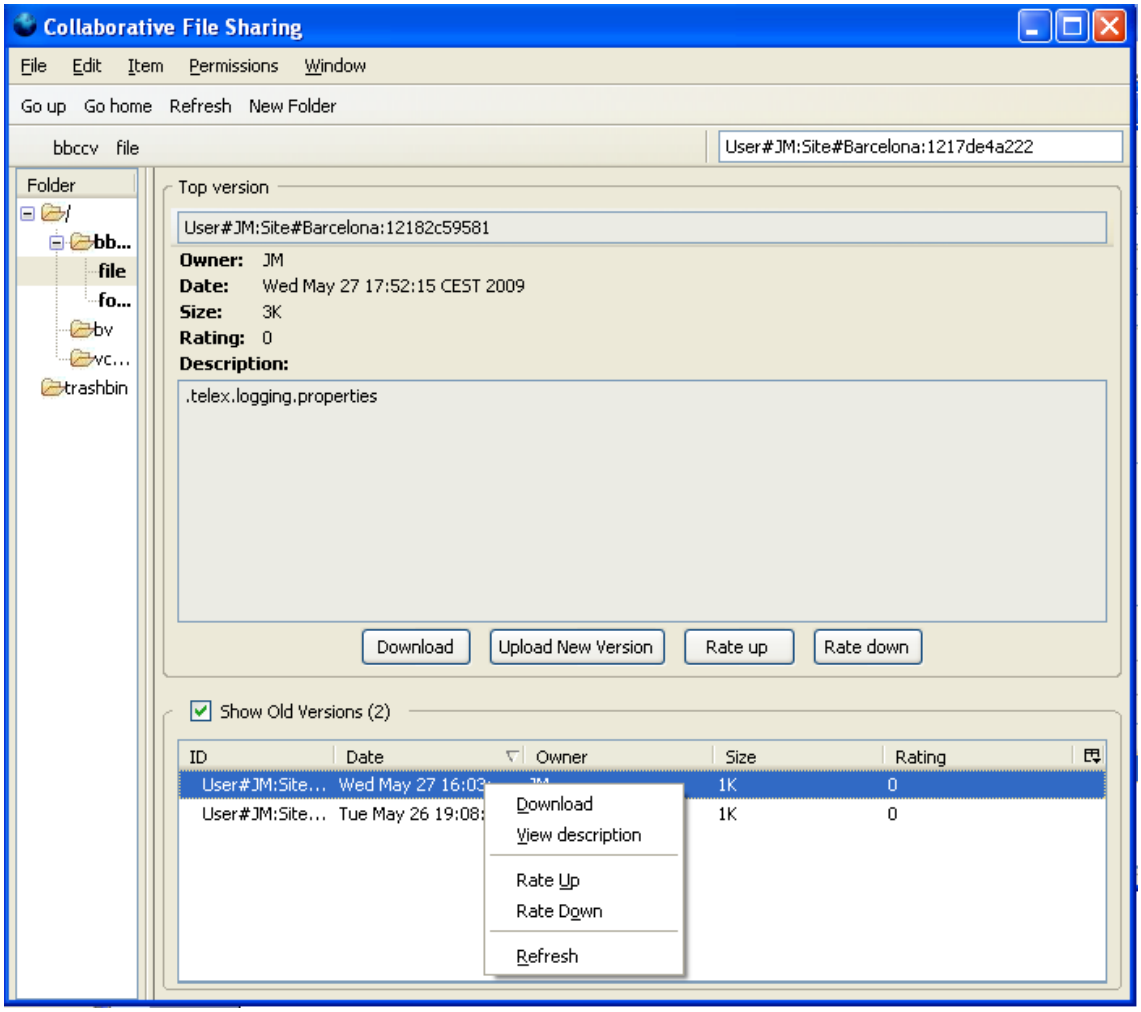


Figure 3.2. Old versions of a file.

4. Delete

Item -- Delete option in application menu or delete option in contextual menu allows deleting folders, files or forums. Deleted items go to “trash bin”.

Figure 4.1 shows trash bin content after deleting workspace **f2ooo**.

Contextual menu allows the recovery of deleted items (and its children items) from trash bin (figures 4.1 and 4.2). When recovering a deleted item, old location and old name are proposed (figure 4.2) by default. In case that the item has to be recovered in a different location or with a different name, user may change them in the recovery box (figure 4.2).

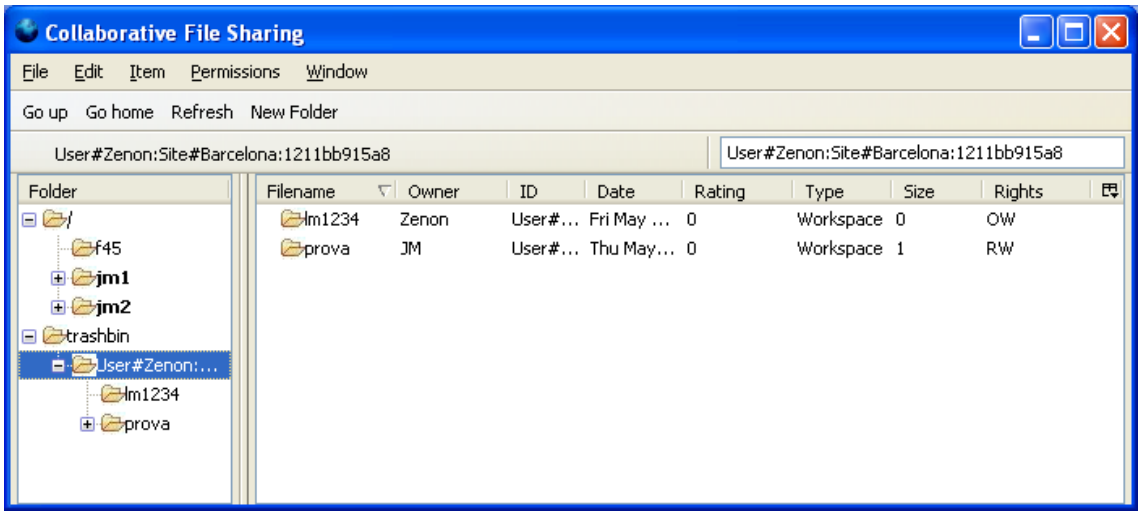


Figure 4.1. Item to recover from trash bin

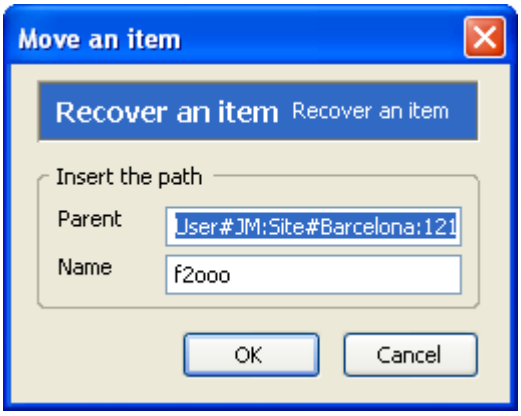


Figure 4.2. By default, deleted items are recovered at deleting location and with its initial name. If desired, both recovery location and name may be changed when recovering.

5. Awareness information

File -- Events on this Item option in application menu shows all events related to a selected item (including reading events on the item). Figure 5 shows all events on item **jm1**.

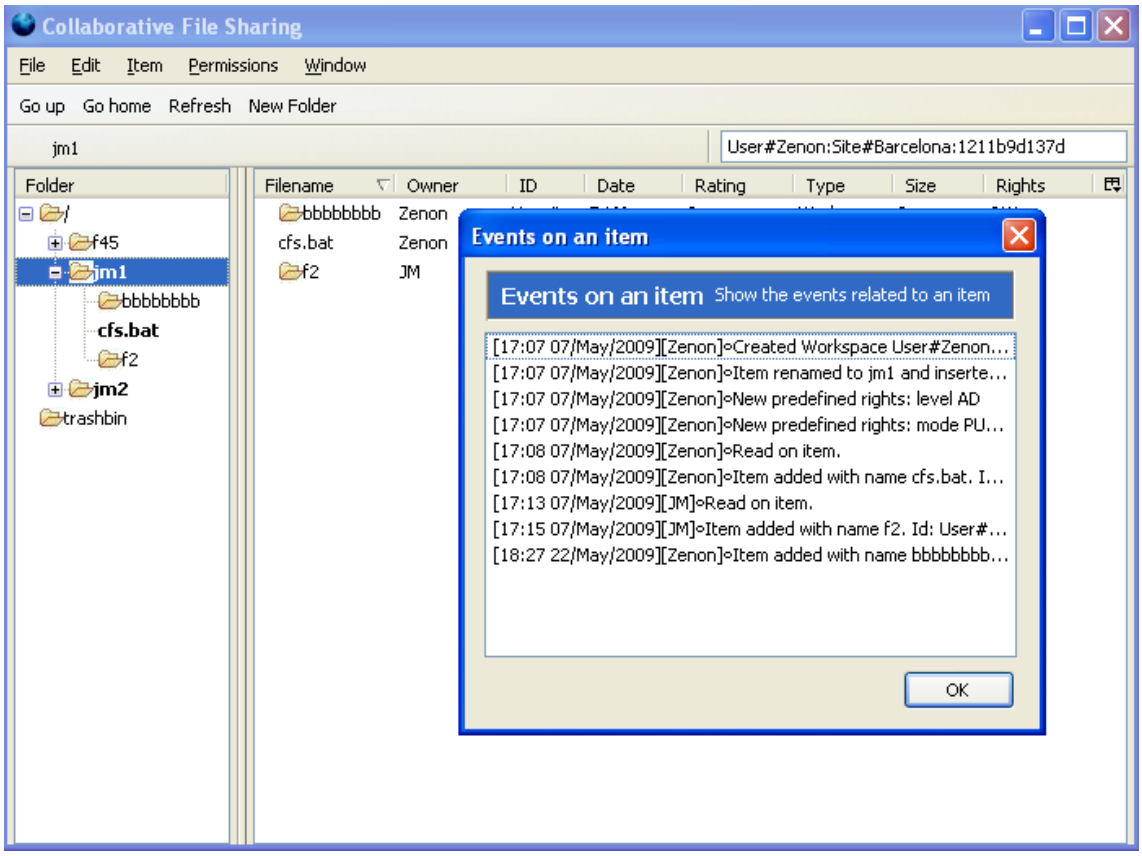


Figure 5. Events on item jm1.

6. Renaming and cut&paste

Items may be renamed or moved.

Rename option in the contextual menu allows the change of name of any item (i.e. folder, file or forum).

Contextual menu also allows the **cut** and **paste** of any item in any existing folder.

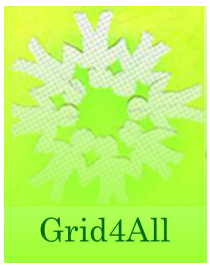
7. Miscellaneous

Operations are propagated using an optimistic approach (using telex middleware). Consequently, when a user does an action, it takes some time to appear in all CFS instances connected to group. This may result on different views of the workspace. CFS and telex guarantees that views will converge.

In addition, when a CFS instance receives an action from another user, the action is not automatically included in the view. CFS has a refresh function that every two minutes refreshes the view that the user has from shared information. This means that, in mean, actions done by other users are presented to other members in a delay of around one minute.

Refresh button forces the refresh of local view of information. It may be done at any moment.

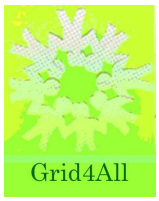
This page is intentionally left blank



Project no. 034567

Grid4All

Annex 12. eMeeting, an online Multimedia Collaborative Environment



e-meeting

User manual

On-line applications

Antares
FORMACIÓN Y COMUNICACIÓN

e-meeting

User manual

Complete application designed for meetings and/or personalised online tutorials, integrated into a work environment enabling access to a series of multimedia resources and functions, acting as a platform for presentations, meetings, tutorials, etc., and making them more attractive.

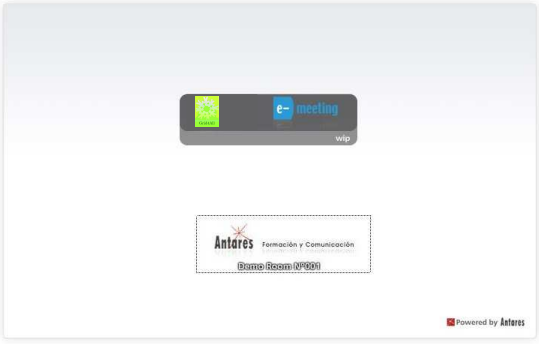
In e-meeting, we wanted to bring together almost all the resources you would find in a real meeting room together in a single application, using all the possibilities of modern technology: multimedia (slides, videos, and animation), notebook, chat room, surveys.

Introduction

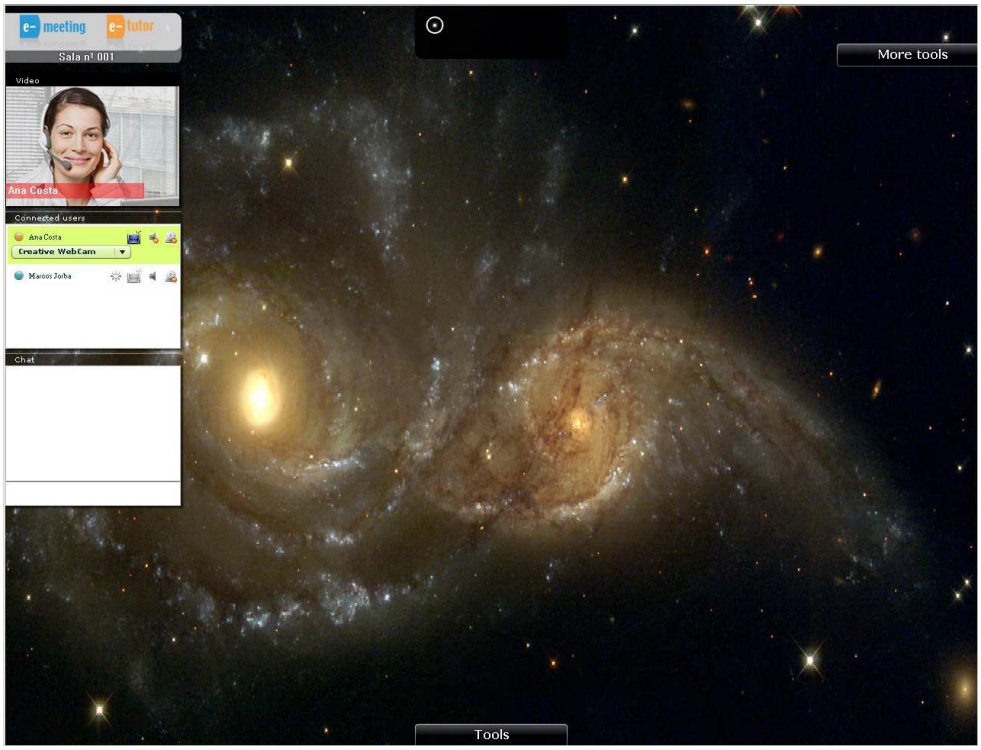
This manual will help familiarise you with all the resources available in the e-meeting application, and these simple instructions will show you how to access the application and understand all of its functions.
From this point on, we will refer to the person directing the meeting or giving the tutorial as the "Host", and the people attending the session, or students attending the course, will be called "Attendees".

How to access

Go to the website address assigned to your organisation for the sessions, and enter your username and password.
Be aware of the profile you will be using to take part: as host or as attendee.
It is important that the first person to log in should be the host, as he or she will have to enable access for the attendees, who can then log in.



After accessing the application, you will find a screen like this one below, split into different sections which we will explain one at a time, with their own resources and functions.

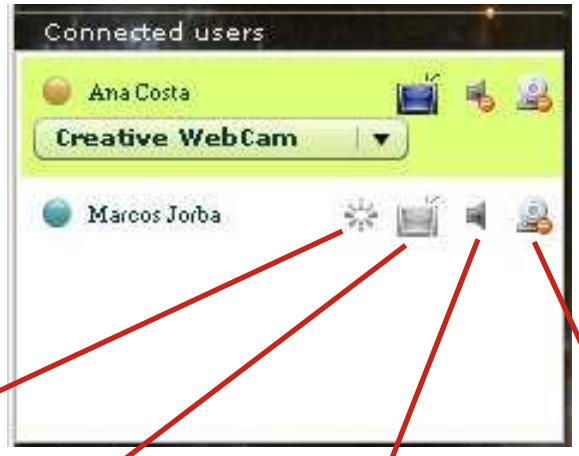


Attendees' Area

- Space to be personalised with the appropriate corporate image
- Image of the host/ attendees
- List with the names of the host and attendees
- Chat room

List of Attendees

Here you will find a list with the names of the host and the attendees taking part in the session.
This space is useful for checking attendee participation, as the host can monitor the time when they log in or out.
The Host has a series of functions available to him or her, represented by some icons which appear to the right of the name of each attendee, and which can be enabled or disabled by clicking on them.



Expel from the session

Icon enabling the host to expel any attendee from the session, if necessary.

Enable Webcams

Icon for enabling your own and the attendees' webcams, if switched on.

Enabling microphone

Icon used to give permission for attendees to use the microphone.

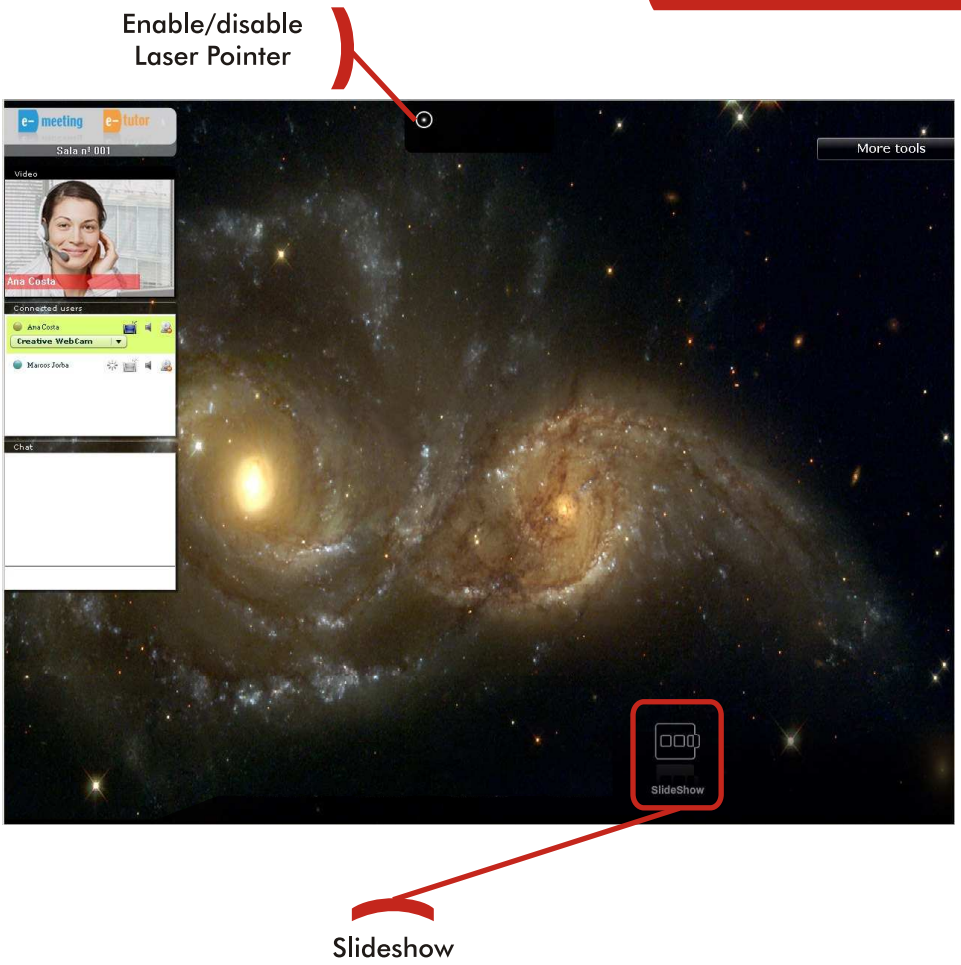
Images in mosaic

Icon used for displaying the host's and attendees' webcam images as a mosaic.

Chat room

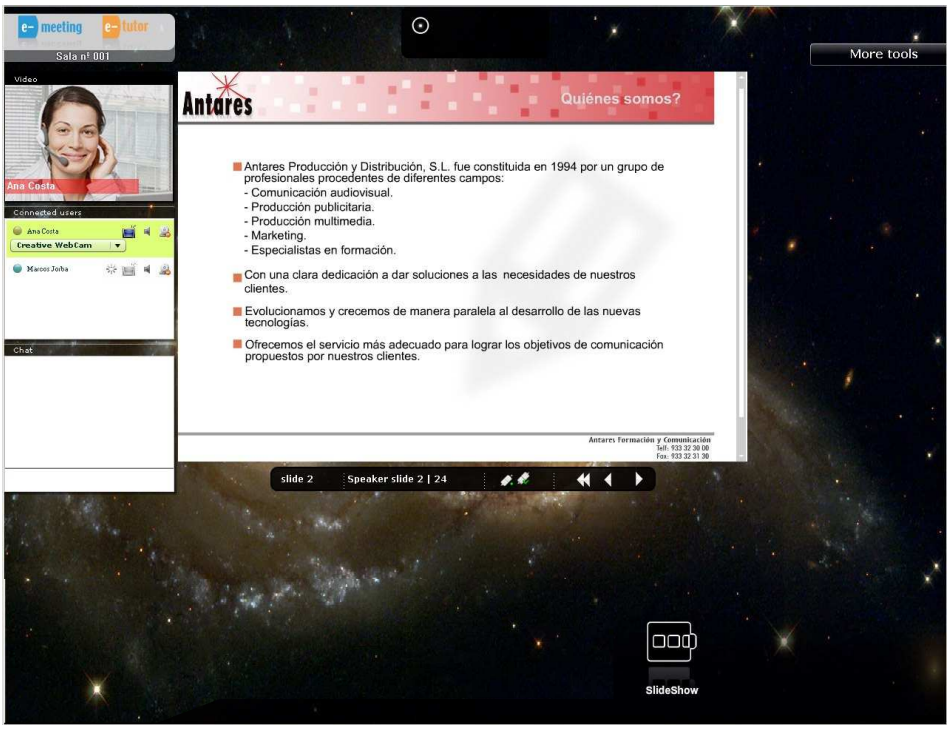
This chat room is provided to enable communication if any attendees do not have a microphone, so that they are not left out of the session.

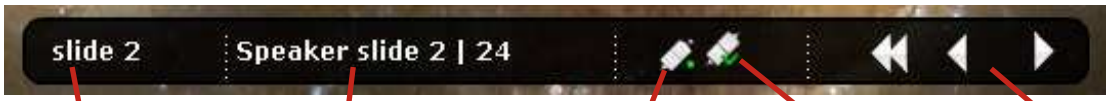
Tools area



"Slide Show"

This function is used to enable slideshows when a PowerPoint presentation has to be shown. It includes a toolbar to go forwards or backwards through the slides, control them and synchronise or de-synchronise a slide in relation to the general presentation; this last function is useful if you want to go backwards or forwards to a particular slide without changing the order of the slideshow as a whole.





Control indicating the number of the slide currently being viewed.

Control indicating the number of the slide currently being viewed and the total number of slides in the presentation.

Function which can be enabled by the host or by attendees, used to synchronise and de-synchronise slides. It affects only the person enabling the function.

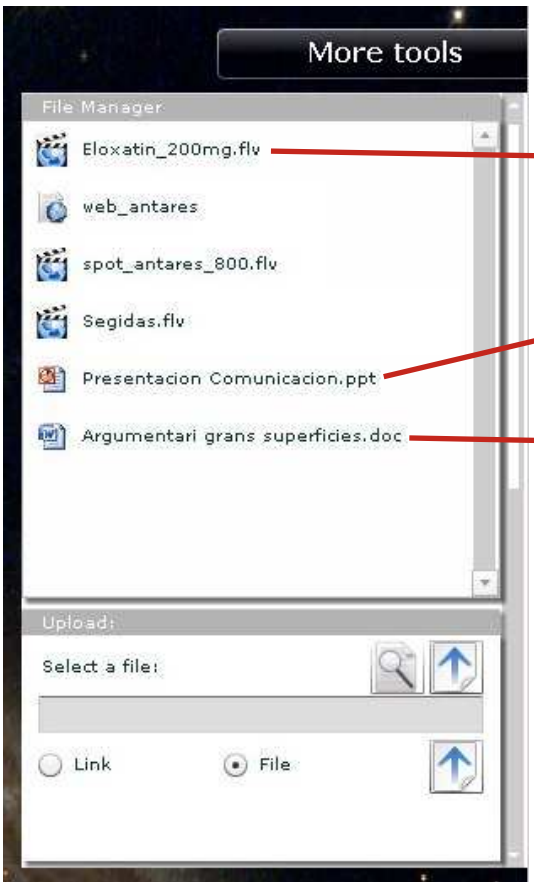
Function which can be enabled only by the host, to enable attendees to use the synchronise/de-synchronise tool.

Buttons for going forwards or back through slides.

More tools

The "More Tools" button is at the top right of the screen. Clicking on it displays the folder containing the files you are going to use during the session: videos, images, PowerPoint, documents, etc., which will enhance the presentation.
This folder contains all the files uploaded to the application before the start of the session.
Immediately below this folder is the "Uploads" resource, enabling you to upload documents directly to the application during the session, when they are going to be used, and which can take the form of "file" or "link".
Some of these archives can be for shared use by attendees, so that each one can download them directly to their desktops.

File folder



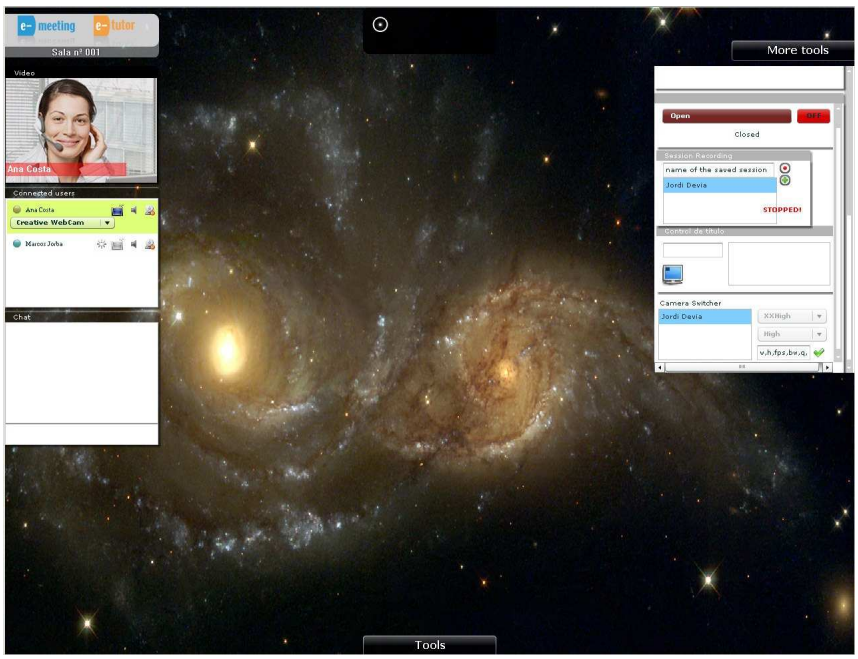
Click to choose slides

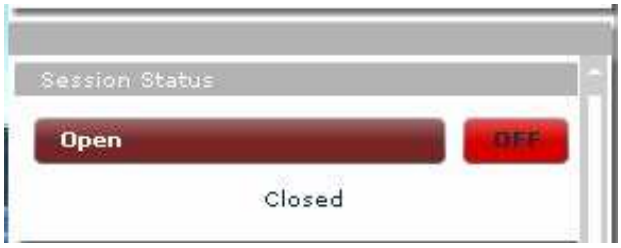
Click to choose document

Click to choose document

Uploads: select documents to upload directly to the application.

Scrolling down, we find the following functions:





Session status:

This function must be enabled at the beginning of each session (**on**) and disabled at the end of the session (**off**).
It is important to enable this function in order to ensure the correct functioning of all the tools available in the application.

Session recording:

Function enabled when starting to record a session. At the end of the recording, remember to give it a name relating to the session for later viewing.

This page is intentionally left blank