Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Reproducible evaluation of a service/resource discovery system D3.2.8

Due date of deliverable: November 30th, 2008
Actual submission date: January 13th,2009

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP3.2
Task number: T3.2.3

Responsible institution: CNR/ISTI
Editor & and editor's address: Massimo Coppola
CNR/ISTI
Via G. Moruzzi 1
56124 PISA
Italy

Version 1.0 / Last edited by Massimo Coppola / January 13th, 2009

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

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	11/12/08	Massimo Coppola, Emanuele Carlini	CNR/ISTI	Initial outline
0.3	20/12/08	Massimo Coppola, Guillaume Pierre, Jeff Napper	CNR/ISTI, VUA	Added VUA contribution
0.9	12/01/09	Massimo Coppola, Laura Ricci, Patrizio Dazzi, Emanuele Carlini, Susanna Martinelli	CNR/ISTI	Final version for internal review
1.0	20/01/09	Massimo Coppola	CNR/ISTI	Minor corrections after internal review

Reviewers:
Jan Stender (ZIB), Samuel Kortas (EDF)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.2.3	Design and implementation of a service/resource discovery system	CNR*, VUA

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

This deliverable presents the current state Service/Resource Discovery System (SRDS), and its evaluation in terms of performance, reliability, robustness. The SRDS will offer to applications and other components of XtreemOS the capability of searching for, and selecting, services and resources.

In the context of XtreemOS, a reproducible evaluation encompasses diverse aspects, namely (1) evaluation of performance and scalability, (2) reliability of the functional behaviour, and (3) deployment robustness within complex, distributed software system. This deliverable covers those three aspects, also building on top of the performance ans scalability evaluation reported in previous deliverable D3.2.4 [14].

The first chapter surveys the SRDS design and SW architecture, whose main component modules are the Resource Selection Service (RSS, realized by the VUA team) and the Application Directory Service (ADS, realized by the CNR team). The SRDS SW architecture is distributed as three different packages of the XtreemOS distribution. We report the current implementation status of those modules, the functionalities they provide in the first public release of XtreemOS, we discuss recent choices made in the development, as well as the procedures followed for testing and packaging. This addresses point (3), motivating and putting in context the following chapters.

The second chapter addresses point (1) above, by reporting additional performance evaluations for ADS and RSS to validate development choices made after M18. A full evaluation of both components at that time had been already performed over diverse platforms (simulators, clusters, Grid5000, DAS, PlanetLab), and was reported in deliverable D3.2.4. We update those results by comparing on a large Grid the performance and scalability of ADS when using the Overlay Weaver Distributed Hash Table (instead of the Bamboo one), and by comparing the RSS scalability against a purely DHT-based resource discovery system.

The third chapter describes the testing methodology employed, which mixes Black-Box testing and Unit Testing. Black-Box tests can be easily employed by end-users, in automated package testing and also *in vivo*, on a running distributed system, while the Unit test suite is mainly useful to developers, and helps tracking functional bugs resulting from changes in the implementation of ADS internal modules. It follows from point (2) above that managing the development and deployment complexity, and certifying functional behaviour of code is an essential feature for a Grid-distributed OS.

Summing up, this deliverable shows that the SRDS prototype performs consistently over diverse platforms, from the performance and functional viewpoints, and it explains how to exploit the testing mechanisms to ensure these properties when modifying and deploying the SRDS code.

Glossary

ADS	Application Directory Service
AEM	Application Execution Management
API	Application Programming Interface
DAS-3	Distributed ASCI Supercomputer
DHT	Distributed Hash Table
DMS	Data Management Services
HTTP	HyperText Transfer Protocol
ID	Identifier
IML	Information Management Layer
IP	Internet Protocol
JSDL	Job Submission Description Language
JSON	JavaScript Object Notation
libDB	Berkeley Database library
M18	Month 18 (December 2007)
M24	Month 24 (June 2008)
M30	Month 30 (December 2008)
MAPI	Module-specific API
NodeID	Node Identifier
OW	Overlay Weaver
OS	Operating System
P2P	Peer-To-Peer
QoS	Quality of Service
QP	Query & Provide
RSS	Resource Selection Service
SRDS	Service/Resource Discovery System
SSL	Secure Sockets Layer
SW	Software
TCP/IP	Transmission Control Protocol / Internet Protocol
TCP	Transmission Control Protocol
TTL	Time-To-Live
UUID	Universally Unique Identifier
VO	Virtual Organization
WP	Work Package
XML	eXtensible Markup Language

Contents

Glossary **2**

1 Service/Resource Discovery System Design **5**
- 1.1 Document Structure 6
- 1.2 Service/Resource Discovery System Design 7
- 1.3 Application Directory Service (ADS) 9
  - 1.3.1 Development Status 10
- 1.4 The DHT layer 12
  - 1.4.1 Overlay Weaver 13
- 1.5 Resource Selection Service (RSS) 14
- 1.6 Development activity up to M30 16
  - 1.6.1 Design of the Integration with Overlay Weaver 17
  - 1.6.2 Testing and Packaging with XtreemOS 20

2 Performance Evaluation **23**
- 2.1 DHT Layer Evaluation – Bamboo vs OverlayWeaver 23
- 2.2 RSS evaluation with respect to a DHT-based approach. 27

3 Functionality Evaluation **30**
- 3.1 Black-box Testing 30
- 3.2 Unit Testing 32

4 Conclusions **35**

Bibliography **36**

Chapter 1

Service/Resource Discovery System Design

This deliverable summarizes current and previous results concerning the evaluation of the discovery services for resources and services which belong to the WP3.2 of XtreemOS. *Reproducible* evaluation of a service within XtreemOS encompasses several aspects of its implementations

1. performance should be predictable and acceptable with larger and larger platform sizes and communication overheads, in order allow execution on future Grids

2. functionality should be easy to test and modify, to allow integration of more and more complex subsystems into an interoperable set of operating system services

3. integration with other system components should be standardized and easy to perform, to allow a robust set of services to be developed, and to ease system deployment for the end-user.

Point 1 above is the obvious first target in designing high availability systems, and for the Service/Resource Discovery System (SRDS) it has been deeply addressed already in the design and early development stages, as documented by the large amount of tests over simulators, clusters and large scale Grids reported in previous deliverable D3.2.4 [14]. These tests already proved that the single components of the SRDS provide the expected performance level to XtreemOS systems and do scale well, thus being of practical use on even larger platforms.

The same deliverable [14] also discussed the properties of modularity and flexibility of the SRDS design that are needed to meet point 2 above, and to bootstrap future open source development of the system.

These results are only briefly recalled and summarized in present document, which focuses instead on the other properties that are implied by "reproducible".

As it was evidenced during XtreemOS system development, especially in the system integration and debugging phase around M24, any component of XtreemOS that is capable of amazing performances in isolation, but that is buggy, hard to build, deploy or configure, it is almost useless for the system, and cannot be practically used and evaluated.

Therefore, this deliverable reports additional results which concern performance aspects only if previously untested, it surveys functionality testing as a tool to ensure robustness and reproducible functional behaviour, and in addition it covers functionalities which simplify the SRDS development, testing and repackaging activities.

We discuss choices we made during the SRDS development in order to speed up system integration, or, conversely, to prepare some changes and upgrades in the system architecture that will soon be needed to ensure reproducibility of system behaviour in the real world, according to all the three meanings we listed.

1.1 Document Structure

The document presents the design of the SRDS starting with its overall organization.

Chapter 1 summarizes the status of the SRDS prototype at M30, discussing the development roadmap and the open research perspectives.

Section 1.2 discusses SRDS design requirements, goals and functionalities based on the decomposition of the SRDS into two cooperating services, the Resource Selection Service (RSS) and the Application Directory Service (ADS), as they have been implemented in the first public release of XtreemOS.

Up to now the SRDS satisfies the requirements gathered from WP3.3 ([7] related to the Application Execution Management) and WP3.4 ([8] concerning the Data Management Services and the Grid-enabled XtreemOS File System) as main test cases in the specification design, and takes into account the overall VO Management and Security infrastructure [9, 5, 6] designed by WP2.1 and WP3.5.

Section 1.3 discusses the specific architecture of the ADS component of the SRDS, whose main layers are the client-oriented interface, a middle-tier providing query resolution algorithms and information encoding, and a low-end tier level which interfaces to one or more DHT implementation libraries. Section 1.4 discusses the issues confronted in implementing the DHT layer, and motivates the choice to switch the default DHT library from Bamboo to OverlayWeaver. Section 1.5 recaps the basic principles and implementation choices of the Resource Selection Service, the module providing the first selection of nodes during resource discovery queries. Section 1.6 summarizes development up to M30, in particular the testing activity and the introduction of automated packaging and testing, also in order to improve system stability.

Chapter 2 reports new performance results which aim at confirming the good scalability and performance results already reported in deliverable D3.2.4 [14], by extensive measurements on simulators, cluster and Grid platforms. The chapter focuses first on the comparison between the Bamboo DHT and the OverlayWeaver DHT, to verify the assumption that the SRDS system can adapt to different DHT implementation layers with no critical loss of performance and reliability. Then it moves on to a comparison between the RSS and a DHT-based support for range queries, showing that for typical resource location queries over a large platform, the RSS leads to a more balanced overhead and avoids critical load spots.

Chapter 3 surveys the mechanism adopted to provide functional verification of the SRDS, to ensure ease of checking and enhance SRDS reliability when integrated within a complex OS like XtreemOS. We employed both Black-Box tests, and Unit tests. Black Box tests can be easily run (even by an end user) in order to verify that the SRDS behaviour satisfies the interfaces requirements defined for a specific client module. Unit testing is adopted for the internal SRDS modules, in order to ensure a safer and more modular development activity.

Chapter 4 summarizes the results presented so far and draws conclusions.

1.2 Service/Resource Discovery System Design

The Service/Resource Discovery System has to provide higly scalable service and resource location functionalities (including Distributed Directory Services as a special case) to a wide range of different *clients*. We define as potential clients all other modules of XtreemOS as well as XtreemOS-aware applications. The set of clients is thus heterogeneous and expandable with time. Flexibility in defining client interfaces and protocols, different query semantics and resolution algorithms is a concrete need. The basic requirements of the SRDS can be summarized as follows.

- The SRDS continuously receives many different kind of data, both static and dynamically variable, associated to nodes, keys, applications and services.
- Different clients may require different communication frameworks, and will most probably rely on different communication protocols (single or multiple phase, blocking, asynchronous or multithreaded interaction) and data marshaling conventions.
- In order to provide organized information to other XtreemOS components, the SRDS has to answer queries with diverse semantics, ranging from simple key-based queries to range-based queries over *dynamic* attributes (i.e. values that are dynamically variable at run-time).
- SRDS has to provide a Quality of Service level that is customizable according to the needs of each SRDS client in several aspects. We cite as examples
  - the performance level, which must scale with the platform, and is in a tradeoff with the complexity of resolving information queries,

  - the reliability service level (e.g. fault tolerance of the distributed storage) that is usually related to the degree of replication of information,
  - additional properties of the service like transaction support, which are needed by specific clients and supported by specific subsystems.

All these functionalities rely on efficient, scalable and reliable ways to gather and organize information concerning the status of resources, services and applications, which are provided by different distributed mechanisms, some of them being of general availability, other ones being state-of-the-art prototypes in P2P technology.

The set of requirements, more thoroughly explained in [14], led to design a composite solution whose software architecture is shown in Figure 1.1, and based on the following design principles.

- Different implementation issues (that is interfacing, providing security and authentication, providing query functionality) have been decoupled into separate subsystems and layers, enhancing modularity and easing future integration with advanced features of XtreemOS.

- Different clients are supported by separate interfaces in the Facade layer, allowing easier customization of interaction protocols, stricter detection and better handling of errors. To avoid proliferation of interfaces and reduce low-level issues in coding them, most of the clients should exploit a common framework, e.g. DIXI[3], to access the SRDS.

- Different implementations of the information management layer (e.g. different DHT libraries) can be used within the same system in order to provide a differentiated QoS offer to the clients. Thus the SRDS architecture (1) has to be flexible w.r.t. the DHT adopted, decoupling the query resolution algorithms from the information management subsystems, (2) has to potentially support multiple overlay network over the same set of nodes and (3) should provide a simple API to tune specific DHTs to the user need, e.g. by selecting different routing algorithms or replication degrees before setting up a new overlay.

- As a special case of previous paragraph, and also as a main task of the SRDS, the performance-critical problem of resource location for Application Execution has to be solved providing high scalability. This particular problem has been dealt with by integrating two different P2P overlays providing complementary features, the RSS and ADS modules, that act as a *"machete"* and *"bistoury"* in extracting the answer from the huge amount of information describing the nodes of a Grid.

Resource queries in XtreemOS involve static and dinamic attributes (fixed properties and running parameters of the computing resources) over very large

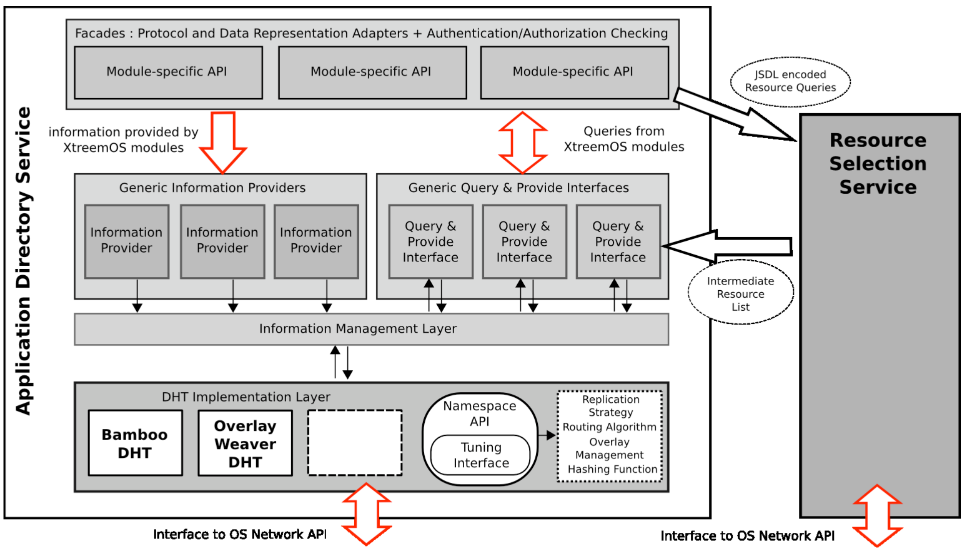

Application Directory Service
Facades : Protocol and Data Representation Adapters + Authentication/Authorization Checking
Module-specific API
Module-specific API
Module-specific API
JSDL encoded Resource Queries
information provided by XtreemOS modules
Queries from XtreemOS modules
Resource Selection Service
Generic Information Providers
Information Provider
Information Provider
Information Provider
Generic Query & Provide Interfaces
Query & Provide Interface
Query & Provide Interface
Query & Provide Interface
Intermediate Resource List
Information Management Layer
DHT Implementation Layer
Bamboo DHT
Overlay Weaver DHT
Namespace API
Tuning Interface
Replication Strategy
Routing Algorithm
Overlay Management
Hashing Function
Interface to OS Network API
Interface to OS Network API

Figure 1.1: SRDS architecture; detail of the ADS architecture and its interaction with RSS.

Grid platforms, and are typically in the form of multidimensional queries where some or all the parameters are value ranges.

No distributed network exists today that can answer this kind of queries efficiently and in a fully scalable way. Thus Resource queries are executed as a two-phase information selection process, in which two modules cooperate exploiting their relative strengths. The Resource Selection Service (RSS) developed by VUA, described in section 1.5, is capable of solving multiattribute range queries over static attributes, returning a set of candidates much smaller than the size of the resource set. A short list of resources satisfying the static part of the initial query is then handed to a DHT-based module of the Application Directory Service (ADS), which is much more flexible in the kind of information dealt with, and can efficiently compute the query answer thanks to the reduced problem size.

1.3 Application Directory Service (ADS)

The ADS internal architecture is depicted in figure 1.1. It constitutes the largest part of the SRDS prototype. The current implementation of the ADS provides interfaces for resource location within a VO (the Application Execution Management queries them) and two Directory Services interfaces, the Job Directory Service, which manages information about running jobs on an XtreemOS Grid, and the Data Management Services interface, which manages information about the server nodes of the XtreemFS file system.

Communication Frameworks Of those interfaces, the AEM and the JDS ones are exported through the DIXI service communication framework, and are currently in use in the public XtreemOS release. The DMS interface is provided over HTTP by a mini server supporting asynchronous and session based communication over a connectionless link. Ad additional RMI interface is provided in all cases that can be used in local to perform tests on the ADS module. Integration with DIXI has initially required a huge effort in debugging, with latest releases of the framework being more and more stable. Currently, DIXI is used within the ADS in a single-thread mode, with the switch to concurrently receiving client requests already planned, as soon as newer versions of the DIXI framework will enable multi-threaded services.

Communication and Marhsaling Protocols The JDS DIXI interfaces exploits the standard passing conventions of the DIXI framework; the AEM interfaces employs DIXI to exchange messages encoded as XML dialects (JSDL [2], GLUE [1]). The DMS interface employ a custom defined protocol based on JSON [11] marshaling rules.

Query Engines Current implementation of the ADS already provides a few query engines: to fetch dynamic attributes of computational resources, a Directory Service engine allowing reverse querying over multiattribute items, which is used by the JDS and DMS subsystems.

Information layer As designed in [14], the Information Layer decouples query processing from the actual P2P approach exploited, which is presented through a generic DHT API. This extra software level, which was meant to help mixing different overlay solutions, has also been already exploited to prepare the migration from the default Bamboo DHT to the OverlayWeaver toolbox, as reported in the next section 1.4.

1.3.1 Development Status

The following points were marked as future work in deliverable 3.2.4 [14, page 63].

- Support for simple and complex queries. Currently the ADS supports complex query defined by the JSDL language.

- Indexing of distributed resources. Besides exploiting the RSS features to index resources in term of their static attributes, research is ongoing about more powerful DHT-based algorithms. At present time we have prototypes developed as research related to XtreemOS, that need further improvement and study, and could be exploited within the ADS. To focus more on this point we need to rely on a stable DHT management system.

- Handling dynamic attributes: we defined and implemented a few backward compatible extensions to the JSDL language in order to support dynamic data for multi-dimensional queries. This works exploiting the RSS and filtering out the intermediate results with dynamicaly updated information held in the ADS DHT.

Currently Supported JSDL Attributes

We designed a minimal extension of the JSDL query language, which by default does not support the concept of dynamic resource attributes, to allow the users to specify range queries over dynamic attributes.

We worked on the assumption that each "dynamic" attribute has a "static" counterpart, e.g. free memory is associated to machine memory, available computing power is proportional to peak performance and the amount of free CPU cycles, and so on.

For each attribute handled by the ADS, an optional *epsilon* tag in JSDL, with a real value between 0 and 1, is used to express a query on the dynamic value associated to that attribute. The epsilon is optional as both attributes (static and dynamic) make sense in a query, but a dynamic query has always a static underlying query (which in the implementation provides the list of candidates to the dynamic query engine).

The threshold value for a query about a dynamic value is computed multiplying the LowerBound (L) attribute of the corresponding static range by the value of the epsilon. This produces and extended range that the value of the dynamic attribute must belong to.

Conversely, when a resource has a dynamic value lower than threshold, the node is discarded.

The epsilon value is expressed in the JSDL query by extending the JSDL syntax with a new attribute "dynamic-epsilon" in the resource tag.

```
<IndividualPhysicalMemory dynamic-epsilon="0.8">
<Range> .. </Range>
</IndividualPhysicalMemory>
```

This mechanism minimizes changes to the JSDL syntax, and allows the same query to address both the dynamic and the static attributes of a resource. As the RSS ignores the dynamic attribute of the query XML tags, the RSS provides resources with the *static* attribute within the range; the ADS will then discard those which, according to the *dynamic* information, do not belong to the range extended by the epsilon parameter[1].

[1]The JSDL standard permits unlimited ranges to either negative or positive infinity. A valid dynamic constraint requires both an epsilon value and a L element.

1.4 The DHT layer

According to the deliverable D3.2.4 the Service/Resource Discovery System exploits DHT subsystems for managing information it is expected to provide to a set of clients. When designing SRDS ISTI - CNR, in order to avoid wheel re-invention, decided to integrate an existing DHT system instead of developing yet another one from scratch. The first DHT system chosen was Bamboo[15, 4] , at the time widely used in the P2P academic community.

The choice of using the Bamboo DHT for the implementation turned out to be unlucky. Indeed, Bamboo was poorly documented, it has been implemented using two different languages, Java and C, integrated together in a very bad way, and the modular design principles are often violated in its implementation. Despite Bamboo has been developed in Java it practically requires to be launched as an external application in a separate JVM, with environmental variables and configuration files properly initialized beforehand.

Moreover, some Bamboo configuration details depend on private global variable, or are hard-coded at each use, so that cannot be changed without modifying the Bamboo source code.

This is the case, for instance, of retransmission timeouts due to lost/late TCP packets. While the timeout employed for some operations are parameters within the Bamboo configuration file, those file-defined values are not always actually used, depending on the operation (get, put, internal routing), and in the end on the choice of the programmer in that particular point of the Bamboo source code. Changing the extremely conservative, default 5 seconds timeouts for many DHT operation has required scattered changes in the Bamboo source code, adding more configuration options to allow the user (that is, the SRDS) tuning the DHT to the actual physical network.

Besides this being a significant development overhead, and in addition to the need of packaging a custom version of Bamboo within the XtreemOS distribution, a few months after its integration with SRDS, it was clear that the development and the maintenance of Bamboo had been discontinued.

This led with time to increasing Bamboo incompatibilities with newer and newer releases of several libraries, frameworks and software components that are used both by Bamboo and by other modules of XtreemOS. In particular, Bamboo has become incompatible with the latest releases of the Java environment. Those incompatibilities were more and more problematic during the preparation of XtreemOS Mandriva distribution. At present time, in agreement with the other development teams, the ISTI - CNR team is replacing the Bamboo DHT with Overlay Weaver (OW) DHT as the default DHT library in the next XtreemOS public release.

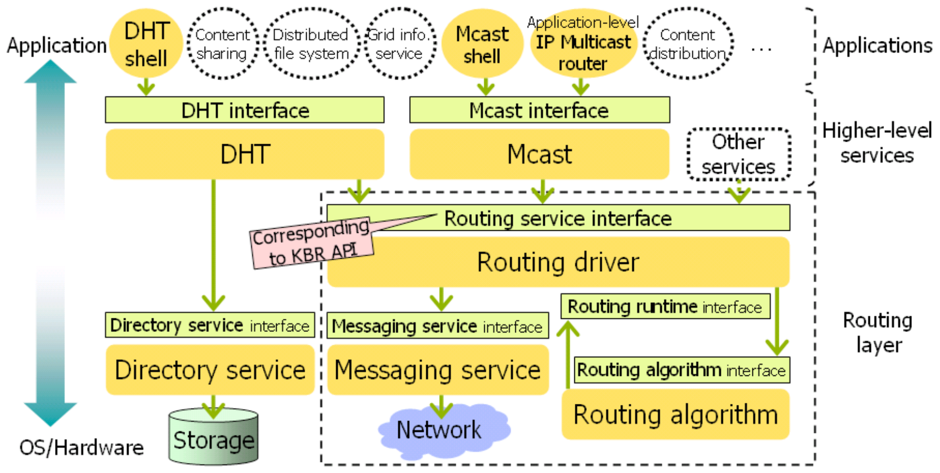


Figure 1.2: Overlay Weaver Architecture, from [16]

1.4.1 Overlay Weaver

Overlay Weaver [17, 16] (abbreviated as OW) is an Open source DHT software developed as a research project. It is highly modular, configurable and customizable. OW is an overlay construction toolkit, which supports overlay algorithm designers in addition to application developers. Indeed, for application developers, the toolkit provides a common API for higher-level services such as distributed hash table (DHT) and multicast. Applications that rely on the common API do not depend on specific transport protocols, database implementation and routing algorithms. The toolkit provides multiple routing algorithms, and besides the several structured network it natively supports (Chord, Pastry, Tapestry, Kademlia and Koorde), it is also possible to customize the network behavior to match specific requirements and develop new algorithms.

OW has a multi-layer architecture. It is structured in four main layers: Applications, High-level Services, Routing Services and Storing Services. Typical usages of the rootkit involve only the first two layers, the other two are devoted to low-level aspects.

OW also includes a *Distributed Environment Emulator*. The emulator aim is to rapidly test new overlays avoiding the overhead of (re)creating them in a real distributed environment. This is particularly useful in the context of the XtreemOS project, where, especially in between public releases, at any given time there might be a limited number of machines available for running a specific version of XtreemOS.

The OW network emulator can host tens of thousands of nodes on a single computer virtually. Overlay designers can improve rapidly their algorithm implementations by testing them iteratively on the emulator. They can also make a large-scale and fair comparison between new and existing algorithms on the emulator. Implemented algorithms do not need any change to work on a real network if they work on the emulator, thus the OW toolkit enables direct transferal of algorithm research results to applications.

The Emulator runs a simulation according to the settings defined in scenario files. The toolkit includes a simple Emulation Scenario Generator for generating a scenario files with it, either by writing them by hand, or through trace collection, by running an existing application and translating execution traces into a scenario.

The OW Emulator has two running modes. In normal mode, the whole emulator runs on a single computer. In the parallel mode, multiple computers form a single emulator in cooperation, making it feasible to run larger simulations. Fig. 1.3 shows the structure of the emulator. Of course, the emulator reads the same scenario file in both cases, and invokes and controls application instances according to the scenario.

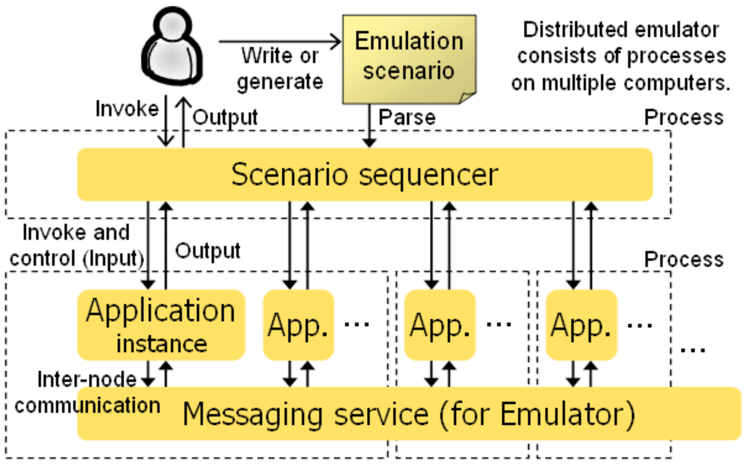
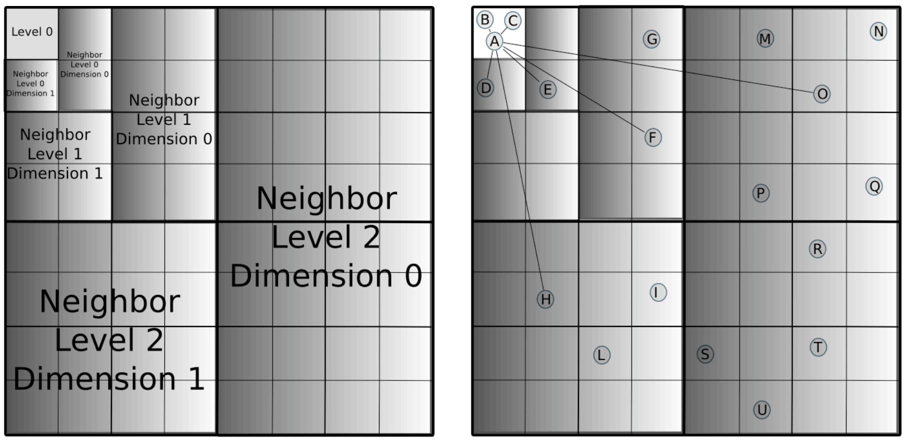


Figure 1.3: Distributed Environment Emulator, from [16]

1.5 Resource Selection Service (RSS)

The Resource Selection Service (RSS) is the module in the SRDS that provides resource location services based on *multi-attribute, range* queries over a set of *static* attributes of the resources. As documented in [14], the P2P approach adopted is designed to provide system scalability exploiting the following assuptions:


Level 0
Neighbor Level 0 Dimension 0
Neighbor Level 0 Dimension 1
Neighbor Level 1 Dimension 0
Neighbor Level 1 Dimension 1
Neighbor Level 2 Dimension 0
Neighbor Level 2 Dimension 1
B
C
A
D
E
G
M
N
O
F
P
Q
R
H
I
L
S
T
U

(a) Neighboring Cells (b) Node A's neighbors

Figure 1.4: Attribute space partition and node neighbours example with $d = 2$.

- all nodes are linked in a hierarchical P2P overlay network,
- each resource node provides information about itself (that is, its own static attribute values)
- the P2P overlay network is built according to the attribute distribution over the resources, employing a definition of *nested cells* that reflects the dimension of the resource attribute space,
- the P2P network allows nodes to join and leave at any time, eventually performing structure maintenance
- query routing avoids centralization-related bottlenecks, also by exploiting a definition of *neighbour nodes* that scatters inter-cell communications among the nodes of any given cell.

The nodes participating the RSS overlay network are assumed to belong to a same Virtual Organization, this ensuring that safe and sound authentication mechanisms exist to certify node identity. The P2P implementation layer employs a gossip-based framework, building a P2P overlay on top of it. The assumption that the attribute values are static guarantees that any node will not change its position, after joining the overlay, thus limiting maintenance overhead to that of nodes joining and leaving.

The evaluation of the RSS has to take into account scalability with respect to both

- the execution platform size and characteristics and
- the kind of requests issued to the RSS (determined by the amount of resources required, the number of attributes and the value distribution of the attributes both in the resources and in the queries).

These aspects were evaluated and reported in [14]. Within XtreemOS, the former condition is much more stringent than the latter, platform scalability being conditioned mainly by the amount of resources linked by the RSS and by the geographical size of the underlying Grid. As such, experiments both on simulators (PeerSim [13]) and real systems (the DAS-3 cluster [10]) have been performed. Experiments on the RSS in isolation also evaluated it under different node churn rates, and its behaviour under massive network failures.

RSS Development Status The RSS has been developed in Java and can be run as a stand-alone service; within the SRDS, the RSS classes are directly called by the SRDS Java code. Thus, RSS initialization is triggered during SRDS initialization, and RSS API is called whenever the SRDS needs to resolve a resource query. The integration work in order to have the two services cooperate within the same Java Virtual Machine has required

- defining a data interchange format, based on the JSDL and GLUE [2, 1] XML standards
- defining an interoperation protocol with the SRDS, and the corresponding API and its Java implementation in the SRDS and RSS modules,
- implement it with some changes to the RSS interface classes
- change the meaning of some of the attributes managed by the RSS (e.g. the disk-space attribute) to better match the JSDL/GLUE interpretation adopted within XtreemOS.

The debug and test activity on the RSS prototype has been performed by the CNR and VUA teams during the reciprocal integration phase, and then in strict collaboration with the XLAB team, the Mandriva team and all the core XtreemOS developers during the XtreemOS integration phase that has led to the first public distribution of the whole OS .

More on this functional test and debugging activity is reported in section 1.6. Further performance results comparing the RSS query resource location approach with classical ones based only on DHTs are discussed in chapter 2.

1.6 Development activity up to M30

This section recaps the development effort performed by ISTI - CNR in the context of XtreemOS project during the last months.

Since M24 the effort on the SRDS partially shifted from internal development toward other main issues: software preparation for the integration of Overlay Weaver as DHT subsystem, SRDS the testing and and debugging activity for XtreemOS packaging, as well as the improvement of packaging procedures.

1.6.1 Design of the Integration with Overlay Weaver

As already discussed in Section 1.4 the Bamboo DHT subsystem caused several problems when integrating SRDS within the XtreemOS Mandriva distribution. In order to avoid those problems in the future, ISTI - CNR started to work on the replacement of Bamboo with Overlay Weaver (as detailed in Section 1.4). For doing this, the DHT abstraction layer inside the Service/Resource Discovery System has been exploited. That layer has been designed to support the integration of different DHT systems. This design choice increase the system flexibility allowing the integration of several DHTs, hence is smarter than a simple substitution of a system with another.

The Structure of DHT abstraction layer

The DHT abstraction layer for integrating new DHT subsystem in SRDS is composed by different Java packages. One main package represents the abstract layer (named `eu.xtreemos.ads.dht`), and additional packages, one for each DHT implementation, behaving as interfaces with the specific DHT subsystems. In the following of this section we describe the main classes composing the abstract DHT layer of the Service/Resource Discovery System.

`AbstractDHT` is the abstract class that represents each DHT subsystem as service. `DHTLowLevelInterface` is an interface for describing low-level functions of a DHT (put, get, remove). The `AbstractDHTHLObject` abstract class defines high-level functions, particularly useful for SRDS (e.g a remove operation performed before a put operation consists in a “publication”). For each DHT system integrated in SRDS, the class `AbstractDHT` as well as the class `AbstractDHTHLObject` must be extended with specific classes concretely implementing the abstract functionalities defined by the abstract classes. Moreover, `FeatureList` class defines the feature that a DHT can exploit. It is associated with the low-level service object.

The `DhtObjectFactory` class consists in the main interface to use for exploiting DHT services both in order to create new namespaces or in order to retrieve a namespace among the existing ones. It defines two main methods:

- *joinnamespace*: this method returns an instance of DHT that provide the specific namespace requested, passed as method parameter, if such a DHT is already running that DHT is returned otherwise a new DHT is created.
- *createnamespace*: this method behaves in a very similar way with respect to the previous one, the main difference consists in the way the method reacts when a requested namespace already exists. Indeed, if this method is called with an existing namespace as a parameter, it returns an error. (In the current version of the software this method has not been implemented yet).

The different DHTs that can be integrated in the Service/Resource Discovery System through the use of the abstraction layer are executed by SRDS as external processes. The `ProcessManager` class is exploited for handling an external process (in particular in this context is used for handling DHTs processes). Roughly speaking, it provides a thread that periodically read the output buffer of a certain process.

The integration of Overlay Weaver

The abstraction provided by means of the DHT layer has been used for integrating Overlay Weaver within the SRDS. The software developed for that integration has been structured as an independent java package (`eu.xtreemos.ads.dht.ow`). It contains the classes implementing the abstract classes that must be extended in order to run overlay weaver from within SRDS. More in detail, the `OWDHT` class is the implementation of the `AbstractDHt` abstract class, and the `OWHLObject` class is the implementation of the `AbstractDHTHLObject` abstract class. Moreover, the package contains two further classes: `OWBridge`, that contains the code for running Overlay Weaver as a separated process, and `DhtAccess` , that contains the methods for allowing a RPC-based communication between the SRDS and the Overlay Weaver DHT (running as a separated process).

The package (`eu.xtreemos.ads.dht.ow`) contains also test classes which has been used to conduct experiments aimed at testing the basic functionalities of the Overlay Weaver Distributed Hash Table (for more details see section 3.2).

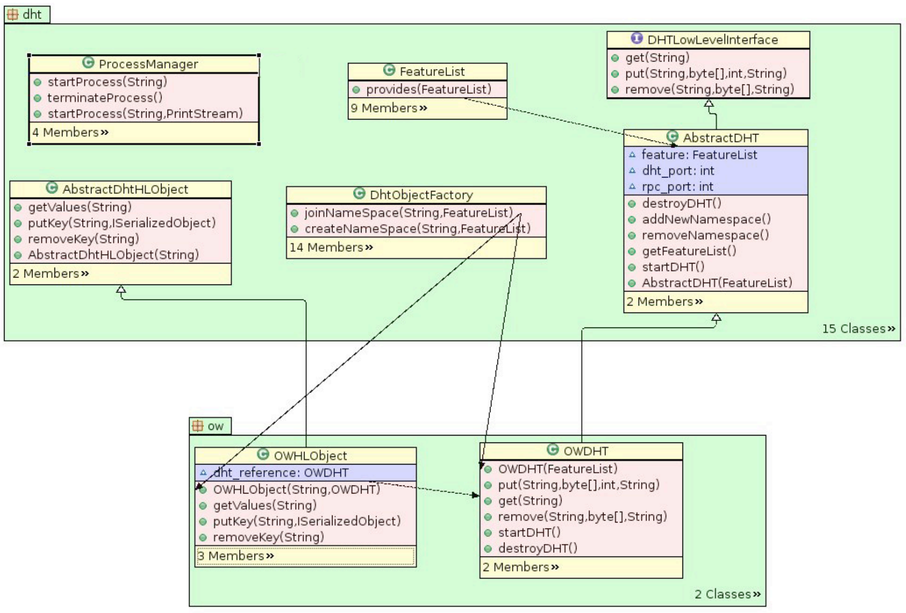


Figure 1.5: example caption

Implementation Notes Despite the effort lavished on the code development, the DHT abstraction layer as well as the Overlay Weaver integration package are still affected by some issues. In this paragraph are reported all the main features and problems characterizing the current version of the Service/Resource Discovery System.

- A DHT is not immediately usable after its creation. Hence, the class implementing `AbstractDHT` must waits for a certain number of milliseconds just after the creation of a new DHT before being able to use it. In the current version of SRDS it is set to 1000 milliseconds. Nevertheless, the amount of milliseconds to wait is configurable and can be tuned changing the `SrdsConfiguration.deleyAfterCreatingDHT` variable.

- Overlay Weaver is executed in the form of an external process. It is done through the usage of a class as wrapper named `OWBridge`. This class instantiates a DHT service performing method calls on the Overlay Weaver package. This class is called via the shell script `runow.sh`, that script is located in the folder `/etc/xos/configuration/OW/`. Once launched, this script looks for the class in the jar specified in the classpath (-cp option to shell line arguments of the Java VM). This implies that for each change on the `OWBridge` class the `srds.jar` has to be rebuilt in order to allow changes to take effect.

- Transport Level Network Ports are hardcoded. It means that they can not be changed without re-compiling the Service/Resource Discovery System. The network port currently used in the software are the standard ones both for Bamboo and for Overlay Weaver.

- The actual configuration of Overlay Weaver in terms of replication degree, routing type, etc.. is hardcoded. More in detail, all those configuration parameters are defined in the `OWBridge` class.

- The *terminateProcess* method defined in the `ProcessManager` class does not work with Overlay Weaver. After calling the method *destroy* defined in the `java.lang.Process` class, the process of Overlay Weaver continues to run, and must be killed explicitly. For this purpose a specialized script has been developed. That script is named `stopow.sh` and can be found inside the folder `/etc/xos/configuration/OW/`.

- Some incompatibilities exist between the Overlay Weaver and Bamboo implementations, due to conflicting version of the Java libraries they employ. Nevertheless, they will not cause any major problem as long as each DHT is running in its own process in different environments.

1.6.2 Testing and Packaging with XtreemOS

Around and after M24, XtreemOS development activities has been requiring increasing efforts on code debugging for inttegration, as well as on packaging prototypes according to linux source/binary distribution standards. Of course, the three SRDS, RSS and Bamboo DHT packages were no exception to the rule.

Testing

In this Section we describe the tests conducted on the SRDS. The goal of those tests was to validate SRDS by means of the analysis of its behavior. The tests aimed at analyzing the behavior of SRDS not only as stand-alone component but as part of XtreemOS, considered both running as single machine and running in the context of a distributed architecture.

We performed the tests using both virtual machines running on Linux (or OSX) systems and chroot environments. In both cases, for the installation, the setup and the configuration of XtreemOS systems ISTI - CNR exploited the official guides available on the XtreemOS versioned system. For the testbeds performed using the chroot environment, ISTI-CNR downloaded from the distribution repository all packages needed for running Service/Resource Discovery System. In this context, the configuration that has been tested is the local-config one. For the tests, CNR used the scripts and classes provided by XLAB. The work with chroot has been very important for taking confidence with whole system without involving too much effort, hence focusing the attention only on the real problems without dealing with problems related to (possible) mismatched system configuration. After several successful tests on chroot environment, CNR moved to virtual machines. For this purpose has been used the Sun VirtualBox. The installation of XtreemOS on virtual machines has been performed exploiting the XtreemOS CD provided by Mandriva. The installation process was always successful. All the packages needed to SRDS in order to work seems to be correctly installed.

Testing Details The tests has been conducted creating three different types of virtual machines. Each machine, having its proper features, had to be configured in a very different way. The types are:

- A local machine (XtreemOS guide Section 4)
- A core node
- A resource node (XtreemOS guide Section 5)

For each one, after the installation, in order to run tests with the very last version of XtreemOS, the distribution has been updated through the use of command: `urpmi -auto-update`. The local machine configuration has been performed simply downloading the local-config package from XtreemOS distribution repository. The other two types of machine nodes has been configured following the instruction specified into the guide. The node used as a bootstrap one (the node which runs the Rss registration service) was the core node that we have setup.

For testing the behavior in a network (even if very small) two virtual node in a local network have been connected. As the nodes were able to see each other, using the `xconsole` provided by `xosd`, it was possible to submit operation requests to the Service/Resource Discovery System. With this testbeds we encountered some problems, and also some issues with the log systems that precluded the straightforward way to identify problems.

In the end, some of the problems actually were caused by other modules, while other were more code-related and due to the aforementioned linking problems with the Bamboo DHT (incompatibilities with specific releases of some Java libraries required by other XtreemOS packages). With a proper network and system configuration, all the connections (between Bamboo and RSS) can be set up to working condition both in a distibuted settings and in the XtreemOS local-config trivial set-up.

Packaging

In order to improve the debugging procedures as well as to simplify the automated code installation and update, ISTI - CNR together with Mandriva worked on the packaging of SRDS, RSS and the Bamboo DHT customized by ISTI - CNR. The three packages are now part of the XtreemOS distribution.

Since ISTI - CNR managed the packaging tasks under the steering of Mandriva, the procedure followed for the packaging as well as the structure of the package are very similar to the procedures and structures of the other XtreemOS packages. This aims to create a more coherent checking of XtreemOS packages with respect to installation and uninstallation procedures, automatic update, dependence analysis, and functionality testbeds.

A key step towards this goal is the design and implementation of tools for the automated testing procedures.

Automated Testing The aim of automated testing procedures is to allow SRDS packagers (typically CNR or Mandriva) having a simple and fast way to determinate if SRDS package is working releasing it. These tests will be used mostly by Mandriva, and have not been thought for being used by external users.

Current SRDS testbeds are intended to run with local-config configuration. Indeed, a complete integration testbed must rely on external nodes, making the test procedure very complex. Nevertheless, in the future ISTI - CNR will move to work with a distributed configuration and will provide testbeds checking SRDS behavior in distributed environments.

The test algorithm defined together with Mandriva before delivering a package consists in the serial execution of the following steps:

1. install local-config packages for XtreemOS

2. install SRDS/RSS on (all) machine(s)
3. run the SRDS/RSS installation tests
4. remove local-config (*)
5. reinstall SRDS/RSS (*)
6. run distributed SRDS RSS installation tests (*)

The basic idea is that every time an update of SRDS affects any local-config files, the SRDS developers will have to provide an update for that package, in order to allow automated testing together with all the other XtreemOS packages.

It should be noticed that the fourth to sixth steps above target testing candidate packages over a properly configured distributed environment. This is not standardized practice in Linux distributions, but it could possibly be required as part of XtreemOS packaging standard. The scheme to follow for distributed testing has thus been defined, but has not yet been implemented or mandated for the current XtreemOS distribution.

Other tests that are useful for the analysis of the behavior of a software. They may include both service status test and functionality test. In case of SRDS a few examples of services status test consists in checking if:

- the correct processes are up and running
- the right transport level network ports are open
- the routing tables are properly configured.

As regards the functionality tests, we are developing a few RMI interfaces that allow to submit request to the SRDS module without using the communication framework typically used on XtreemOS. This permit to run some post install test without the need of other services running. As example, a typical feature test is to check if the system reply in a proper way to various requests.

Chapter 2

Performance Evaluation

In this chapter we report additional performance results regarding the SRDS. A key factor in evaluating reproducibility of results is the invariance of the system behavior with respect to the kind of platform and of its scale. These system features were already experimentally evaluated in a first design stage, as reported in project deliverable D3.2.4, for both the ADS and the RSS subsystems.

Here we mainly show

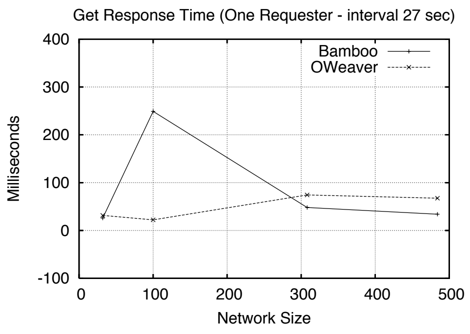
- additional results on a large Grid concerning the evaluation of the ADS exploiting a different DHT implementation layer from Bamboo, showing that the SRDS directory service over a large DHT network is of practical utility also adopting the OverlayWeaver implementation layer;

- a comparison among the dedicated P2P network of the RSS and a mainstream approach for implementing multiattribute range queries over a DHT support; we show that the RSS approach scales well, does not cause any average overhead, and achieves a much better balancing of the query processing load scattered among the P2P network than approaches based on a DHT layer.

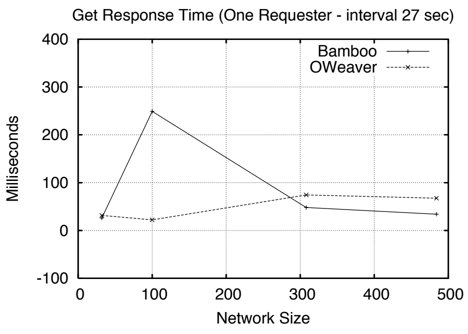
2.1 DHT Layer Evaluation – Bamboo vs OverlayWeaver

As we plan to support multiple DHT libraries and DHT configurations in the SRDS, to match the different service qualities target that different clients will impose on the Directory Services, it is important to show that the SRDS can keep a sound degree of scalability with the different DHT implementations supported. This requirement is even more strict for the DHT library used by default, that is going soon to change from Bamboo to OverlayWeaver (reasons were explained in Chapter 1).

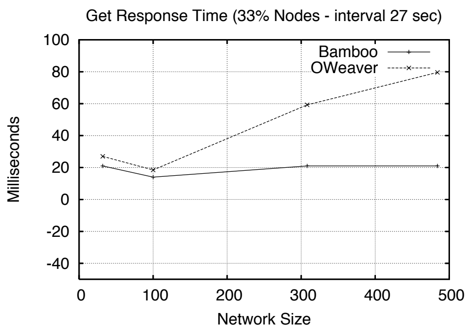
We show a comparison on a large Grid platform among the behaviour of the SRDS employing OverlayWeaver and results previously obtained employing Bam-



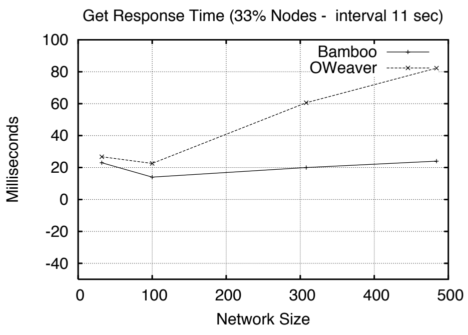
(a) Get request every 27 sec, 1 Requester



(b) Get request every 11 sec, 1 Requester



(c) Get request every 27 sec, 33% of Requester nodes.



(d) Get request every 11 sec, 33% of Requester nodes.

Figure 2.1: First and Second Scalability Test - Grid5000 Platform at 32 – 484 nodes scattered among 1 – 5 clusters, all nodes provide information. First test, one Requester invokes 20 Get requests with assigned time interval. Second test, 33% of nodes concurrently run 20 get requests with assigned time interval.

boo. It should be noted however that the OW tests were performed using the Chord DHT algorithm [18].

We exploit the same Grid platform (Grid5000), and apart from the actual selection of Grid nodes, which is hard to reproduce exactly, we employ the same testing conditions as in deliverable [14]. To summarize, all nodes involved in each test provide information about themselves to the DHT, once every 30 seconds, and the DHT performance is measured in three different test conditions, for a subset of the whole platform ranging from 32 to 484 nodes.

Scalability 1 — A basic latency measure, where one Requester Node periodically asks information about another nodes (essentially a "get" operation on the DHT).

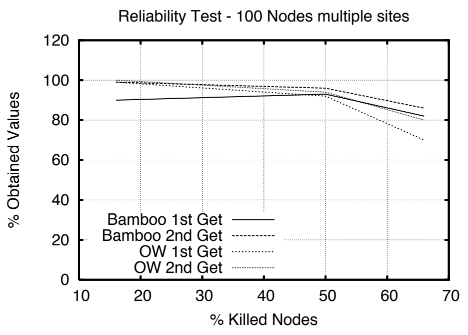
Scalability 2 — A latency measure when the network is normally congested; a constant fraction (33%) of the nodes concurrently and periodically query information about other nodes.

Reliability — A measure of information recall to show DHT overlay rebuilding after (massive) node churn or failure. Here a fraction of the DHT nodes are killed (choice is random) and all the information in the DHT is queried twice, immediately after the "failure", and one minute later.

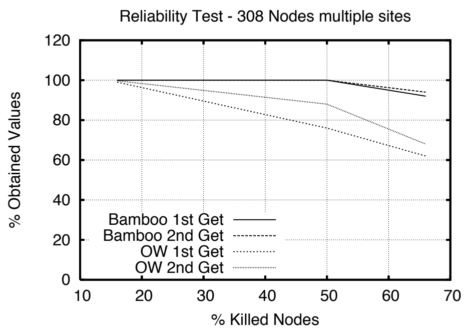
In Figures 2.1(a) and 2.1(b) we see the First Scalability test; in the comparison, using OverlayWeaver leads sometimes to a larger latency but more regular latency. Figures 2.1(c) and 2.1(d) show that OverlayWeaver leads to a cost on a loaded network that increases with network size faster than that of Bamboo does, but is still logarithmically growing. We can extrapolate that the SRDS latency will be in the order of one hundred milliseconds even for networks with the same characteristics and one order of magnitude larger. Considering the fact that OverlayWeaver in these tests has not been deeply tuned[1], and that Bamboo code is no longer maintained by its original developers, it is clear that the more modular structure of OverlayWeaver is a good substitute for Bamboo both on the performance side and on the development efficiency side.

Similar conclusions can be drawn from the Reliability tests. In Figure 2.2 we see a comparison of the Reliability tests of Bamboo and OverlayWeaver. In this case what should be observed is that even in presence of massive faults the overlay network is able to rebuild. The key point in this test is that we spread information about all the nodes in the DHT ring. When a large number nodes are killed, information is lost, and part of it is later restored as the providing nodes still alive update their information, which is reassigned. The fact the the amount of information after one minute from the failure (2nd get in the graphs) grows toward the original amount means the network is able to heal. Note that the DHT can restore up to 100% of the original information content even if much less than the

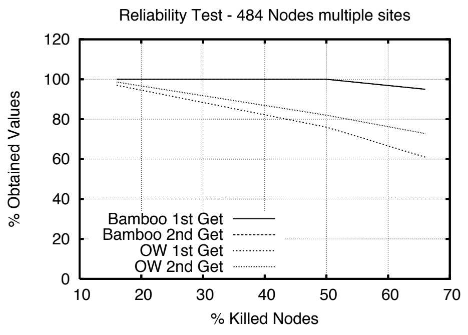
[1]We recall that in order to obtain a good performance from Bamboo, the CNR development team had to change several timeouts hard-coded to 5 seconds in the Bamboo code.



(a) Result comparison with 100 nodes



(b) Result comparison with 308 nodes



(c) Result comparison with 484 nodes

Figure 2.2: Reliability test - Comparison between Overlay Weaver and Bamboo with respect to overlay self-repairing and information recall features, when 16%, 33% and 66% of the nodes leave the network. We measure the percentage of information recovered immediately after the failure (1st get) and one minute later (2nd get).

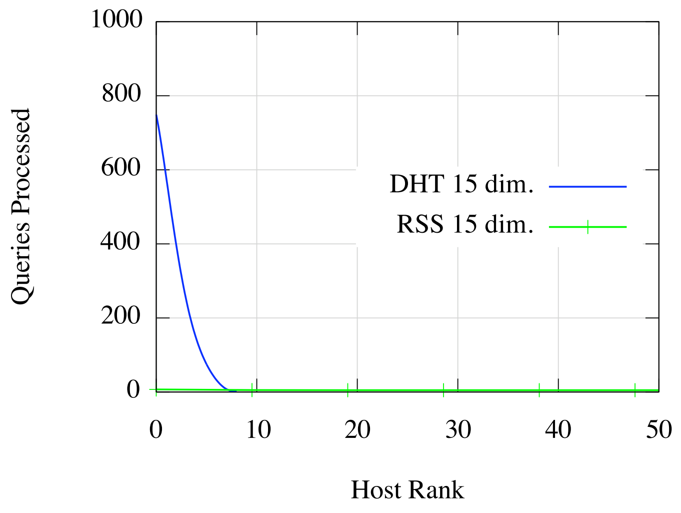


Figure 2.3: Histogram of query overhead (# of query processed - 50) on 1000-node networks with the RSS and with a SWORD-like DHT based approach. Nodes ranked and sorted by diminishing load.

initial amount on nodes still belong to the overlay, as the information is always hashed and replicated across the DHT ring. How close to the initial content a DHT can get depends on the actual DHT algorithm used, on the amount of replication degree selected, and on the interplay between the hashing and the physical node distribution. In the test reported, we see that on average Overlay Weaver is able to recover less information. As Overlay Weaver has not yet been extensively tuned for these tests, it hashing and routing function are different and and the two tests were performed on different subsets of the Grid5000 clusters, some difference in the degree of information recall is normal.

2.2 RSS evaluation with respect to a DHT-based approach.

We compare the execution of multi-attribute range queries on the RSS with the behaviour of a DHT (in our case Bamboo) with a SWORD-like implementation [12] of range query resolution.

In SWORD, node attributes are stored in the DHT at keys corresponding to the values of each attribute (plus a random skew to spread apart equal values). Thus, there are d keys that will store a node in the DHT if there are d attributes. We employ the same key generation method of SWORD.

The tests use a uniform distribution of all nodes attribute values over many different dimensions (as described for each graph). Queries are also distributed uniformly among the nodes (a different node begins each query). Queries are for 50 nodes (out of 1000) and the selectivity is .125.

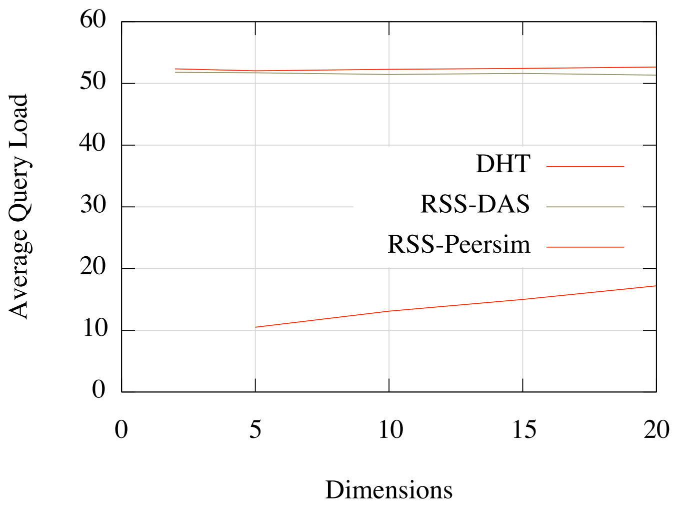


Figure 2.4: Average query load on nodes over multiple runs, plotted for a varying number of dimensions of the attribute space.

Figure 2.3 shows the graph for the query load. The y-axis represents the number of requests seen by each node over the course of 50 queries (that is, the number of times the node has had to process a query). The x-axis is ranked and sorted by the y-axis value. The RSS numbers go off the right side of the graph, of course, out to above 800.

We've scaled[2] the y-axis for the DHT values to put them in line with the overhead. The definition of overhead we use is: for a query for X resources, the overhead is the number of overlay nodes serving the query, minus X. In the RSS P2P approach, we have to contact at least X nodes, and actual query execution will be the more closer to optimal the less nodes in excess it needs to contact.

Of course, most nodes are never contacted by a query in a DHT approach, so the DHT "overhead" will be negative according to this definition. On the other hand, what we really care for is that there are no hot-spots which suffer critical load under a normal query traffic, and this is the property that RSS has to verify.

It is already apparent that the DHT approach causes a small number of nodes to withstand a query load which is roughly proportional to the network size, while the RSS essentially balances query load among the nodes.

[2]The scaling has been performed recording the number of requests for keys to the storage manager in Bamboo and the number of DHT is queries. We record both numbers as a form of double check on the estimate of the number of DHT nodes contacted, as the Bamboo implementation is quite involved and cumbersome, and often unnecessarily performs some internal operations.

As for the overhead, the DHT is queried on average 15 times to satisfy a query for 50 nodes. By that record, the routing overhead (as defined for RSS) is -35 (the leaf sets are about 11-12 nodes, so forwarding appears to be minimal and doesn't require a database lookup anyways).

Figure 2.4 represents something of the converse. The y-axis is an average (over multiple runs of 50 queries) of how many nodes process each request, but as query cost rather than routing overhead, so this time the DHT numbers are not negative.

Here query load is number of nodes that actually process the query on average (that is 50 + routing overhead in previous graphs for RSS experiments) for a varying number of dimensions (that is the number of attributes). The numbers for RSS on the Peersim simulator are from previous experiments on a smaller number of nodes.

As we average over multiple runs, the DHT approach also converges to the same figures than the RSS, the hot-spots being canceled out in the average by the lower load imposed on most nodes (the "negative overhead" of the DHT with respect to the RSS). Besides, the two approaches exhibit the same average load, so the RSS does not pay a significant cumulative overhead in order to balance the load among its nodes.

The two graphs combined show that while a DHT can put little load on most of the nodes for range queries (and scale sublinearly with the number of dimensions), the load can concentrate on a small set of nodes and thus overwhelm them on any large scale implementation.

This confirms that the two-stage approach used in resource query resolution by the SRDS is useful for queries that are not degenerate[3], as it avoids potential bottlenecks in selecting a relatively small number of interesting nodes, and eventually exploits from a DHT in order to narrow down the first candidate list taking into account information that it is not convenient to access by the RSS approach (e.g. heavily dynamic data and unfrequently used attributes).

[3]asking for a large fraction of the nodes in the platform would be inefficient anyway

Chapter 3

Functionality Evaluation

In this chapter we discuss how functional testing is conducted in the SRDS development. The test methodology is actually composed by two different kind of tests: Black-Box Testing and Unit Testing.

- Black-Box testing is performed directly on the high-level interfaces, which have to meet specific protocols defined for the client using them.

- Unit testing is exploited on the key modules composing the SRDS, in order to minimize error propagation during code development, and increase modularity.

3.1 Black-box Testing

Test definitions for the JDS client interface of the Service/Resource Discovery System are reported in table 3.1. The table includes both positive tests, which return an operation complete message, and negative tests, which shall return errors. For our purpose each file represents a possible JDS query. These requests are submitted via a local RMI interface to the SRDS module. Afterwards, the reply obtained is checked against the expected result.

The same testing approach described above is employed also for AEM client testing. In this case, the context is only slightly different w.r.t. the JDS client. Each file is converted into the JSDL format specified by XtreemOS and submitted via RMI to the system. During the query resolution process, the request is refined by the RSS module. The response for an AEM request is in the GLUE format; its correctness is checked once received back from SRDS.

At the current implementation status, the RMI interface is only used for the above described internal testing. However, in the next future, this interface will be also used to automate the tests for the packaging process (more details were reported in section 1.6.2).

Table 3.1: List of test input files, their purposes and expected test outcomes.

Input Script	Description	Expected Behaviour
InsertJob.input	Insert the information about a new Job: the `jobId`, the `@jobManager` and the `userId`.	Success
InsertDuplicateJob.input	Insertion of a job with `jobId` parameter unspecified.	Raise Exception: ADSJobIdAlreadyPresent
InsertDuplicateJob.input	Insertion of two jobs with the same `jobId`.	Raise Exception: ADSJobIdAlreadyPresent
Add.input	Insertion of a new job attribute, together with its proper value, specifying a valid triple `jobId`, `attributo`, `valore`.	Success
AddAttributeExist.input	Insertion of two new attributes named in the very same way.	Raise ADSWrongAttributeException
AddAttributeNull.input	Insertion of a new attribute with no value assigned to.	Raise ADSWrongAttributeException
AddJobIdNotExist.input	Insertion of a new job attribute to an non-existing `jobId`.	Raise ADSKeyNotFoundException
Update.input	Update a job value for an existing job.	Success
UpdateJobIdNotExist.input	Update an attribute value of an non-existing job.	Raise ADSKeyNotFoundException
UpdateAttributeNotExist.input	Update a non-existing job attribute.	Raise ADSWrongAttributeException
GetAttribute.input	Returns an attribute value of a job, it takes as parameters: `jobId`, `attribute`.	Success
GetAttributeJobIdNotExist.input	Fetch an attribute value of an non-existing job.	Raise ADSKeyNotFoundException
GetAttributeNotExist.input	Fetch an attribute value of an non-existing attribute.	Raise ADSWrongAttributeException
RemoveJob.input	Remove a job using its `jobId`.	Succes
RemoveJobNotExist.input	Remove a job using an non-existing `jobId`.	Raise ADSKeyNotFoundException
RemoveAttribute.input	Remove a job attribute passing as parameters: `jobId` and `attributo`.	Success
RemoveAttributeNotExist.input	Remove an non-existing attribute.	Raise ADSWrongAttributeException
RemoveAttributeJobIdNotExist .input	Remove a job attribute to an non-existing `jobId`.	Raise ADSKeyNotFoundException

Table 3.2: List of all Junit classes along with a description of features tested.

Test class name	**Testing subject**
JDSClientImplementationTest	Add, update and remove operations on jobs and attributes. List of methods: testAddRemoveJob(), testUpdateJobAttribute(), testAddRemoveJobAttribute(), testGetJobIDByValue().
DhtObjectFactoryTest	DHT creation and namespaces management. List of methods: testCreateNameSpace(), testJoinNameSpace_1(), testJoinNameSpace_2().
OWDHTTest	Low level DHT operations: get, put and remove. Similar class is provided for Bamboo. List of methods: testPut(), testGet(), testRemove().
OWHLObjectTest	High level DHT operations: getValues, putValues and removeValues. Similar class is provided for Bamboo. List of methods: testPutKeys(), testGetValues(), testRemoveKey().
QueryProviderTest	General logic for direct and reverse queries. List of methods: testUpdate(), testDuplicateUpdate(), testGetID(), testGetReverse(), testRemoveFreeAttribute().

3.2 Unit Testing

In the context of SRDS, JUnit testing aims to validate the correctness of internal ADS module integration. The tests are used to guarantee that the implemented operation semantics is consistent with the specification. Hence, each single step of operations is handled as expected. Test classes have been designed for being used to analyze the main functions of the classes, i.e. testing a single method or simple method combination. Currently, they do not support the test of the behavior in-the-large, hence on a running distributed system.

The current SRDS JUnit tests deal only with the core aspects of the package. Nevertheless, in the future they will be extended in order to support the testing of the whole set of modules of the SRDS package. Regarding the functionality details, the tests cover all the DHT subsystem, in particular the low-level and high level operations for both Bamboo and Overlay Weaver. Moreover, tests are provided for the query engine of the JDS operations.

The table 3.2 lists all the JUnit classes used to test core functionalities. Along with each class a brief description is given of the tested features, and a list of corresponding class methods.

As an example, we consider how the sequence of operations implementing a JDS attribute update is completely tested in all its steps inside the SRDS.

Junit test for JDS Client

The *UpdateJobAttribute* method updates the value of an attribute associated to a job, assuming the job is already present in the system. The JUnit code depicted in Figure 3.1 tests this update operation. It adds a new job into the system, then adds it a couple of attributes, then it modifies the value of the last attribute. Finally it tests, performing a request, that the corresponding value has been updated correctly.

```
@Test
public void testUpdateJobAttribute()
{
jds = new JDSClientImplementation();
jds.addNewJob("JOB", "jbmng", "me");
jds.addJobAttribute("JOB", "att1", "67");
jds.addJobAttribute("JOB", "att2", "997");

String result = jds.getJobAttributeValueByName("JOB", "att1");
Assert.assertEquals("67", result);
result = jds.getJobAttributeValueByName("JOB", "att2");
Assert.assertEquals("997", result);
}
```

Figure 3.1: A sample method of the JDSClientImplementationTest class.

Junit test for Query Provider

The semantics of a resource update is defined by the Update method of the Query-Provider engine. This method retrieves data from underlying overlay, updates the values, and then re-writes data into overlay. This behavior is tested by the QueryProviderTest class (figure 3.2). A new container class for attribute is created (`SerializedHash`). Then, it is inserted into the system as an attribute of a job. After, another slightly different container is associated to a job in order to simulate an update operation. Finally, data are retrieved to check if the values insertion has been correctly performed.

Junit test for DHT layer

The QueryProvider class relies on a DHT for storing and retrieving information. The class `OWHLObject` contains the method needed for interfacing with a DHT (in this case OverlayWeaver). The example method (Figure 3.3) only verifies that a couple <key, values> has been correctly inserted into the DHT.

```
@Test
public void testUpdate() throws Exception
{
SerializedHash sh = new SerializedHash();
sh.addAttribute("att1", "value1");
sh.addAttribute("att2", "value2");
ResourceInfo res = new ResourceInfo("jobid", "jobmng", sh);

qp.doReversePubblication(res);

sh.removeAttribute("att2");
sh.addAttribute("att2", "value2.1");
res = new ResourceInfo("jobid", "jobmng", sh);

qp.doUpdateResource(res);

SerializedHash result = qp.doIDSearch("jobid", "jobmng");
Assert.assertNotNull(result);
Assert.assertEquals("value2.1", result.getAttributeValue("att2"));
}
```

Figure 3.2: Sample method of QueryProviderTest class.

```
@Test
public void testPutKey() throws Exception
{
int r = dht.putOverwriteKey(key, values);
Assert.assertTrue(r >= 0);
}
```

Figure 3.3: Sample method of OWHLObjectTest class.

Chapter 4

Conclusions

The results presented in this deliverable complement those already shown in previous project deliverables, which provided early evaluation of the design and prototyping approach. The results presented here show that the good performance of the Service/Resource Discovery System is not due to a single specific implementation solution or a specific test conditions, but are repeatable given the described condition.

The performance of the main modules of SRDS has been tested over different execution platforms (simulated machines, clusters, structured Grids à la Grid5000 and DAS, loosely coupled distributed environments like PlanetLab). In each platform SRDS performance resulted quite good. Moreover, SRDS behavior showed to be coherent across the several different architectures.

In addition to isolated tests, additional activities were entailed to ensure that those good results are reproducible on a wide range of test cases. In particular,

- careful engineering of the SRDS prototype in order to ensure it expandability, flexibility, robustness; the synergic combination of the ADS and RSS systems, when deployed within the XtreemOS complex software architecture must easily deploy, interoperate and scale over different hardware platforms

- providing automated test and packaging procedures to ease integration within XtreemOS

The SRDS approach exhibits good performance independently of the DHT used. In order to give an empirical proof of it, this deliverable discussed the recent development of the SRDS, showing in Chapter 1 the need of moving to a different DHT layer, and reporting in Chapter 2 the comparable performance obtained w.r.t. the previous DHT layer Bamboo, when used over a large Grid.

Besides, we also showed that the RSS module outperforms DHT-based approaches in terms of load balancing and ability to avoid hot-spots, while enjoing the same degree of scalability in asymptotic terms. This confirms that the choice of exploiting a sinergy of two different P2P approaches is a scalable way to support the complex resource queries over very large Grids, those that XtreemOS targets.

The detailed description of the repeatable testing procedures has been given in Chapter 3, describing both Black-Box testing and Unit testing of the SRDS architecture. Some of the testing procedures described will be put in place within the SRDS package release routine.

Summarizing, this deliverable contains experiment results showing that the SRDS prototype performs consistently over diverse platforms, from the performance and functional viewpoints. Moreover it explains how to exploit the testing mechanisms to check its behaviour when modifying and deploying it.

Bibliography

[1] Sergio Andreozzi. XML Schema mapping of the GLUE Schema specification V.1.2, May 22nd 2006. `http://glueschema.forge.cnaf.infn.it/Mapping/XMLSchema`.

[2] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) Specification, Version 1.0. Technical Report GFD.136, Open Grid Forum, 2008. Available from OGF at `http://www.ogf.org/gf/docs/`.

[3] Matej Artac. XtreemOS Distributed Framework — DIXI. Technical report, XtreemOS Consortium, 2009. In preparation.

[4] Bamboo DHT web site. `<http://bamboo-dht.org/>`.

[5] Linux XOS specification. Deliverable D2.1.1 , XtreemOS WP 2.1.

[6] Design and implementation in Linux of basic user and resource management mechanisms spanning multiple administrative domains. Deliverable D2.1.2 , XtreemOS WP 2.1.

[7] Design of the Architecture for Application Execution Management in XtreemOS. Deliverable D3.3.2 , XtreemOS WP3.3.

[8] Requirements Documentation and Architecture. Deliverable D3.4.1, XtreemOS WP3.4.

[9] Security requirements for a Grid-based OS. Deliverable D3.5.2 , XtreemOS WP3.5.

[10] DAS-3: The next generation grid infrastructure in The Netherlands. `http://www.cs.vu.nl/das3/`.

[11] JSON web site. `<http://www.json.org/>`.

[12] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Scalable wide-area resource discovery. Technical Report CSD04 -1334, University of California Berkeley, Berkeley, CA, USA, 2004.

[13] PeerSim. `http://peersim.sourceforge.net`.

[14] Guillaume Pierre, Paolo Costa, Massimo Coppola, Domenico Laforenza, Laura Ricci, and Martina Baldanzi. XtreemOS Research Project Deliverable D3.2.4 Design and Specification of a Prototype Service/Resource Discovery System, December 2007.

[15] Sean Rhea, Dennis Geels, Timothy Roscoe, , and John Kubiatowicz. Handling churn in a dht. In *Proceedings of the USENIX Annual Technical Conference*, June 2004. Online at `http://www.usenix.org/publications/library/proceedings/usenix04/tech/`.

[16] Kazuyuki Shudo. Overlay weaver overview. Web site, 2008. `http://overlayweaver.sourceforge.net/`.

[17] Kazuyuki Shudo, Yoshio Tanakaa, and Satoshi Sekiguchi. Overlay Weaver: An overlay construction toolkit. *Computer Communications*, 31(2):402–412, February 2008. Special Issue: Foundation of Peer-to-Peer Computing.

[18] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.