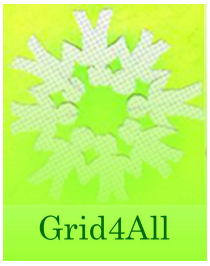



Project no. 034567

# Grid4All

Specific Targeted Research Project (STREP)

Thematic Priority 2: Information Society Technologies

# D2.2 Specification and initial prototype of the Grid4All VO management services

Due date of deliverable: 1st June 2008

Actual submission date: 11th July 2008

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

Organisation name of lead contractor for this deliverable: INRIA

Contributors : Nikos Parlavantzas, Jean-Bernard Stefani, Christophe Taton, Florent Métral, Vladimir Vlassov, Leif Lindbäck

Revision [1]

| Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006) | | |
|---|---|---|
| **Dissemination Level** | | |
| **PU** | Public | X |
| **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 |
|---|---|
| VO | Virtual organisation |
| DCMS | Distributed component management system |

# 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 deliverable presents the current state in the development of the core VO management system, which provides basic support for the creation and operation of VOs. The system is decomposed into a set of interacting services (membership, basic resource management, deployment, registry, security, reservation management, remote node addition) and is implemented following a component-based, decentralised architecture. The deliverable describes how the services are used, their design, and their current implementation status. Most of the services are prototyped and integrated into a functional VO management system.

# 2. Introduction

A central capability of Grid platforms is managing virtual organisations (VOs). A VO is a dynamic set of users that pool a dynamic set of resources into a single virtual administrative domain for some common purpose. VO management must include functions such as creating and deleting VOs, maintaining members and their attributes, discovering, allocating, monitoring, and controlling VO resources, deploying application code, and enforcing security. VOs in democratic grids are diverse, complex, and constantly changing, which requires that VO management supports ease of use, scalability and resilience in the face of failures and churn. These requirements have driven the design of the management system, presented in this document.

The Grid4All VO management system is responsible for realising the VO abstraction, and for managing the main VO elements: users, resources, and applications. The system underpins higher-level Grid4All services and applications and has a decentralised, overlay-based implementation using the DCMS platform. The system is decomposed into a set of interacting services (section 3). The *membership service* supports authentication of VO members and keeps associated information (section 4). The *security infrastructure* supports policy-based authorisation (section 5). The *basic resource management* supports providing resources to the VO, and discovering and allocating resources (section 0). *Remote node addition* supports requesting remote machines to join the VO (section 7). *Reservation management* supports reserving resources in advance (section 8), and the *registry* supports registering and looking-up VO services (section 9). Finally, *deployment* supports installing and configuring component-based applications (section 10). In terms of deployment, we present a language-independent reference model (10.1), an ADL-based deployment service (section 10.2) as well as a framework for defining complex deployment workflows (10.3).

# 3. VO management system

The VO management system provides basic, generic mechanisms for supporting the creation and operation of VOs. Specifically, the system supports creating and dissolving VOs, keeping track of VO membership, discovering and allocating resources, monitoring and controlling the resource usage of VO members, deploying component-based applications, registering and finding services, and enforcing security. The system builds on the DCMS platform and overlay services and is used by higher-level Grid4All services and applications.

VO management relies on the following conceptual model. *Users* register with a VO, and may contribute their *resources* to the VO, deploy *components* that execute on VO resources, or find and use VO services. *Services* are executing components that have been published so that they can be discovered by users.

Note that the VO management system embodies no policy decisions concerning the sharing of resources among users and services (more precisely, resources are allocated using a first-come first-served strategy). The VO management system is intended to form the substrate for higher-level, VO-specific management strategies programmed using the DCMS framework.

## 3.1 Architecture

VO management (see Figure 1) is decomposed into four core services (membership, basic resource management, registry, deployment), the security infrastructure, and two utility services (reservation management, and remote node addition).

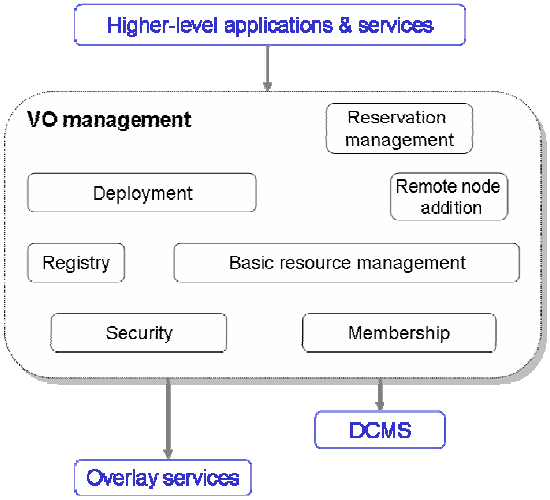


***Figure 1. VO management***

*Membership* manages members and their associated information, including roles, their rights and obligations, and their on-line status. *Basic resource management* handles providing resources to the VO, resource discovery, and allocation. *Deployment* supports installing, configuring, and activating component-based applications. The *registry* supports registering and looking-up VO services. The *security infrastructure* supports implementing VO-wide authorisation policies. Finally, *reservation management* supports reserving resources in advance, and *remote node addition* supports requesting remote machines to join the VO.

The following scenario illustrates the interactions between the VO membership service (VOMS), basic resource management (RM), deployment (D), registry (R), and the security infrastructure.

- A user invokes VOMS to log in to the VO.
- The user invokes RM to contribute some computational and storage capacity to the VO.
- RM obtains the user information from VOMS and verifies that the user has contributed a minimum capacity as specified in the member information.
- The user invokes R to ask for a given service; R responds that the service is not available.
- The user invokes D to deploy an application that implements the given service.
- D uses RM to discover and allocate appropriate resources for hosting the application components.
- RM verifies that the user has the right to allocate these resources based on the security infrastructure.
- D instantiates the application components on the allocated resources, binds them, and activates the application. D also registers the given service using R.
- The user invokes R to obtain the service and then invokes it.

More details on each service are provided in following sections.

## 3.2 Implementation

VO management is implemented as a set of components integrated into the *Grid4All containe*r—the customised Jade system for supporting autonomic management of VOs. The container is based on the Java implementation of the Fractal component model, and has an initial configuration described in Fractal ADL.

Apart from VO management, the container integrates the DCMS/Niche system as well as components that implement or wrap higher-level services and applications.

A VO physically corresponds to a set of Grid4All containers connected in the same structured overlay network. The current implementation distinguishes between two configurations of the container: *JadeBoot* and the *JadeNode*. A VO contains one or more JadeNodes, and a single JadeBoot that bootstraps the system and includes any centralised VO functionality, such the LDAP server in VOMS (section 4).

# 4. Membership

The goal of the VO membership service (VOMS) is to handle authentication of VO members and to keep track of member information, including their roles and associated policies, and their current on-line status. VOMS distinguishes two basic types of users: *administrators* and *members*. Administrators may create/dissolve VOs, add and remove members, create roles and associated VO-level policies, assign roles to members, retrieve information on members, and create invitations for members. Members may register with a VO using invitations, log in/ log out, and retrieve information on other members.

## 4.1 Scenarios

VOMS supports four main use cases: (1) create a VO, (2) register with a VO (i.e., become a member), (3) add member, and (4) log in the VO. The following scenarios illustrate how VOMS is used:

Create VO and invite users

- The user starts a Grid4All container, which activates a browser and brings up a web page.
- Through the web interface, the user asks to create a VO and enters a name and password. The system creates the VO, and registers the user as an administrator.
- The user (now an administrator) invites others to become members; invitations are sent as e-mails that contain the contact details of the VO.

Register with VO

- After receiving an invitation, the user obtains the contained contact details, and uses them to configure and start a Grid4All container.
- Through the web interface, the user asks to be registered to the VO providing details, such as name, email, and password.
- The system sends an e-mail to the administrator.
- The administrator adds the user as a member (see next).
- The user receives a registration confirmation by email, and can now log in and view other members.

Add member

- After receiving a registration request by e-mail, the administrator adds user as VO member through the web interface.
- The administrator assigns roles to the user.
- The system sends registration confirmation to the user.

Log in to VO

- Through the web interface, the user signs in to the VO providing name and password.
- The system validates that the user is a member.

## 4.2 Design and implementation

VOMS follows a conventional client-server architecture that relies on an LDAP server for storing membership information. A decentralised implementation using the DHT capability of Niche is possible, but beyond the scope of this work.

Specifically, the main part of the VOMS implementation lies at the JadeBoot and comprises the following three components: the *web application* (wrapper for a collection of servlets and the Apache Tomcat), the *LDAP component* (wrapper for the Apache Directory Server), and the *membership component*, which exposes the VOMS API. The web application provides the graphical interface to VOMS and uses the membership component, which in turn builds on the LDAP component. LDAP stores the following attributes for each member: user name, password, first name, last name, email, description, on-line status, and a set of roles. Each role has a name and a description. The JadeNode contains the *security component* which stores the memberId, a unique identifier generated by VOMS to identify the current login session.

The main operations of the VOMS API are shown in Figure 2. The operation *connect()* returns the memberID (represented as a random string), which is passed as argument to the remaining operations.

```
String connect(String userName, String pass, String jadeNodeName)
String connectAdmin(String userName, String pass, String jadeNodeName)
boolean addMember(Member mbr, String memberId)
boolean removeMember(Member mbr, String memberId)
Member getMember(String mbrName, String memberId)
List<Member> listMembers(String memberId
boolean updateMember(String userName, String newMail, String newDescr,
String memberId)
List<String> getRolesFromUser(String fullName, String memberId)
boolean addRole(Role role, String memberId)
boolean removeRole(Role role, String memberId)
Role getRole(String roleName, String memberId)
List<Role> listRoles(String memberId)
boolean assignRoletoMember(Member mbr, Role role, String memberId)
boolean unAssignRoletoMember(Member mbr, Role role, String memberId)
void createInvitation(String emailto, String memberId)
HashMap<long, Member> getOnlineMembers(String memberId)
```

***Figure 2. The VOMS API***

When the administrator starts the JadeBoot, the web application and the LDAP server are configured and activated. When a user starts a JadeNode, the web application is accessed through HTTP and allows the user to request registration or to log in, if already a VO member. After a successful login, the JadeNode receives and stores the user memberId. An overview of the interactions between users, JadeBoot, and JadeNode is shown in Figure 3.

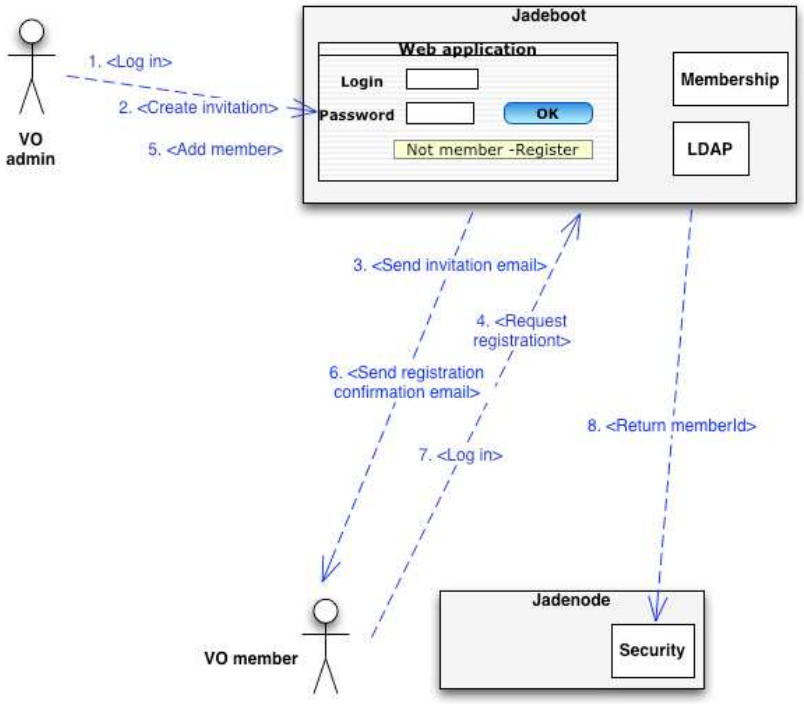


***Figure 3. VOMS interactions***

# 5. Security Infrastructure

The goal of the security infrastructure is to provide authorization—that is, to ensure that users can only access resources that they have the right to access. Authorization is policy-based; policies are expressed in XACML (eXtensible Access Control Markup Language).

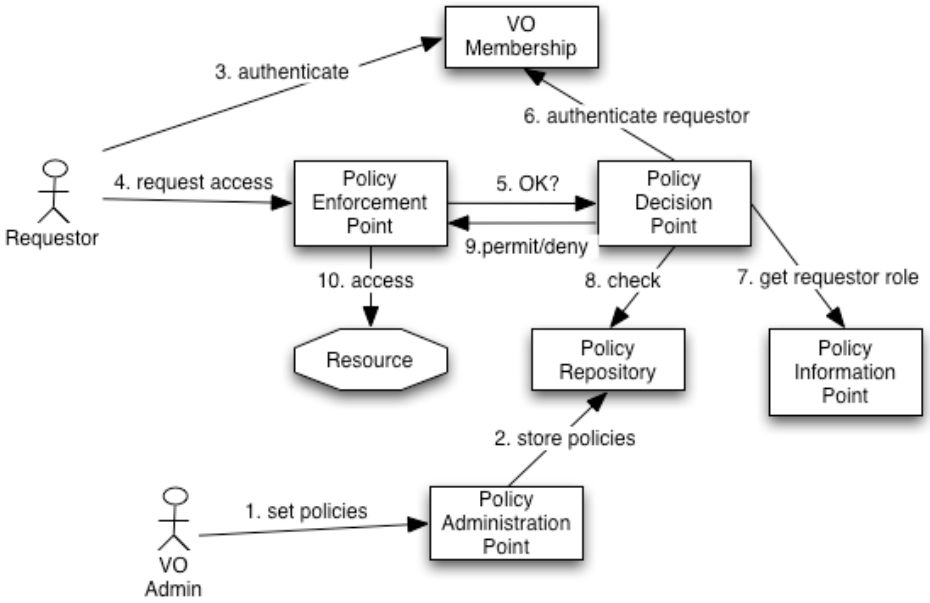


***Figure 4. The security infrastructure***

## 5.1 Scenario

The goal of the security infrastructure is to provide authorization—that is, to ensure that users can only access resources that they have the right to access. Authorization is policy-based; policies are expressed in XACML (eXtensible Access Control Markup Language).

## 5.2 Scenario

A typical scenario of the interactions between the security framework and the other components of a Grid4All system in depicted in Figure 4. We distinguish three different phases: setting security policies, authentication, and access control (i.e. policy enforcement.)

1. **Setting policies** (1) The VO policy administrator uses the Policy Administration Point, PAP to set policies. (2) The PAP stores the policies in the Policy Repository, PR. The PAP will invalidate the PDP's policy cache and the PDP will invalidate the response cache of the PEP.
2. **Authentication** (3) The requestor logs in to the VO membership service (VOMS). This should be done before the user tries to access any resource. If the user is authenticated, VOMS returns a memberId that can be used to identify the requestor for a limited amount of time. After successful authentication the user is not required to log in again until the memberId expires.
3. **Resource access** (4) The requestor uses an application that tries to access some resource, e.g. to perform an operation on the resource. The application sends the user name, and memberId to the Policy Enforcement Point, PEP, which protects the resource. (5) The PEP asks the Policy Decision Point, PDP, to evaluate the request, i.e. to check whether the requestor has rights to perform the requested operation on the resource. (6) The PDP contacts VOMS to check if the memberId is valid and if the requestor is who she claims to be. (7) The PDP asks the Policy Information Point (PIP) for the requestor's roles. (8) The PDP evaluates the policies stored in the PR. (9) If access was granted, the requestor accesses the resource.

## 5.3 Design and implementation

We have used the Sun's XACML implementation in order to develop and to implement a prototype of the Grid4All security framework. The components of the Grid4All security infrastructure are described next; VOMS was described in section 4.

1. **PEP (Policy Enforcement Point)** acts as a filter protecting the resource. PEP sends authorization requests to the PDP and caches the responses. We provide a PEP that handles the cache. It can be accessed through a stream or by direct method calls. The application developers can extend the generic PEP (with cache) provided in our framework. The developers are also allowed to write their own PEPs.
2. **PDP (Policy Decision Point)** evaluates requests from the PEP according to the policies in the PR. The result of the evaluation is returned to the PEP. If multiple policies give contradictory results, the answer will be Deny. Only if all policies relevant to the request permit access, the result will be Permit. In addition to the response, the PDP also sends responses to requests similar to the actual request (slight variations in resource and action). This is to improve performance since the responses will also be cached by the PEP. Policies are cached in memory by the PDP. Invalidation of the PDP's cache also invalidates the PEP's cache. The PDP must contact VOMS to validate the requestor's memberId. The answer from VOMS is cached. The PDP can be accessed through a TCP socket (used by the provided PEPs), through standard input, or through RMI.

   Important methods of the PDP class are:

   - `public String evaluateRecursively(String role, String resource, String action)`

     First checks for a policy for the resource specified by *resource*. 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 parameters are *role*, the role of the subject wanting to access the resource, *resource*, the resource about to be accessed and *action*, the action the subject wants to take on the resource. The method returns the PDP's

response. This is one of *Permit*, *Deny*, *NotApplicable* or *Indeterminate*. This method shall be used for example by file system applications.

- `public String evaluate(String requestCtxXml)`
  Evaluates policies without recurring, otherwise works like evaluateRecursively.

- `public void reload()`
  Discards all policy information cached by the PDP. Then loads all policies described in the XACML files in the policy directory.

3. **PIP (Policy Information Point)** returns the requestor's roles given the requestor's id. If the PIP has to contact the VOMS to do this, the answer from the VOMS is cached.
4. **PAP (Policy Administration point**) is a server that makes updates to the PR (Policy repository). The PAP is protected by PEP. The provided PAP can be called through a TCP socket, the stream reading PEP or through RMI. We also provide a command line client that uses TCP to contact the PAP.

   Some methods of a PAP API are shown below.
   - `public void clearACL(String issuerRole, String resource, String subjRole)`
     Removes an ACL entry from the policy repository.

   - `public void **setACL**(String issuerRole, String resource, String role, String permission, Boolean positive)`
     Handles the ACL creation requests.
   - `public String **listACL**(String issuerRole, String resource)`
     The method returns the access rights of the given parameters. The issuer must have enough permission to issue the listacl command. The required permission is *read* for the specified resource path and all parent paths.
5. **PR (Policy repository)** is a persistence storage used to store the policies as XACML files. The repository is continuously monitored by a special service Repository Access Monitor that notifies (via a callback interface) PDPs whenever some file in the repository is updated or deleted, or a new file is created. These notifications cause invalidation of PDP caches and as a consequence invalidation of PEP caches.

# 6. Basic resource management

The basic resource management service (called RM) provides core support for sharing resources among VO members. Specifically, RM enables members to discover, allocate, and monitor VO resources and to contribute resources to the VO. All supported operations can be subjected to VO policies by exploiting the membership service and the security infrastructure.

The goal of RM is to provide only simple, lightweight mechanisms for resource sharing. For example, RM neither supports co-allocation and advance reservations, nor does it dynamically coordinate resource allocation in order to optimise VO-wide measures. Indeed, RM is intended to form a basis for more sophisticated Grid resource management services, such as the reservation management service described in section 8.

## 6.1 Conceptual model

In the context of RM, a *node* is the unit of resource contribution by a VO member. A *resource* represents a share of the processing, storage, communication capacity of a node that is allocated to a particular VO member. Resource discovery results in nodes with desired properties. Resource allocation takes as input a node and returns a resource. Discovery and allocation are based on node and resource properties; that is,

name-value pairs such as {“storage” : 100000000}. Resources are used to deploy components; specifically, a component is deployed on a single resource.

## 6.2 Design

The RM service follows a decentralised architecture in which each Grid4All container includes an RM component supporting all management operations. The main RM operations are shown in Figure 5.

```
NodeId contribute (Map underlyingConfig, String memberId)
NodeId[] discover(ResourceSpec spec, String memberId)
ResourceId allocate(NodeId node, Map properties, String memberId)
release(ResourceId resource, String memberId)
Map query (ResourceId resource, String memberId)
```

***Figure 5. The basic resource management interface***

The *contribute()* operation enables configuring the local Grid4All container to become a node, owned by the member identified by the argument. The *discover()* operation takes as input a specification and returns a list of node references that satisfy the specification. The specification (of type ResourceSpec) represents constraints on node properties and takes the form of LDAP search filters, e.g., “((storage>=500000000) and (networkSpeed=medium))”. Discovery is implemented by using the corresponding low-level overlay service, which currently relies on an efficient broadcast mechanism. The *allocate*() operation takes as input a node reference and a set of properties representing the requested capacity {e.g., “storageShare” : 100000000} and returns a resource reference. To implement this operation, RM contacts the remote machine through the overlay network and interacts with its remote counterpart, which updates the node properties and keeps track of the association between resources and members. The *release()* operation terminates the allocation, and the *query()* operation returns the full set of properties associated with a node or resource. Importantly, RM is responsible to validate all RM operations according to member roles and policies by acting as a PEP in the security infrastructure. For example, RM could validate that members of the “student” role cannot allocate more than 1GB of storage in the VO.

## 6.3 Implementation

RM is implemented as a Fractal component that depends on DCMS/Niche and is integrated in the Grid4All container. The current prototype handles resource properties related to CPU time, physical memory, storage space, and network bandwidth. Note, however, that the prototype cannot control the resource consumption of deployed components, which would require appropriate OS-level support. Moreover, the prototype currently does not enforce any authorisation or resource consumption policies.

# 7. Remote node addition

This is a utility service that allows a member to contribute a target machine to the VO without having to log in to the VO from the target machine. Specifically, in contrast to the corresponding resource management operation (*contribute()*), this service is not invoked by a local user that has logged in to the VO using the web-based membership interface. Rather, the service runs on the to-be-added machine and is remotely accessible from anywhere. It takes as input the VO contact details, and the memberId of the contributing member, and returns a node reference. The service has the effect of configuring the machine to become a VO node, owned by the identified member.

## 7.1 Implementation

The planned implementation of this service will be integrated in the Grid4All container, which must be installed on the target machine. The implementation builds on the membership service for joining the VO, and the basic resource management service for contributing the node. The service will be accessible from clients external to the VO through a standard protocol, such as Web Services or RMI.

# 8. Reservation management

The reservation management service (RvM) supports advance reservation of resources available within the VO or obtained on demand from resource markets. It builds on basic resource management described previously and on the resource brokering support described in deliverable D2.3.

## 8.1 Conceptual model

This service assumes that users send reservation requests and RvM first checks whether the allocation is possible using only the VO's available resources. If not, RvM initiates negotiation at the resource market and leases additional resources from resource brokers. Resource requirements are expressed in term of *resource descriptions* (of type ResourceDesc), which encapsulate a resource specification (of type ResourceSpec), the times when the resource is required, and the number of units of the resource. RvM responds to reservation requests with a *reservation* object. This object contains a set of *resource holders*, which are bound to concrete resources (of type ResourceId) when these become available.

## 8.2 Design

The structure of the RvM API is shown in Figure 6.

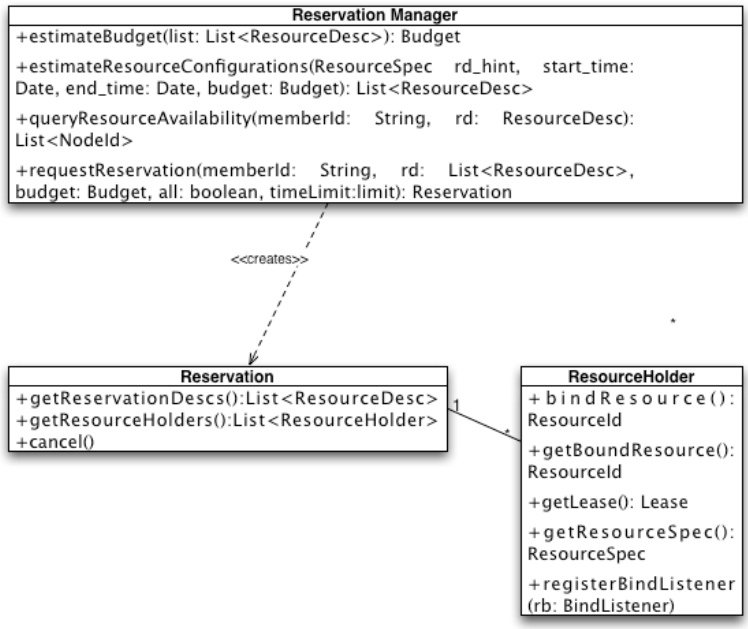


***Figure 6. The reservation management API***

The main RvM operation is *requestReservation()* which accepts the following arguments: (1) the memberId of the client; (2) a list of *resource descriptions* specifying the characteristics of the required resources; (3) the *all* argument, which is true if the client requires the total quantity or resources or nothing; (4) the maximum time to reserve resources (*timeLimit*); and (5) the maximum *budget* that the client is willing to pay to acquire the resources. The operation returns a *reservation* object that provides information about the reserved resources. Moreover, the reservation provides access to a set of *resource holders*, which may be either bound or unbound. When bound, the holder provides access to a concrete resource (i.e., a ResourceId). Clients request the binding of holders using *bindResource()*. Finally, RvM provides convenience operations to obtain the current availability in the VO, to estimate the budget required to lease resources from the market, and to estimate the resource configurations given a specific budget.

A simple example of using the RvM API is shown in Figure 7.

```
// Estimate budget
List<ResourceDesc> preferences = ...
Budget budget = mgr.estimateBudget(preferences);

// Estimate the set of resources that can be obtained for a given budget
ResourceSpec rspec_hint = ...
List<ResourceDesc> rsrcEstimate = mgr.estimateResourceConfigurations(rspec_hint,
                          starttime, endtime, budget);

// Create a resource reservation
Reservation reservation = mgr.requestReservation(myMemberId, rsrcEstimate, budget,
false, latestTimeToReserve);

// Inspect resource reservation
List<ResourceHolder> rHolders = reservation.getResourceHolders();
ResourceSpec spec= rHolders.get(0).getResourceSpec();

// Allocate reserved resources by, for example, setting up timer-based tasks that
// run at scheduled times
Timer timer = new Timer();
timer.schedule(new TimerTask() { public void run() {
        ResourceRef res = rHolders.get(0).bindResource();
        // Use resource res, e.g., to deploy components
}}, 1200000);
```

***Figure 7. Using the reservation management API***

Typically, the client first decides on the budget and the required resources using some combination of *estimateBudget()* and *estimateResourceConfigurations()*. Next, the client creates a reservation and obtains information about the reserved resources, which may be a subset of the required resources (if the *all* argument is false). When the lease period is active, the client obtains the resource through the *bindResource()* operation.

## 8.3 Implementation

We are planning to provide a simple RvM implementation that builds on the RM, the remote node addition service, and the Negotiator module (described in D2.3). Each Grid4all container will contain an independent instance of this implementation, thus not supporting VO-wide management of reservations. To implement *requestReservation()*, RvM will first use RM to discover and allocate any available resources within the VO. If necessary, RvM will then invoke the Negotiator to obtain leases to nodes bought from external providers. To implement *bindResource()*, the RvM will invoke the remote node addition service on the leased node, causing it to join the VO. The leased node will be owned by the member that requested the reservation. After the node is successfully incorporated into the VO, RvM will invoke the allocation operation of RM and return the resulting ResourceId to the member.

# 9. Registry

The registry service supports a simple means of finding services within the VO. In particular, it supports registering names for executing components, and looking up components using their names. The registry is used as a bootstrapping mechanism to obtain access to the core VO management services (e.g., basic resource management) as well as higher-level services and deployed applications.

## 9.1 Design and implementation

The registry API is shown next. Names are provided by clients, they are unique within the VO, and the name space is flat.

```
Component lookup(String name)
void register(String name, Component comp)
void unregister(Component comp)
```

***Figure 8. Registry API***

The returned *Component* reference gives access to a local Fractal component, which may be a proxy to a remote component. Proxies communicate with remote components via DCMS-based bindings exploiting the overlay network (see D1.2).

The core VO management services are registered with well-known names. Deployed applications are registered by the deployment service with names of the form < *ADLname*>_<*deploymenId*>, which allows designating different instances of the same application. There is currently a centralised implementation of the service that relies on a Fractal RMI registry, deployed on the JadeBoot. A decentralised implementation that uses the Niche DHT is in our future plans.

# 10. Deployment support

Deploying distributed applications is a complex process involving multiple distributed activities, such as assembly, installation, configuration, and activation. To deal with this complexity, we adopt an architecture-based approach relying on a language-independent reference model for deployment and (re)configuration. Using this model, we have developed two forms of deployment support:

- ADL-based deployment service, fully integrated in the VO management system
- FructOz framework, the first version of an Oz-based framework for defining complex deployment workflows, currently independent of the VO management system

The reference model, the deployment service, and the FructOz framework are described in turn next.

## 10.1 Reference model

The architectural background for our work takes the form of a programming-language-independent reference model. It consists of two main elements: a component model, and local node structure. The component model is a refinement of the Fractal component model (described in detail in [Bruneton0]) to make explicit notions of software component packages and their dependencies. The local node structure identifies a set of abstractions and functions for local installation and deployment.

A Fractal component is a run-time entity, whose presence is manifest during execution. Deploying and configuring a Fractal application informally implies: installing the appropriately configured executables (e.g. source code, binaries) at chosen nodes (i.e. physical or virtual machines), creating and activating the necessary components (including the necessary bindings constituting interaction pathways with other

components). Our reference model refines the Fractal model by making executables explicit and enabling them to be manipulated by means of *component packages.*

A component package (or package for brevity) is a bundle that contains executables necessary for creating run-time components, as well as data needed for their correct functioning, and metadata describing their properties and requirements with respect to the target environment. A package is itself a Fractal component, which means it can contain other packages, be shared between multiple containing packages, provide certain interfaces, and require other interfaces from its environment. A containment relation between packages corresponds to a requirement dependency between packages: the composite package requires the presence of the sub-package in the target environment. A package metadata may contain additional requirements, such as version information, resource requirements, or conflicts (identifying that certain packages, or components in general, must not be present in the target environment). Note that there are two aspects of configuration involved in our model: the first one is related to package resolution, and the second one corresponds to setting appropriate activation parameters and attributes, and establishing necessary bindings between run-time components.

A distributed deployment and configuration process for Fractal systems can be understood as a distributed application executing on a set of nodes. Nodes that form the targets of the deployment process are called *managed nodes*. To effect deployment and configuration, managed nodes must be equipped with a minimal structure. This comprises, for each managed node:

- One or more *binding factories* for establishing bindings supporting remote communication with components residing on the managed node.
- An *installation store* component, which is a repository of packages.
- One or more *loader* components, which are responsible for loading executables from packages in the installation store.

At least one binding factory per managed node is required to enable communication with component interfaces located on managed nodes, and to transfer packages to managed nodes (either in a pull or a push mode). The installation store constitutes a local target environment for deployment, and provides the context for deciding whether a package is installable or not, i.e. whether all the mandatory dependencies of a package can transitively be resolved. The installation store component need not execute on the managed node it belongs to, but may be located on a different node (e.g. because of resource constraints). Loader components (loosely analogous to Java class loaders) can take many forms, performing strictly binary loads for execution, or engaging in run-time resolution of package dependencies, with installation of required packages in the local installation store. By definition, loader components execute on the managed node they belong to.

## 10.2 ADL-based deployment service

The deployment service relies on declarative application descriptions written in Fractal ADL extended with Grid4All-specific concepts. Specifically, the ADL-based application description includes the following information: (1) the application architecture in terms of components, their composition relationships, and their bindings; (2) component configurations in terms of attribute values; (3) packaging information (e.g., the names of software packages with the code); and (4) resource requirements for components. The deployment service automatically discovers and allocates appropriate VO nodes for hosting the application, and performs instantiating, binding and configuring the necessary components. Note that this service supports establishing initial application configurations; performing dynamic reconfiguration requires using the Fractal-based DCMS API described in D1.2.

The deployment service in conjunction with DCMS provide an implementation of the reference model presented in section 10.1. Managed nodes correspond to Grid4All containers. Packaging relies on the OSGI specification; component packages correspond to OSGI bundles, the installation store corresponds to the OSGI repository, and the loader corresponds to a modified Oscar OSGI implementation. Finally, binding factories correspond to the DCMS-based one-to-one and group communication mechanisms that use the overlay network.

### 10.2.1 Using the deployment service

To address deployment of Grid4All applications, the standard Fractal ADL has been extended with new concepts concerning component placement and binding. To illustrate the placement-related extensions, consider a simple application consisting of two primitive components (client and server) inside a composite component (see Figure 9). The server component provides a server interface named "s" of type Service. The client component provides a server interface named "m" of type Main, and a client interface named "s" of type Service.

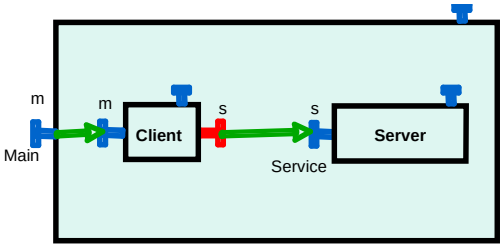


Figure 9. Simple Fractal application

A description of this application as given as input to the deployment service is shown next.

```
<definition name="helloworld">

  <interface name="m" role="server" signature="Main"/>

  <component name="client">
    <interface name="m" role="server" signature="Main"/>
    <interface name="s" role="client" signature=" Service"/>
    <content class=" ClientImpl"/>
    <virtual-node name="node1" resourceReqs="physicalMemory>1000000000"/>
  </component>

  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
    <packages>
        <package name="ServerPackage v1.3" >
              <property name="local.dir" value="/tmp/j2ee"/>
        </package>
    </packages>
     <virtual-node name="node2" resourceReqs=" physicalMemory >2000000000"/>
  </component>

  <binding client="this.m" server="client.m" />
  <binding client="client.s" server="server.s" />
  <virtual-node name="node1">
</definition>
```

This description illustrates the main concepts of the standard Fractal ADL (i.e., definition, component, interface, binding, content) as well as the Grid4All-specific extensions for component placement (i.e., packages, and virtual nodes). In particular, the *packages* element provides information about the OSGI bundles necessary for creating the component; packages are identified with their unique name in the OSGI bundle repository. The *virtual node* element describes resource and location requirements of components. At deployment time, each virtual node is mapped to a concrete resource that conforms to the given

requirements (*resourceReqs* attribute); this resource is used to host the associated component. In the example, the client and the composite component should be co-located at a node with memory larger than 1GB, and the server should be deployed on a different node with memory larger than 2 GB.

The binding-related extensions enable the ADL to represent group communication patterns as supported by DCMS. In the following example (see Figure 10), a client component is connected to a group of two server components (server1, server2) using one-to-any invocation semantics.

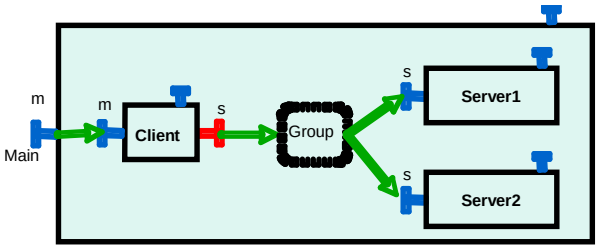


Figure 10. Fractal application with group communication

The ADL description of this application is shown next.

```
<definition name="hellogroup">

  <interface name="m" role="server" signature="Main"/>

  <component name="client" definition=”ClientType”/>

  <component name="group1">
      <interface name="s" role="server" signature="Service"/>
      <interface name="clients" role="client" signature="Service"
          cardinality="collection"/>
      <content class="GROUP"/>
  </component>

  <component name="server1"> definition=”ServerType”/>
  <component name="server2" definition=”ServerType”/>

  <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 the example, group communication is represented by employing a special component with content "GROUP", which is bound to both the clients and members of the group. Group components abstract over the DCMS-provided group mechanism and are logically distributed over different nodes. One-to-all bindings are similarly represented using the value “groupAll” as the *bindingType* attribute.

## 10.2.2 Implementation

The implementation of the deployment service is integrated in the Grid4All container and comprises the OSGI-based installation support and an ADL interpreter. The interpreter is a customised Fractal ADL factory that supports the Grid4All-specific ADL extensions. More details on the ADL factory were given in D2.1. The interpreter analyses the application description, uses the basic resource management service to find and allocate resources, and then creates, configures, and binds the components through the DCMS API.

# 10.3 FructOz framework

This section presents a software framework, called FructOz, which supports the development of complex and highly flexible deployment and configuration workflows. FructOz relies on the Fractal component model and the Oz programming language. At this time, FructOz is not integrated into the VO management system, but this is included in our plans.

## 10.3.1 Motivation

The ADL-based deployment service described in section 10.2 provides users with a declarative interface that greatly facilitates the deployment and configuration process. However, the deployment service supports a fixed, built-in deployment workflow (essentially, sequential deployment), which restricts its applicability in different application scenarios. In particular, the service cannot support complex deployment, configuration, and reconfiguration processes, such as those envisaged by [Coupaye00], which require handling multiple constraints, policies, synchronization, and error conditions.

In order to overcome this limitation, we are developing the FructOz framework, written in the Oz programming language. In particular, the requirements that motivated FructOz and are not adequately addressed in the existing literature are:

1. Ability to define complex deployment workflows, including support for well-known control flow and exception patterns.
2. Ability to define parameterized and higher-order workflows for the concise specification of complex distributed architectures.
3. Ability to finely control the activities involved in the deployment and (re)configuration of a distributed software architecture, and the moment they are triggered.
4. Ability to internalize deployment and (re)configuration activities in the description of a self-configurable software architecture.

Requirement 1 concerns the ability to directly support the main patterns of synchronization and exception handling that have been identified by the workflow community [Aalst03]. These patterns are useful yardsticks for the specification and programming of deployment and configuration processes for they embody higher-level abstractions and operators for process synchronisation and task composition which have proved useful in the design of enterprise-wide formal processes and on-line services.

Requirement 2 concerns the ability to define parametric deployment and configuration processes, which is essential for supporting scalable and compositional definitions of deployment and configuration processes. Parameters of a deployment and configuration process may include, for instance, the size and structure of the target environment, or binding and interconnection schemas between deployed components as in multi-tier systems. Requirement 3 concerns in particular the ability to delay the deployment and configuration of individual components up to the point where they are finally needed. This lazy deployment capability in turn supports different performance and adaptation tradeoffs. Finally, requirement 4 is a necessary step towards autonomic components and systems.

## 10.3.2 Related work

Previous approaches for supporting complex distributed deployment processes fall roughly into four categories: (i) approaches that rely on a fixed deployment algorithm or more extensive task framework, driven by (static) software architecture descriptions, including [Deng05, Flissi06, Kon05, Lan05, Mikic02, Quema04, Abdellatif05, Akkerman05, Chazalet07, Ketfi05, Kirby05, Wang06]; (ii) approaches that rely on a workflow language for describing deployment processes, including [Goldsack03, Badonnel04, Rowanhill07, Valetto03]; [Anderson03, Goldsack03, Badonnel04, Rowanhill07, Valetto03]; (iii) approaches that rely on a constraint solving algorithm for determining a deployment target, including [Bastarrica98, Malek05, Tibermacine07, Mikic05]; (iv) approaches that rely on an AI planner for generating deployment processes, including [Arshad07, Kichkaylo04, Heydarnoori06]. Approaches that rely on fixed deployment algorithm or a supporting framework, do not meet the requirements we have identified in the introduction. However, the work reported in this report could be exploited in a number of the frameworks referenced above. In particular, the work in this report complements our previous work on component deployment and configuration in Jade [Abdellatif05], and would be directly usable in the FDF [Flissi06] framework.

Approaches that rely on constraint solving and AI planning are interesting for their high degree of declarativity, but they do not provide sufficient support for dealing with complex deployment workflow, and to support effectively our four requirements above. Approaches based on constraint-solving tend to focus on specific deployment problems and objectives (e.g. such as ensuring a certain degree of availability [Malek05]). They must be complemented with additional mechanisms to deal correctly with exceptional conditions or to ensure synchronization conditions which are not enforceable within their scheduling framework. Approaches based on AI planning are interesting because they have the potential to automate the generation of complex deployment workflows, but it is not clear at this point that they can be employed successfully beyond simple system configurations. There is certainly promising work for integrating AI planning techniques with workflow management systems [Moreno02] but issues with respect to expressivity and scalability are still open.

Approaches that rely on a workflow language are closer to ours. They include the SmartFrog system [Goldsack03], the SmartFrog system [Anderson03, Goldsack03], the Workflakes system [Valetto03], the Andrea system [Rowanhill07], and the use BPWS4J workflow engine with the IBM Tivoli deployment engine for the provisioning of application services [Badonnel04]. Compared to our approach, which relies on the coordination and distributed programming capabilities of the Oz programming language, these systems do not provide direct support for parameterized, higher-order workflows (requirement 2), and they do not provide direct support for lazy execution (requirement 3). Also the use of a workflow or process management engine remains essentially external to the components used or the system being managed, and no provision has been made for the construction of self-deployable components and self-configurable hierarchical components (requirement 4). This is also the case with the SmartFrog system, even though its workflow constructs take the form of SmartFrog components, because these are used for the structuring of the initial deployment process. Note that, in our approach, even if components are not programmed in Oz, internalizing a complex deployment and configuration behavior is made simpler by the reflective character of the Fractal model. We could, for instance, make use of the aspect-oriented capabilities of the reference implementation of Fractal in Java, to program advices interfacing directly to the (meta-level) deployment and configuration behavior written in Oz.

Our work is also related to architecture description languages for dynamic architectures, i.e. ADLs that can describe dynamically evolving architectures. A recent survey of such ADLs can be found in [Bradbury04]. The survey shows that the subject of ADLs for dynamic architectures is well researched but that none of the surveyed approaches provided support for unconstrained evolution. Interestingly, this can be traced to the fact that none of the surveyed approaches are higher-order, e.g. none can specify the receipt of a new component from the environment, which is not already specified in the original architecture description. Although some of the works surveyed such as Darwin [Georgiadis02] provide a supporting distributed infrastructure, none deal directly with deployment issues. Another ADL for dynamic architectures is Plastik [Joolia05], which benefits from a supporting infrastructure that provides a causal connection between architecture descriptions and the supporting OpenCOM component model [Coulson08]. The Plastik infrastructure provides basic support for local deployment, through its loader component, and supports reconfiguration scripts coded with the Lua programming language. Plastik provides good support for local reconfiguration but does not provide direct support for our requirements above.

### 10.3.3 Overview

The FructOz framework allows Fractal developers to leverage the Oz programming language for the deployment and configuration of Fractal systems (be they programmed in Oz or in some other language). FructOz also allows the development of self-configurable and self-deployable distributed components, i.e. components whose implementation may span several nodes, and which can reconfigure themselves during execution, possibly proceeding to changes in subcomponents and in supporting component packages. FructOz currently comprises:

- A lightweight implementation of the Fractal model in Oz.
- A “dynamic FPath” library providing support for navigating, querying, and monitoring Fractal component structures.

The Fractal implementation which FructOz provides is “lightweight” in the sense that it does not provide all the features and capabilities of the reference Java implementation of Fractal. Also, we have not aimed at this point for a highly optimized implementation. The “dynamic FPath” library has been inspired by the FPath language [David06], and allows navigating and querying a Fractal component structure, much like XPath

allows to navigate and query XML documents. In our case, navigating and querying can take place in a component structure that spans multiple nodes. FructOz can typically be complemented by a library implementing distributed control flow and exception patterns, such as e.g. those documented in the workflow literature [Russell06, Aalst03]. We have not yet built a comprehensive workflow library to complement FructOz, but section 10.3.8 provides some hints on supporting workflow patterns as higher-order abstractions in Oz.

FructOz provides a simple implementation of the reference model presented in section 10.1. The construction of Fractal components is explained below. Component packages are constructed as Oz functors, with package dependencies corresponding to import clauses in functors. Loaders in FructOz correspond to Oz module managers, which support the resolution of import clauses in functors and return components. The installation store is not reified in FructOz: it just corresponds to the Oz memory store since FructOz packages are plain Oz functors, and thus language values. Binding factories are not reified either: they just correspond to the communication capabilities of the Oz infrastructure.

To facilitate the understanding of program fragments below, here are a few indications on the Oz language (see [MozTut] for tutorial material).

- Variables in Oz are logic variables: they can be unbound or bound. Once bound, i.e. once a value has been assigned to it, a variable is immutable. Assignment is denoted by an = sign. Variable declarations typically take the form `X in ...` or `X = E in ...`, where E is some statement.
- Values in Oz can be integer, strings, atoms, records, cells, ports, or procedures. A record takes the following form `r(f1:X1 ... fn:Xn)` where r (the label of the record) and f1 ... fn (the fields of the record) are typically atoms, and X1 ... Xn are variables. Access to a record field is noted with a . sign (thus: `r(f1:X1 f2:X2).f1` returns X1). Special cases of records are pairs, noted with an infix # sign (thus: X#Y corresponds to the pair (X,Y)), and lists, noted with a | sign (thus: `H | T` denotes a list whose head is H and whose tail is T).
- A cell corresponds to a mutable reference. A cell content is a variable. Thus `@C` denotes the content of a cell C, and `C := X` (assignment) updates the content of cell C with variable X.
- A procedure call takes the form `{P A1 ... An}` where P is a procedure name, and A1 ... An are variables denoting the arguments of the call. A procedure can update several of its (unbound) arguments, and thus return several results. A procedure declaration takes the form **`proc`** `{P X1 ... Xn} E` **`end`** where statement E is the body of the procedure. A function, declared with a statement of the form **`fun`** `{F X1 ... Xn E` **`end`**`}` is a special case of procedure which returns a single result. Anonymous procedures and functions can be declared with the anonymous marker $.
- A new concurrent thread is spawned by a statement of the form **`thread`** `S` **`end`**, where S is the statement to be executed in parallel with the current thread.

### 10.3.4 Entities

This section describes how FructOz implements interfaces, components, and bindings.

*Interfaces* are implemented as Oz ports: “An Oz port is an asynchronous channel that supports many-to-one communication. A port P encapsulates a stream S. A stream is a list with unbound tail. The operation Send P M adds M to the end of S. Successive sends from the same thread appear in the order they were sent.” (quotation from [MozTut]). Considering a port more in details, the port itself constitutes the input of the interface, i.e. where clients address their messages, while the stream encapsulated in the port constitutes its output, i.e. where the implementation reads the messages it processes.

Actually, a FructOz interface combines a port with a cell (a mutable reference), so as to allow changing the implementation which processes clients messages at runtime, thus featuring dynamic reconfigurations.

A FructOz *component* is represented by its membrane, a structure holding a mutable set of interfaces. Some interfaces are server interfaces and export a functionality outside the component, while others are client interfaces that import functionalities inside the component. There is no distinction between primitive and composite components in FructOz as in Fractal/Julia: a component may well implement and make direct use of some of its interfaces, and at the same time, have subcomponents bound to other interfaces.

```
%% Create a new empty component membrane
```

```
Comp = {CNew}

%% Create and add an interface to the component membrane
Itf = {INew server [/*tags list*/]}
{CAddInterface Comp Itf}
```

Interfaces can be bound with each other and, eventually, to some native code. Technically, a *binding* between two interfaces is an active pump implemented with an Oz thread waiting for messages on the output stream of one interface and forwarding these messages to the input port of the other interface. Establishing a binding between two interfaces is realized through the BNew primitive.

```
IClient = {INew client [/*tags list*/]} % declare a client interface
IServer = {INew server [/*tags list*/]} % declare a server interface
% bind the client interface to the server interface
Binding = {BNew IClient IServer}
```

Bindings may be chained to connect two interfaces through multiple reconfiguration points. On the endings of a binding chain, the interfaces are connected to native Oz code. From an interface provider perspective, this boils down to define what to do with the messages in the output stream of an interface. The `IImplements` primitive allows one to define the procedure to invoke an every message. The primitive may be invoked multiple times on the same interface to replace the previous procedure with another one.

```
%% How to implement a server interface of a component
Itf = {INew server [/*tags list*/]} % declare the server interface
{IImplements Itf % bind the interface to a procedure
 proc {$ Message}
       /* process the Message here */
 end}
```

A convenient way to implement an interface is to associate it with an Oz object, because the object methods determine the different types of message expected on the interface:

```
Itf = {INew server [/*tags list*/]} % declare the server interface
{IImplements Itf % bind the interface to an object
 {New class $
       meth message1(Parameter1 ...)
             /* process Messages of type message1 */
       end
       ...
 end} init}
```

From an interface user perspective, this involves resolving the Interface into a proxy for the implementation procedure of the interface. The primitive IResolveSync (resp. IResolveAsync) performs the resolution of an interface into a synchronous (resp. asynchronous) proxy.

```
%% How to use a client interface inside a component
Itf = {INew client [/*tags list*/]} % declare the client interface
ItfProxy = {IResolveSync Itf} % resolve the interface into a proxy
% invoke the implementation procedure through the proxy
{ItfProxy message1(Parameter1 ...)}
```

FructOz represents the interfaces, components and bindings presented before as objects inheriting an abstract *Entity* base class, that provides generic navigation facilities through tags filtering: an Entity instance may be tagged with a set of keywords. The set of entities can then be filtered based on their tags. Finally, FructOz exports a set of *primitives* for creating and manipulating those entities; for example, it provides primitives for adding/removing interfaces to/from a component, and retrieving the set of bindings connected to/from a given interface.

### 10.3.5 Components as packaging and deployment entities

A functor is a function which creates (i.e. instantiates and exports) a new module (essentially, a record of Oz values), as a result of linking a module definition to its module dependencies (imports):

$\mathrm{functor} : \{\mathrm{modules}\} \rightarrow \mathrm{module}$.

The choice of the modules to use as imports, i.e. the resolution of imports, is done by a module manager.

A component is implemented as an Oz module exporting the component membrane. The functor which initializes the module thus corresponds to the deployment procedure of the component. Deploying the component means instantiating the module with a module manager. Note that a module (i.e. a component) may be instantiated multiple times by the same module manager. A module manager is thus a component factory.

```
functor ComponentPackage
export membrane: Membrane
define
 %% Create a new empty component membrane
 C = {CNew}
 %% Create the component content (interfaces, implementation, subcomponents,
 %% bindings, etc)
 ...
 %% Now that the component is deployed and ready to be
 %% used, make the membrane available
 Membrane = C
end
```

Note that the module imports in the functor declaration representing the component do not represent the client interfaces of the component (in the example functor ComponentPackage above, there is no import section). The deployment of the component depicted above into a running entity can be achieved in the local running virtual machine with the following deployment primitive example:

```
fun {Deploy PackedComponent}
 %% Create a new module manager
 ModuleManager = {New Module.manager init}
 %% Ask the module manager to apply the functor procedure,
 %% this instantiates the component and exports a reference to its membrane
 C = {ModuleManager apply(PackedComponent $)}
in
 %% Return the membrane exported by the instantiated module
 C.membrane
end
```

### 10.3.6 Distributed environments

The Mozart/Oz platform integrates a distributed programming environment. The platform aggregates several nodes into a unique global store in which objects (values) can be freely shared and accessed. The control of the distribution over the nodes is explicitly handled through the joint use of functors and module managers (i.e. functor executors). An Oz module manager is tied to the machine it has been created on, and thus always deploys modules locally on that machine.

In a centralized, non-distributed environment, components may be deployed on the same machine and in the same virtual machine, using the `{Deploy PackedComponent}` primitive presented in section 10.3.5. To represent a non-distributed environment consisting of a single machine, FructOz provides a Host component acting as a component factory: the Host component exports a factory interface, with a deploy operation relying on the Deploy primitive. A Host component represents a managed node in the reference model of Section 10.1.

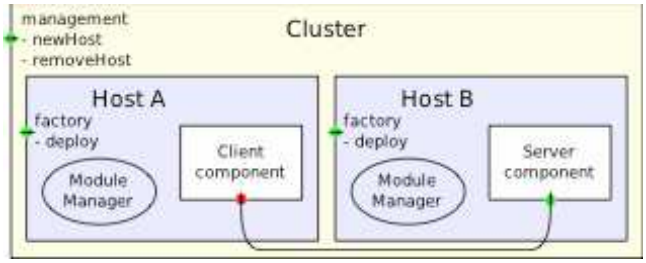


***Figure 11. Distributed environment***

As an example showing how to extend this to a distributed environment, a Cluster component has been implemented to represent a logical set of fully interconnected hosts (see Figure 11). The Cluster component provides operations to add and remove machines from the set of nodes, hence acting as a Host factory. Introspection of the Cluster component thus allows to discover the distributed infrastructure. Distributed deployments may then be parameterized with a cluster parameter identifying the scope of the deployment. Deploying a component on this infrastructure now requires to make explicit the host a component should be deployed on. For this purpose, we provide a new deployment primitive: `{RemoteDeploy Host PackedComponent}` relying on the factory interface of the Host component to dispatch the deployment on the remote hosts. The Deploy primitive remains available to deploy components locally.

```
fun {RemoteDeploy Host PackedComponent}
 %% Get and resolve the factory interface of the Host component
 FactoryItf = {CResolveSyncInterface Host factory}
in
 {FactoryItf newComponent(PackedComponent $)}
end
```

As an implementation detail, the cluster component creates and integrates new hosts in the distributed environment as follows: it remotely starts an Oz virtual machine via an SSH shell command; the new remote Oz virtual machine opens a connection with the source virtual machine, creates a module manager and transmits it (more exactly, a proxy for it) back to the source machine, thus making it available to the cluster component. Finally the cluster component instantiates a host component remotely on the new virtual machine using the proxy for the remote module manager.

```
fun {NewRemoteHost Hostname}
 %% First start a new Oz virtual machine on the remote Host and obtain a %% proxy
to a module manager in this virtual machine
 RemoteModuleManager = {New Remote.manager init(host:Hostname fork:ssh
detach:false)}
 %% Remotely deploy a Host component on the new virtual machine
 HostModule = {RemoteModuleManager apply(PackedComponent $)}
in
 %% Returns the membrane of the new Host component
 HostModule.membrane
end
```

### 10.3.7 LactOz: a dynamic FPath library

FPath is to Fractal architectures what XPath is to XML documents: FPath is a compact notation inspired by XPath for navigating through Fractal architectures and for matching elements according to some predicate. However, contrary to XML documents, Fractal architectures are dynamic and their structure and content may evolve over time. The evaluation of a standard FPath expression produces a static result, only meaningful with respect to the original architecture it has been applied on, and which might not reflect the architecture as it may have evolved since the evaluation of the FPath expression. LactOz provides *Dynamic FPath* expressions that capture the dynamicity of architectures within FPath expressions. LactOz endows FPath expressions with the ability to be dynamically updated following to architecture evolutions.

Updates to dynamic FPath expressions may happen synchronously or asynchronously. Thus FPath variables and expressions that may be static, dynamic with synchronous updates or dynamic with asynchronous updates.

**Predicates** FPath predicates are built using Oz expressions on top of FructOz introspection primitives. We demonstrate here how to build the dynamic FPath expression `GetExternalComponentsBoundFrom` contained in the dynamic FPath library.

Given a component $C$, we want to select all the components $C'$ where there exists a binding from a client interface of $C$ to a server interface of $C'$. The exploration process to achieve this computation starts from $C$, then gets $C$ client interfaces, then $C$ client bindings, then the server interfaces of $C$ client bindings, and finally the components pointed to by these bindings.

```
%% Auxilliary introspection functions
fun {CGetClientInterfaces C} % get the dynamic set of client interfaces of a
component
 %% The kind of an interface cannot change, so we optimize
 %% using a dynamic set filtering with a static filter function
 {SStaticFilter AllItfs (fun {$ I} I.kind == client end)}
end

fun {BGetServerComponent B}
%% get the component owning the interface the binding points to
 {IGetComponent {BGetServerInterface B}}
end

fun {CGetExternalComponentsBoundFrom C}
 ClientItfs = {CGetClientInterfaces C}
 %% SMap maps the Set of (client) Interfaces into a Set of Set of Bindings,
 %% that we flatten thanks to SUnion into a Set of Bindings
 ClientBindings = {SUnion {SMap ClientItfs (fun {$ I} {IGetBindingsFrom I} end)}}
 %% and finally, map the Set of Bindings into a Set of Components
 {SMap ClientBindings (fun {$ B} {BGetServerComponent B} end)}
end
```

Now suppose you want to filter the set of components obtained with some predicate. For example, we want only those having a sub-component tagged with the tag “interesting”, you might build and use the following predicate:

```
%% filter the dynamic set of components generated with the previous function
{SFilter {CGetExternalComponentsBoundFrom C}
 fun {$ C}
        %% get the sub-components of C
        Children = {CGetSubComponents C Context}
 in
        %% only keep components for which the sub-component set contains at least
```

```
            %% one child component tagged with interesting, i.e. for which the %% sub-
            component set filtered with respect to the interesting tag is not empty
            {BNot {SIsEmpty {SFilterHasTag Children interesting}}}
      end}
```

The predicate here is dynamic as its prototype is: $\mathcal{C} \rightarrow \mathbb{B}$ where the boolean might change if the component is added or removed some children or if some children are tagged or untagged with “interesting”.

LactOz comes with a general set of primitives such as `BNot`, which creates a dynamic boolean computing a logical not of another dynamic boolean, `SIsEmpty`, which tests the emptiness of a set, `SMap`, `SUnion`, `SFilter` etc. All these primitives can be tuned to be static or dynamic, synchronous or asynchronous.

## 10.3.8 Case studies

In this section, we present examples of dynamic architectures illustrating how our framework supports the requirements we identified in section 10.3.1.

### Parameterized architectures

This example shows how to describe highly parameterized architectures. For this purpose we present an architecture composed of two sets of components and where a parameter defines the interconnection scheme between these two sets (see Figure 12).

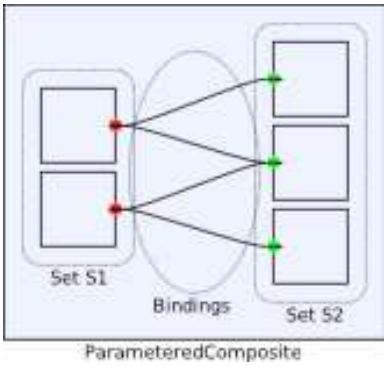


***Figure 12. Parameterized architectures: interconnection scheme***

The parameters for this architecture are:

- the size of the first set of components
- the size of the second set of components
- an interconnection scheme function: $f : \mathcal{S}\{\mathcal{C}\} \times \mathcal{S}\{\mathcal{C}\} \rightarrow \mathcal{S}\{\mathrm{Desc}(\mathcal{B})\}$ where $\mathrm{Desc}(\mathcal{B})$ represents a descriptor for a binding (in our scenario, a descriptor is a pair of interfaces IFrom#ITo).

```
fun {ParameterizedComposite N1 N2 BindingScheme}
 functor $
 export Membrane
 define
       C = {CNew nil}
       %% Create the first set S1 with N1 instances of component C1.
       S1 = {SNew} % create a new empty set S1
       for I in 1..N1 do
             SubComp = {Deploy C1} % deploy a component C1 locally
       in
             {S1 add(SubComp)} % add the new component into the set S1
```

```
        end
        %% Create S2 similarly to S1
        ...
        %% Invoke the interconnection scheme function: generate a list of
        %% binding descriptors.
        LDescs = {BindingScheme S1 S2}
        %% Translate these descriptors into real bindings.
        LBindings = {List.map LDescs
               fun {$ IFrom#ITo}
                      {BNew IFrom ITo}
               end}
        Membrane = C
 end
end
```

A trivial example of interconnection scheme is the full interconnection scheme which realizes the complete two-party graphs.

```
fun {FullInterconnect S1 S2}
 L = {NewCell nil} % create a new empty descriptor list
in
 %% for all pairs (CFrom, CTo) in (S1 x S2)
 {S1 forAll(
        proc {$ CFrom}
               {S2 forAll(
                      proc {$ CTo}
                             % locate the client interface
                             IFrom = {CGetInterface CFrom [/*tags list*/]}
                             % locate the server interface

                             ITo = {CGetInterface CTo [/*tags list*/]}
                             Desc = IFrom#ITo % make the descriptor
                      In
                             % prepend the new descriptor to the list
                             L := Desc | @L
                      end)}
        end)}
 @L % return the list of descriptor
end
```

## Synchronization and workflows

In the example depicted in the previous section, the sub-components are deployed sequentially, one after the other. We describe here how to describe and integrate distributed synchronizations. For this purpose, we define a Barrier synchronization pattern in the `Barrier` procedure, analogous to a parallel AND workflow pattern. The procedure is used to parallelize and synchronize on the completion of the deployment of $N_2$ instances of component $C_2$. Expressing the synchronization pattern as a procedure allows to consider the synchronization pattern as a parameter of an architecture (such as the `BindingScheme` parameter of the example shown previously). We can in the same way encapsulate exception handling patterns in higher-order procedures.

```
%% Barrier synchronization pattern
proc {Barrier P N}
 Bar = {Tuple.make barrier N}
 for I in 1..N do
        thread
```

```
                {P} % invoke the procedure
                Bar.I = true % ready signal once the procedure is over
        end
 end
 {Record.forAll Bar Wait} % wait for all N ready signals
end

%% Apply the barrier pattern to deploy N2 instances of C2 components
{Barrier
 proc {$}
        SubComp = {Deploy C2}
 in
        {S2 add(SubComp)}
 end
 N2}
```

## Lazy deployments

In this example we show how to implement lazily deployed components. We consider 3 levels of deployment: (i) level 0: the component is represented by a lazy variable and is not deployed at all; (ii) level 1: the component membrane is deployed, which allows to introspect its interfaces and immediate sub-components; the implementation of the component is not deployed yet. (iii) level 2: the component is fully deployed.

Contrary to the transition from level 0 to level 1, which is atomic, the transition from level 1 to level 2 may contains several intermediary states with partial deployments of the component. For instance, a composite component with many sub-components may be somewhere between layer 1 and layer 2 with only a few of its sub-components deployed.

This relies on lazy bindings instantiated via the `BNewLazy` primitive. Lazy bindings require that the client interface is determined, but allow the server interface to be lazily determined. The `BNewLazy` primitive differs from `BNew` in that the server interface will only be made needed, thus deployed, on usage of the binding (for either functional or control purpose such as introspection).

```
%% Lazily deployed implementation
Itf = {INew server [tags ...]}
{IImplements Itf
 {ByNeed fun {$}     % Implementation will be lazily instantiated
        {New class $ ... end}}}

%% Lazily deployed component
Client = {ByNeed fun {$} {Deploy LazyClient} end}

%% Lazily deployed binding between a Client component and a Server
%% component
B = thread
        %% Wait until someone needs and thus triggers the deployment of the     %%
 Client component. Once this has happened, get the client
        %% interface
        IFrom = {CGetInterface {WaitQuietValue Client} [client]}
        %% Create a lazy reference to the server interface
        ITo = {ByNeed fun {$} {CGetInterface Server [service]} end}
 in
        %% Create a lazy binding between Client (now determined) and Server  %%
        (might still be lazy)
        {BNewLazy IFrom ITo}
 end
```

## Self-configurable architecture

We present now how to extend the example of parameterized architecture (presented earlier) with dynamic parameters. The original example is a parameterized architecture `{ParameterizedComposite N1 N2 BindingScheme}` where N1 and N2 are two constant integers and where `BindingScheme` is a function which generates a set of binding descriptors given two sets of components.

The architecture dynamicity with respect to $N_1$ means: (i) to dynamically deploy or undeploy instances of $C_1$, and (ii) to dynamically add and remove bindings between $C_1$ instances and $C_2$ instances. For this purpose, the interconnexion scheme takes the dynamicity into account in generating a dynamic set of binding descriptors. The composite component listens to this dynamic set of descriptors and translates new (resp. removed) set entries into binding creations (resp. removal).

The dynamic set of descriptors is implemented as an object integrated into an event-based system. The object is parametered with two dynamic sets S1 and S2 and is thus listening and reacting to these sets updates. For example, in response to an update event related to S1 such as a removal of elements, method `removeS1(SElements)` is invoked, which adjusts the state of the descriptor set. Adjustments to the descriptor set made in `removeS1` generate new events in cascade propagated to listeners of this set, such as a dynamic architecture.

```
%% How to construct a dynamic set
class DynamicBindingDescSet from ImplicitSet
 meth init
        %% Listen to S1 updates
        self.s1_listener = {New SetListenerForwarder init(self S1 sync addS1
removeS1)}
        {S1 listen(self.s1_listener)}
        ...
 end
 ...
 meth removeS1(SElements) % On removal of elements from S1:
        lock % mutual exclusion to prevent race conditions
               %% Update our local view of S1
               s1 := {FSet.removeAll @s1 SElements}
               %% Remove all descriptors referencing components that are
               %% removed from S1
               {FSet.forAll SElements
                      proc {$ CFrom}
                             Set,filterInPlace(
                                    fun {$ Desc}
                                           ({IGetComponent Desc.iFrom} == CFrom)
                                    end)
                      end}
        end
 end
 ...
end

%% How to listen and react to events generated by the dynamic descriptor ser
class MyDescSetListener from SetListener
 ...
 meth remove(SElements) % On removal of descriptors:
        {FSet.forAll SElements
               %%  destroy the bindings corresponding to removed descriptors
               proc {$ Desc}
                      {BBreak Desc.binding}
               end}
 end
end
```

```
%% Create and activate our listener on DescSet
{DescSet listen({New MyDescSetListener init(C DescSet sync)})}
```

## Deployment scenarios

We present here how to describe different deployment strategies. We consider the deployment of a set of identical components on a cluster of nodes and organized as subcomponents of the composite component described as follows. The size of the set of components drives the complexity of the overall deployment process.

```
fun {CompositePackage Cluster N DeployProc}
        functor $
        export Membrane
        define
                Comp = {CNew}
                {DeployProc SubcomponentPackage Cluster N Comp}
                Membrane = Comp
        end
end
```

The composite component description is parameterized by a deployment procedure which is responsible of the deployment of the $N$ identical subcomponent of the composite.

**Sequential deployment**. Our first deployment strategy consists in deploying all subcomponents sequentially from a single centralized controller as follows:

```
proc {DeploySeq CompPackage Cluster N Parent}
 NextRoundRobin = {MakeRoundRobin Cluster}
 for I = 1..N do
       Host = {NextRoundRobin}
       NewComp = {RemoteDeploy Host CompPackage}
       {CAddSubComponent Parent NewComp}
 end
end
```

**Centralized asynchronous parallel deployment**. Adding uncontrolled parallelism to the previous strategy is trivial thanks to Oz:

```
proc {DeployPar CompPackage Cluster N Parent}
 NextRoundRobin = {MakeRoundRobin Cluster}
 for I = 1..N do
       thread
             Host = {NextRoundRobin}
             NewComp = {RemoteDeploy Host CompPackage}
             {CAddSubComponent Parent NewComp}
       end
 end
end
```

**Centralized synchronous parallel deployment.** Relying on the barrier synchronization described in section 10.3.8, we build a centralized deployment strategy that spawns and synchronizes on concurrent deployments as follows:

```
proc {DeployParallel CompPackage Cluster N Parent}
 NextRoundRobin = {MakeRoundRobin Cluster}
 %% Deployment script
 proc {DeployProc}
       Host = {NextRoundRobin}
       NewComp = {RemoteDeploy Host CompPackage}
       {CAddSubComponent Parent NewComp}
```

```
  end
in
  %% Execute and synchronize on the scripts execution
  {Barrier DeployProc N}
End
```

**Tree distributed deployment.** All deployment strategies presented up to now are executed on a single centralized controller node. Here is how to build a distributed deployment process based on a tree distribution strategy. The deployment process is initiated on the root node of the tree. Each node of the tree locally deploys one instance of the component; each node also initiates the deployment process on all their branches and synchronizes on them.

```
NextRoundRobin = {MakeRoundRobin Cluster}
proc {DeployTree CompPackage Arity Depth Parent}
  functor DistributedDeployProc
  export Membrane
  define
        if (Depth > 0) then
              {DeployTree CompPackage Arity (Depth - 1) Parent}
        end
        Membrane = {Deploy CompPackage}  % deploy component locally
  end
  proc {DeployProc}
        Host = {NextRoundRobin}
        NewComp = {RemoteDeploy Host DistributedDeployProc}
        {CAddSubComponent Parent NewComp}
  end
in
  {Barrier {MakeCopyList DeployProc Arity}}
end
```

## 10.3.9 Evaluation

### Microbenchmarks

We show here a performance evaluation to compare the deployment process and the cost of remote method invocations on SmartFrog/Java RMI, Julia/Fractal RMI and FructOz/Mozart. Additionally we evaluate the deployment process of a “FructOz/Julia bridge” (see the details at the end of this section). For this purpose, we measure the latency of the deployment of the distributed composite component depicted in Figure 13 and we also evaluate the cost of a synthetic remote method invocation between the client and the server subcomponents.

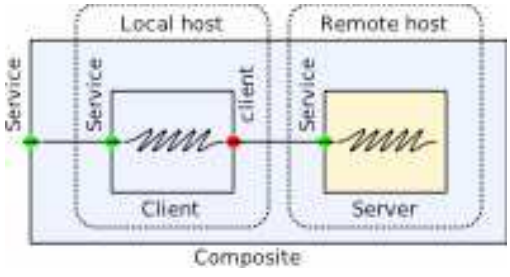


***Figure 13. Simple distributed component***

The experiments have taken place on the Grid’5000 infrastructure, and the measures are realized on dual-opteron 252 (2.6GHz) machines with 4Gb of memory and all interconnected with 1Gb network interfaces. We used SmartFrog 3.12.000 and Fractal 2.0.1, running on the Sun Java virtual machine v1.6.03, and the

Mozart/Oz virtual machine v1.3.2. To evaluate the remote method invocation cost, we evaluate the time required to complete 10000 synthetic method calls. The garbage collector is manually triggered between measures sequences so as to minimize its impact on the performance. The experiments have been repeated 30 times to finally report averaged measures. Table ***1*** summarizes the results of this evaluation.

The deployment of the distributed component is more efficient on FructOz, which may be explained because FructOz does not need any descriptor parsing as this is the case for SmartFrog and Fractal ADL. In fact, SmartFrog reported during the experiments an average descriptor parsing time of 150 ms, which is about half the time of our SmartFrog deployment process. Remote method invocations is more efficient on the Mozart/Oz platform. This may be explained because the Mozart/Oz marshaler is directly implemented and optimized in C++ while Java and Fractal RMI marshalers are implemented in Java and use the Java reflection. Additionally, Mozart/Oz heavily relies on lazy marshalling, and serializes complex structures when this is required by a remote process only. Overall, these microbenchmarks show that the basic operations in our platform compare favorably with other environments.

*FructOz/Julia bridge*: We extended the FructOz framework with an Oz/Java bridge that provides FructOz with the ability to drive Java tasks. Moreover, the bridge exports specific hooks to manipulate the Fractal/Julia component model in the Oz world. This way, FructOz constitutes a deployment engine for

***Table 1: Deployment and remote invocation costs***

| | Component deployment | Remote invocation |
|---|---|---|
| SmartFrog/Java RMI | 295.5 ms | 0.097 ms |
| Julia/Fractal RMI | 159.5 ms | 0.099 ms |
| FructOz | 146.3 ms | 0.0048 ms |
| FructOz/Julia Bridge | 262.3 ms | N/A |

Fractal/Julia components. The Oz/Java bridge is built as a set of XML-RPC handlers, and we used the Apache XML-RPC v3.1 Java implementation and the Oz XML-RPC client modules for the experimentation.

The deployment of the distributed component in this configuration thus adds the cost of XML-RPC invocations and Fractal/Julia primitive executions to the latency measured on the regular FructOz environment, which explains the higher deployment cost (262.3 ms). Finally, remote method invocations could happen either as Oz invocations or as Fractal RMI invocations, thus mixing the performance of both configurations. This demonstrates the applicability of the FructOz framework on heterogeneous non-Oz environments. More concretely, driving the deployment of legacy applications such as J2EE servers would require the design of component wrappers for these legacy applications (see e.g. [Abdellatif05] for details).

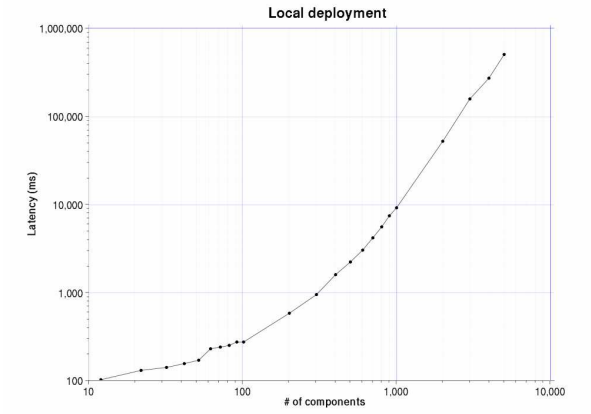


***Figure 14. Local deployment evaluation***

## Local deployments

In the following we evaluate the scalability of local single-machine deployments on the Mozart/Oz platform. As an evaluation base, we deploy a composite component containing a single client subcomponent bound to a number of server subcomponents. The number of server subcomponents thus constitutes the size of the deployed component. All measures are repeated 3 times and averaged.

Results are presented in Figure 14 show that the deployment cost increases with the number of components deployed. This behavior may to be correlated with the garbage collector of the Oz virtual machine. Indeed we reported an issue preventing the correct collection of generated values, leading to an increase in the heap size, thus slowing down the collection mechanism. Moreover, the FructOz framework has not been designed with performance in mind, and circumvents a number of limitations of the current implementation of the Oz virtual machine with some design and performance overhead. Thus there is room for significant optimizations.

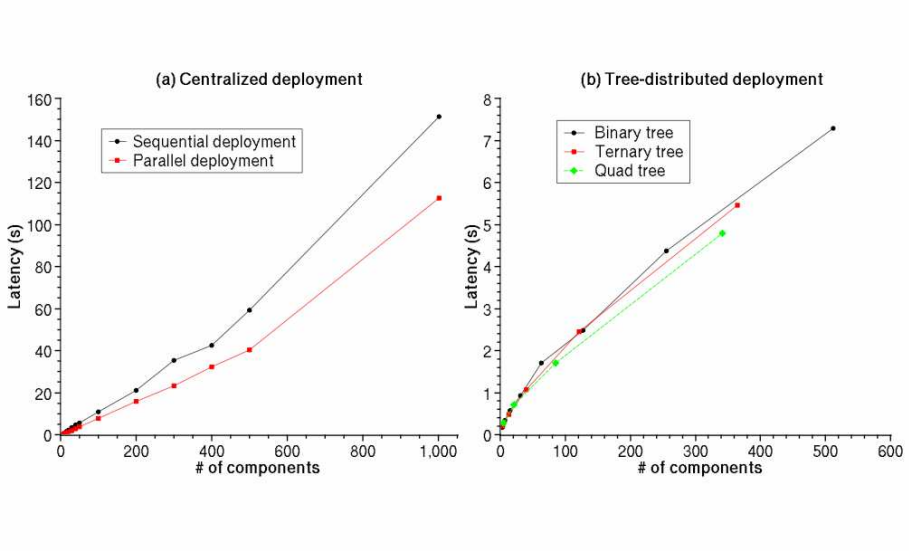


***Figure 15. Distributed deployments evaluation***

## Distributed deployments

The following experiments consist in deploying increasing numbers of components on a distributed environment made of a cluster of 16 machines of the Grid'5000 infrastructure (described more in details in section 10.3.9). The machine on which a new component is deployed is determined with a simple round-robin policy over the machines available in the cluster. In Figure 15(a) and Figure 15(b) we demonstrate the gains that can be obtained by defining the appropriate deployment workflow. Figure 15(a) compares two simple workflows, executing on a single machine: the first one is just the sequential deployment of a number of components on different machines; the second one is the parallel deployment of the same number of components on the same number of machines. Just increasing the parallelism provides an interesting improvement.

A more drastic speed-up is obtained when changing the workflow to a distributed one. In Figure 15(b), we show the result of distributing the deployment workflow on a tree of machines. We experimented with different forms of trees, as reported in the figure. One can notice a huge improvement (5× speedup) over a centralized sequential workflow. Interestingly, changing the workflow configurations with our framework, is only a matter of changing parameters in a higher-order procedure call.

Note that in Figure 15(b) we have not been able to obtain results beyond 600 components, because of instabilities of the current Oz virtual machine. These are due, we believe to some interplay between the garbage collector and distribution, but we have not been able at this point to ascertain the exact source of the problem.

### 10.3.10 Integration and plans

FructOz is an experimental system, currently not integrated within the VO management system. We intend to extend FructOz to become a deployment engine for Grid4All applications on VO nodes. This will mainly involve extending the prototype FructOz/Julia bridge to allow manipulating Fractal/DCMS components through the DCMS API. Application developers will then be able to concisely express various deployment strategies, such as those discussed in section 10.3.8. Furthermore, we intend to study and collect patterns for deployment and use them as the basis for more specialised FructOz primitives. In the long term, we plan to investigate exposing the full DCMS sensing and actuating capabilities through an Oz-based framework in order to combine the scalability and robustness supported by the former with the usability benefits of the latter.

# 11. Conclusion

This deliverable has discussed the core VO management system, a set of interacting services built using the Fractal component model and the DCMS overlay-based platform. Most of the services have prototype implementations. The main areas of future work involve implementing the reservation management and remote node addition services, and integrating FructOz as a deployment engine.

# 12. References

[Aalst03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1), 2003.

[Abdellatif05] T. Abdellatif, J. Kornas, and J.B. Stefani. J2EE Packaging, Deployment and Reconfiguration Using a General Component Model. In Component Deployment, 3rd International Working Conference, CD 2005, volume 3798 of Lecture Notes in Computer Science. Springer.

[Arshad07] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3), 2007.

[Badonnel04] A. Keller and R. Badonnel. Automating the Provisioning of Application Services with the BPEL4WS Workflow Language. In *15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM*, number 3278 in Lecture Notes in Computer Science. Springer, 2004.

[Bastarrica98] M. C. Bastarrica, A. A. Shvartsman, and S. A. Demurjian. A Binary Integer Programming Model for Optimal Object Distribution. In *2nd Int. Conf. On Principles Of Distributed Systems. OPODIS 98*. Hermes, 1999.

[Bouchenak05] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, N. De Palma, V. Quéma, and J.B. Stefani. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*. IEEE Computer Society, 2005.

[Bouchenak06] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton. Autonomic Management of Clustered Applications. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2006.

[Bradbury04] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004*. ACM, 2004.

[Bruneton06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[Claudel06] B. Claudel, N. De Palma, R. Lachaize, and D. Hagimont. Self-protection for Distributed Component-Based Applications. In *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006*, number 4280 in Lecture Notes in Computer Science. Springer.

[Coulson08] G. Coulson, G. S. Blair, P. Grace, F. Taïani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1), 2008.

[Coupaye00] T. Coupaye and J. Estublier. Foundations of enterprise software deployment. In *Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2000.

[David06] P.C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *Software Composition, 5th International Symposium, SC 2006*, number 4089 in Lecture Notes in Computer Science. Springer, 2006.

[Deng05] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. S. Gokhale. DAnCE: A QoS-Enabled Component Deployment and Configuration Engine. In *Component Deployment, 3rd Int. Working Conference, CD 2005*, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[Dolstra04] E. Dolstra, M. de Jonge, and E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *18th Conference on Systems Administration (LISA 2004)*. USENIX, 2004.

[Flissi06] A. Flissi and P. Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In *OTM Confederated International Conferences, Grid computing, high performAnce and Distributed Applications (GADA 2006)*, volume 4276 of *Lecture Notes in Computer Science*. Springer, 2006.

[Garlan04] D. Garlan, S.W. Cheng, A.C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10), 2004.

[Garlan00] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[Georgiadis02]I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *1st Workshop on Self-Healing Systems, WOSS 2002*. ACM, 2002.

[Goldsack03] P. Goldsack. SmarFrog: Configuration, Ignition and Management of Distributed Applications. Technical Report http://www-uk.hpl.hp.com/smartfrog, HP Research Labs, 2003.

[OMG06] Object Management Group. *Deployment and Configuration of Component-based Distributed Applications Specification*, 2006.

[Hall99] R. Hall, D. Heimbigner, and A. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *21st International Conference on Software Engineering (ICSE'99)*, pages 174–183, Los Angeles, CA, May 1999.

[Haridi04] S. Haridi and N. Franzen. Tutorial of Oz, 2004. Available at the URL: `http://www.mozart-oz.org/documentation/tutorial/index.html`.

[Heydarnoori06] A. Heydarnoori, F. Mavaddat, and F. Arbab. Towards an Automated Deployment Planner for Composition of Web Services as Software Components. *Electr. Notes Theor. Comput. Sci.*, 160, 2006.

[Hoek99] A. van der Hoek. Configurable software architecture in support of configuration management and software deployment. In *21st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1999.

[Joolia05] A. Joolia, T. Batista, G. Coulson, and A. Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In *5th IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE Computer Society, 2005.

[Kephart03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), 2003.

[Kichkaylo04] T. Kichkaylo and V. Karamcheti. Optimal Resource-Aware Deployment Planning for Component-Based Distributed Applications. In *13th International Symposium on High-Performance Distributed Computing (HPDC 2004)*. IEEE Computer Society, 2004.

[Kon05] F. Kon, J. R. Marques, T. Yamane, R. H. Campbell, and M. D. Mickunas. Design, implementation, and performance of an automatic configuration service for distributed component systems. *Software - Practice and Experience*, 35(7), 2005.

[Lan05] L. Lan, G. Huang, L. Ma, M. Wang, H. Mei, L. Zhang, and Y. Chen. Architecture Based Deployment of Large-Scale Component Based Systems: The Tool and Principles. In *Component-Based Software Engineering, 8th International Symposium (CBSE)*, volume 3489 of *Lecture Notes in Computer Science*. Springer, 2005.

[Liu06] Y. Liu and S. Smith. A Formal Framework for Component Deployment. In *20th ACM Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[Malek05] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. In *Component Deployment, Third International Working Conference, CD 2005*, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[Mancinelli06] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. IEEE Computer Society, 2006.

[Mikic02] M. Mikic-Rakic and N. Medvidovic. Architecture-Level Support for Software Component Deployment in Resource Constrained Environments. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD'02)*, number 2370 in Lecture Notes in Computer Science. Springer, 2002.

[Moreno02] M. D. Rodriguez-Moreno and P. Kearney. Integrating ai planning techniques with workflow management system. *Knowledge-Based Systems*, 15(5-6), 2002.

[Puhlmann05] F. Puhlmann and M. Weske. Using the Pi-Calculus for Formalizing Workflow Patterns. In *Business Process Management, 3rd Int. Conference, BPM 2005*, number 3649 in Lecture Notes in Computer Science. Springer, 2005.

[Quema04] Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet, and Serge Lacourte. Asynchronous, Hierarchical, and Scalable Deployment of Component-Based Applications. In *Component Deployment, 2nd Int. Working Conference, CD 2004*, number 3083 in Lecture Notes in Computer Science. Springer, 2004.

[Rowanhill07] J. C. Rowanhill, G. S. Wasson, Z. Hill, J. Basney, Y. Kiryakov, J. C. Knight, A. Nguyen-Tuong, A. S. Grimshaw, and M. Humphrey. Dynamic System-Wide Reconfiguration of Grid Deployments in Response to Intrusion Detections. In *High Performance Computing and Communications, 3rd International Conference, HPCC 2007*, number 4782 in Lecture Notes in Computer Science. Springer, 2007.

[Russell06] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Workflow Exception Patterns. In *Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006*, number 4001 in Lecture Notes in Computer Science. Springer, 2006.

[Tibermacine07] C. Tibermacine, D. Hoareau, and R. Kadri. Enforcing Architecture and Deployment Constraints of Distributed Component-Based Software. In *10th International Conference Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *Lecture Notes in Computer Science*. Springer, 2007.

[Valetto03] G. Valetto and G. E. Kaiser. Using Process Technology to Control and Coordinate Software Adaptation. In *25th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2003.

[Wang06] Wang, X., Li, W., Liu, H., and Xu, Z. 2006. A Language-based Approach to Service Deployment. In *Proceedings of the IEEE international Conference on Services Computing* (September 18 - 22, 2006). IEEE Computer Society, Washington, DC, 69-76.

**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.