Project no. IST-033576

XtreemOS

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

Design and Specification of a Prototype Service/Resource Discovery System
D3.2.4

Due date of deliverable: November 30th, 2007
Actual submission date: December 10th, 2007

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 / December 10th, 2007

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)	$\surd$

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	11/10/07	Martina Baldanzi, Massimo Coppola, Domenico Laforenza, Laura Ricci	CNR/ISTI	Initial outline
0.2	18/10/07	Martina Baldanzi, Massimo Coppola	CNR/ISTI	Added current prototype, experiments.
0.3	24/10/07	Laura Ricci	CNR/ISTI	Added future works.
0.4	25/10/07	Guillaume Pierre	VUA	Initial version of Chapter 3.
0.5	29/10/07	Martina Baldanzi, Massimo Coppola	CNR/ISTI	Minor corrections, figure tweaking, initial version of the ADS overall design.
0.5	30/10/07	Paolo Costa	VUA	Refined version of Chapter 3.
0.51	1/11/07	Paolo Costa	VUA	Added concluding remarks on future work in Chapter 3.
0.52	06/11/07	Massimo Coppola	CNR/ISTI	Chapter 1 first version, ADS overall design reworked.
0.53	07/11/07	Massimo Coppola	CNR/ISTI	Further material into ADS overall design.
0.54	08/11/07	Massimo Coppola, Martina Baldanzi	CNR/ISTI	Reworked introduction, new test results in chapter 4.
0.6	14/11/07	Massimo Coppola, Martina Baldanzi	CNR/ISTI	Added new figures and extensions of some sections.
0.61	15/11/07	Domenico Laforenza	CNR/ISTI	First overall revision of deliverable, added executive summary and various corrections.
0.62	15/11/07	Paolo Costa	VUA	Minor fixes.
0.63	16/11/07	Domenico Laforenza	CNR/ISTI	Various corrections on chap 1 and 2.
0.64	16/11/07	Martina Baldanzi, Massimo Coppola, Laura Ricci	CNR/ISTI	Corrections and revision
0.7	19/11/07	Martina Baldanzi, Massimo Coppola, Domenico Laforenza, Laura Ricci	CNR/ISTI	Candidate final revision
0.71	26/11/07	Guillaume Pierre	VUA	Small updates to take review into account
0.72	29/11/07	Martina Baldanzi, Massimo Coppola	CNR/ISTI	Small changes after revision
0.73	5/12/07	Massimo Coppola	CNR/ISTI	More changes in account of both internal reviews
0.74	10/12/07	Paolo Costa	VUA	More changes in account of both internal reviews
0.80	10/12/07	Massimo Coppola	CNR/ISTI	Merged chapters 1, 2
1.0	10/12/07	Massimo Coppola	CNR/ISTI	Final revision

Reviewers:
Yvon Jégou (INRIA/IRISA), Gregor Pipan (XLAB)

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

Abstract

This deliverable presents the current state of the design and specifications of the XtreemOS component called "Service/Resource Discovery System (SRDS)". The SRDS is a key component of the highly available and scalable infrastructure described in the deliverable D3.2.1 (Design of an Infrastructure for Highly Available and Scalable Grid Services) under the responsibility of WP3.2.

The SRDS will offer to other XtreemOS components the capability of discovering and selecting services and resources. The system is based on two main components: the Resource Selection Service (RSS) and the Application Directory Service (ADS). The specifications of the interfaces of these two components and the interactions between them (intra-SRDS interactions) are described in this document. The document presents a first specification of the interactions among the SRDS (as composed by the RSS and the ADS) and other XtreemOS components, developed by other project WPs. In particular, current specification was produced by using the requirements gathered from WP3.3 (Application Execution Management) [7] and WP3.4 (Data Management) [8], and taking into account the design constraints imposed by security and VO management requirements [9, 5, 6].

In order to describe the services/resource discovery and selection process in the document we use the "machete and bistoury" metaphor, where the "machete" is the rough cutting tool used to open the path in a wild and dense bush, whereas the "bistoury" is the fine and precise cutting tool used in surgery. In our metaphor the "wild and dense bush" represents "a large-scale, multi-domain, distributed systems" (a.k.a. Grid) composed of a large number of heterogeneous resources/services. The services/resource discovery and selection process is a two-step process. The first step involves RSS that will act as a "machete" handling the first-level of resource/service selection in a Virtual Organization (VO) by leveraging an overlay network which will hold node information, and answering to multidimensional range queries based on static attributes of the resources/services, and returning to the ADS a list of node identifiers (NodeIDs). In order to meet the project requirements of large VO management, the RSS will implement an efficient and highly scalable distributed algorithm handling range queries.

The second step of the discovery and selection process is performed by the ADS. The ADS will behave as a "bistoury", handling the second-level of resource and service selection, and answering queries expressed as predicates over the dynamic attributes of the resources/services. The ADS can also create an application-specific "directory service", exploiting the NodeID list received from the RSS. To provide scalability and reliability, and to support dynamically changing attributes and complex queries, state-of-the-art improvements to DHT techniques are being evaluated and will be exploited. The document describes the interaction between

the RSS and the ADS. Moreover, in order to allow the SRDS interoperability inside XtreemOS, the interaction protocols among the SRDS and other XtreemOS components are also summarized.

The document is structured in 5 chapters. The first chapters introduces the SRDS, its structure, and the main design issues. The 2nd and 3rd sections describe the Resource Selection Service and the Application Directory Service respectively. Those two sections have the same structure: an introduction, a detailed description of the system model adopted, of the internal component modules, of their interactions, and of implementation issues, then performance evaluation of the solutions/approaches adopted is reported. Finally, both chapters include notes about future development and research to be conducted, in the next phases of the XtreemOS project, in order to enhance the degree of innovation of the proposed solutions. The 4th chapter of this document is dedicated to the description of the interactions among the SRDS and other XtreemOS components. The 6th section concludes the document presenting possible future development of the whole SRDS design after M18 and in the long term perspective.

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

DOS Denial Of Service

HTTP HyperText Transfer Protocol

ID Identifier

IML Information Management Layer

IP Internet Protocol

JSDL Job Submission Description Language

JSON JavaScript Object Notation

KKRS Kernel Key Retention Service

libDB Berkeley Database library

M18 Month 18 (December 2007)

M24 Month 24 (June 2008)

MAPI Module-specific API

MEM Main Memory

NodeID Node Identifier

OS Operating System

P2P Peer-To-Peer

QP Query & Provide

RSS Resource Selection Service

SRDS Service/Resource Discovery System

SSL Secure Sockets Layer

TCP/IP Transmission Control Protocol / Internet Protocol

TCP Transmission Control Protocol

TTL Time-To-Live

UTF-8 Unicode Transformation Format, 8 bit

UUID Universally Unique Identifier

VO Virtual Organization

WP Work Package

XML eXtensible Markup Language

Contents

Chapter 1

Service/Resource Discovery System Design

This deliverable presents the current state of the design and specification of a prototype Service/Resource Discovery System (SRDS). The SRDS will offer to applications and other components of XtreemOS the capability of searching for and selecting services and resources.

In this chapter we describe the overall architecture of the Service/Resource Discovery System (SRDS), its main component modules – the Resource Selection Service (RSS) and the Application Directory Service (ADS) – and the interaction protocol between them. This chapter includes also some interoperability requirements derived by other XtreemOS software components in order to guarantee an appropriate level of interoperability of the SRDS prototype due at M18.

1.1 Document Structure

The document presents the design of the SRDS starting with its overall organization. Here the high-level specification is presented, based on the decomposition of the SRDS into two services, the Resource Selection Service (RSS) and the Application Directory Service (ADS), and including their interfaces and interactions.

The following Chapters 2 and 3 describe in detail the RSS and ADS architectures respectively, including their implementations at M18, and the future development. Test results validating the proposed solution for the M18 prototype are an important part of both chapters.

Chapter 4 presents a first specification of the interactions with XtreemOS components developed by other project WPs. We used 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 took into account the overall VO Management and Security infrastructure [9, 5, 6] designed by WP2.1 and WP3.5.

Chapter 5 summarizes the status of the SRDS prototype at M18, discussing the development roadmap and the open research perspectives.

1.2 The Service/Resource Discovery System

The SRDS continuously receives many different kind of data, both static and dynamically variable, associated to nodes, keys, applications and services. In order to provide organized information to other XtreemOS components, the SRDS has to perform several tasks, from simple key-based queries to range-based queries over dynamic attributes, while providing a storage service level (e.g. reliability) that is customizable according to the needs of each SRDS user. All these functionalities (more details will be provided in Section 1.3.3 and in Chapter 3) rely on efficient, scalable and reliable ways to gather and organize information concerning the status of resources, services and applications, as well as to deliver the information to other services and applications.

The issue of controlling and organizing a large number of computing resources in performing information-related tasks naturally arises and leads to the exploitation of distributed and peer to peer (P2P) techniques. Besides being inherently distributed, the software architecture of the SRDS has to cope with several different information kinds we mentioned, has to perform information processing tasks, and support various query semantics. This is the result of the SRDS interfacing to different XtreemOS modules, services and applications. To efficiently achieve this aim, we followed simple and effective general design principles.

- The *machete and bistoury* metaphor has been applied to complex query processing, to gain the advantage of existing scalable solutions without sacrificing more elaborate functionalities and future expansion to more refined techniques.

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

1.3 SRDS Architecture

As shown in Figure 1.1, the SRDS includes two main components: the **Resource Selection Service** (RSS) and the **Application Directory Service** (ADS). RSS and ADS cooperate in answering the most complex information queries, in order to provide both query semantics flexibility and overall system performance and scalability. Both the ADS and the RSS have a distributed implementation, based on P2P overlay networks. These are established on the set of computing resources managed by XtreemOS, where the SRDS is globally deployed.

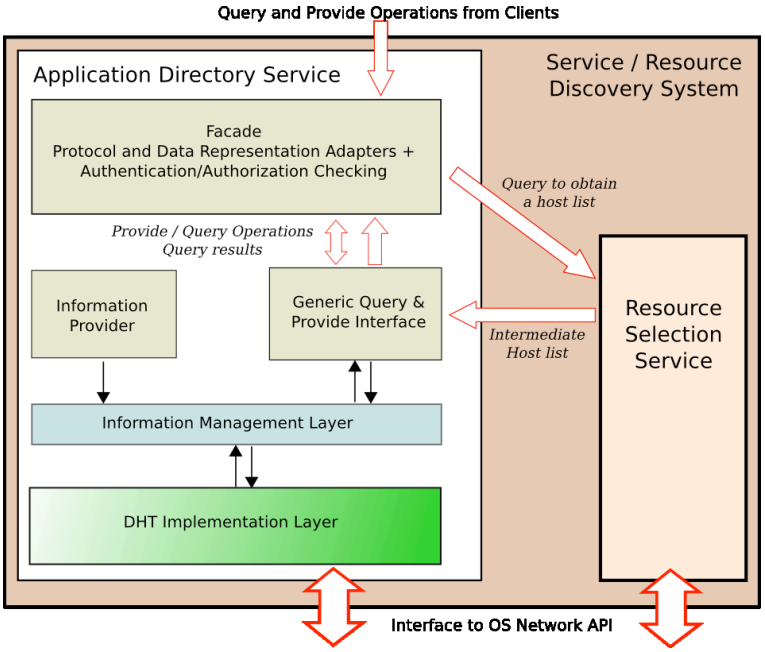


Figure 1.1: High-level software architecture of the SRDS. Here we show the interaction between the Application Directory Service and the Resource Selection Service in order to answer resource queries from the Application Execution Management

The SRDS *clients* are other XtreemOS modules and services, as well as user applications. The ADS thus provides the main interface of the SRDS toward the clients. The Facade exposes both very simple directory service primitives, as well as more complex information queries, required and invoked by other XtreemOS services and by Applications. The ADS Facade provides a basic encapsulation form for all kinds of requests.

The SRDS version due at M18 was designed taking primarily into account the requirements coming from WP3.3, thus the first prototype implements the features that are needed to resolve requests submitted by the AEM. In particular, the AEM asks to the SRDS to select a set of computational resources, according to some user-specified constraints and targets, in order to execute user Applications.

This is a kind of information query that can be hard to perform, as it involves range constraint over attributes, dynamic information on resources, and multiple fitting criteria expressing the user needs. This is also a good example to describe the SRDS architecture without immediately confronting with the heterogeneity of the SRDS interfaces, that is discussed later on. Nowadays efficient implementations of simple directory service functionalities over very large set of nodes are thoroughly studied. Unfortunately, the possibility to formulate/invoke more complex queries requires state-of-the-art techniques and is still considered an open research issue.

Although complex query semantics is difficult to implement over large P2P networks, especially where dynamically variable information has to be dealt with, nevertheless answer accuracy is critical to the efficiency of the following AEM negotiation phase.

To split up the complexity of the problems and ensure a readily available prototype, we have adopted a modular design where performance-critical operations are executed as a two-phase information selection process, in which the RSS and the ADS act as a *"machete"* and *"bistoury"* respectively.

The interaction between the ADS and the RSS resolves the AEM requests in an efficient and scalable way. In particular, when the SRDS has to satisfy an AEM request, it starts a process that returns a reply in two phases. The information collected at the end of these two phases is routed to the AEM again passing through the ADS Facade.

1.3.1 The RSS "Machete"

In this two-phase selection process the Resource Selection Service (RSS) will behave as a "machete" handling the first-level of resource selection in a Virtual Organization (VO). RSS leverages an overlay network that will hold static information about nodes.

The RSS answers to multi-dimensional range queries over static attributes, returning to the ADS a list of node identifiers (NodeIDs) that match the query. This task is performed with low overhead and high scalability thanks to a structured P2P overlay network ad search algorithm, that are specifically designed to solve the problem. A full description of this approach and its validation are reported in Chapter 2.

The output produced by the RSS will be a set of NodeIDs matching the specified criteria. This node set is larger than the required amount of resources for the application execution, but much smaller than the total number of resources belonging to the XtreemOS platform.

1.3.2 The ADS "Bistoury"

Starting from the results obtained by the RSS, the ADS performs a second-level of resource selection – acting as "bistoury" – that is more discerning than the previous phase.

The algorithm and the overlay network exploited by ADS cannot be AEM specific only, they have to be of more general applicability, as they are also used to solve simpler directory service queries coming from XtreemOS components other than the AEM. Beside simple query primitives, ADS provides specific access functions, that are tuned to solve range queries over resource attributes that can dynamically change their values (e.g. CPU load).

The ADS answers AEM queries expressed also as predicates over the static and dynamic attributes of the resources (e.g. a certain set of software and hardware

resources are currently available at the site). Solving range queries over dynamic attributes is inherently less scalable than answering queries about static values. Extensions of Distributed Hash Table (DHT) techniques and to dynamic attributes and complex queries can be employed. However, in order to improve the efficiency of the refining selection process, the ADS can exploit the size reduction of the candidate set for the query that is provided by the RSS. To this purpose, the ADS can also create an application-specific "directory service" using the NodeIDs received by RSS, those related to the resources (possibly) involved in the application execution. The design and implementation of the ADS modules are fully described in Chapter 3.

1.3.3 Query Semantics and Namespaces

Different clients (XtreemOS services) need to exploit different semantics of the queries, and key uniqueness generally won't allow matching among keys from different clients. This requires us to implement some form of namespaces within the SRDS. To show that this is possible, we anticipate some design concepts of the ADS which we will explain in full in chapter 3.

The actual query semantics depends on the implementation of each SRDS external operation as an algorithm, involving operations with the internal Information Management Layer of the ADS (actually, a DHT). From the point of view of the SRDS, the ADS can be seen as a black box providing different storage and query semantics at the same time.

By properly designing and matching Client Adapters and Query/Provide Interfaces, modules which are shown in Fig. 1.1 and are fully defined in Chapter 3, it is possible to exploit the DHT layer to perform complex operations. These will require the format and meaning of key employed within the DHT layer to be different from that exposed by the SRDS to its clients. Sophisticate key/value transformations can provide

- reverse, range and proximity query implementation,
- key-associated values that are complex data structures, with multiple fields to represent different *attributes* of the key,
- tuning and optimization of queries under specific assumptions on the attribute semantics.

For example, concatenation of the client type id (which is unique and obviously known to the client-specific interface) with the proposed key generates unique DHT keys for any kind of object. Choosing unique ID schemas and performing translation of the (key, value) couples sent to the DHT easily provides separate key spaces to different clients.

We stress the fact that the same technique can be (and will have to be) applied also to generate application- and VO-specific key spaces, as long as reverse queries

are never needed (a reverse query in this context can have the form "which client registered key ID?").

Thus, to fully obey the utilization requirements, the SRDS has to implement not only *visibility* scopes, but also *activation scopes*. Performance constraint on the implementation may suggest to return a different ADS instance to each running application. For instance, this is already planned for resource selection, a new ADS instance possibly being spawn whenever the Application Execution Management needs a list of resources for a job. This abstraction may actually reflect in the implementation, with a new P2P layer being brought up on demand, or the new ADS instance may be a proxy, linking the client to a new virtual key space allocated within an already existing DHT structure. The topic is discussed in section 3.2.7.

On the other hand, some SRDS clients explicitly require persistent, separate namespaces, where keys are able to even survive catastrophic failures as they are backed up on persistent storage. A typical example is the use of the SRDS to store data and metadata server lists for the Data Management Services. Other clients will need to specify their own structure of namespaces.

While at present time we are not going to explicitly address arbitrarily nested namespaces, a second design dimension in the SRDS is thus connected with XtreemOS-native and user-specified namespaces,

- that will generally be related to VO policies and exhibit a specific lifespan,
- that will not always be linked to the activation of an application or service.

These issues will have to be dealt with by the various modules composing the SRDS.

1.3.4 Pitfalls to Avoid in Defining the SRDS APIs

As the ADS is also used as a front-end for the whole SRDS, it will provide the SRDS toward the rest of XtreemOS. The SRDS will need to adapt to the different protocols and data formats specified by the XtreemOS work packages. Two points need to be explicitly mentioned, to avoid a costly refactoring process at a later stage.

- All interactions with the SRDS eventually have to comply to a common form of encapsulation (e.g. XML based) in order to provide authentication and authorization elements (e.g. keys or certificates) with each request. The encapsulation will allow placing basic security hooks within the SRDS (in the ADS Facade). These hooks will consistently interact with the VO support system, when needed, in order to check the validity of requests against current VO policies, or to extract SRDS relevant information from the request.
- The semantics of each kind of request, maybe except the simplest ones, will have to be completely defined to allow the SRDS optimize the exploitation of the underlying information management layer.

1.4 Internal Interactions among SRDS Modules

In this section we explain the interaction between Resource Selection Service and Application Directory Service. This interaction is needed to efficiently perform resource queries issued by the Application Execution Management (AEM, WP3.3). The basic architecture is depicted in figure 1.1, where the arrows represent the interactions among the entities composing the SRDS. Full explanation of the ADS architecture is provided in Chapter 3. All data exchanged between the ADS and the RSS are sent via (local) sockets. This can also happen, but it is not mandated, between the AEM and the ADS, evenntually the ADS Facade being in charge of the adaptation.

1.4.1 Internal Data Specification

Results are exchanged among the ADS and the RSS as XML encoded data. The list of static and dynamic attributes that describe each computational resource is based on the JSDL specification [18] and its XML schema. An in-depth analysis of the requirements from WP3.3 will be given in Chapter 4. We are still investigating about the set of useful attributes for the XtreemOS project.

Homogeneous resources (e.g. clusters) are represented in compact form, as single list elements. Beside saving in message size among the RSS and the ADS, which is probably negligible even on very large NodeID sets, preserving resource grouping helps avoiding excessive query fragmentation in the RSS and in the ADS.

WP3.3 Module-Specific API / Resource Selection The RSS receives requests by the ADS module-specific API related to AEM. In particular the RSS receives an XML file that encodes the submitted query.

Resource Selection / Query & Provide Interface The RSS (our “Machete”) exploits its overlay network to retrieve an host list satisfying static attributes specified in the WP3.3 query. Then the RSS sends a XML file - containing a list of IP addresses, ports and related static attributes - to the Query & Provide Interface of the Application Directory Service.

Query & Provide Interface / WP3.3 Module-Specific API When the Query Interface (our “bistoury”) retrieves a filtered node list exploiting the dynamic information within its DHT Layer, it returns it to the AEM through the WP3.3 Module-Specific API. The returned list contains all static and dynamic attributes of resources that are useful to the AEM tasks.

Chapter 2

The Resource Selection Service

2.1 Introduction

As detailed in Chapter 1, the role of the Resource Selection service is to select nodes based on static attributes. So far, such allocations are often controlled in a centralized or statically hierarchical manner. This is certainly the case for most grid-based resource management systems. We envision that for next-generation platforms, as those targeted by the XtreemOS project, consortia will follow an approach akin to that of PlanetLab and team up to provide a joint, very large, shared infrastructure consisting of tens to hundreds of thousands of nodes, if not more. Unlike PlanetLab, nodes will be able to join and leave the system easily. End users can subsequently install a wide variety of applications on such an infrastructure.

The size of such an infrastructure simply precludes having a consistent view on the current allocation of resources to applications, effectively making it infeasible to have any static solution to resource management. We claim that scalable resource allocation needs a fully decentralized and scalable solution in which participating nodes play equal roles to avoid introducing any bottlenecks.

One way of handling these problems is to use DHT-based peer-to-peer systems. In that case, each node becomes responsible for managing part of the complete set of resources, while looking up resources can be done relatively efficiently (see, e.g., [2, 27]). The problem with these solutions, however, is that we need to rely strongly on the willingness and ability of node A to manage information on the resources of node B. Moreover, we need to ensure that the information maintained by A is consistent with B's resources. A much better solution, to our opinion, is not to separate management of information on resources from those resources. Instead, each node having resources to offer, should be directly responsible for providing accurate information on those resources when asked for. This approach requires that queries for discovering resources should preferably be directed only to nodes that can provide those resources, and no other ones.

In this Chapter we present the solution that we devised within the XtreemOS project to address this problem. In our approach, each node is placed in a multi-

dimensional space, with each dimension representing a resource attribute type. As detailed in Chapter 1, Resource Selection Service focuses only on *static* attributes whereas *dynamic* attributed will be handled by the Application Directory Service, described in the next Chapter. A query is specified as a list of (*attribute*, *value interval*) pairs, effectively demarcating a subregion in this multi-dimensional space. Each node maintains a few links to other nodes, such that when it receives a query it can forward that query to a node that either lies in the associated subregion, or to a neighboring node closest to that region. Once a query is being processed within the associated subregion, it need merely be forwarded to enough nodes within that subregion as was asked for.

The efficiency of this solution is primarily dictated by the number of hops a query needs to take before arriving at its associated subregion. We evaluate our system using both simulations (with infrastructures consisting of up to 100,000 nodes), emulations (of up to 1000 nodes running on a wide-area grid cluster) and actual deployment on PlanetLab (up to 300 nodes). All experiments show that the system can support high churn and scales extremely well according to the number of nodes and the number of dimensions. Notably this last property is hardly ever satisfied by other solutions. To the best of our knowledge, letting nodes be responsible for management of their resources and information on those resources, combined with the scalability of resource discovery make our solution unique.

2.2 System Model

In our model, each node is characterized by a set of (*attribute*,*value*) pairs including, for instance, memory, bandwidth, and CPU power. For sake of simplicity, we assume that the number of attributes is fixed and known *a priori*. We also assume that attribute values can be uniquely mapped to natural numbers (although they need not be represented as such).

We model a network as a $d-$dimensional space $\mathcal{A} \triangleq \mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_d$, with $\mathcal{A}_i$ being the set of all possible values for attribute a_i and d the total number of different attributes considered. Every node X can therefore be represented as a single point with coordinates $(v_1, v_2, \ldots, v_d)$, with v_i being the value of attribute a_i for node X. A query is defined as a binary relation over $\mathcal{A}$, i.e. $q : \mathcal{A} \rightarrow \{0,1\}$ that selects which nodes satisfy the application requirements. The set of nodes for which q yields 1 represents the set of *candidates* to be allocated to the application. Note that q identifies a subspace $\mathcal{Q}(q) \triangleq \mathcal{Q}_1 \times \mathcal{Q}_2 \times \cdots \times \mathcal{Q}_d$, where $\mathcal{Q}_i \subseteq \mathcal{A}_i$.

As an example, consider a space based on five attributes: CPU instruction set, memory size, bandwidth, disk space, and operating system. Ignoring strict notational issues, an example query could then be formulated as:

CPU	=	IA32
MEM	$\in$	$[4GB, \infty)$
BANDWIDTH	$\in$	$[512Kb/s, \infty)$
DISK	$\in$	$[128GB, \infty)$
OS	$\in$	{Linux 2.6.19-1.2895, ..., Linux 2.6.20-1.2944}

A query can be issued at any node; there are no designated nodes where queries should initially be sent to.

At a more mundane level, and in order to keep matters simple so that we can concentrate on the core allocation algorithm, we assume that each node is allocated to only a single application at a time, after which it becomes unavailable for others. Furthermore, the network is connected: each node can reach any other node.

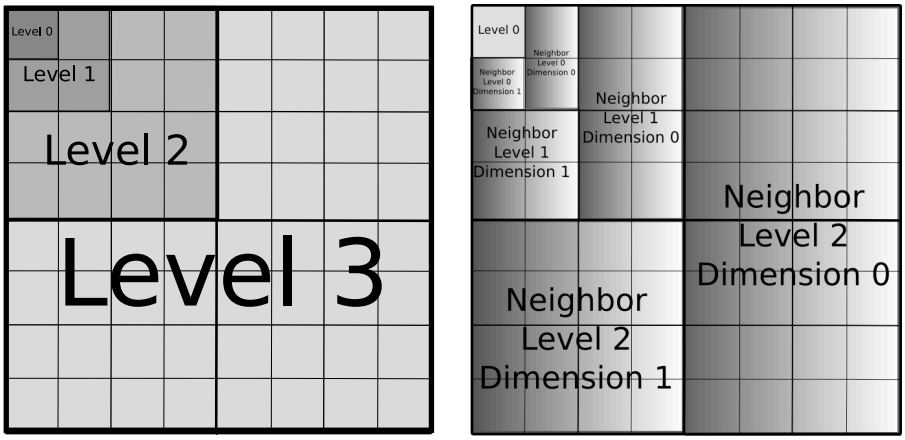
2.3 Protocol Description

In this section we illustrate the protocol for discovering resources that are to be allocated to a given job. We start with describing the properties of the overlay and will then detail how routing is performed. Finally, we discuss how we can effectively build and maintain this overlay structure in the presence of faults.

2.3.1 Overlay Network Topology

The model described in Section 2.2 is naturally represented as a hypercube. In order to scale up to thousands, possibly millions, of nodes, we must limit the amount of knowledge that each node needs to maintain. A naive, inefficient solution is to connect every node, for each dimension, with its most immediate neighbor, i.e., those nodes having the most similar attribute values. This way, when a node receives a query message q, it can simply forward the message in a greedy fashion to the neighbor closest to the area $Q(q)$. Unfortunately, this approach can dramatically increase both the latency and the traffic overhead: since the query can be issued at any node, it may need to traverse many nodes along every dimension to reach the target area Q.

We therefore opted for a hierarchical approach by recursively splitting the d-dimensional space into smaller spaces, called *cells*, and providing each node with a link to increasingly larger subspaces of which it is a member. This approach is akin to maintaining finger tables in DHTs. An example for $d = 2$ is shown in Figure 2.1(a). The largest cell has been partitioned into four smaller cells which each, in turn, have been split in four even smaller cells. To distinguish among the different cells, we introduce the notion of level l. The smallest cells are at level zero because no further nesting has occurred. These are denoted by the symbol C_0. C_1 cells are those obtained by composing four C_0 cells. Similarly, four C_1 will create a single C_2 cell and so on. More formally, given a hypercube of d dimensions and a level l, we say that a C_l cell is obtained by joining 2^d adjacent C_{l-1} cells. Obviously, every node X belongs to a unique C_l cell, which we denote as $C_l(X)$.



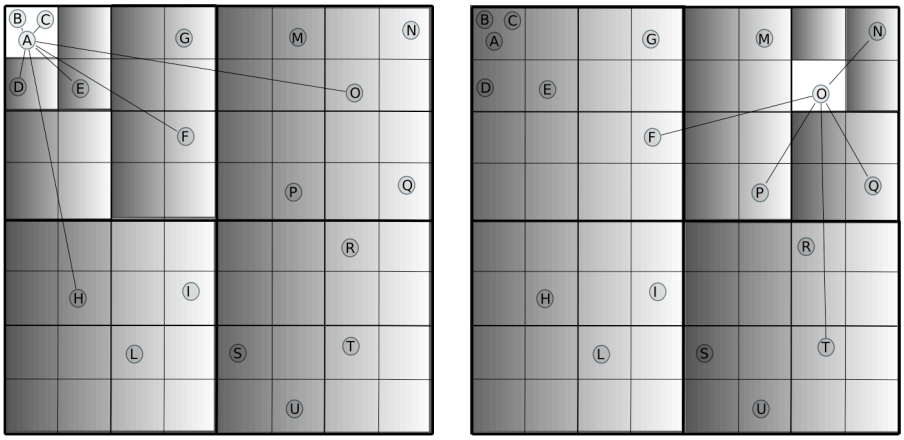
(a) Nested Cells. (b) Neighboring Cells

Figure 2.1: Attribute space partition with $d = 2$

Key to our approach is that when a node X is requested to handle a query q, that X forwards the query to a lowest level cell $\mathcal{C}_l(X)$ that overlaps with $Q(q)$. We will explain the details below. For now, this approach requires that for each level l, X in principle knows about nodes in $\mathcal{C}_l(X) \backslash \mathcal{C}_{l-1}(X)$. To this end, we construct, for each dimension, a neighboring subcell of $\mathcal{C}_{l-1}(X)$ by first splitting $\mathcal{C}_l(X)$ into two along dimension #0. The half in which $\mathcal{C}_{l-1}(X)$ is contained, is then split into two along dimension #1. This procedure is repeated until all dimensions have been considered, so that we will then have created d subcells at level l of $\mathcal{C}_l(X)$, each of which is adjacent to one "side" of $\mathcal{C}_{l-1}(X)$. Figure 2.1(b) shows the neighboring cells for a node A with the corresponding levels and dimensions.

We require that a node knows at least one other node (*neighbor*) falling in one of these subcells for each level $l > 0$. If no node is present in a given subcell, then no link must be maintained. The nodes in $\mathcal{C}_0(X)$ are arranged in such a way that X can efficiently broadcast a message to each of them, for example, through an epidemic protocol [12]. We use the symbol $\mathcal{N}_{(l,k)}(X)$ to identify the neighboring cell of node X on level l and dimension k. Similarly, the selected neighbor in $\mathcal{N}_{(l,k)}(X)$ is denoted as $n_{(l,k)}(X)$. It is worth noting that while the number of $\mathcal{C}_l$ cells grows exponentially with the number of dimensions, the number of $\mathcal{N}_{(l,k)}$ subcells (and hence the number of neighbors required per node) grows only linearly, and will thus not hinder scalability.

Figure 2.2(a) shows an example for a node A (for sake of clarity, we omit the connections among the other nodes). First, A is connected with all the other nodes in $\mathcal{C}_0(A)$ i.e., B and C. Then, for each neighboring cell $\mathcal{N}_{(l,k)}(A)$ depicted in Figure 2.1(b), it must choose one node $n_{(l,k)}$ to connect with. For $l = 1$, it has chosen nodes D ($k = 1$) and E ($k = 0$). For $l = 2$, it has two available nodes for $k = 0$ (F is selected). There is no node in $\mathcal{N}_{(2,1)}(A)$ so that no link is created. The same procedure is repeated for $l > 1$ (nodes O and H are selected). Similarly,



(a) Node A's neighbors (b) Node O's neighbors

Figure 2.2: Neighbor links for node A and O.

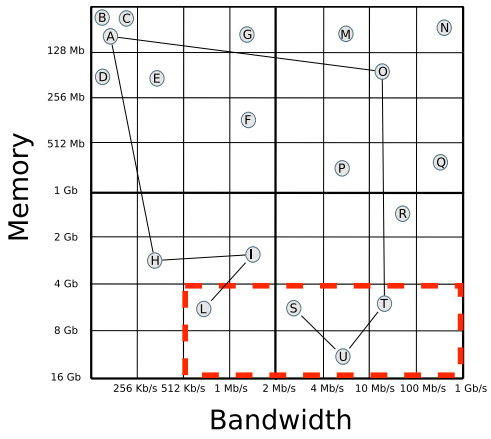


Figure 2.3: Example of query routing.

Figure 2.2(b) reports the links[1] of node O. As we will see next, these neighbors will be used to disseminate queries throughout the network.

2.3.2 Query Routing: The Rationale

We illustrate how query messages are routed by means of the example depicted in Figure 2.3. Assume that node A is interested in collecting $\sigma = 4$ nodes that have a network connection of at least 512 kb/s and 4 Gb of RAM. Graphically, this is represented by the dotted rectangle in Figure 2.3, representing the interest area Q. According to the aforementioned partitioning scheme, node A will first check whether itself or any other nodes in $\mathcal{C}_0(A)$ match the requirements. Since

[1] Note that links may be asymmetric. For instance, in this example O is a neighbor of A but not vice versa.

this operation is not successful ($\mathcal{C}_0(A)$ does not belong to Q), it will increase its scope considering higher level neighboring cells, until it finds one overlapping Q. In our example, this process ends with $l = 2$, since an overlap is found between $\mathcal{N}_{(3,0)}(A)$ and Q. Hence, node A will forward the query to its neighbor $n_{(3,0)}(A)$ responsible for that subcell (node O in the example). The latter will proceed in the same way but to avoid backward messages, it considers only $\mathcal{N}_{(3,1)}(O)$ or lower level cells.

Node O finds that $\mathcal{N}_{(3,1)}(O)$ partially overlaps Q and, hence, the query is forwarded to T, i.e, to $n_{(3,1)}(O)$. T will first include itself in the candidate set as it matches the query requirements. Then, since both $\mathcal{N}_{(3,0)}(T)$ and $\mathcal{N}_{(3,1)}(T)$ cannot be further considered to avoid backward propagation of the query, it can just consider $\mathcal{N}_{(l,k)}(T)$ with $l < 2$. Therefore, it routes the query towards $n_{(2,0)}(T)$, namely U, which fulfills the query requirements. Since A asked for 4 nodes, U continues to disseminate the query to S (which also matches). Now, S cannot propagate the query further and hence it replies back to U. Also U, T and O do not have alternative paths and hence, following the return path, the query goes back to A. Node A, however, can forward the query to H, since also $\mathcal{N}_{(3,1)}(A)$ overlaps Q. Here the propagation occurs as above and in the end the query reaches node L, whose attributes also match the query.

As shown in Figure 2.3, query propagation follows a depth-first tree rooted at the originating node (A in our example). This ensures that no loops are created and no useless transmission occurs. It is worth noting, however, that this tree is created on the fly each time a new query is issued, exploiting the links of the overlay network. Compared with traditional approaches, where a single tree is used, we argue that this solution is more efficient due to a better load distribution and lower maintenance costs.

2.3.3 Query Routing: The Pseudo-code

When a user needs a collection of nodes to perform her tasks, she contacts one node in the overlay network (possibly the one running on her PC) and passes the query to it. A query message contains the address of the querying node and the desired range of values for each attribute. A range is a pair of values representing the lower and upper bound, respectively. The job may specify both of them, only one or even none (if it does not care for any particular value for that attribute).

In addition, a job can impose an upper bound σ on the number of nodes it is interested in. This information will be used by our decentralized protocol to halt the propagation of the query once that threshold is met.

Finally, each query also contains two additional fields, *level* and *dimensions*, which will be exploited to forward the query. Initially, they are set to their default values (see Figure 2.4).

Each node stores two tables containing an entry for each query it receives (Figure 2.5). The first one, *pending*, is used to keep track of on-going queries. Each entry is associated with a time out $T(q)$: when it expires, the neighbor is consid-

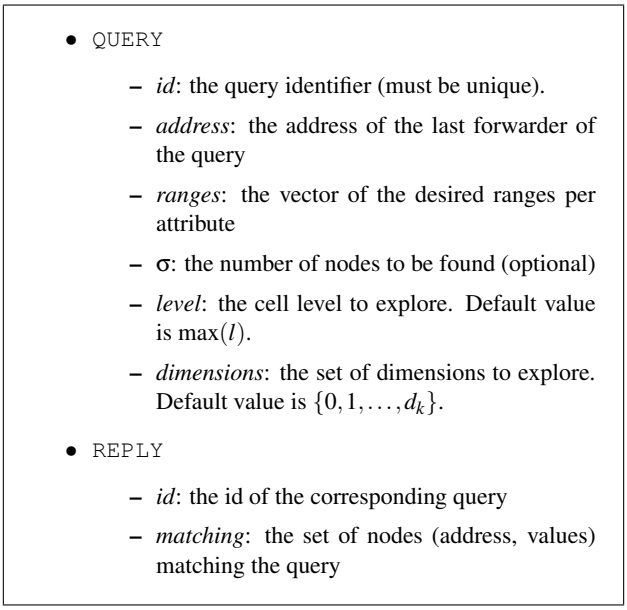


Figure 2.4: Message Formats.

ered to have failed and the query is forwarded again. The second table, named *matching*, includes the list of candidates that the node has retrieved so far for each query in *pending*.

As shown in Figure 2.6, when a node X receives a query it first adds a new entry in the two aforementioned tables (lines 1–2). Then, it checks whether its own attributes satisfy the request in which case it adds itself to the *matching* list (lines 3–4). Then, if further nodes are needed, it invokes the `forward` procedure to route the query to its neighbors (line 6). Otherwise, it replies back to the sender (lines 8–12).

Starting from the lower level, all neighboring cells are scanned sequentially until a cell overlapping Q is found. In this case, the query message is forwarded to the neighbor responsible for that cell and the procedure terminates (lines 1–7). To prevent this neighbor from sending the query back, the corresponding dimension of the neighboring cell is removed from the query (line 5). This way, when the neighbor wants to forward the query, it will be prevented to send the message along the same dimension.

Conversely, if no suitable neighbor has been found, nodes belonging to $\mathcal{C}_0(X)$ are checked to verify whether possible matches exist (lines 9–15).

Finally, if the query could not be forwarded, X immediately replies to the node from which it received the query by sending the set of matching nodes it found. This set can either be empty or contain X itself (lines 16–21).

Upon the receipt of a REPLY message, a node retrieves the corresponding query from the *pending* list and adds the addresses of the nodes included in the

Each node X hosts a set *neighbors*, containing one neighbor $n_{(l,d)}(X)$ per each neighboring cell at level l and dimension d. Neighbors in $\mathcal{C}_0(X)$ are kept in a separate set *neighborsZero*.
For each neighbor the following information is stored:

- *n.address*: the TCP/IP address of n
- *n.values*: a vector of all the attribute values of n
- *n.level*: the level of this neighbor
- *n.dimensions*: the dimension of this neighbor

To deal with queries, a node is also provided with the following vectors (maps) indexed by the query id:

- *pending*: contains the queries that have not been answered yet.
- *matching*: contains the addresses and the attributes of the nodes, matching the query, collected so far.

Finally, each node stores information about its own state in a record *self* with the following fields:

- *address*: its own TCP/IP address.
- *values*: a vector of all its attribute values.

To provide a more concise representation of the pseudocode, we will also rely on the following Boolean functions:

- overlaps(q, l, d, X): returns TRUE if $\mathcal{Q}(q)$ overlaps $\mathcal{N}_{(l,d)}(X)$
- matches(n,q): returns TRUE if node n's attributes fulfill query q's requirements

Figure 2.5: Data Structures.

Invoked by a user willing to query for a collection of k nodes.
`create QUERY` q

1: create a QUERY message q
2: $q.address \leftarrow self.address$
3: **for all** $\mathcal{A}_i \in \mathcal{A}$ **do**
4: $q.ranges[i] \leftarrow (min_i, max_i)$
5: $q.\sigma \leftarrow k$
6: $q.level \leftarrow \max(l)$
7: $q.dimensions \leftarrow \{0, 1, \ldots, d\}$
8: receive_query(q)

Invoked by a node receiving a QUERY message.
`receive_query QUERY` q

1: $pending[q.id] \leftarrow q$
2: $matching[q.id] \leftarrow \varnothing$
3: **if** *matches(self, q)* **then**
4: $matching[q.id] \leftarrow matching[q.id] \cup \{self\}$
5: **if** $|matching[q.id]| < q.\sigma \wedge q.level > -1$ **then**
6: forward(q)
7: **else**
8: create REPLY r
9: $r.sender \leftarrow self.address$
10: $r.id \leftarrow q.id$
11: $r.matching \leftarrow matching[q.id]$
12: **send** $r \rightarrow sender[q.id]$

Invoked by a node receiving a REPLY message.
`receive_reply REPLY` r

1: $q \leftarrow pending[r.id]$
2: $matching[r.id] \leftarrow matching[q.id] \cup r.matching$
3: **if** $|matching[r.id]| < q.\sigma \wedge q.level > -1$ **then**
4: forward(q)
5: **else**
6: create REPLY r'
7: $r'.sender \leftarrow r.id$
8: $r'.id \leftarrow r.id$
9: $r'.matching \leftarrow matching[r.id]$
10: **send** $r' \rightarrow q.address$

Invoked by a node to forward a QUERY message.
`forward QUERY` q

1: $sent \leftarrow$ FALSE
2: **while** $q.level \leq max(l)$ **do**
3: **for all** $d \in q.dimensions$ **do**
4: **if** *overlaps(q, q.level, d, self)* **then**
5: $q.dimensions \leftarrow q.dimensions \setminus \{d\}$
6: **send** $q \rightarrow neighbors[l, d]$
7: **return**
8: $q.level \leftarrow q.level + 1$
9: **if** $q.level > max(l)$ **then**
10: $sent \leftarrow$ FALSE
11: $q.level \leftarrow -1$
12: **for all** $n \in neighborsZero$ **do**
13: **if** *matches(n, q)* **then**
14: **send** $q \rightarrow neighbors[l, d]$
15: $sent \leftarrow$ TRUE
16: **if** $\neg sent$ **then**
17: create REPLY r
18: $r.sender \leftarrow self.address$
19: $r.id \leftarrow q.id$
20: $r.matching \leftarrow matching[q.id]$
21: **send** $r \rightarrow sender[q.id]$

Figure 2.6: Query routing protocol.

reply to its own *matching* list (lines 1–2). Then, if the number of candidates found so far is still lower than what was initially required and there are still some neighboring cells to explore (i.e., $q.level > -1$), the forwarding procedure is invoked

again (lines 3–4). Otherwise, a new reply message is created and filled with the addresses of the discovered candidates and sent back to the node from which the query message had been received (lines 6–10).

2.3.4 Securing the Protocol

In order to ensure the proper level of security, in accordance with the WP 3.5, we decided to maintain one separate overlay network per each Virtual Organization (VO). This way we can enforce that each node taking part in the overlay must have formerly been authenticated by the VO Manager component which is being developed within the WP 3.5. We also assume that, once authenticated, each node receives a signed certificate containing its own attributes list. When a node gossips with a previously unknown node, it can easily check whether it is a trusted node and which are its attribute values.

Each certificate will include an expiration time after which the certificate is not valid any more. Periodically each node is required to contact the VO Manager to renew its certificate. This way, undesired nodes can be prevented from being part of the overlay network by simply not renewing their certificates.

Finally, secure communication is ensured by means of SSL connections using the public key provided in the certificate to bootstrap encryption.

2.4 Overlay Maintenance

A major issue in running the above protocol is to efficiently maintain the overlay network in the presence of frequent node joins and leaves, for example, as caused by failures. Even if the topology were static, attributes might change during a system's lifetime due to hardware or software upgrades, thus introducing another source of dynamicity in the system.

To address this concern, we adapted and extended previous work on letting nodes dynamically self-organize into a connected overlay [32]. We exploit a two-layered approach. The lowest layer, executing the CYCLON [31] protocol, connects all nodes into a randomly structured overlay, essentially allowing every node to arbitrarily select another live peer from the current set of nodes [19]. On top of this, we run an *anti-entropy* protocol that periodically exchanges information on (*attribute*,*value*) pairs between two connected nodes, allowing each node X to keep only (higher layer) connections to live nodes discovered in neighboring $\mathcal{N}_{(l,k)}(X)$ cells. As discussed extensively in [32], this two-layered gossip-based approach for self-organization is extremely fast and responsive to changes in node membership. We evaluate the self-organization properties of our system in Section 2.6.6.

In this light, note also that query execution itself can easily handle failures. If a node does not succeed in forwarding a query message to an apparently failed node, it can select another node in the same subcell. Such a node will have been discovered using our epidemic protocol. Alternatively, it may also decide to explore a

completely different branch. At worst, the total execution time of a query will be longer, but nothing prevents its from making normal progress. Of course, if there are not enough nodes to choose from, the query will eventually fail.

2.5 Implementation

To assess the performance of our protocol, we built two implementations. We deployed the first implementation on the DAS-3 cluster [11]. The cluster is composed of 85 dual-CPU/dual-core 2.4GHz AMD Opteron DP 850 compute nodes. We emulated a system with 1000 nodes by running 20 processes per node on a total of 50 nodes. The second implementation runs on top of PeerSim, a discrete event simulator written in Java [21]. This allows us to explore setups with up to 100,000 nodes. Finally, for one experiment we deployed our system on PlanetLab.

Hereafter, we provide some details about the first implementation which will be merged into the first XtreemOS prototype[2]. This implementation is written in Java 1.6 and consists of two main components. The first one, `Recorder`, should run on a separate hosts and it is used to bootstrap the nodes by providing them with a random subset of address of nodes already part of the overlay network. It also serves to collect statistics, mainly useful for debug and evaluation purposes. Typically, this role will be played in the future by the VO Manager, which will perform all the necessary operations to join a new node.

The second component, `XOSNode`, instead, will run on every node and implement the protocol logic. In particular, it will start four different threads:

- `Cyclon`: this is the thread responsible to maintain a random network connectivity among nodes;
- `Vicinity`: this builds and maintain the hypercube-shaped overlay network;
- `Protocol`: this is responsible of forwarding queries and replies, using the routing information provided by `Vicinity`;
- `Dispatcher`: this thread receives messages and dispatches them to the appropriate thread.

Communication is built on top of TCP to ensure strong reliability. Thanks to the use of the `Dispatcher` class, only one TCP port is used, thus simplifying firewall configuration. Message are implemented in XML and are defined in separate classes extending the `Message` interface. This way, future changes in the format will not affect the rest of the code. Furthermore, the use of a text-based format like XML allows for more flexibility, since Java and non-Java implementations can co-exist.

[2]The interested reader can find additional details in the documentation in bundle with the code.

Parameter	Default value
Network size (N)	100,000 (PeerSim) 1,000 (DAS)
Query selectivity (f)	0.125
Max. no. requested nodes (σ)	50
Dimensions (d)	4
Nesting depth ($max(l)$)	3
Gossip period	10 seconds
Gossip cache size	20

Table 2.1: Default simulation parameters.

All configuration parameters, including the TCP port (default is 1905) and the address of the Bootstrap node, are parsed from a configuration file. The final format of this file and of the network messages will be decided later in the project, according to the needs of the other members using our services.

2.6 Evaluation

Based on the setups described in the previous section, we evaluated the performance of our system in terms of efficiency and correctness. Efficiency is measured in terms of overhead, that is the average number of queries that were routed through nodes that did *not* match the query themselves. Correctness means that each node that matches a query must be hit exactly once. We note that we systematically obtained 100% delivery in all experiments where the system does not experience churn. We return to the effect of churn on delivery in Section 2.6.6.

In all experiments, including the ones on the DAS, we first randomly populate the space with nodes following a uniform distribution, and give them sufficient time to build their routing tables. Depending on the number of attributes we use in our experiments, we thus build up a d-dimensional hypercube with a uniform distribution of nodes. Effectively, this allows us to consider the space as nicely built up from equally-sized d-dimensional cells. In later experiments, we drop the uniform distribution of nodes.

We generate queries by selecting a subspace in the hypercube such that it approximately contains a desired fraction f of the total number of nodes N, which we refer to as the *query selectivity*. Each query will therefore consist of a request for $f \times N$ nodes. Different queries refer to different subspaces. Each query is then issued repeatedly from every node in the system. Unless otherwise specified, simulations are based on the default parameters depicted in Table 2.1.

In the following sections we focus on the performance of the resource discovery algorithm, and ignore the costs due to the maintenance of the overlay. These costs depend on the gossip frequency: for each gossip cycle, each node sends one

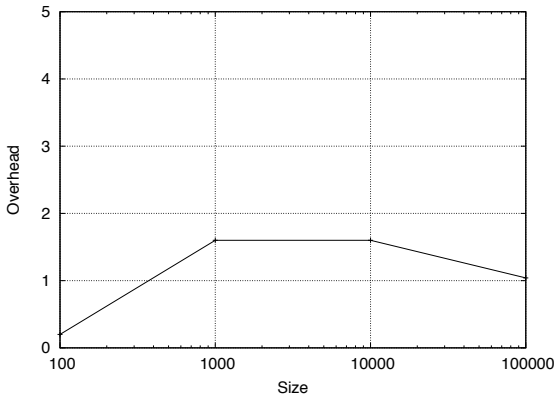
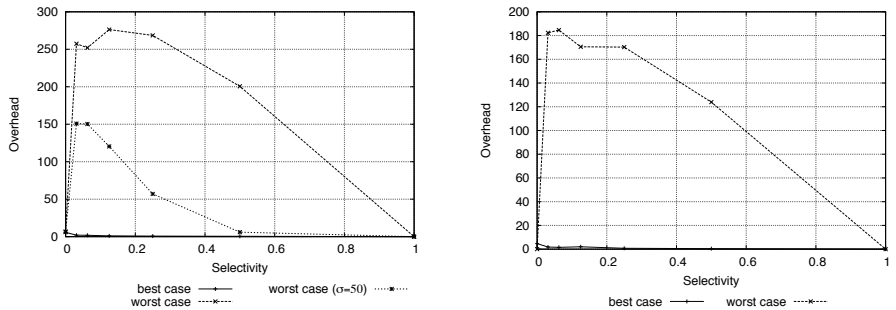


Figure 2.7: Routing overhead against network size (PeerSim)



(a) PeerSim (b) DAS

Figure 2.8: Routing overhead against query selectivity

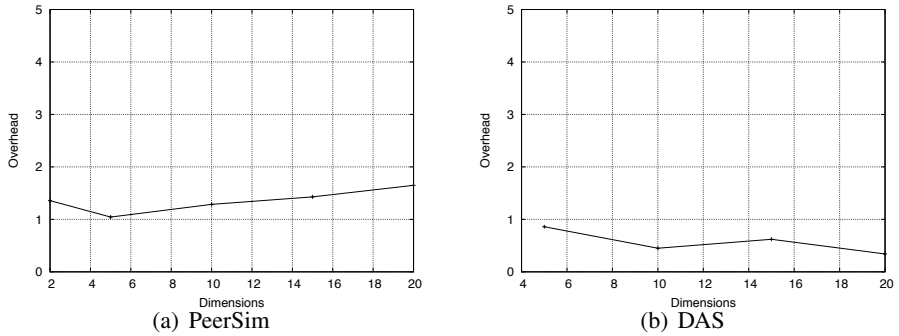
short message to another node in the system. Given a gossip frequency of 10 seconds, we consider this maintenance cost as negligible.

2.6.1 Effect of Network Size

Figure 2.7 plots the routing overhead of our system for different network sizes N. In all configurations, the overhead remains very small, on average below two messages per query. Interestingly, the overhead increases approximately logarithmically until 10,000 nodes, then decreases for large network sizes. This is due to the threshold $\sigma = 50$ (the maximum number of nodes requested): when the network is densely populated, a query often reaches its requested threshold very early and does not need to iterate through all cells that may overlap with the query.

2.6.2 Effect of Query Selectivity

We now turn to study the cost of queries with different selectivity, that is queries that match different fractions of the total system nodes. We studied two query workloads corresponding to a best-case and a worst-case scenario. In the best-case



(a) PeerSim (b) DAS

Figure 2.9: Routing overhead against the number of dimensions

scenario, each query is built such that it spans only a single cell and matches exactly the required number of nodes. In contrast, the worst-case scenario consists of queries that span multiple subcells such that every dimension and cell level is represented. Note that in practice cell boundaries can be set to specific, discrete values, and that we can force queries to respect those boundaries. Thus, for example, an application in need of 1.2–2.9 GB of memory, will be forced to request 1–3 GB. In our experiments, we did not make use of such forced query delineations.

Results based on the PeerSim and DAS implementations are shown in Figure 2.8. In the best-case scenario, the overhead remains negligible for all selectivity values. The worst-case scenario, however, shows different behavior, with higher overhead values, albeit still reasonable: e.g., in Figure 2.8(a), for $f = 0.12$ we have an overhead of just 257 messages against 12,500 matching nodes. This is due to the fact that queries that span multiple subcells must be split to cover all requested cells. This overhead decreases for queries with very high selectivity: in these cases, the system contains less nodes that do not match the query, and that can potentially create overhead.

In most cases, however, it is reasonable to assume that a user is interested in identifying a limited number of nodes out of a large population of candidates that match the query. Due to the depth-first search of our algorithm, such queries can easily be stopped when they reach the query's threshold σ. This explains why experiments with a value $\sigma = 50$ always exhibit very low query overheads.

Interestingly, the overhead in the worst case does not change significantly between 100,000 (Figure 2.8(a)) and 1,000 nodes (Figure 2.8(b)). This stems from the fact that the number of additional nodes to contact to reach the matching ones does not depend on the size of the network but on the topological properties of the $d-$dimensional space (i.e., the number of dimensions and the nesting depth), which are the same in both systems.

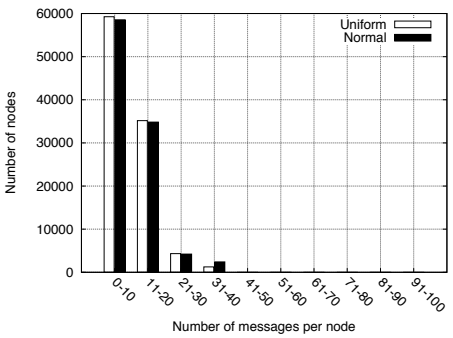


Figure 2.10: Node load distribution (PeerSim).

2.6.3 Effect of the Number of Dimensions

A major difficulty in multidimensional peer-to-peer systems is to be able to handle a large number of dimensions. Figure 2.9 charts the performance of our algorithm when using different numbers of dimensions, in both PeerSim and DAS setups. In the PeerSim experiments, the overhead increases slightly with the number of dimensions, while in the DAS setup it remains roughly constant. These variations, however, remain difficult to interpret, as such low overhead values typically fall within normal statistical error margins. What is noticeable, however, is that in all cases the routing overhead remains extremely low.

2.6.4 Load Distribution

In a large-scale system as the one we propose, it is important that the load imposed by the protocol is evenly distributed among nodes. This property is illustrated in Figure 2.10 where we show the load in terms of messages (queries and replies) dispatched by each node. We exercised the PeerSim system with two different node distributions across the space. In the first configuration, each parameter of each node is selected randomly in the interval $[0,80]$ using a uniform distribution. The second configuration creates a hotspot around the coordinates $(60,60,\ldots,60)$. Nodes were distributed around that coordinate, with a standard deviation of 10.

In both cases, we observe that no node receives a load significantly higher than the others. This is due to the gossip-based construction of the neighbor lists. Even in dense areas of the hyperspace, each node selects its neighbors independently. The inherent randomness of this neighbor-selecting protocol evenly distributes the links across all nodes of a given cell, which, in turn leads to an even distribution of load among those nodes.

2.6.5 Number of links per node

The next evaluation concerns the number of links that each node in the system must maintain. These links belong to two different categories. First, a node must

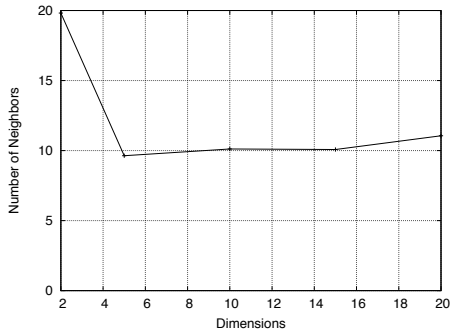


Figure 2.11: Number of links per node against the number of dimensions (PeerSim).

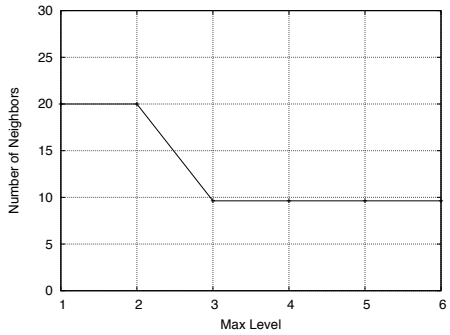


Figure 2.12: Number of links per node against the number of nesting levels (PeerSim).

maintain its *neighborZero* list which links to every other node present in the same lowest-level cell. The number of cells in the system is $\left(2^d\right)^{\max(l)}$, where d is the number of dimensions and $\max(l)$ is the nesting depth. This number grows extremely fast with d and $\max(l)$, so we expect that in practice a lowest-level cell will contain only nodes strictly identical to each other (nodes belonging to the same cluster, for example). However, even if that is not the case, we can relax this condition by demanding that the nodes in the same lowest-level cell are connected in an overlay. Such overlays are easy to construct and maintain, as discussed in depth in [19].

Second, every node must maintain one link to a node in every neighbor cell for each dimension and level. Each node thus has $d \times \max(l)$ neighbor cells. However, because of the huge number of cells, even a 100,000-node system such as in our PeerSim example will leave most cells empty. Nodes do not need to maintain a link to empty cells, so in reality the number of neighbor links that node must maintain will be significantly lower than $d \times \max(l)$.

This intuition is confirmed in Figures 2.11 and 2.12. The drop in number of neighbors per node from 2 to 5 dimensions, and from 2 to 3 levels, corresponds to

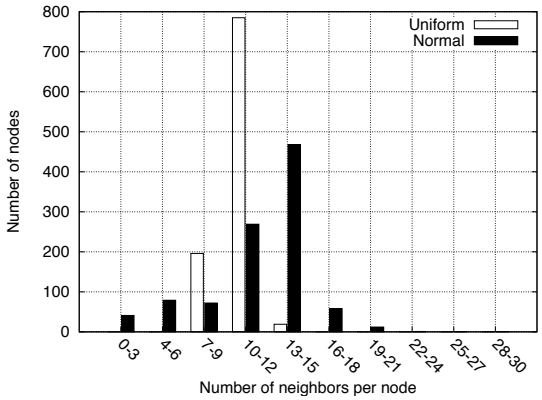
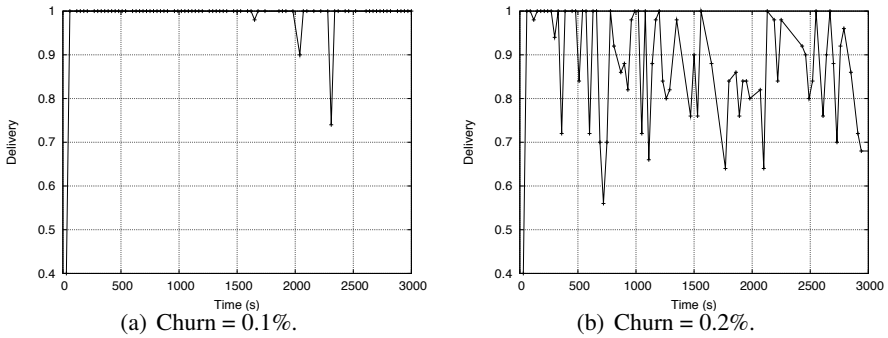


Figure 2.13: Neighbor number distribution (PeerSim).



(a) Churn = 0.1%. (b) Churn = 0.2%.

Figure 2.14: Delivery under churn (Peersim).

the reduced size of the *neighborZero* lists[3]. For higher numbers of dimensions and levels, the number of links that a node must maintain, both in its *neighborZero* and to its neighbors, is virtually constant.

Figure 2.13 plots the distribution of the number of links that nodes must maintain in the PeerSim system, under uniform and normal distribution. In both cases, these numbers remain very manageable, under 20 links in total[4]. We note, however, that the normal distribution case requires slightly more links per node. This is due to the fact that *neighborZero* lists will grow in the cells around the hotspot.

2.6.6 Delivery under Churn

Experiments presented so far assume that the list of nodes taking part in the system remains stable. This is of course unrealistic, and we should rather foresee that the network will exhibit a large degree of dynamicity due to massive node joins and

[3]For $d < 5$ and $\max(l) < 3$, the number of neighbors maintained by each node is bounded by the gossip cache (equal to 20 in our configuration).

[4]This number takes into account only the links used to route queries. This number can thus be significantly lower than the gossip cache size.

leaves. In particular, ungraceful node departures may represent an issue, since the routing tables of other nodes should be updated to maintain correct routing. We however claim that no particular measure should be taken to handle churn. Instead, we expect the underlying gossip-based protocol to continuously maintain correct routing tables.

To support this claim, we evaluated the delivery of the PeerSim system when 0.1% (resp. 0.2%) of the system nodes leave the system and re-enter it under a different identity every 10 seconds. The 0.2% value corresponds to the churn rate observed in Gnutella peer-to-peer networks [25]. Note, however, that grid systems are considerably more stable, and that recent studies of large-scale grids show much lower churn values [16].

We measure the delivery per time interval by issuing one query every 30 seconds. Queries do not use any threshold value, so a delivery of 1 means that the query reached $f \times N = 12,500$ matching nodes. As shown in Figure 2.14, a churn of 0.1% is not sufficient to significantly disrupt the delivery. Under a churn of 0.2% the delivery decreases, but remains around 0.8. Note that we expect most users of a real system to issue queries with a threshold. In such cases churn would only slightly reduce the number of reachable matching nodes to choose from, but most queries would be satisfied according to their specification.

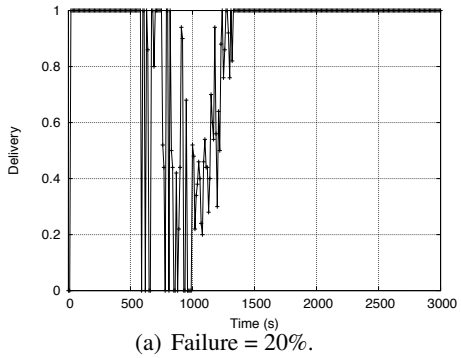
Also note that in these experiments, to avoid any bias, we did not adopt the strategy described at the end of Section 2.4. If we did, delivery would be steady, although latency would increase, because nodes, upon the detection of a failed neighbor, would wait for the overlay network to be repaired before forwarding the query.

2.6.7 Delivery under massive failure

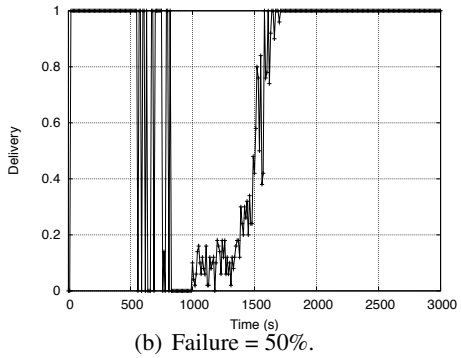
The next experiments show the behavior of our system when facing a massive simultaneous failure of a large fraction of the system. Again, we measure the delivery over time before and after the failure. Similarly to the previous section, we do not use any threshold values nor the just mentioned strategy so as not to bias the evaluation

Figure 2.15 shows how the PeerSim system handles massive failures of respectively 20%, 50% and 90% of the total system size. Here, we evaluated the delivery once every 10 seconds. At the time of the failure, a number of routing paths get disrupted, so the delivery oscillates across a broad spectrum. Rapidly, though, the system re-organizes itself until the delivery returns to 100%. In the case of 50% simultaneous node failures, the system needs only around 15 minutes to recover. This value may however be tuned by changing the gossip frequency. Only in the case where 90% of the system fail simultaneously, the delivery could not be restored. In this case, the overlay was partitioned by the massive failure so full recovery was impossible.

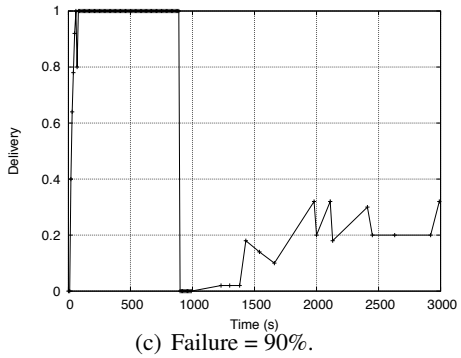
Similar graphs are shown in Figure 2.16 for massive failures on the DAS. Here, for practical reasons, we measured the delivery only once every 30 seconds. We



(a) Failure = 20%.



(b) Failure = 50%.

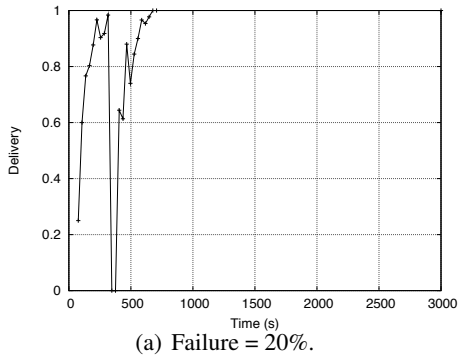


(c) Failure = 90%.

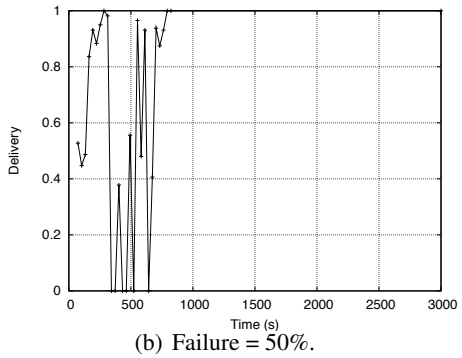
Figure 2.15: Delivery after massive failure (PeerSim).

observe similar behavior as in PeerSim, with a drop in delivery at the time of the failure, followed by a progressive recovery.

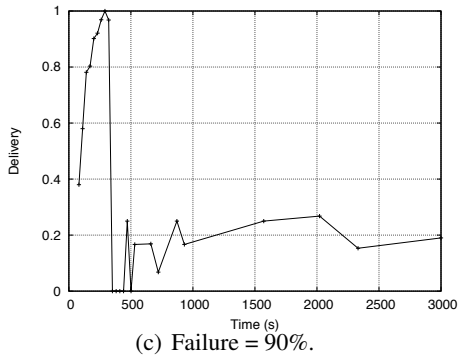
A similar experiment run on PlanetLab is presented in Figure 2.17. Here, we started our system on 302 nodes, and artificially increased the natural churn of PlanetLab by killing 10% of the network every 20 minutes. These nodes were not replaced, so the system shrinks over time. Here again, we observe near-optimal delivery.



(a) Failure = 20%.



(b) Failure = 50%.



(c) Failure = 90%.

Figure 2.16: Delivery after massive failure (DAS).

2.7 Future Development and Research

Future grid systems will be too large to support (semi-)centralized resource discovery, as is currently done in many systems. We have presented a fully decentralized protocol to identify nodes according to their properties. Each node represents itself in an overlay where resource discovery queries can be routed. We have shown based on simulations and actual deployments that this protocol scales well with the number of nodes and with the number of dimensions. The overlay is based on a gossip-based infrastructure which continuously maintains its routing tables. This makes our system extremely resilient to churn.

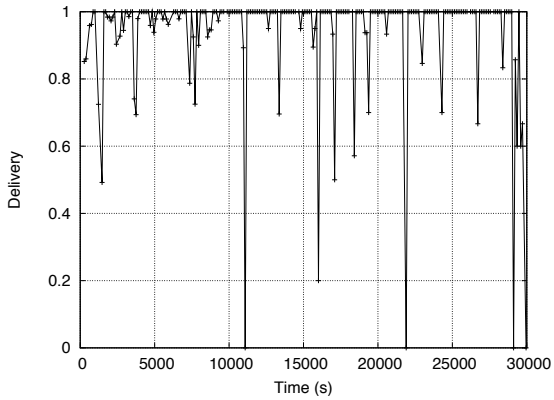


Figure 2.17: Delivery under repeated massive failures (PlanetLab).

To improve the speed of the discovery process, one can further add support for caching and network proximity. We present them as separate optimizations because their inclusion is still under discussion and, in case, they will be included only in a later stage of the project.

Runtime Reconfiguration. During protocol execution, one may want to add or remove attributes, effectively changing the search space. The simplest option would be to shutdown the system and bootstrap it from scratch. This, however, may be practically infeasible. We propose an alternative solution. The basic idea is to use the gossip protocol to also disseminate information on the overlay structure. As soon as a node notices that an attribute has been added, it simply starts contacting known nodes to discover which ones will become its neighbor in the new dimension. It is worth noting that even during this transient phase the protocol still behaves correctly, although further constraints on the added dimension will be ignored.

Query Caching. In many scenarios, we can observe that there are hosts that are accessed more often than others. This set may include highly powerful machines or ones equipped with popular software. Relying on such properties, a node can decide to update its neighbor set by selecting more popular nodes. Indeed, since any node in a given cell can be chosen as that cell's representative, we can bias node selection to the most popular ones to speed-up query propagation. For instance, if in the example in Figure 2.3, node A realized that node S is selected very often, it would replace the connection with O with a new connection with S, thus reducing the number of hops. Note that this process can take place at each node differently without any need of distributed synchronization.

Network Proximity. A general overlay routing does not consider the topology of the underlying network, often leading to suboptimal performance. In our approach, this can easily be compensated for by following a preferential attachment scheme

that also takes network proximity into account instead of only (*attribute*,*value*) pairs.

Chapter 3

Application Directory Service

In this Chapter we describe the architecture and the features of the Application Directory Service (ADS).

Section 3.1 reports the main issues related to the targets and the general requirements that the ADS has to satisfy. In section 3.2, we describe the ADS architecture as it has been designed, discussing all its basic modules and the foreseen developments in the long term. Section 3.3 reports about the first ADS prototype, actually a subset of the main architecture, available at M18. For this prototype, we describe in detail the implementation and the features of the existing modules. In section 3.4, we report about the experiments performed to evaluate the scalability and reliability of the ADS. We summarize results in section 3.5, and finally in section 3.6 we describe the future development plans, and the foreseen research issues concerning the ADS after M18.

3.1 Design Issues

The ADS is required to provide a flexible directory service supporting

simple queries – basic key-value queries with general content issued by any other XtreemOS modules and by running jobs and applications

complex queries – more refined forms of querying (range queries are one example) issued by some modules of XtreemOS. Here a specialized semantics has to be defined on the supplied data, which is exploited within the ADS architecture to optimize the queries.

dynamic attribute queries – simple and complex queries can be performed on key-value pairs which are subject to dynamic update, e.g. in order to account for real-time measured quantities and properties, for the sake of monitoring or to support decision making.

We target a large collection of nodes which are on average stable, but not immune to occasional failures and churn. In order for the system to be practically

useful and scalable, the overhead of running the ADS on top of the physical nodes should be valuable and bounded. We care for the scalability of the service with respect to

- efficiency of resource exploitation, measured possibly by the average and maximum amount of local resources employed (open sockets or allocated memory), by the number of exchanged messages per node and volume of the network traffic,
- time needed to answer queries,
- the amount of churn and failures the ADS can tolerate without degrading performance
- the amount of churn and failures the ADS can tolerate without loosing information (where a threshold has to be defined relative to the failure probability).

The most scalable approach to answer simple queries is based on Distributed Hash Tables (DHT). Appropriate DHT algorithms have been studied since several years, and provide the scalability and reliability properties that we need for large systems, being logarithmic in the number of hops and messages sent per query.

It is also known that complex queries and queries over dynamic attributes are a tougher issue, involving more recent research results, and that more complex algorithms are needed in order to efficiently answer those queries.

While the ADS is bound to support both, a general architecture has been devised in order to split the problem and solve it by stages. A specific ADS architecture has been designed to perform each query according to the most efficient algorithm available for that kind of query, and to allow releasing an early prototype which provides essential functionalities to allow the first integration at M18.

3.2 Overall ADS Design

We give an overall figure of the ADS by quickly describing all of its composing modules as shown in Fig. 3.1. We will dwell in details and mention in-depth issues in the following sections (from 3.2.2 to 3.2.7). From now on, and unless otherwise stated, we will address as **clients** of the ADS all other XtreemOS modules which need to interact with it.

Facade (sec 3.2.2) The facade is a generic adapter, providing basic encapsulation for all client requests; a common form of request boxing is needed as a hook to apply the VO policies which control authorization and access to XtreemOS services as the ADS. We stress the fact that the Facade module is used to actually reach the whole SRDS, so defining the ADS interfaces is paramount to defining the Service/Resource Discovery System (SRDS) ones.

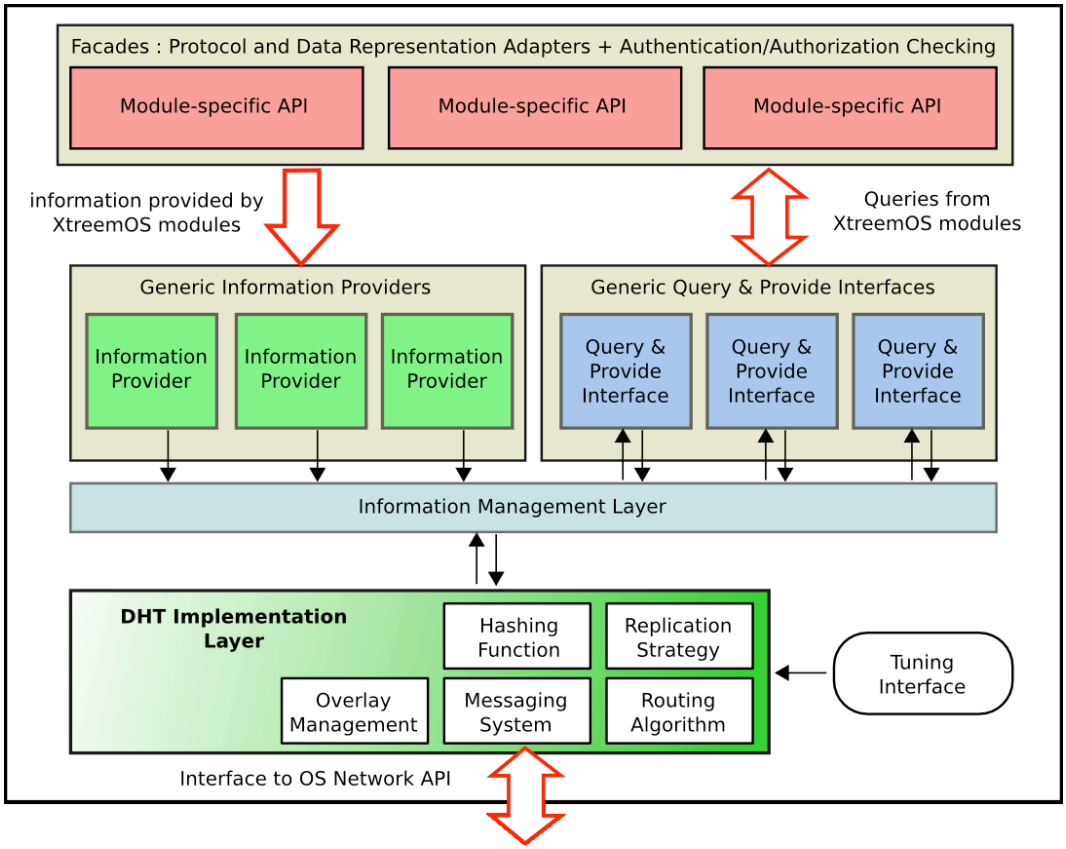


Figure 3.1: Overall design of the ADS

By exploiting the facade design pattern, the ADS is designed to be independent of the actual protocol used to access it (e.g. custom XML messages as opposed to an HTTP subset, see Chapter 4).

Module-specific API (sec 3.2.3) Generally speaking, each client will have a specific API adapter (that we will call MAPI) performing custom data representation translation as needed.

Generic Query & Provide Interfaces (sec 3.2.4) Each query interface performs a kind of information translation algorithm required for a set of query and provide operations (usually associated to a specific client type). The double name is meant to remind that matching algorithms have to be implemented to translate into elementary operations all the *provide* and *query* operations with corresponding semantics and use. The translation allows structured and dynamic information to be efficiently stored in the underlying information layer, and allows to optimally exploit the DHT layer for complex queries.

Generic bidirectional modules can still be associated with a specific client's need, but are always accessed through the upper level API.

Generic Information Providers (sec 3.2.5) These modules are exclusively intended to provide information to the DHT layer. Some of them will retrieve information directly from the local host. Providers may exploit a simple implementation as they never return any result.

Information Management layer (sec 3.2.6) The common interface of the whole ADS toward the all information management functionalities. This is intended to provide a common put/get approach over (key,value) pairs As implementation and research proceed, and as the integration with XtreemOS will show necessary, extensions to the basic paradigm will be added (e.g. governing the expiration time of the information), to allow full use of the DHT Implementation layer.The Information Management Layer API shall allow to implement all the needed high-level SRDS operations, and once the IML API has been fixed, almost any concrete DHT solution should suffice to implement the IML.

DHT Implementation layer (sec 3.2.7) The DHT Implementation layer provides and manages the Peer-to-Peer (P2P) layer of nodes. DHT behavior may be tuned according to client needs by having ADS modules supplying additional information (calling module, namespace restrictions) which allow the IML to select among several policies or P2P networks.

3.2.1 The Process of Query Translation

Through the Facade and its contained MAPI modules the ADS receives abstract *provide* and *query* operations that are translated to (sequences of) simpler ones, to be executed exploiting the DHT layer.

We took as a starting point the kind of queries that all DHT systems support, i.e. get and put operations on (key, value) pairs. The set of information-related functionalities required by the various XtreemOS modules and services is rather broad, as we evidenced in Section. 3.1.

A general form of information query[1] has several components.

key a unique key value used to store and retrieve a unit of information.

value, or attribute list the value linked to a key can be set, retrieved and updated; it is often convenient to see the value as a data structure (in most cases a list or a set) containing all the elementary attributes of a given key; operation on elementary attributes given a specific key may be supported by the IML.

execution context the specific execution context can affect how the actual query is performed in various ways:

[1] We will use a DHT to implement the IML, but the discussion is not restricted to DHTs only.

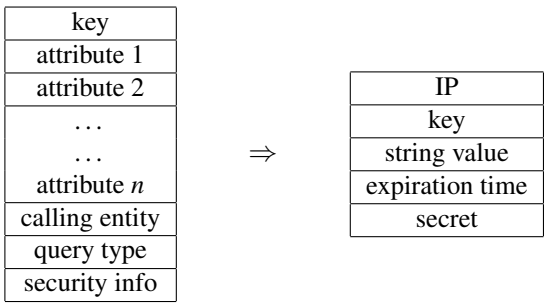


Figure 3.2: Schematic translation from high-level information queries to DHT-based queries. In the example, we use the Bamboo API, described in Sect. 3.3.1.

- some kind of information is only accessible to a specific service, e.g. to the modules implementing it; the XtreemFS Grid file system may need to cache user secrets and certificates that should never leak to the level of user applications;
- some types of queries will have optimized / enhanced implementations, relying on specific assumption about the keys and values to broaden / streamline information processing;
- user identities and roles within a Virtual Organization (VO) may affect the kind of query and the kind of data that a specific query is allowed to return;
- application, VO and user specific spaces may need to be provided at different levels, in oder to simplify real-time collaboration of users and application within the same VO.

security information authentication and authorization tokens may be needed for a specific information management operation, security being a special form of execution context, with its own set of mandatory rules and special system support. Security information has to be translated in a form suitable to the IML in all cases where the IML is entitled to perform a check and provide a given security level. Note that this does not happen as a rule, as most VO-related policies will be checked *before* reaching the IML level. The IML is in charge of security enforcing when

1. Security checks depend on the actual information queried, and are related to the semantics of that information.
2. IML operation go through the network, and data confidentiality has to be ensured on the layer.
3. IML operation go through the network, and authorization/security checks will help improving the IML performance or reducing the chances of Denial-Of-Service (DOS) attacks to the SRDS.

The SRDS, and in particular the ADS, have to implement the functionalities required by their clients in term of simple operations on the IML. Correspondingly, the various components of an information query have to be eventually mapped to the parameters of the IML API. The situation is exemplified in Fig. 3.2.

The minimal API being the standard one, based on get/put/remove over (key,value) pairs, most DHT implementations provide additional features which can be used to help perform the translation. For instance, the Bamboo DHT used in the M18 ADS prototype provides

attribute lists — Bamboo values are managed as list elements for any given key, the list monotonically increasing with each matching put

security through passwords — deleting any specific key and its linked value is only possible when knowing a key-related secret.

The full list of Bamboo features is given in Sect.3.3. Here we only underline the differences between high level properties ensured by the SRDS, and DHT features that the SRDS can exploit and affect the query translation process.

A translation example

As an example of translation we consider the Grid File System wanting to manage a list of data/meta-data servers, and examine its translation into Bamboo DHT primitives.

Several modules of the Grid File System implementation will have to register a *server address* (attribute) with a given *unique id* (key). More attributes will be defined by XtreemFS, including a geographic area of the server. As requirements in this simple case, (1) we don't want any unrelated XtreemOS module to fiddle with the server list, and (2) the list of servers (keys) maintained will never expire.

```
SRDS-register-server(ID, attribute-list, authorization)
SRDS-query-server-by-ID(ID, attribute-list,
authorization)
SRDS-query-servers-by-area(area, ID, attribute-list,
authorization)
SRDS-deregister-server-by-ID(ID, authorization)
```

The register operation has to first check the authorization (i.e. who is the caller, and if it is allowed to register a server), then it issues a

```
put(DHTID, attribute-list, never-expire, secret)
```

where the secret is to be derived from the authorization parameter. Note that the identifier (ID) passed to the DHT layer is generated starting from the ID parameter by adding some kind of prefix to generate a separate namespace. This is done

to prevent any chance of name clashes of IDs defined e.g. by the Application Execution Management (AEM) with those defined by user applications.

Actually, to allow retrieving servers from a certain area, the registration operation may need an additional

```
put(area, DHTID, never-expire, SRDS-secret)
```

to enlist the new server under the "area" key. This approach allows the "query by area" operation to exploit the DHT layer in a straightforward way.

3.2.2 ADS Facade

The Facade module of the ADS provides a framework for the functionalities that are common among all MAPI modules, to avoid reimplementation, in order to

- allow future evolution of the common functionalities, e.g. tracking improvements and changes in the API of security services with minimal effort
- allow easier customization of the SRDS system for specific needs; for instance for accepting/rejecting requests from other hosts via proxies.

The Facade module is a container class of the MAPI modules. In the following we describe the groups of functionalities provided to the contained modules.

Encapsulation Message encapsulation format with boxing and unboxing primitives. A common format of boxing communications toward the ADS is used, that is independent of the actual message exchanged and of its format.

- The format of the contained message is free (e.g. ASCII, XML) as long as it can be effectively encapsulated. For this reason, although we have no requirements to support them at the moment, binary message formats would likely have to be re-encoded, e.g. in Base64 or hexadecimal notation.
- The message wrapper also contains information about the interaction with the ADS, that is needed in order to provide a proper level of security to the interaction. This information is encoded in the format of the wrapper, so as to allow a single couple of routines in the Facade to perform message parsing and generation tasks for all kind of messages, regardless of the contained MAPIs they concern to.

Security Authorization and security primitives are provided, based on the information contained in the message wrapper, and performing the needed calls to VO authentication and authorization mechanisms provided by WP 2.1 and 3.5. Security and VO hooks will generally be contained only within the Facade, to minimize the development cost of security, and they will be able to exploit knowledge about the kind of interaction ongoing. For instance, access to functionalities related to AEM will likely obey to VO policies different from those concerning the Grid File System.

External Routing The Facade provides routing functionalities of messages between the MAPIs and the clients, to allow more MAPIs to use a common transport protocol (e.g. HTTP) and share a single communication point.

Internal Routing The facade can also provide homogeneous communication mechanisms toward other modules within the SRDS (e.g. the Resource Selection Service – RSS) as a primitive functionality to the contained MAPIs.

Communication Management It is under evaluation if the Facade should handle a standard communication channel, that is socket operation possibly through Secure Sockets Layer (SSL), in order to further simplify the implementation of MAPI modules in the most common case.

3.2.3 Module-Specific API

Module-Specific API modules provide functionalities which are tailored to dealing with a given client (module, service of XtreemOS, or application). In particular, a MAPI is the endpoint of the communications between clients of a given type and the SRDS.

- Each MAPI deals with a specific communication channel, protocol and with a specific client. It can exploit these assumptions in processing each operation request. As a rule, a MAPI handles all interaction with a specified client.
- A MAPI can provide additional security and access controls when appropriate, beside those performed by the Facade functionalities.
- A MAPI can read and translate the message format used by the client within the message envelope (e.g. JSON [20] for the Data Management Services); can decode and translate the data into a format suitable to all lower level modules.
- If needed for its task, a MAPI is assumed to know about the attribute semantics for the type of operation submitted by its client.
- A MAPI can dispatch query and provide operations (as well as any other operation and result) to the proper Query & Provide module (and back to the right client) exploiting the aforementioned information about the kind of operation, the kind of client involved (possibly the identity of the client), and the data posted with the query or returned with the results.

Namespace Translation in MAPI Modules Here we give a more formal definition of the translation of operations, starting from their form on the ADS interface:

operation_{ADS} = { **request, XOSCert, ClientInfo**}

where the first level of encapsulation is a standard envelope structure defined by the SRDS. The contained fields can be described as

request	= { **op, key, value, parameters** }
ClientInfo	= { **ClientType, ClientId** }
XOSCert	authorization certificates to check against XtreemOS VO policies

where **op** is the operation code.

A MAPI receives such an operation and augments it with information received from the Facade module, to obtain a representation operation_{MAPI}, where information implicitly stored in the message envelope, e.g. in the security related fields, has been extracted from the operation wrapper.

The MAPI produces as output an operation description for a Query & Provide module, containing the following fields

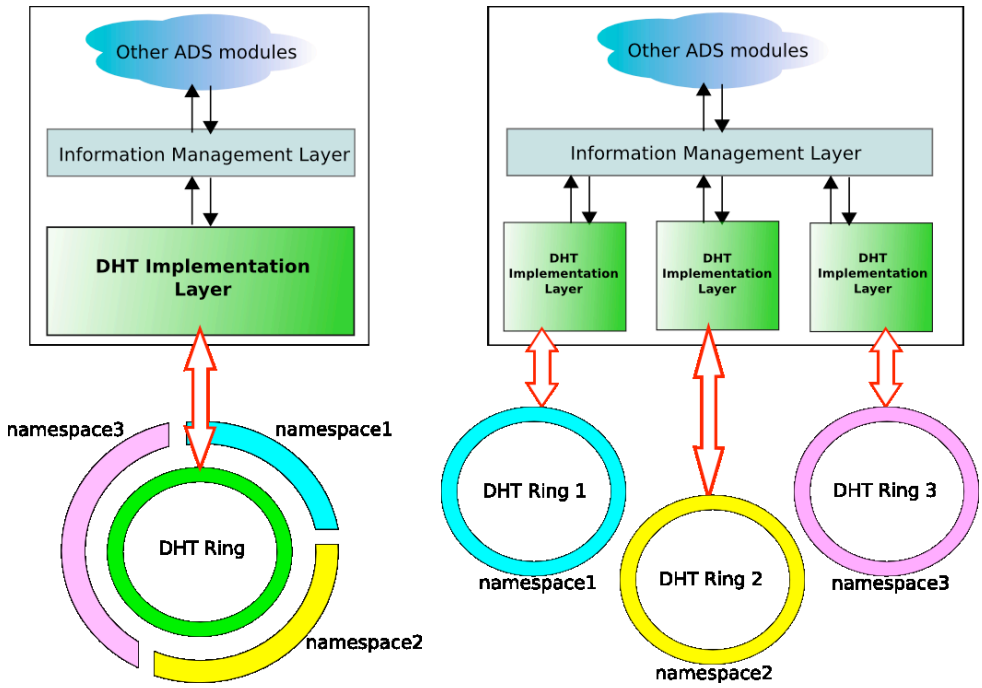
operation_{QP}	= { **op, key**$_M$**, value**$_M$**, Nspace, ClientType, ClientId** }
Nspace	= f(**key, value, parameters, ClientInfo**)

where Nspace is a namespace specification derived from the operation parameters and the client information.

3.2.4 Query & Provide Interfaces

A Query & Provide (QP) interface is a module that translates operations defined in the client interface into those provided by the IML, which are closely related to DHT primitives. As we have seen in the example in Section 3.2.1, a single query may be resolved by an algorithm performing several operations on the IML. Each QP interface is a separate design, depending on the kind of operations that we need to perform. The common traits of all QP modules are the following ones.

- High level operations are interpreted by algorithms defined on top of the IML supported operations.
- The translation of functionalities in not 1 to 1, each provided high-level functionality can be an algorithm over the DHT layer, composed of several steps.
- Clearly, the algorithm implementing each operation is dependent on the data semantics, thus a QP module is strongly related to a client type.
- Client identification information has to reach the QP modules, as a separate namespace shall be used for different clients (e.g. a distributed applications exploiting the SRDS to store data may want to set up a private key space).
- A QP module collaborates with the IML in managing the abstraction of namespace (a very simple implementation being by unique key concatenation), by always attaching namespace information to each IML operation. QP actively ensures that keys of different clients are kept separately.



(a) Single DHT Ring (b) Multiple DHT Rings

Figure 3.3: Different implementations of the DHT layer, exploiting either a single DHT ring to hold the information of multiple namespaces (key space partitioning) or a distinct DHT ring for each namespace.

3.2.5 Information Providers

Information Providers exploit simpler interfaces and algorithms to provide information to the IML. They are used when there is no need for security support and result management.

- In simple cases, local information has to be fed to the DHT (e.g. the local daemon providing dynamic attributes used by WP3.3)
- Whenever information is provided by interfaces to other systems and services and a proxy is needed, a dedicate Information Provider is a simpler solution than a MAPI.

3.2.6 Information Management Layer

The interface to the Information primitives is the common ground for all the high-level operations to be implemented in the ADS. It is provided by the DHT abstraction of put/get/remove operations over (key,value) pairs. We explicitly include a few additional parameters that simplify the translation of ADS operation, and are

either already supported by existing DHT implementations (e.g. Bamboo) or are necessary in order to ensure proper flexibility to the ADS architecture.

- *get*, *put* and *remove* operations are supported.
- Key and value parameters are obviously present.
- A Namespace parameter provides a concept of separate key spaces (see 3.2.4, and the following discussion).
- Additional features of a key are expiration times, and secrets needed to operate on the key. For instance, Bamboo provides such a type of elementary authorization mechanism.

At the IML level a namespace identifier holds all information that might have been conveyed by call parameters at the more abstract levels for the sake of disambiguating the particular meaning the key has, and the context in which it is used, to avoid clashes with similar keys (i.e. a namespace makes explicit information that was implicit in the SRDS primitive invoked, in the client type or identity).

The name space is provided by the calling QP module, and can be used to implement separate key spaces in two different ways, that are shown in Figure 3.3. We assume that keys are stored in a P2P ring of nodes, where a set of processes maintains a DHT ring.

A name space, being a unique id, can be concatenated with the key value to generate a unique key over the whole DHT ring. As shown in Figure 3.3(a), the keys from different namespaces (e.g. different clients of the ADS) will distribute over a single DHT ring according to the hash function of the DHT.

A second solution, which brings a higher overhead but may result more scalable over large Grids, is to have several separate DHT rings, replicating the instances of DHT implementation module 3.2.7. This approach naturally provides separate key spaces. The overhead of setting up, possibly on the fly, several P2P networks, can be counterbalanced by the higher efficiency due to the smaller size of each P2P ring (not all nodes need to belong to all the DHT rings) and by the opportunity to tune the implementation of each DHT according to the semantics of the data that it has to store.

3.2.7 DHT Implementation Layer

The DHT implementation provides the actual storage functionalities of the IML. As we saw in the previous section, the IML layer is in charge of selecting a namespace for each operation, and this can lead to two different implementations.

Actually, the two implementations are not incompatible, and can be merged to achieve the best tradeoff. However, it is too early at this development stage to discuss such an issue, and we will assume that only one of the two solutions in Figure 3.3 is selected.

Besides providing the additional functionalities defined by the IML layer (e.g. the expiration of requests), the DHT implementation has to allow a certain degree of customization, concerning for instance the hash function used, or the replication degree. This is needed in order to tune the DHT for different usage pattern, and to extend the ADS to support complex queries and dynamic attributes (Section 3.6). The two main ways to support separate namespaces that we describe have an impact on the amount of possible tuning of the DHT layer, and on how to perform it.

- In the implementation of Figure 3.3(a), a single DHT is used to implement the IML within the ADS. As a consequence, all XtreemOS nodes have to be part of the P2P DHT ring as long as they stay in the Grid.[2] Note that providing a separate instance of the ADS for each application is still possible, as more instances of the ADS can share the local DHT ring instance.

  In this case, namespace information is used to dynamically select hashing and replication characteristics for the given set of keys, i.e. we can map namespaces to additional parameters of the DHT implementation that optimize its behaviour. One simple example is turning the hash function to a linear affine function, or to a space filling curve mapping, in order to map locality for a given set of keys to locality on the DHT overlay.

  As the DHT ring is unique, it will not be possible to tune all of its parameters according to namespaces, e.g. P2P ring repair strategies will have to be common.

- In the implementation shown in Figure 3.3(b), it should be possible to customize all parameters and policies of a DHT ring for its specific use at ring set-up time, that is when that specific overlay network is created. Rings dedicated to key spaces essential to XtreemOS will have to always remain active, while smaller rings supporting application specific tasks, or temporary VOs, will have a shorter lifespan (that of an application or a VO) and likely a lower initialization cost too.

Namespace Translation and the DHT interface Starting from the operations received by the QP modules, the IML layer has to map them to the DHT implementation layer. As we discussed in sections 3.2.6 and 3.2.7, this requires mapping an abstract concept of namespace (a separate space of keys) into one of the implementations shown in figure 3.3.

The abstract format of operations

operation_{QP} = { **op, key**$_M$**, value**$_M$**, Nspace, ClientType, ClientId** }

[2]This does not take into account the issue of XtreemOS for mobile devices. As we may not want these devices to be involved in DHT operation, e.g. for reasons of better reliability and power consumption, those devices may access instances of the SRDS which are hosted by non-mobile XtreemOS platforms.

will be mapped to a concrete operation on the DHT implementation

operation$_{DHT}$ = { **op, key**$_D$**, value**$_D$**, auxInfo** }

The fields of the DHT operation are functions of the fields in the high-level operation.

key$_D$ = g(**key**$_M$**, Nspace, ClientType, ClientId**)
value$_D$ = **value**$_M$
auxInfo = h(**key, Nspace, ClientType, ClientId**)

We do not assume any value change at the DHT implementation level: values are provided by the query translation algorithms embedded in the QP module, and go uninterpreted in this step. We do assume, however, that key translations may happen as a result of namespace implementation.

In particular, the IML and the DHT behaviour we discussed in previous sections will map into two translation functions $g()$ and $h()$. In a generic implementation we have that the $g()$ function can be defined as a key concatenation.

key$_D$ = $(K_1 \bullet K_2 \bullet K_3)$
K_1 = **Nspace**
K_2 = **key**$_M$
K_3 = **marshall(ClientType, ClientId)**

That is, we generate unique keys in the DHT ring by concatenation of the namespace unique id, the actual key, and a unique ID function of the client, to ensure client separation when needed. Note that, when client separation is not needed, K_3 may be unused. The length of the three key portions may vary from case to case, provided that the beginning of the namespace part conveys enough information to prevent key clashes.

The $h()$ function provides hints to the DHT implementation layer, by inferring information, from the namespace and other parameters, about DHT ideal tuning (hash functions, data expirations, and so on).

We note that depending on the implementation chosen (see Figure 3.3(b)) the resulting key K_D may be split again in two parts, a ring selector identifier and a DHT key. The ring selector can be seen, in a special case, as the concatenation $K_1 \bullet K_3$. Besides, in the multi-DHT implementation overlay rings may be dynamically created, hence an analogous of the $h()$ function has to be defined to provide tuning information at overlay initialization time.

3.3 ADS M18 Prototype

In this section we describe the ADS prototype due to M18. It is based on the general architecture depicted in figure 3.1, thus the implemented features are a subset of our ADS general ones, and we take into account mainly the AEM (WP3.3) requirements, see the figure 3.4.

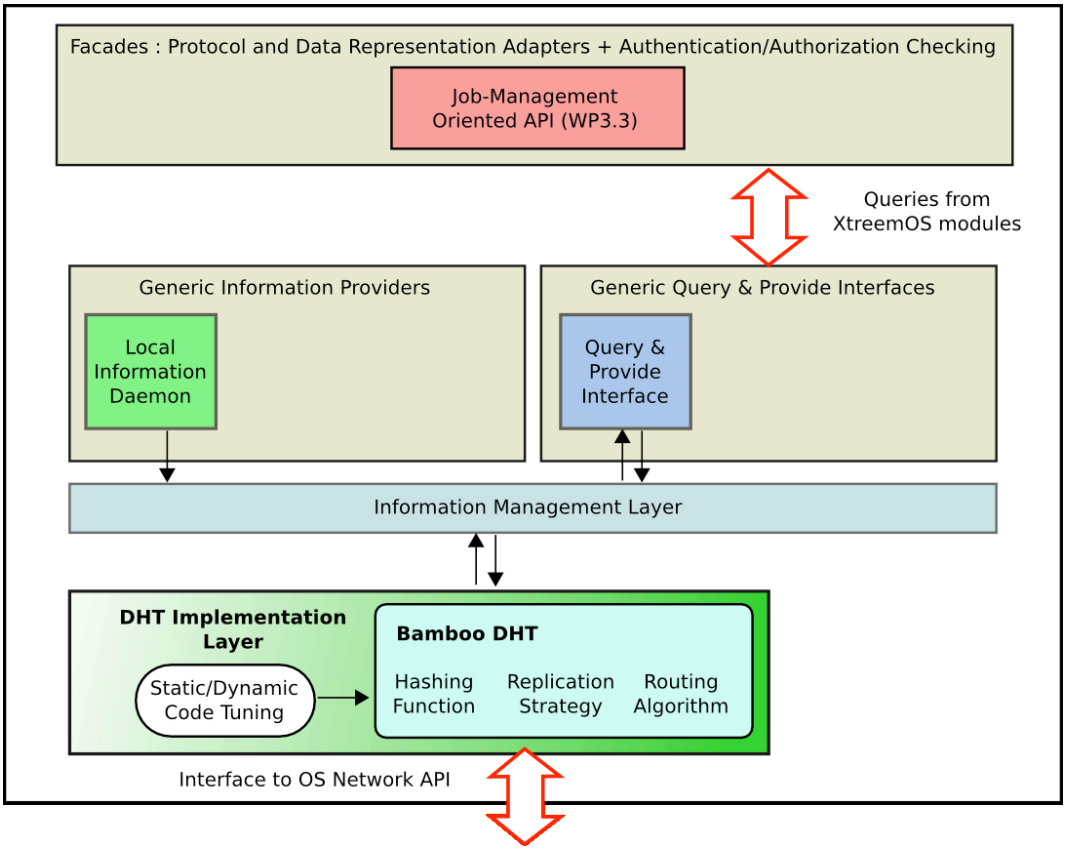


Figure 3.4: First prototype of ADS (M18 prototype)

Static attributes	Dynamic Attributes
Number of CPUs, Clock Frequency	CPU Usage, percentage for each CPU
Total Ram Size	Ram Space Usage
Total Swap Size	Swap Space Usage
Total Hard Disk Size	Hard Disk Space Usage
OS version	

Table 3.1: List of static and dynamic attributes published in the underlying DHT

Inside the Directory Service prototype we can find three kinds of modules:

- **Local Daemon Information:** retrieves the host state information from the **/proc** file system (table 3.1) and translate the gathered information in a XML file (figure 3.5). Then, its content is published onto the underlying DHT.
- **Query & Provide Interface (WP3.3):** receives simple or complex queries from the AEM. The main task of this module is to translate them in order to be resolved using the DHT capabilities. Furthermore, having received some data from the underlying network, WP3.3 Query & Provide Interface has to translate them in the data format expected by the AEM module. In the current implementation, the algorithm to resolve range queries has not been implemented yet.
- **DHT Implementation Layer:** is the module responsible to handle and maintain the overlay network and it provides some capabilities exploited by the higher layers to implement publish/retrieve operations. We have used as overlay network the Bamboo DHT [1].

The first two modules can interact with the underlying DHT thanks to an interface (the IML, a DHT Interface – figure 3.4). We use this kind of architecture already in the M18 prototype to guarantee that the ADS implementation is independent by any DHT solution. In other words, we keep open the option of choosing any one of the several libraries implementing DHTs (e.g. [23, 24, 28, 29]) with no impact on the ADS design.

3.3.1 Information Management Layer

The **IML** is composed by a list of methods and, in the following, we describe the meaning and the signature of each of them. It is important to underline that we use additional parameters – such as **secret** word – that are dependent by the Bamboo DHT implementation. The list of methods require the following parameters:

1. the **key** is the key of the hash table. Its hash value suggests the point in the DHT keyspace;
2. The **value** is the shared object and is the XML file content of ADS instance;

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <XtreemOS>
    <Node>
      <TimeStamp>Wed Oct 03 00:36:19 CEST 2007</TimeStamp>
      <IP>paravent-19.rennes.grid5000.fr</IP>
      <Port>3630</Port>
      <Static>
        <Hw>
          <CPU>
            <CPUsNumber>2</CPUsNumber>
            <CPUFreq_0 Peck="MHz">2009</CPUFreq_0>
            <CPUFreq_1 Peck="MHz">2009</CPUFreq_1>
          </CPU>
          <RamTotal Peck="MB">2010</RamTotal>
          <SwapTotal Peck="MB">3812</SwapTotal>
          <HardDiskTotal Peck="MB">5636</HardDiskTotal>
        </Hw>
        <Sw>
          <OSVersion>Linux 2.6.19.1.070611</OSVersion>
        </Sw>
      </Static>
      <Dynamic>
        <CPUsage_0 Peck="%">2</CPUsage_0>
        <CPUsage_1 Peck="%">1</CPUsage_1>
        <RamUsage Peck="MB">324</RamUsage>
        <SwapUsage Peck="MB">0</SwapUsage>
        <HardDiskUsage Peck="MB">2457</HardDiskUsage>
      </Dynamic>
    </Node>
    <Notification/>
  </XtreemOS>
```

Figure 3.5: XML file content

3. the **ttl** represents the Time-To-Live (TTL) and it is the number of seconds that the shared object is still valid.

The three parameters described above are the essential ones, that we assume any DHT solution will support.

Two features provided by Bamboo DHT are transparent to the higher modules and increase the reliability and reliability of a distributed directory service: automatic replication and expiration of out-to-date information.

- As other DHT implementations, Bamboo handles replication of the tuples stored on each node. If the replication degree is configured to **k**, in the overlay network we will have **k** copies of each tuple.

- Removal of out-of-date information is performed by associating a **TTL** parameter to each key-value pair every time a new object is put on the network. When the TTL expires, the shared information is automatically removed from the Bamboo network.

Bamboo allows the developer to further customize its behavior, in order to run different kind of tests. In particular, Bamboo allows to change the timeout used to detect that a DHT operation has failed (get timeout), and the percentage of dropped messages (to simulate lossy network connections). These two features are important to evaluate the behavior of Bamboo network, making it possible to study an overlay network with a tune-able degree of unreliability, and according to that, experiment with different values of the get timeout.

As an example, if we have a Bamboo network that loses the 70% of messages we can choose a low value of the get timeout, to reduce the waiting time before re-sending the same request. These experiments are important because an excessively low value for the get timeout can lead to an overload of the Bamboo network due to repeated messages.

In the following, we describe in details the signature of the methods in the IML.

Put key-value. Method that publishes an object (**value**) in one specific point of the network (point described by the **key**).

```
public int PUT(
InetAddress serverHost,
int serverPort,
String key,
byte[] value,
int ttl,
String secret);
```

The first two parameters are the **IP address** and **port number** of the DHT layer instance (i.e. ip and port of Bamboo node).

The **secret** is a secret word associated with key-value pair and it is useful as security aspect. In fact, the removal is permitted if and only if the operation is invoked passing the key-value pair and the corresponding secret parameter. A successful put will return zero.

Get key. Method that retrieves the **values' list** corresponding on the **key**.

```
public byte[] GET(
InetAddress serverHost,
int serverPort,
String key,
String maxVals);
```

The first two parameters are the **IP address** and **port number** of the DHT Layer instance.

The **maxVals** field specifies how many values should be returned by a single get. By default it is set to $2^{31} - 1$.

If the returned value is a valid array reference, the get request was correctly resolved.

Remove key-value. Removes the **key-value pair** from the network.

```
public int REMOVE(
InetAddress serverHost,
int serverPort,
String key,
byte[] value,
int ttl,
String secret);
```

The first two parameters are the **IP address** and **port number** of the DHT Layer instance.

The **secret** is a secret word associated with key-value pair. If the secret word is different from the put one, the DHT will not remove the pair. A successful remove will return zero.

The features described above, which are directly invoked by the modules on top of the DHT interface, are supported by the Bamboo API, that has been used in their implementation. We have chosen to have two separated layers (the IML and the DHT implementation) to achieve a good code engineering. In this way, we can replace Bamboo with another DHT at the cost of modifying the IML layer.

3.3.2 Local Daemon Information

This module is responsible to publish and update the host state information on the underlying DHT, using **Put key-value** and **Remove** methods.

Publication. Local Daemon Information retrieves the host state information of the current machine thanks to the data stored in the **/proc** file system. The list of static and dynamic attributes are in the table 3.1. Starts from this set of data, a parser generates a XML file (see the figure 3.5) and its content is the **value** parameter of **Put key-value** method. In this way, each node can share its computational resource in the ADS overlay network.

It is important to underline that Local Daemon Information **doesn't know** the overlay network location of the XML content file due to DHT definition.

Update. The Local Daemon Information periodically has to update the XML content file because it contains also a list of dynamic attributes that continuously change their value. This module updates its host state information every 30 sec. To do so, the Local Daemon Information periodically retrieves from the **/proc** file system the current computational resource information, the parser generates the corresponding XML file and its content is put inside the DHT. After that, the Local Daemon Information removes the previous host state using the **Remove** method. This solution relies on the Bamboo behavior of storing a list of values for each key. Moreover, the removal is an optimization because the XML file contains a

timestamp of last modification. In general, given a key, we always obtain the last host state by looking at timestamps. We prefer to exploit the remove feature to reduce the space and communication overhead.

3.3.3 WP3.3 Query & Provide Interface

The WP3.3 Query & Provide Interface performs queries that are resolved by the underlying network. This module receives queries from the AEM and it is responsible to translate the requests in simpler ones that can be resolved using the **Get key** method. On the other hand, the WP3.3 Query & Provide Interface obtains XML file content – for each get request submitted – and it has to retrieve the information to set the corresponding data structure expected by the AEM.

3.4 Experiments

In this section we describe the experiments performed to evaluate the scalability and reliability of the ADS prototype.

We have used two very different testbeds to probe the application capabilities first "in-the-small" and then "in-the-large".

Pianosa Pianosa is a homogeneous RLX blade cluster with RH Linux 7.3, composed by 32 nodes with Pentium III 800MHz and 1Gb Ram. Every node has got a triple 100 MB ethernet. It was used for functionality tests and in-the-small tests.

Grid'5000 Grid'5000 [14] is a reconfigurable, controllable, monitorable Grid composed by heterogeneous clusters. 2/3 of the Grid'5000 nodes are dual CPU 1U racks equipped with 64 bits x86 processors (AMD Opteron or Intel EM64T), the ones that we selected for our experiments. Grid'5000 is geographically distributed, being composed by 9 sites, each one hosting one or more clusters. Each cluster is accessible through a "head" node and using a standard set of management tools for reservation, deployment, and monitoring. All the nodes of each cluster are interconnected through at least Gigabit Ethernet. More performing network equipments can be present on some clusters, (e.g. Myrinet 10G on 33 nodes, and Infiniband on 66 nodes in the Rennes site[10]).

Grid'5000 is a real testbed for an XtreemOS prototype, because of its geographic size and the heterogeneity of the network. It is not a dedicated platform, our experiments run concurrently with other applications (on different nodes of the same clusters).

We have used up to 484 of the 1249 nodes of Grid'5000, arranged as 5 subsets of the 9 sites.

- Rennes;

- Lille, Sophia-Antipolis;
- Lille, Rennes, Sophia-Antipolis;
- Bordeaux, Lille, Rennes, Sophia-Antipolis;
- Orsay, Rennes, Sophia-Antipolis.

First, we have studied and observed the behavior of the ADS prototype in one homogeneous cluster where the hosts are connected by LAN Ethernet (Pianosa). Then we have repeated the tests on Grid'5000 to check the ADS prototype behavior on larger and larger networks.

Test Organization. On each node involved in the testing phase, we run an ADS instance (see figure 3.4). In particular, on each computing resource there are a **Bamboo DHT** instance as DHT Layer module, and a **Local Information Daemon** that publishes the host state information every 30 sec. The WP3.3 Query & Provide module is not always running on each node in all tests.

- In the first scalability test we measured the response time of **Get** request done by a single node of the ADS network (the **Requester**).
- In the second scalability test we measure the response time of **Get** requests done by an increasing number of (**Requester**) nodes.
- In the third test we evaluate the system reliability. Again we run the WP3.3 Query & Provide only on one node (**ReliabilityRequester**) while interfering with the DHT network by killing random nodes.

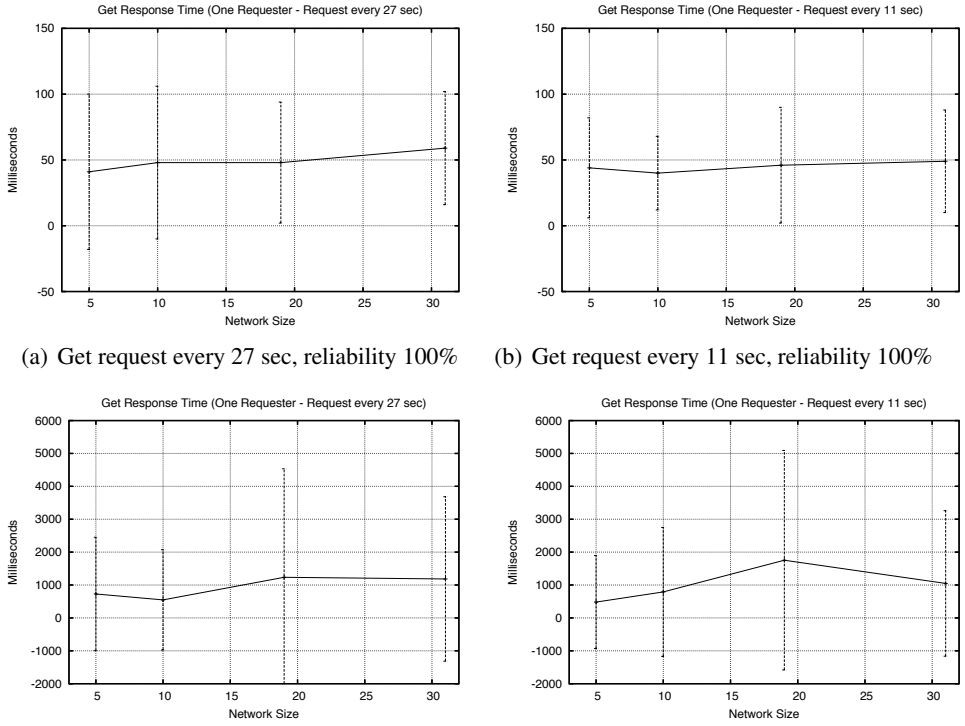
In all tests performed, the Bamboo replication degree was set to 4 (**k = 4**). In both Scalability Tests we set the **Get timeout** to the default 5000 ms, and for each measurement we consider first a perfectly reliable overlay network, and then a Bamboo network that loses 10% of the messages (this is done through the "drop messages" Bamboo configuration). The reliability test is performed only with the second, lossy network configuration.

3.4.1 First Scalability Test

The Requester invokes the Get method periodically, with the key being chosen randomly from the whole set of nodes in the test. This experiment has been repeated twice, respectively with the Requester interval set to 27 sec and 11 sec. We have chosen these values because are less than 30 sec, and reasonable with respect to the overlay network size.

As we said before, we measure the Get response time compared to the overlay network growth:

- Pianosa network size: 5, 10, 19, 32 nodes;



(a) Get request every 27 sec, reliability 100%

(b) Get request every 11 sec, reliability 100%

(c) Get request every 27 sec, reliability 90%

(d) Get request every 11 sec, reliability 90%

Figure 3.6: First Scalability Test - Pianosa Cluster, Requester invokes 20 Get requests with assigned time interval - 100% and 90% overlay network reliability.

- Grid'5000 network size: 32, 100, 308, 484 nodes.

The graphic trend represents the Get response time average while the horizontal line is the standard deviation, that is:

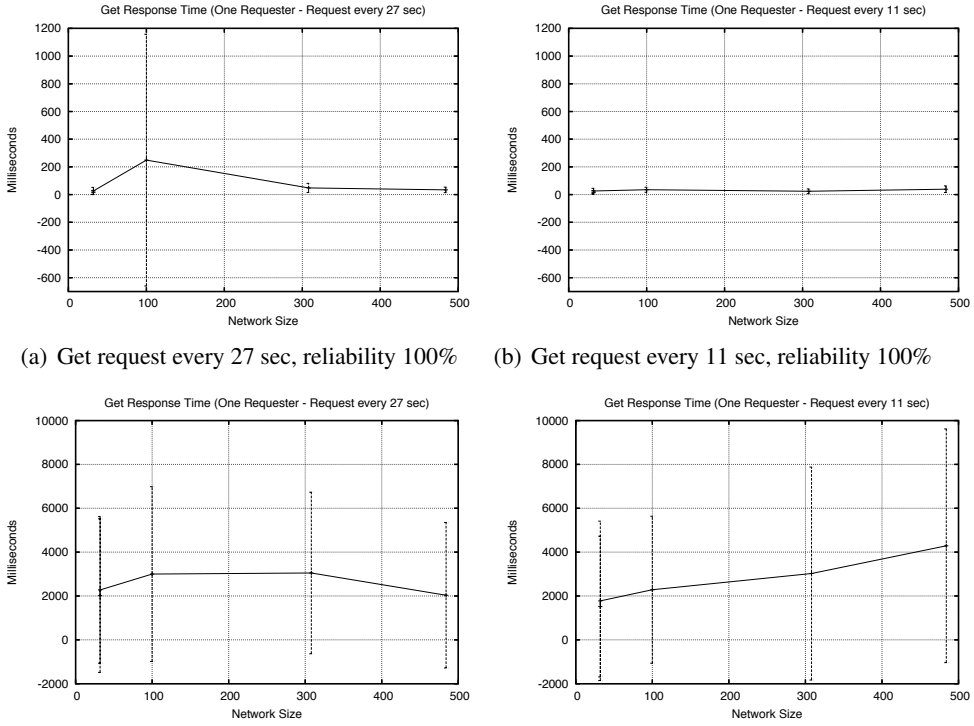
$$\sigma_{1,2} = \pm\sqrt{\frac{\sum_{i=1}^{n}(x_i-\mu)^2}{n}}$$

where n is the network size and μ is the average value of the x_i.

The figures (3.6) show the measure of the Get response time using the Pianosa cluster.

In the graphics (3.6(a), 3.6(b)), we can see that Bamboo has good performance, as the average response time is around 50 ms, with a worst case we of 100 ms, when no messages are lost.

In figures (3.6(c), 3.6(d)), the average value is higher and a high standard deviation shows up, is due to the reduced reliability of the network. This is simulated by configuring Bamboo to accept only 90% of the messages on the overlay network (90% reliability). The high get timeout (5000 ms) thus impacts on the performance of the ADS.



(a) Get request every 27 sec, reliability 100%

(b) Get request every 11 sec, reliability 100%

(c) Get request every 27 sec, reliability 90%

(d) Get request every 11 sec, reliability 90%

Figure 3.7: First Scalability Test - Grid5000 Cluster, Requester invokes 20 Get requests with assigned time interval - 100% and 90% overlay network reliability.

The graphics (3.7) show the measurements of the Get response time using Grid'5000 under the same test conditions, for much larger network sizes.

At a glance, we can observe that the ADS prototype is scalable in respect with the growth of the network size, both in the case that we have an unreliable overlay network and reliable one. Clearly, the average performance is nevertheless heavily influenced by the reliability of the network

In the next three graphics (3.8, 3.9(a), 3.9(b)), it is shown on a logarithmic scale the distribution of the response times during some typical test runs with an unreliable overlay network. As we can see, on each experiment the number of expiration timeout increases with the amount of messages exchanged, roughly logarithmically with the network size. Multiple timeouts occur on the same get: in figure 3.9(a), we can see that in two cases the results is 15 sec (iterations 14 and 19) and in one case we obtain 10 sec (iteration 9), while in figure 3.9(b), we obtain even worse results.

The severe impact of an unreliable network on the DHT performance is clearly due to the high value of the timeout on remote interaction among the overlay nodes. This is the result of Bamboo being optimized for loosely coupled, dispersed and

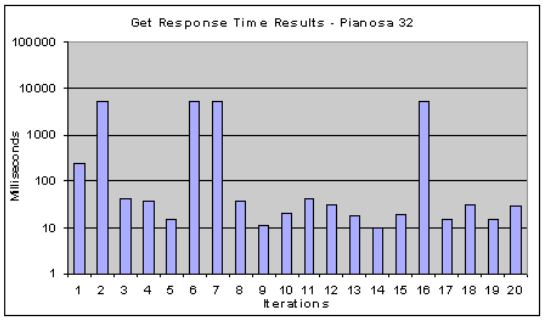
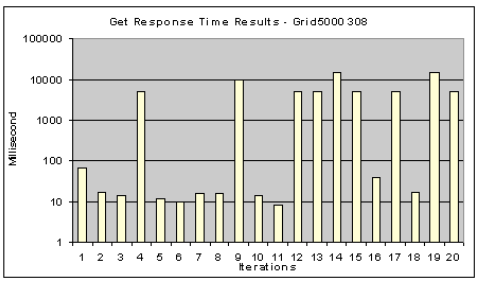
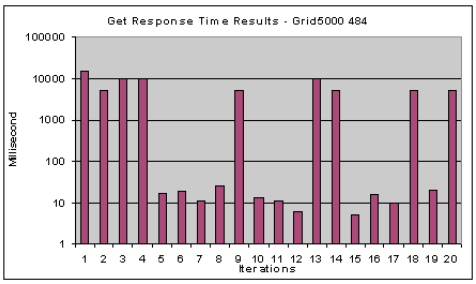


Figure 3.8: First Scalability Test - Get response time results distribution - Pianosa 32 nodes - 90% overlay network reliability.

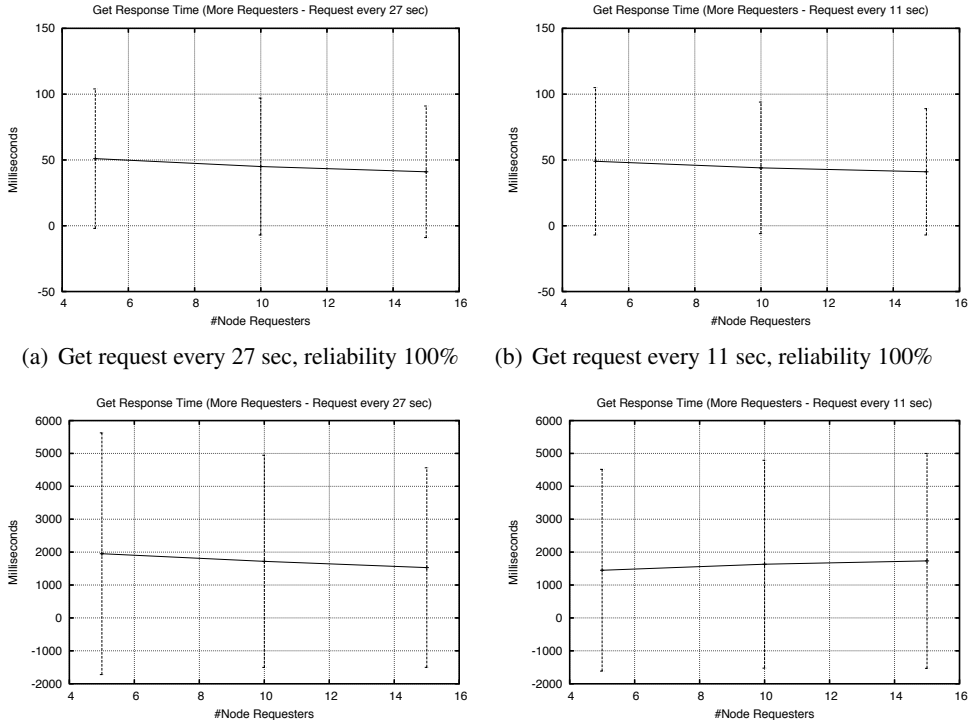


(a) 308 nodes



(b) 484 nodes

Figure 3.9: First Scalability Test - Get response time results distribution on Grid'5000 - 90% overlay network reliability.



(a) Get request every 27 sec, reliability 100%

(b) Get request every 11 sec, reliability 100%

(c) Get request every 27 sec, reliability 90%

(d) Get request every 11 sec, reliability 90%

Figure 3.10: Second Scalability Test - Pianosa Cluster, more Requesters invoke 20 Get requests with assigned time interval - 100% and 90% overlay network reliability.

best-effort networks. This already points out to the need for optimization of the timeout parameters when using modern, fast networks.

3.4.2 Second Scalability Test

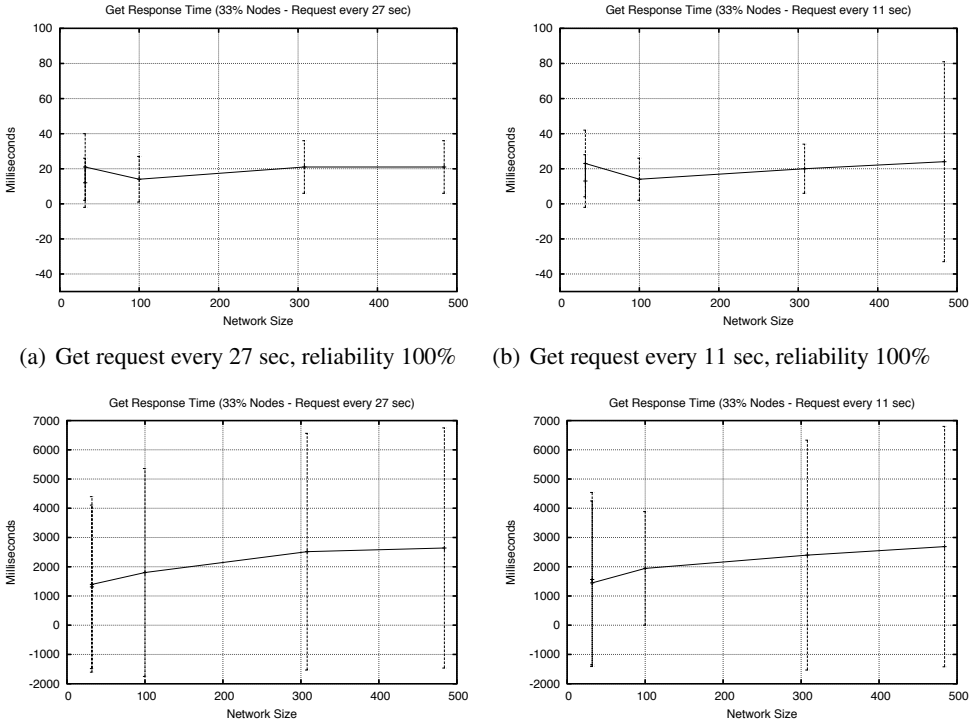
In the second scalability test, on each involved node it is running one ADS instance like in the previous experiment. Now we increase the number of Requester nodes.

- Pianosa Requesters: 5, 10, 15 nodes (network size fixed to 32 peers);
- Grid'5000 Requesters: 33% nodes belonging to the network size (32, 100, 308 and 484).

Figures (3.10) show the Get response times with the Pianosa cluster.

In both cases (3.10(a), 3.10(b)), the average time is 50 milliseconds on a reliable overlay network, while figures (3.10(c), 3.10(d)) show the measurements when 10% of the messages are lost (90% network reliability).

Figure (3.11) shows the times on Grid'5000.



(a) Get request every 27 sec, reliability 100%

(b) Get request every 11 sec, reliability 100%

(c) Get request every 27 sec, reliability 90%

(d) Get request every 11 sec, reliability 90%

Figure 3.11: Second Scalability Test - Grid'5000 Cluster, more Requesters invoke 20 Get requests with assigned time interval - 100% and 90% overlay network reliability.

Even in this second scalability test the ADS prototype is scalable with the network size, in spite of a a large fraction of the nodes actually performing queries.

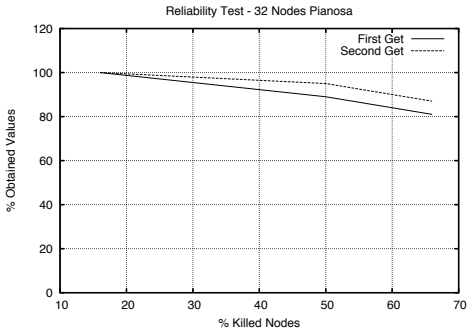


Figure 3.12: Reliability Test - Pianosa cluster - The x axis represents the number of killed nodes - 90% overlay network reliability.

3.4.3 Reliability Test

The reliability test is performed differently from the previous ones.In all cases the overlay network reliability is 90%, and we simulate unreliability (churn) of the nodes themselves.

The ReliabilityRequester asks to the ADS the host state of *every node* involved in the experiment. That is, the ReliabilityRequester performs one Get request for each key. This is run once to verify the initial correctness, then, a fraction of the nodes chosen randomly nodes are killed. The ReliabilityRequester then invokes the Get function twice on **each** key, once immediately, to verify the number of lost keys after a fault. It then waits for an update by the ADS instances that are still alive (it has to wait at least 30 s), and it repeats all the Get calls. We varied the amount of killed nodes with respect to the network size.

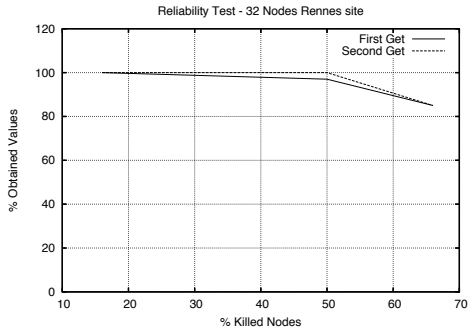
- Pianosa killed nodes: 5, 13, 20 nodes (network size fixed to 32 peers);
- Grid'5000 killed nodes: 16%, 50% and 66% of nodes belonging to the network size (32, 100, 308 and 484).

Since Bamboo DHT performs automatic replication of degree $k = 4$, a key is lost if and only if all nodes that store the copy are killed, and the host providing the lost key is killed too. Thus the second round of get requests, actually retrieves more keys, since the sources of some keys are still alive.

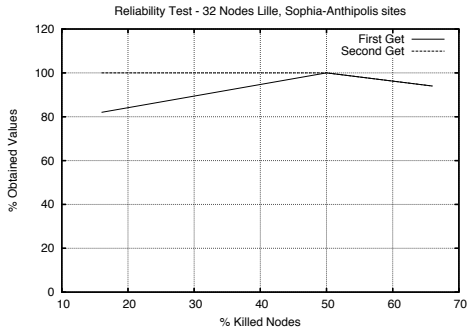
In the following figure (3.12), we have the reliability test on Pianosa cluster. The figures (3.13(a), 3.13(b), 3.13(c), 3.13(d), 3.13(e)) show the reliability test on Grid'5000.

The reliability test too results in a similar behavior on Grid'5000 and the Pianosa cluster, and it shows that the ADS provides a high level of reliability, thanks to the automatic replication and self-healing properties of the Bamboo DHT.

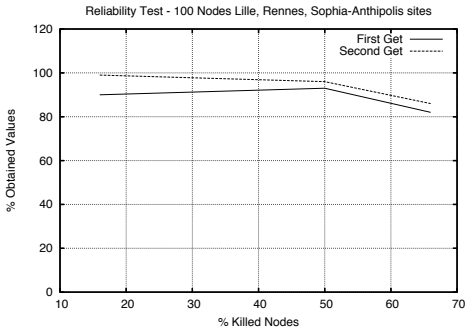
It is important to underline that in all graphics about reliability test, if we kill the 66% of the nodes of the overlay network, the number of the retrieved objects



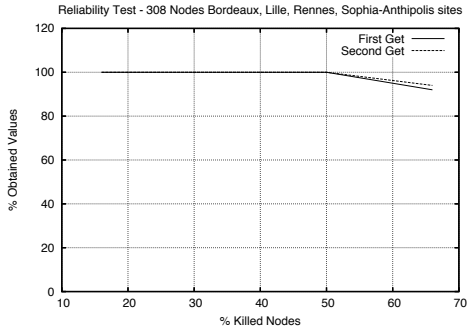
(a) 32 Nodes - Rennes site



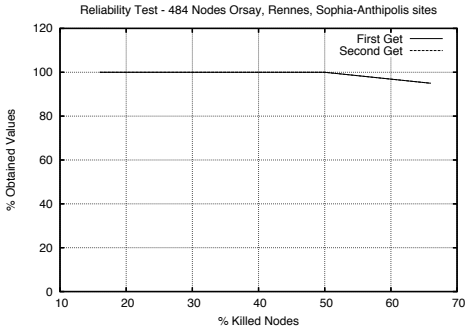
(b) 32 Nodes - Lille, Sophia-Anthipolis sites



(c) 100 Nodes - Lille, Rennes, Sophia-Anthipolis sites



(d) 308 Nodes - Bordeaux, Lille, Rennes, Sophia-Anthipolis sites



(e) 100 Nodes - Orsay, Rennes, Sophia-Anthipolis sites

Figure 3.13: Reliability Test - Grid'5000 - The x axis represents the percentage of killed nodes - 90% overlay network reliability.

O is always greater than the 80% of the total number of the shared object T. Note that, when the TTL expires we obtain that $O = T$, but the self healing property is important for those kinds of information that do *not* expire. As long as a key is not lost, it can be quickly replicated again in order to preserve the reliability of the network.

3.5 Lessons Learned

In the previous section we have evaluated the ADS based on Bamboo DHT. The obtained results show that the ADS is scalable in respect with, both the growth of network size and the increase of Requester nodes.

The experiments demonstrate that Bamboo DHT is reliable because it handles effectively the lost messages – sending the same request after timeout expiration – and it is resistant to nodes failures. In fact, if we kill the 66% of the nodes in the overlay network, we retrieve more than 80% of the total number of keys.

Another advantage of exploiting Bamboo DHT, is the ease of configuration and of setting different test conditions during the evaluation. For example, we can choose to change the Get timeout according to the reliability and latency of the overlay network.

The tests described above can be ran on a simulator or on a real clusters. Testing with a large real testbed as Grid'5000 has proved precious to us not only in verifying our system in the large, but also in detecting real-life problems and issues. An example of these issues is related to prototype portability. The ADS and Bamboo DHT are written using Java [17], hence they are platform independent. However, Bamboo DHT needs a native library (the Berkeley Database library libDB) that is of course dependent on the underlying CPU architecture (e.g. 32/64bit CPUs).

While the issue is easily solvable by recompiling the open-source code of the libDB, this is an example of the module dependencies that XtreemOS development WPs need to track down and record, as they are candidate to become prerequisites/packages of the XtreemOS distribution.

The current ADS prototype *doesn't resolve the range query* and this problem needs more investigation because using pure DHT approaches may not scale at all. Solutions that resolve multi-attribute range queries implementing this feature as a primitive function of the DHT have several problems related to the kind of query translation they use.

Among future developments, we will run the ADS on a simulator as PeerSim [21] to study the system behavior over larger grids (tens of thousands of nodes and above).

3.6 Future Development and Research

This section describes the future progress we are planning for the definition of the ADS. We recall that *ADS* should offer at least the following functionalities.

- indexing of distributed resources
- supports for *simple and complex queries*
- handling of *static*, but in particular, *dynamic data*.

The definition of *indexing structures* to support *complex queries*, is currently investigated in an active research area[30]. We plan to evaluate different approaches in order to test their effectiveness for the definition of the *ADS*. Since distributed resources are generally defined by a set of k attributes, $k > 2$, we are mainly interested in solutions supporting k-dimensional queries. As far as concerns the constraints defined on the attribute values, besides *exact match query*, more complex constraints should be investigated as well. As a matter of fact, a directory service should support *partial match queries* including wild cards or subset of the attributes describing the resources, for instance (CPU-Speed=3Ghz *and* RAM=256MB *and* Arch=*), or *range queries* defining a range of value for at least one of the resource attributes, for instance (1Ghz<CPU-Speed<3Ghz *and* 512MB<RAM<1Gb).

Similarity (nearest neighbor) queries should be supported as well. The definition of a metric for the attribute space is required by this kind of queries. A similarity query is initiated by submitting an exact match query which defines a point P in the k-dimensional space, afterwords the k resources nearest to P, according to the defined metric, are returned.

Finally, different types of attributes, *static*, *dynamic* and *semi-dynamic* should be considered. A challenging issue is the definition of a set of techniques to reduce the refreshes of the resources status required by dynamic attributes.

We plan to exploit *DHTs* as the basic building block of the *ADS*. The adoption of this approach to support *complex queries* on *dynamic attributes* requires the solution of a set of problems. First of all, it is worth noticing that *range* and *similarity* queries are characterized by a strong degree of *locality*, i.e. data searched are close in the attribute space. One the other hand, one of the main goal of the *DHT* approach is a *uniform distribution* of the data onto the nodes of the overlay. This means that data close in the attribute space may be allocated to far away nodes. The definition of an *indexing layer* above the *DHT* is therefore mandatory. The main goal of this layer should be to define a support to recover this loss of locality.

Several proposals have been recently presented. The first set include approaches based on the definition of a *locality preserving function*. *MAAN* [3], for instance, defines a locality preserving function for 1-dimensional-range queries and it implements k-dimensional queries by intersecting the results returned by k 1-dimensional queries. A more complex approach [26] is based on the definition of a *locality preserving mapping* from a k-dimensional space to a 1-dimensional space based on the use of *space filling curves*. In this case the k-dimensional space is linearized by mapping each point/data in the k-dimensional space to a point of a straight line. The index I of each point (data) corresponds to its position on the straight line. I is exploited to map each data onto the *DHT* nodes. It is worth noticing that this mapping may guarantee that points which are close on the straight line are also close in

the k-dimensional space, while points close in the k-dimensional space may be far away on the straight line. Due to this issue, a range query generally requires data stored on a set of overlay nodes.

As alternative approach requires to partition the k-dimensional attribute space into a set of *zones* and to map zones to peers [13, 22]. The resulting partition is generally described by a *hierarchical structure* which has to be distributed among the peer. The definition of highly distributed strategies for the management of this data structure is the main challenge of this approach.

As far as concerns dynamic attributes, we plan to investigate if *DHT* are suitable for their support. An alternative solution exploits the *DHT* only for the static attributes, by applying the hash function to the static attributes referred in the query. This allows a fast detection of the group G of nodes characterized by the same value of the static attributes afterwords each node in G can be queried to detect the current state of its dynamic attribute. The routing layer defined by the *DHT* can be exploited in the second step to define group communication strategies.

Chapter 4

Interaction with other XtreemOS Modules

This chapter describes the main requirements provided by the Application Execution Management (AEM, WP3.3) and the Data Management Services (DMS, WP3.4). Section 4.1 is based on written [7] and spoken interaction with the WP3.3 members, in which we discussed the requirements and protocols for the AEM to interact with the SRDS, and the type of exchanged data.

In the same way, section 4.2 summarizes DMS-related requirements [8] for the Application Directory Service (ADS), the data attributes and the operations to be implemented. This section is also based on a preliminary technical report [33] written by the WP3.4 group.

Within Sections 4.1 and 4.2 we also detail what features are going to be supported by the ADS prototype at M18, and which ones will be implemented later on or are still being investigated.

Section 4.3 summarizes the interaction with security and VO management people [5, 6, 9]. Due to the approach XtreemOS has toward the matter, there have been consequences on the design of the SRDS, but no specific interface has to be defined.

In this chapter we don't analyze application requirements (WP4.2), as we focused first on requirements concerning XtreemOS system integration. However, we took into account in designing the SRDS that applications will use it, and several functionalities (e.g. namespaces, information management) have been expanded in order to support generic use of the SRDS Directory Service functionalities.

4.1 Application Execution Management (WP3.3)

We will support most WP3.3 related requirements [7] already in the M18 prototype. A more refined version of these requirements will take into account detailed comments and second-level contributions also from WP 4.2.

The Service/Resource Discovery System (SRDS) receives a set of resource requirements in the form of a RRL (Resource Requirement Language) and returns an unordered list of set of resources that fulfills these requirements, to start the negotiation process.

The resource requirement language will be an XML schema based on the JSDL [18] resource requirements specification. In the following we summarize the JSDL attributes that the SRDS is going to support, and those whose semantics is yet to be matched with their XtreemOS usage. As JSDL is designed to interact with batch schedulers and reservation systems, which is not the case for XtreemOS, some attributes do not apply.

Requirements to be supported are the following ones:

1. Resource filtering existing in JSDL. Semantics in JSDL must be supported [18]. Some of the attributes are enumerations and other ones are ranges. Multiple attributes within the same query correspond to the logical AND of all of them.

   - ExclusiveExecution
   - CPUArchitecture
   - IndividualCPUSpeed
   - IndividualCPUCount
   - IndividualNetworkBandwidth
   - IndividualPhysicalMemory
   - IndividualVirtualMemory
   - TotalCPUCount
   - TotalPhysicalMemory
   - TotalVirtualMemory
   - TotalResourceCount

2. Additional XtreemOS resource requirements

   User Availability Resources must be accessible by the grid user submitting the job.

   Policy Fulfillment Resources returned must fulfill VO policies. This implies contact with VO.

   Geographic Distribution Resources could be geographically limited.

3. Requirements forced by the scheduler

   - at the end of the resource selection, the AEM obtains a list of resources, and starting from them, it can begin the negotiation. If this operation fails, the scheduler asks for a new set of resources suggesting the ones that don't pass the negotiation. In this way, the Application Directory Service (ADS) will return a new list of resources that match the AEM request without the failed resources.

4.1.1 Requirement implementation

We discuss those of the listed requirements which are not straightforward.

JSDL Resource Requirements Some of the JSDL related requirements need to be analyzed more carefully.

- OperatingSystem maybe should always just match "XtreemOS", and will definitely do in the M18 prototype.
- IndividualCPUTime, TotalCPUTime have unclear semantics at the moment, as an XtreemOS node is not a reservation system. These requirements may be interpreted as expected (maximum) execution times to be fed to VO policies.

User Availability This can be verified in the SRDS architecture at two different stages. (1) the request is accepted by a facade module, which can check that the user is authorized within the VO by contacting the VO support on the node receiving the request. (2) the user and VO information is exploited within the query in order to rule out resources which will not allow executing the job.

Policy Fulfillment User authorization is just a special case of applying a VO policy. Whether the job execution on the returned resources will fulfill all relevant VO policies is a complex issue, depending on the exact policy language and evaluation scheme that will be implemented within XtreemOS. If the VO policy evaluation exploits information from the local environment[1] then it is not possible in the general case to anticipate the result of the policy evaluation that will happen during the negotiation phase. On the other hand, any evaluation of VO policies that are homogeneous across the VO can be performed at the SRDS site, exploiting VO support functionalities provided by XtreemOS.

Geographic Location We should allow resource localization in the form of a geographic tag (e.g. country) associated to each resource. This tag is static and does not take into account mobility issues, mobile machines addressed through IPV6 routing will report "unknown" location. The approach will not try to approximate real distance with calculations, as in the general case they would have no relationship anyway with the network topology and features.

4.2 Data Management Services (WP3.4)

DMS are distributed services, and rely on the SRDS as a central instance for registering and querying information about file system services and volumes. More

[1] A simple but realistic instance is *"give access to local resources to users from group G, only if current load is less than* 0.3*"*, which is node-specific, user-specific and relies on dynamic, local information.

details are reported in [33], in the following we enumerate the DMS requirements [8] for the SRDS:

- **Persistent & Transient Storage:** the Directory Service has to store static configuration information and has to provide dynamic information about the current state of services
- **Authorization and Security:** there has to be some kind of access control on the Directory Service
- **Atomic Modifications:** it should be possible to make changes to DMS entries in the Directory Service in an atomic fashion. When changing a service or volume entry, consistent results are required despite concurrent access.
- **High Availability:** it is of particular importance that failures of single Directory Service hosts do not have a major impact on the availability of the entire service. Most data stored in the Directory Service should be replicated in order to survive permanent host failures.
- **Simple Setup and Administration:** installation, configuration and maintenance of the Directory Service ought to be as easy as possible.

Protocol The communication protocol used between a DMS client and the Directory Service is based on a subset of HTTP v1.1 [15] and messages exchanged are encoded using JSON [20] and UTF-8.

Interface Description The Directory Service stores information about entities. In connection with DMS, an entity may contain information about a volume or a service. Such information is represented by a set of attribute-value pairs.

Attributes are associated with a name which is used to identify the attribute, and they may be either persistent or transient. Persistent attributes are stored in stable storage, i.e. they will survive a shutdown and restart of the Directory Service and are suitable for configuration information. Transient attributes reside in memory, are lost with a restart and could be subject to garbage collection.

Any entity *must* have the following attributes:

Attribute	Storage	Description
UUID	persistent	A globally unique identifier for the entity.
version	persistent	A string representing the current version of the entity.
lastUpdated	transient	A Unix time stamp representing the server time of the last update.
owner	persistent	A String representing the owner of the entity.

Any entity *may* have the following attributes:

Attribute	Storage	Description
type	persistent	This attribute can be used to define groups of entities.
name	persistent	Unlike the UUID, this attribute can be used to assign a human-readable name.
organization	persistent	The organization to which the entity is assigned.
country	persistent	The country of the organization to which the entity is assigned.

Any entity representing a service *should* have the following attributes:

Attribute	Storage	Description
uri	persistent	The URI of the service.
publicKey	persistent	The public key of the service.

Directory Service Operations

- **registerEntity(uuid, data)**. Registers a new entity or updates an existing one. The **uuid** is the UUID of the entity and the **data** is the set of key-value pairs associated with the entity.
- **getEntities(query, atts)**. Returns information on one or more registered entities. The **query** restricts the set of entities that are part of the result set. While **atts** defines which attributes of each matching entity are included in the result set.
- **deregisterEntity(uuid)**. Cancels the registration of a set of entities. The **uuid** is the UUID of the entity to deregister from the Directory Service.

4.2.1 Realization

In this section we describe the feature implemented by the ADS in respect with the requirements of WP3.4.

- **Persistent & Transient Storage:** the ADS provides a feature to store and retrieve static and dynamic information thanks to the underlying overlay network. In particular, the attribute list described above can be inserted in a XML file as we saw in Chapter 3.
- **Authorization and Security:** this aspect is realized through the support of WP2.1

- **Atomic Modifications:** this feature can be implemented using the DHT primitives but the consistency can be guaranteed if the performed updates are infrequent. It is important to remember that the consistency on a distributed system as a DHT is an open research problem.
- **High Availability:** the experiments described in chapter 3 show that the network used by the ADS is reliable, because it is resistant to node failures (thanks to the key-value pair replication) and to network failure (thanks to its P2P organization).
- **Simple Setup and Administration:** the installation of the Application Directory Service is easy and automatic. Every time a new XtreemOS instance is running, it automatically creates a new instance of SRDS and both of its modules – Application Directory Service and Resource Selection Service – are responsible to connect to the appropriate overlay network.

The **registerEntity(uuid, data)** and **deregisterEntity(uuid)** are operations that can be actually supported by the ADS exploiting the DHT primitives. For example, the former can be implemented using the following algorithm:

> **get(uuid)** to verify if it is a registration or an update **put(uuid, data)** to update the entry of the DHT.

The **getEntities(query, atts)** operation needs a more complex DHT primitive such as **RangeQuery(...)**, that is not implemented in the M18 prototype.

4.3 Virtual Organization Support (WP2.1) and Security (WP3.5)

Interaction with the virtual organization support in XtreemOS does not lead to a detailed protocol specification like in the case of AEM and DMS. Instead of tying the system to a specific API and set of functionalities, we included in the design of the SRDS the principles adopted by the WP 2.1 and 3.5 in designing the VO support at the node level and at the Grid system level, respectively. This is in line with one main goal in XtreemOS, that VO management be as transparent as possible to the application programmer, and that VO attributes are seamlessly mapped to standard UNIX abstractions every time this makes sense, enhancing application portability.

The research groups of WP2.1 and WP3.5, involved in VO Management and Security for XtreemOS, have described in several deliverables [5, 6], internal reports and publications [34, 4], how extensions to the (local) LINUX kernel will allow seamless application of VO policies and authorization checks. From our point of view, this will happen mainly by exploiting the kernel key retention service (KKRS) to convey authorization tokens and certificates.

We designed the SRDS to use this feature, providing a set of interface points (the ADS Facade module) where VO-related information can be provided to the

KKRS, and defining a set of modules (Query-Provide Interfaces, DHT implementation) where additional VO policy checks may be performed.

VO configuration will provide a set of policies defining what operations the modules of the SRDS are allowed to perform on behalf of their clients. This will be a generally valid property for any VO and any XtreemOS service, not only to the SRDS. VO policies will also specify what kind of operations are allowed to the SRDS implementation modules on the local system. Where appropriate, the local VO management system will be responsible of contacting VO management services on remote servers, to retrieve all needed information about VO policies. The occasional need to contact remote VO services does not affect the SRDS implementation.

Most of the hooks toward VO management will be within the SRDS Facade module, providing information to the XtreemOS kernel that is checked every time it is appropriate. In case specific authorization checks are needed within the SRDS, they are implemented in the Facade itself. The resulting design is modular, flexible and readily expandable, and it allows for mandatory checking of access rights before executing any functionalities of the Service/Resource Discovery System, exploiting the XtreemOS VO management mechanisms.

Chapter 5

Conclusions

This deliverable has described the general architecture of the Service/Resource Discovery System (SRDS), starting from high-level requirements and down to the design and validation of the current prototype due to M18. The SRDS provides diverse information management functionalities within the XtreemOS system, with a common denominator of aiming at high scalability and efficiency over large and geographically distributed Grids.

We can distiguish two main perspectives in the document, the short term one, concerning the development of the M18 prototype of the SRDS, and the long term one, taking into account future developments and research issues.

Requirements from other WPs of the project constrain the SRDS, and while all of the needs have been considered, in this first design we focused more on requirements from the Application Execution Management (WP 3.3) and from the Data Management Services (WP 3.4). On the other hand, the need to interact with the systems and services realized by other WPs of the project led us to the design of a modular, extensible architecture.

The overall architecture of the SRDS has been decomposed into interacting modules, a very important one being the Resource Selection Service, and the other ones composing the Application Directory Service (ADS).

We have discussed in chapters 1 and 1 the reasons of the decomposition, that we summarize in the following.

- Exploitation of the "machete and bistoury" approach to ensure fast implementation and scalability to large platforms since the first prototype, exploiting standard DHT technology.
- Provision of configurability of the directory service functionalities to the varying needs of XtreemOS modules and applications.
- Future extendability to more advanced strategies of query resolution, supporting a richer query semantics and dynamically changing information.

The modules composing the SRDS prototype due to M18 have been analyzed and validated. Their structures have been described, and their properties have been

verified with extensive testing on simulators, and on real computing platforms, most notably on the Grid'5000 testbed. The test conditions included both reliable and unreliable platforms, affected by network errors and resource churns. From the description and the test results shown we can conclude that the SRDS prototype meets the XtreemOS scalability and performance targets, although some of the more advanced functionalities of the SRDS will not be available at M18.

The first target approaching is that of the M18 prototype. The VUA and ISTI/CNR groups worked to finalize and integrate prototypical implementations of the RSS and ADS modules, and make them available. The following roadmap sketches the most important steps after M18.

5.1 Roadmap

Our foreseen roadmap includes both more concrete steps, concretely oriented toward evolving the M18 prototype, and more open issues we plan to investigate, namely advanced features we wish to implement in the next prototype, and research topics that look promising in a 4-year perspective.

As previously outlined, requirements coming from key WPs have been prioritized over the rest. Further steps to take immediately after M18 will be:

- to validate the integration of the SRDS with other XtreemOS services;
- to extend the architecture with more detailed requirements collected by other WPs;
- to continue the research activity toward scalable and efficient techniques to support the SRDS functionalities over large networks.

The main directions for future work and the open research issues, that we have outlined in sections 2.7 and 3.6, encompass:

- enhancing the support for query result caching;
- exploiting network proximity in building the overlay network;
- efficiently managing *dynamic* and *semi-dynamic* attributes;
- supporting *range-*, *neighborhood-*, *proximity-queries*, through hierarchical approaches or space filling techniques;
- studying the tradeoff between dynamic creation of instances of the ADS, to act as a proxy for a common DHT, and the run-time startup overheads of overlay networks, to separately manage the information for each new application;
- providing dynamic load balancing over DHT rings;
- providing dynamic namespace creation functionalities, that could support changing needs on the part of applications and Virtual Organizations.

Bibliography

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

[2] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the SIGCOMM Conference*, pages 353–366, August 2004.

[3] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *Fourth International Workshop on Grid Computing (Grid2003)*, 2003.

[4] Massimo Coppola, Yvon Jegou, Brian Matthews, Christine Morin, Luis Pablo Prieto, Oscar David Sanchez, Erica Y. Yang, and Haiyan Yu. Virtual organization support within a grid-wide operating system. *IEEE, pending revision*, 2007. Available from XtreemOS gforge in articles/Submitted.

[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] First version of XtreemOS testbed user manual D4.3.1. Deliverable D4.3.1, XtreemOS WP4.3.

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

[12] Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67, May 2004.

[13] Jun Gao and Peter Steenkiste. Efficient Support for Similarity Searches in DHT-based Peer-to-Peer Systems. In *IEEE International Conference on Communications (ICC'07)*, 2007.

[14] Grid'5000 web site. <https://www.grid5000.fr/>.

[15] Hypertext Transfer Protocol – HTTP/1.1. <http://www.w3.org/Protocols/rfc2616/rfc2616.html/>.

[16] Alexandru Iosup, Mathieu Jan, Ozan Sonmez, and Dick H.J. Epema. On the dynamic resource availability in grids. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, September 2007.

[17] Java web site. <http://java.sun.com/>.

[18] JDL web site. <http://www.ogf.org/documents/GFD.56.pdf/>.

[19] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), August 2007.

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

[21] PeerSim. http://peersim.sourceforge.net.

[22] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables. ACM PODC, 2004.

[23] Sylvia Ratnasamy, Paul Francis Mark, Handley Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. ACM SIGCOMM, 2001.

[24] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[25] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems*, 9(2):170–184, August 2003.

[26] Christina Schmidt and Manish Parashar. Enabling Flexible Queries with Guarantees in P2P Systems. *IEEE Internet Computing*, 8(3):19–26, May/June 2004.

[27] David Spence and Tim Harris. XenoSearch: Distributed resource discovery in the XenoServer open platform. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, pages 216–225, June 2003.

[28] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. ACM SIGCOMM, 2001.

[29] Schütt Thorsten, Schintke Florian, and Reinefeld Alexander. Chord#: Structured Overlay Network for Non-Uniform Load-Distribution. Technical report, Zuse Institute Berlin (ZIB), 2005.

[30] Paolo Trunfio, Domenico Talia, Charis Papadakis, Paraskevi Fragopoulou, Matteo Mordacchini, Mika Pennanen, Konstantin Popov, Vladimir Vlassov, and Seif Haridi. Peer-to-Peer resource discovery in Grids: Models and systems. Technical report, Future Generation Comp. Syst. 23(7): 864-878, 2007.

[31] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2), 2005.

[32] Spyros Voulgaris and Maarten van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Proceedings of the 11th International Euro-Par Conference*, pages 1143–1152, August 2005.

[33] WP3.4. Requirements and Interface Proposal for the XtreemOS Directory Service.

[34] Erica Y. Yang, Brian Matthews, Amit Lakhani, Yvon Jegou, Christine Morin, Oscar David Sanchez, Carsten Franke, Philip Robinson, Adolf Hohl, Bernd Scheuermann, Daniel Vladusic, Haiyan Yu, An Qin, Rubao Lee, Erich Focht, and Massimo Coppola. Virtual organization management in xtreemos: an overview. In Thierry Priol and Marco Vanneschi, editors, *Towards Next Generation Grids*, CoreGRID series, pages 73–82, August 27-28, 2007, Rennes, France, August 2007. Springer. ISBN 978-0-387-72497-3.