Project no. IST-033576

XtreemOS

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

Extended version of a service / resource discovery system D3.2.13

Due date of deliverable: December 1st, 2009
Actual submission date: December 23rd, 2009

Start date of project: June 1st 2006

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

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

Version 1.0 / Last edited by Massimo Coppola / December 23, 2009

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

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	15/10/09	Guillaume Pierre	VUA	document skeleton
0.1	23/10/09	Jan Sacha	VUA	RSS self-management section
0.2	02/11/09	Corina Stratan	VUA	RSS manual management section
0.3	03/11/09	Guillaume Pierre	VUA	Wrapped up the RSS section as a whole
0.4	03/12/09	Susanna Martinelli	CNR	SRDS architecture, Vivaldi module, monitoring interface
0.5	03/12/09	Emanuele Carlini	CNR	SRDS JDS/AEM text
0.6	10/12/09	Susanna Martinelli, Emanuele Carlini, Patrizio Dazzi, Massimo Coppola	CNR	Corrections and additions
0.7	15/12/09	Massimo Coppola, Susanna Martinelli	CNR	Restructured SRDS chapter, added Executive Abstract, Introduction
0.8	21/12/09	Massimo Coppola, Emanuele Carlini, Susanna Martinelli, Laura Ricci	CNR	Corrections according to comments from internal reviewer
1.0	23/12/09	Massimo Coppola	CNR	Corrections according to comments from internal reviewer

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

Tasks related to this deliverable:

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

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

Contents

Executive summary

In this document we recap the overall structure of the Service/Resource Discovery System (SRDS). The SRDS software module, developed by CNR, acts as a front-end of the highly scalable services toward the rest of the XtreemOS system, and implementing Resource Discovery and the Application Directory Services. The Resource Selection Service (RSS), a specialized P2P overlay developed by VUA, is a core component of the system, targeting highly scalable resource location. Previous development of the whole SRDS distributed service is documented in deliverables D3.2.4 and D.3.2.8.

After briefly surveying the overall structure of the SRDS and of the RSS, we discuss in detail the improvements and changes of these modules that were performed until PM40, leading to the reference implementation for the XtreemOS revision 2.1. Both the SRDS and RSS modules have been extended and reworked in order to satisfy more stringent requirements of (1) reliability (2) performance (3) usability in the real system.

Concerning the SRDS, this deliverable discusses the changes in the software architecture in Section 2.1, which address improved network fault tolerance, extended configurability, stronger checking of system interaction. The new SRDS monitoring interface, improving system management in-the-large, is discussed in 2.2. New mechanisms and semantics of queries for resource location, casted as extensions to the JSDL XML dialect, are discussed in Section 2.3 and 2.4, leading to a tighter integration of the SRDS with the rest of the XtreemOS system. Performance evaluation of the changed functionalities of the SRDS prototype is reported in Section 2.5, concerning the AEM and JDS services used in resource discovery and management. The test results confirm that the overall SRDS system is scalable to large networks.

The RSS research was focused on the dynamic management and tuning of the overlay's parameters. To address these aspects, we have designed a set of extensions to the current RSS protocols and tested them through simulation. One goal is to be able to change the set of nodes' attributes at runtime (thus restructuring the overlay), without having to restart the RSS daemons. The discussion, analysis and experimental evaluation of this first feature are reported in Section 3.1, and lead us to conclude that the cost of updating the RSS network related to the attribute set change is low enough to allow the system to reconfigure while running. Building on top of this flexibility, a self-management procedure is also developed, allowing to tune the RSS overlay automatically at run-time. Details, a behavioural model and its evaluation based on extensive testing are presented in Section 3.2, which allow us to start the integration of the new features within the XtreemOS system, increasing its flexibility and efficiency.

1 Introduction

The Service/Resource Discovery System (SRDS) is the main tool used in XtreemOS to locate resources and services over a large platform without incurring in scalability and reliability issues. As the platform size grows, the SRDS services have to overcome the effects of scale in term of access latencies, resilience to faults and to node churn. For this reason, the SRDS has been structured since the beginning as a set of interoperating P2P networks, which by their very nature exhibit high scalability and fault tolerance. The work involved in the developing of such a distributed service has lead to the results already documented in deliverables D3.2.4 and D.3.2.8.

In this document we recap the overall structure of the SRDS, a software module developed by CNR to act as a front-end of the highly scalable services toward the rest of the XtreemOS system, and of its core component the Resource Selection Service (RSS), a specialized P2P overlay developed by VUA, targeting highly scalable resource location.

The document is structured into two main sections, namely Section 2 concerning the SRDS and Section 3 discussing the RSS, where we describe the improvements made to these XtreemOS modules until project month 40.

Both modules have been extended and reworked in order to satisfy more stringent requirements of reliability and usability in the real system. Concerning the SRDS, this deliverable discusses the changes in the software architecture in Section 2.1, its monitoring interface in 2.2, improvements to the resource query functionalities in Section 2.3 and 2.4, and performance evaluation of the module in Section 2.5.

The RSS has undergone a deep evolution, with the introduction of manual, on-line reconfiguration capability to change the set the attributes and parameters of the overlay network without having to restart it. The discussion, analysis and experimental evaluation of this first feature are reported in Section 3.1. Building on top of this feature, a Self-management procedure is also developed, allowing to tune the RSS overlay automatically at run-time. Details, a model and its evaluation based on extensive testing are presented in Section 3.2.

2 SRDS

The SRDS is a complex module tying together a number of separate overlays, which cooperate in providing a set of highly available and scalable services throughout the XtreemOS platform. Its architecture and functionalities have been discussed in detail in project deliverables D3.2.4 and D3.2.8 [19, 4].

SRDS Architectural Overview The architecture shown in Figure 1 is deployed on each physical node of the system (core and resource nodes). As all computational nodes within an XtreemOS system are at the same time part of more different overlays, which differ in their characteristics and provided functionalities, the upper layer of the SW architecture of Figure 1 provides a set of common interfaces (Facades) to the SRDS services. Local and remote XtreemOS components that need to access the information stored in those overlays will access these interfaces, which can be customised and whose use can be restricted to a single XtreemOS module.

An intermediate SW layer (comprising Generic Information Providers and Query/Provide Modules) provides common information management and query processing capabilities.

The lower layer in the figure wraps the interfaces of different overlay network implementations into a common abstraction, and provides the ability for each XtreemOS machine to start up new overlays, and to join existing ones.

All sub-modules managing specific overlays at the lowermost level in the SRDS architecture currently are variations on the basic Distributed Hash Table (DHT) paradigm, with the important exception of the RSS overlay. The RSS and the OverlayWeaver P2P overlays have a primary role in the process of resource selection, thus the integration with RSS is tighter and less generic, but more optimized from the functional and performance viewpoints. It is important here to remind that the SRDS is capable of dealing with resources which have both **static**-valued attributes, that is attributes whose value does not change for the life of the resource itself, and **dynamic**-valued attributes, whose value is free to change at any moment, and will likely change quite often as the platform is used (e.g. the node load and free memory). We will simply distinguish resource attributes into **static** and **dynamic**, where no ambiguity can arise.

The combination of the RSS approach (designed to deal with static attributes and to be fast and highly scalable in the overlay size) and the DHT approach of SRDS (less scalable, but suited and optimized to deal with dynamic attributes, and thus capable to refine the query result taking dynamic values into account) provides flexible and scalable resource location services within XtreemOS. In the SRDS architecture shown, the RSS overlay is queries for resources first, and its

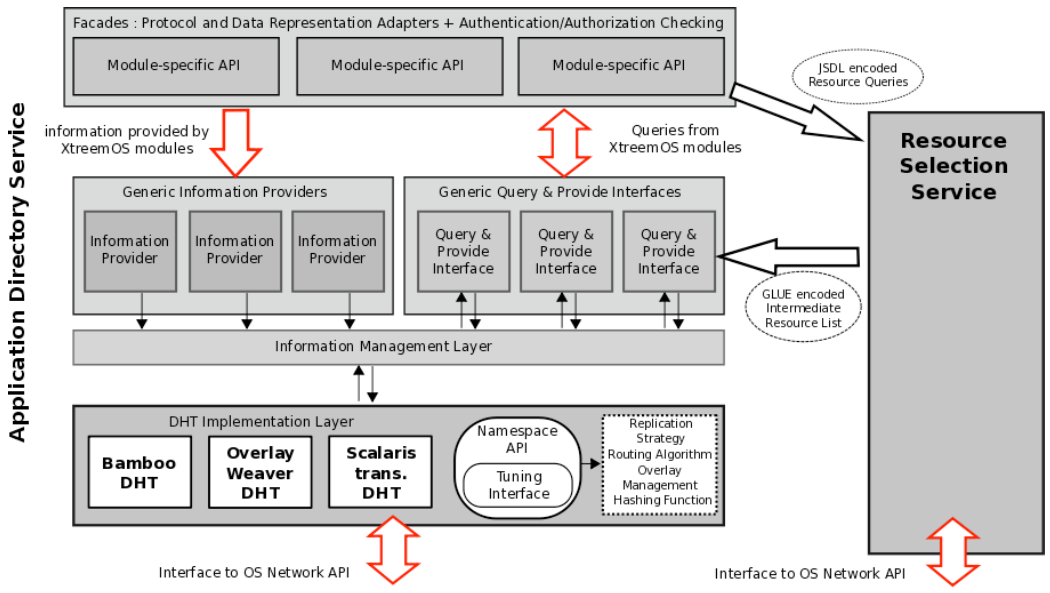

Facades : Protocol and Data Representation Adapters + Authentication/Authorization Checking
Module-specific API
Module-specific API
Module-specific API
JSDL encoded Resource Queries
Application Directory Service
information provided by XtreemOS modules
Queries from XtreemOS modules
Resource Selection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

Figure 1: Overall software architecture of the SRDS, including the RSS, deployed on each node of the XtreemOS system.

answer is refined by filtering out nodes whose dynamic attributes do not match the constraints expressed in the query (a "machete and bistoury" metaphor).

Recent changes to the SRDS design During the last year, a significant effort has been made to increase both the performance and the flexibility of use of the SRDS. The first part of this deliverable lists the improvements related to the SRDS code developed and maintained by CNR. The SRDS code interfaces with the DIXI/XATI communication framework for the management of dynamic resource information and various directory-like services like for instance the Application Directory Service. In order to provide such heterogeneous services, the SRDS coordinates and exploits three different overlay networks (RSS, Scalaris, OverlayWeaver).

The remaining part of this section will summarize the different improvements made, with respect to

- increased usability and ease of SRDS configuration
- robustness of code and of integration within XtreemOS
- scalable performance of SRDS provided services, with respect to overlay size and system load.

Section 2.1 describes changes introduced in SRDS architecture with the release 2.0 of XtreemOS. Most changes are related to increasing the reliability and upgrade-ability of SRDS. Section 2.1.1 describes the improvements in fault-tolerance with respect to network connectivity, Section 2.1.3 regards increased configurability, and Section 2.1.4 concerns the introduction of new XML validation points for the JSDL and GLUE data. Some changes were aimed at providing new functionalities, like the improved query layer described in Section 2.1.2.

Section 2.2 describes an addition to the user interface, the new Web interface introduced to check the status of the DHTs that SRDS exploits.

Section 2.3 describes the extensions made to the JSDL XML format to improve its expressiveness, addressing the issue of providing a dynamic semantics for JSDL elements either already existing or not described by the JSDL standard.

Section 2.4 presents the new functionality of neighbourhood query for Vivaldi coordinates.

Finally, section 2.5 reports performance tests, made on Grid5000 platform, regarding functionalities that SRDS provides to Job Directory Services and Application Execution Management

2.1 SRDS Architectural Changes

2.1.1 Increasing P2P Overlays Reliability

While two of the overlay networks exploited by the SRDS are developed within the XtreemOS project (namely Scalaris and RSS) and are hence tested for scalability and reliability over XtreemOS by their own developers, the third P2P network, OverlayWeaver, is a commodity P2P implementation.

OverlayWeaver (OW) [22] is a Java P2P and DHT framework which is used by SRDS mainly as a support for the Discovery System. OW integration with SRDS has been described in previous projects deliverables. It is a project policy to test library upgrades before accepting them in the main repository.

In the release 2.0 of XtreemOS we upgraded OW, moving from 0.8.9 to release 0.9.7, in order to addresses an issue of TCP socket pollution issue, due to OW not properly releasing used sockets, that caused network port congestion after several days of XtreemOS uptime. A fault tolerant join routine has been added to the code of the SRDS on top of the OW functionalities, to avoid that delays or disconnections at boot time prevent the overlays from correctly forming.

What happened with the plain OW routine was that a node, failing to connect to the bootstrap node, went on and created a single-node overlay all by itself. The SRDS release 0.3.0 has been added an exception catch and retry mechanism, which activates whenever OW detects that the current node is not connected to the overlay bootstrap node as reported in the SRDS configuration. It is now possible to exploit multiple-bootstrap-nodes overlay configurations (each OW peer is potentially a bootstrap) without the risk of partitioning the overlay.

2.1.2 Generalized Query Layer

The Query Engine used to support the ADS queries on top of DHT networks has been extended and made more general, by allowing the user to choose a concurrency-reliable data management when needed, which in turn relies on the properties of the Scalaris transactional DHT.

The introduction of a general query layer improves the extendability of the ADS, by easing the integration of new services, and allows to separate DHT namespace usage (typically a series of put/get operations on DHT) from DHT namespace management (creation, deletion and storing of namespaces).

The typical situation the query layer addresses is that of a client (e.g. the Job Directory Service) providing structured data (job descriptions) and eventually issuing queries to retrieve data records. The queries are based on some attributes of the stored data (e.g. the Job id).

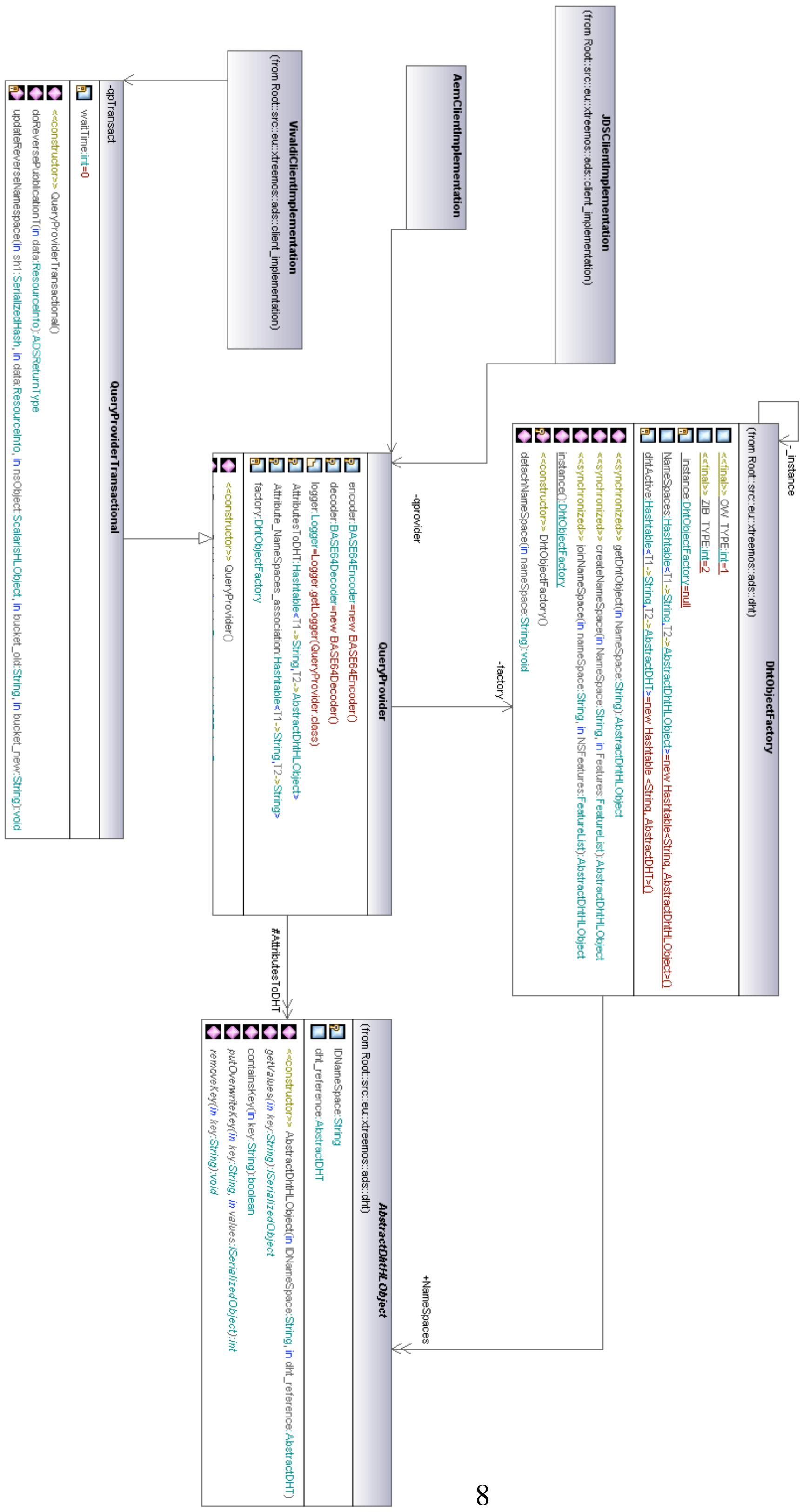
The new query layer manages all the data provided by its client as single resource. A resource is defined as a list of attribute-value pairs. As an example, if the data represent a job the attributes may be the "jobID", the "userID" associated with that job, as well as the version of the job.

The Qprovide layer provides an API to XtreemOS services for accessing the underlying DHTs. The API must match different services, hence different operations and attributes. To achieve this flexibility we defined a resource specification that handles three different classes of resource attributes, with different management requirements.

- The *ID* attribute identifies the resource, thus we assume it has an unique value and a resource must have only a ID attribute. The ID attribute is used to directly retrieve the single resource associated with it (direct query).

- The remaining attributes are divided into *strict* and *free* attributes. There is a sharp distinction between them: strict attributes are mandatory in each resource submitted (each client, beside the ID attribute, will describe all the strict attributes at service initialization). Strict attributes can be used in queries to retrieve a list of resources matching a particular attribute value.

- Free attributes are unrestricted, and still memorized with the resource description, but no efficient way is provided to query a resource from the values of its free attributes. Free attributes are not searchable and can be retrieved only with a direct query.



JDSClientImplementation
(from Root::src::eu::xtreemos::ads::client_implementation)
AemClientImplementation
VivaldiClientImplementation
(from Root::src::eu::xtreemos::ads::client_implementation)
QueryProviderTransactional
-qpTransact
waitTime:int=0
<<constructor>> QueryProviderTransactional()
doReversePublicationT(in data:ResourceInfo):ADSReturnType
updateReverseNamespace(in sh1:SerializedHash, in data:ResourceInfo, in nsObject:ScalarIsHLObject, in bucket_old:String, in bucket_new:String):void
QueryProvider
-qprovider
encoder:BASE64Encoder=new BASE64Encoder()
decoder:BASE64Decoder=new BASE64Decoder()
logger:Logger=Logger.getLogger(QueryProvider.class)
AttributesToDHT:Hashtable<T1->String,T2->AbstractDHtHLObject>
Attribute_NameSpaces_association:Hashtable<T1->String,T2->String>
factory:DhtObjectFactory
<<constructor>> QueryProvider()
-factory
DhtObjectFactory
(from Root::src::eu::xtreemos::ads::dht)
-_instance
<<final>> OW_TYPE:int=1
<<final>> ZIB_TYPE:int=2
_instance:DhtObjectFactory=null
NameSpaces:Hashtable<T1->String,T2->AbstractDHtHLObject>=new Hashtable<String,AbstractDHtHLObject>()
dhtActive:Hashtable<T1->String,T2->AbstractDHT>=new Hashtable<String,AbstractDHT>()
<<synchronized>> getDhtObject(in NameSpace:String):AbstractDHtHLObject
<<synchronized>> createNameSpace(in NameSpace:String, in Features:FeatureList):AbstractDHtHLObject
<<synchronized>> joinNameSpace(in nameSpace:String, in NSFeatures:FeatureList):AbstractDHtHLObject
instance():DhtObjectFactory
<<constructor>> DhtObjectFactory()
detachNameSpace(in nameSpace:String):void
#AttributesToDHT
+NameSpaces
AbstractDHtHLObject
(from Root::src::eu::xtreemos::ads::dht)
IDNameSpace:String
dht_reference:AbstractDHT
<<constructor>> AbstractDHtHLObject(in IDNameSpace:String, in dht_reference:AbstractDHT)
getValues(in key:String):SerializedObject
containsKey(in key:String):boolean
putOverwriteKey(in key:String, in values:SerializedObject):int
removeKey(in key:String):void

Figure 2: UML diagram representing the Java implementation classes of the eu.xtreemos.ads.qengine package.

In principle a resource may contain any number of *strict* attributes. In practice, the service should keep this number as low as possible to increase the overall performances, as each strict attribute S requires an auxiliary data structure within the DHTs in order to allow reverse queries on that attribute. Maintaining this auxiliary information is costly, since a list (or other data structure) of all IDs that contain a specific value of S must be managed at the location of that value within the DHT. For example, in the JDS client implementation, the userID is a strict attribute, so that it is possible to retrieve a list of the jobID resources which share a given userID (i.e. all the jobs of a user).

Two different implementations of the Query Layer have been developed, a low-overhead one which suffices for the AEM and JDS needs, and a concurrency-aware one that enforces data coherence by exploiting a transactional DHT where appropriate.

In order to enforce the consistency of data, it's worth to point out that the simpler implementation does not need to use Scalaris, but

- it is not concurrent-safe for several different client instances to modify the values of attributes defined as strict,
- it is possible, but not concurrent-safe to delete a resource by its ID.

That is, the simplest version currently supports only one writing client and one reading client at time, which is enough for the AEM/JDS service needs, where jobs are handed by a single instance of the AEM at a time. Conversely, the more general implementation of the Query layer can enforce distributed consistency by employing transactions when updating linked data structures, and properly re-executing the transaction if the distributed state of the system has changed.

Two new Java classes called *QProvider* and *QProviderTransacional* have been created, which implement the operations described above, respectively without and using the transactional support. The class *ResourceInfo* represent the resource. All these classes are contained into *eu.xtreemos.ads.qengine* package. Figure 2 shows the UML diagram for these classes.

2.1.3 Configuring the SRDS Dynamic Data Monitoring Function

The *dynamic data monitor* is a process of the SRDS that automatically monitors the dynamic attributes of interest to the Resource Discovery Service, and publishes them on the DHT within the appropriate resource descriptor. Although by default all the measurements known to the SRDS are enabled, it is now possible to choose to enable/disable a particular attribute measurement via the *srds.properties* file.

Currently, this feature allows to save a little overhead both in the measurement process itself and allowing to send shorter messages on the network. However, as different modules measuring the dynamic data can be made available, it will be possible to configure which probes are going to be used for the measurement. We plan to introduce in SRDS a new module based on the REMED P2P algorithm [3] that will improve the performance of range queries along with reducing of the traffic generated by resource updates. The basic concept of REMED lies in the exploitation of the temporal locality of the measurements to skip a percentage of the attribute updates. This approach inherently gains great advantages in terms of traffic over the network when the number of attributes published is low.

2.1.4 XML validation

The SRDS continuously exchanges information about XtreemOS resource nodes with other modules of the system, basically the AEM and the RSS. While existing standard data encodings, mostly based on XML, have been used when available, several of the new functionalities of XtreemOS called for the definition of new XML formats or the extension of existing languages. Action has been taken within the project to reach a consensus toward common internal standards, and possibly promote the inclusion of XtreemOS extensions within existing standards.

The approach raises the need to enforce strict adherence to precisely defined XML interpretation and validation rules, that can enforce an existing standard or provide a reference implementation for extension proposals. Two formats which are candidate for extensions are the JSDL one (used to describe Jobs and search for resources) and the GLUE one (GLUE encodes resource lists for application to be run onto). The SRDS has been thus provided with strict XML validation behaviour, that for testing purposes can be disabled or configured to enforce a specific XML schema and extension (e.g. for the GLUE schema this boils down to editing the property *srds.configuration.files.glueSchema* in the srds.properties config file).

- Resource requests coming from the AEM are subject to validation against the JSDL standard schema with POSIX and XtreemOS custom extensions. The JSDL extensions more strictly related to SRDS are described in Section 2.3.

- GLUE validation is performed on the resource list returned by the RSS service before the SRDS filters them according to dynamic attribute values. Concerning the GLUE standard, we are currently using the standard OGF Schema, version 1.2 with no custom extension. We plan to upgrade from the 1.2 to the 2.0 GLUE schema version for the next release, and a possible extension with custom resource attributes is foreseen.

2.2 Monitoring the Status of Overlays via HTTP

The latest SRDS release introduced a new feature that permits to check the status of the DHTs that the SRDS exploits. Both the Overlay Weaver and Scalaris DHT implementation provide by themselves a web monitoring server that allows to check the current node status, the overlay status and allows to put/get values on the DHT for testing purposes.

Tools like these are useful when deploying a large system such as XtreemOS, and testing the correct configuration of the nodes. The OW and Scalaris HTTP interfaces are available at each node of the overlay at specific TCP ports (although those can be configured). Of course, in a general settings we will not wish to have these ports open to the casual users, and it would be quite poor usability to force the administrator to access several different monitoring tool just in order to check the overall SRDS configuration is correctly working. We addressed this problem by grouping those interfaces together within the SRDS.

The *Overlay Weaver status interface* contains first a short node presentation, which lists:

- the node ID
- the routing algorithm
- the lookup style.
- the number of stored keys from the web interface

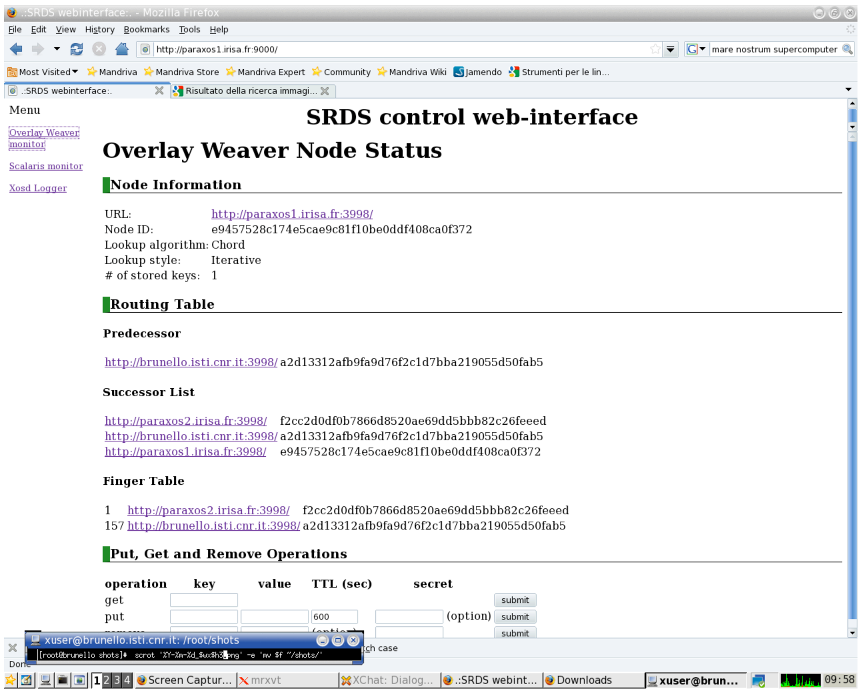
It also shows information regarding the structure of the DHT, showing the list of nodes belonging to the the predecessor set, the successor set and the the finger table of the current node.

The Scalaris status interface similarly displays the overlay status, with a node list and a debugging tool. The full Scalaris web interface can be viewed only from the Scalaris boot node, while on regular node only minimal information is reported.

Both the OW and Scalaris interfaces allow getting/putting informations on the DHT. Deletion is not allowed on Scalaris, and is available with Overlay Weaver (as it is from the programmer's API, due to different implementation constraints of the two DHTs).

SRDS aggregates the OW and Scalaris HTTP interfaces into a single web interface, in order to simplify the Sysop management. The aggregate interface provides also a visualizer of recent messages from the xosd daemon local to the same node, allowing easier checking of any errors/warnings occurred inside the XOSD service, including those within the SRDS and its linked overlays.

The aggregation is achieved by means of a small local HTTP proxy/web server, and security can be enforced by restraining regular users and remote connections from accessing the monitoring services.



SRDS control web-interface

Overlay Weaver Node Status

Node Information

URL:	http://paraxos1.irisa.fr:3998/
Node ID:	e9457528c174e5cae9c81f10be0ddf408ca0f372
Lookup algorithm:	Chord
Lookup style:	Iterative
# of stored keys:	1

Routing Table

Predecessor

http://brunello.isti.cnr.it:3998/ a2d13312afb9fa9d76f2c1d7bba219055d50fab5

Successor List

http://paraxos2.irisa.fr:3998/	f2cc2d0df0b7866d8520ae69dd5bbb82c26feeed
http://brunello.isti.cnr.it:3998/	a2d13312afb9fa9d76f2c1d7bba219055d50fab5
http://paraxos1.irisa.fr:3998/	e9457528c174e5cae9c81f10be0ddf408ca0f372

Finger Table

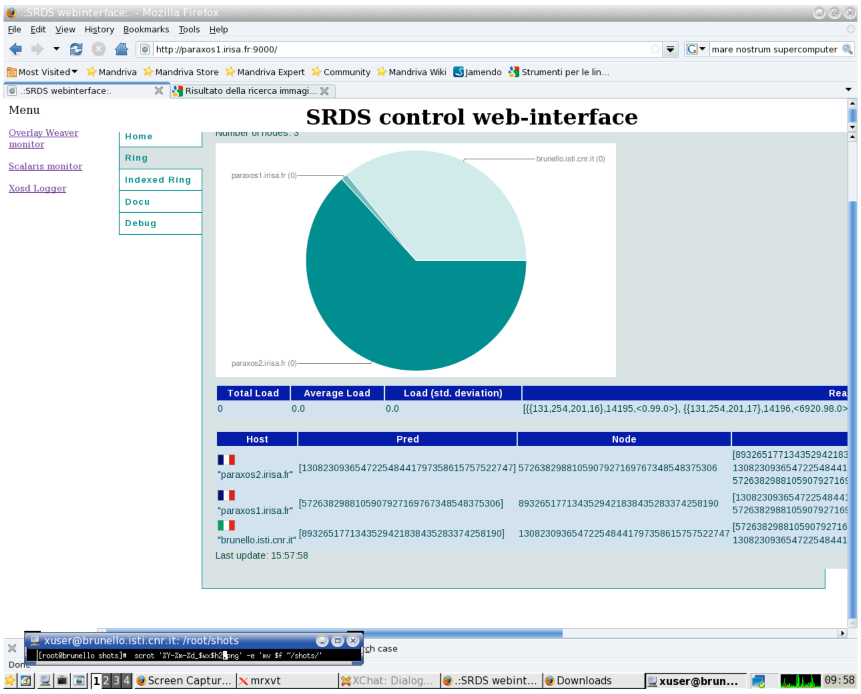
1	http://paraxos2.irisa.fr:3998/	f2cc2d0df0b7866d8520ae69dd5bbb82c26feeed
157	http://brunello.isti.cnr.it:3998/	a2d13312afb9fa9d76f2c1d7bba219055d50fab5

Put, Get and Remove Operations

operation	key	value	TTL (sec)	secret	
get					
put			600	(option)	

```
xuser@brunello.isti.cnr.it: /root/shots
[root@brunello shots]#  scrot '%Y-%m-%d_$wx$h3.png' -e 'mv $f ~/shots/'
```

Figure 3: The Overlay Weaver manager interface.

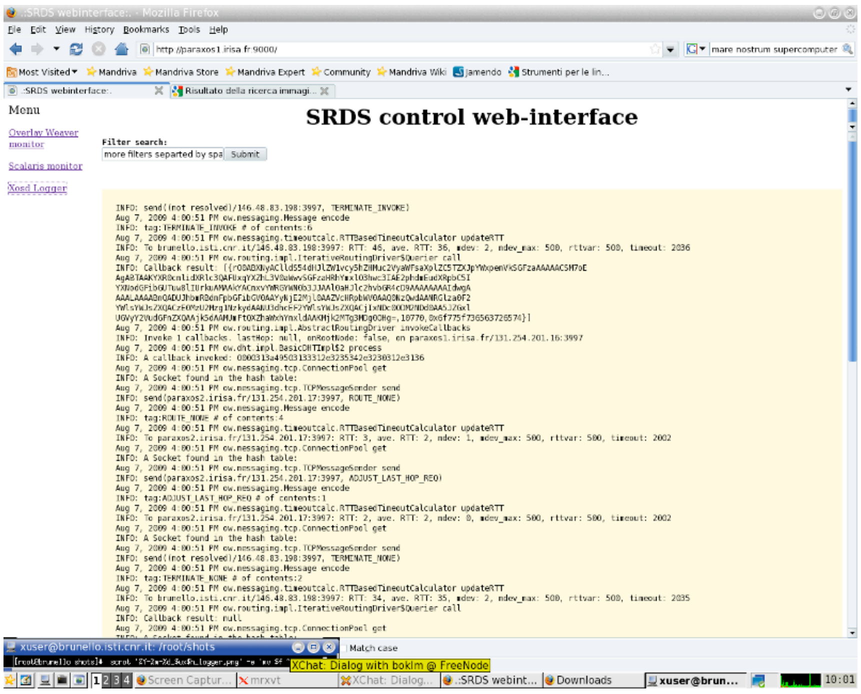


Total Load	Average Load	Load (std. deviation)	Rea
0	0.0	0.0	[{{131,254,201,16},14195,<0.99.0>}, {{131,254,201,17},14196,<6920.98.0>

Host	Pred	Node	
🇫🇷 "paraxos2.irisa.fr"	[130823093654722548441797358615757522747]	5726382988105907927169767348548375306	[893265177134352942183 130823093654722548441 572638298810590792716
🇫🇷 "paraxos1.irisa.fr"	[5726382988105907927169767348548375306]	8932651771343529421838435283374258190	[130823093654722548441 572638298810590792716
🇮🇹 "brunello.isti.cnr.it"	[8932651771343529421838435283374258190]	130823093654722548441797358615757522747	[572638298810590792716 130823093654722548441

Last update: 15:57:58

Figure 4: The Scalaris manager interface.


.:SRDS webinterface:. - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://paraxos1.irisa.fr:9000/
Menu
Overlay Weaver monitor
Scalaris monitor
Xosd Logger
SRDS control web-interface
Filter search:
more filters separted by spa
Submit

```
INFO: send((not resolved)/146.48.83.198:3997, TERMINATE_INVOKE)
Aug 7, 2009 4:00:51 PM ow.messaging.Message encode
INFO: tag:TERMINATE_INVOKE # of contents:6
Aug 7, 2009 4:00:51 PM ow.messaging.timeoutcalc.RTTBasedTimeoutCalculator updateRTT
INFO: To brunello.isti.cnr.it/146.48.83.198:3997: RTT: 46, ave. RTT: 36, mdev: 2, mdev_max: 500, rttvar: 500, timeout: 2036
Aug 7, 2009 4:00:51 PM ow.routing.impl.IterativeRoutingDriver$Querier call
INFO: Callback result: [{rO0ABXNyACllddS54dHJlZW1vcy5hZHMuc2VyaWFsaXplZC5TZXJpYWxpemVkSGFzaAAAAACSM7oE
AgABTAAKYXR0cmlidXRlc3QAFUxqYXZhL3V0aWwvSGFzaHRhYmxlO3hwc3IAE2phdmEudXRpbC5I
YXNodGFibGUTuw8lIUrkuAMAAkYACmxvYWRGYWN0b3JJJAAl0aHJlc2hvbGR4cD9AAAAAAAAIdwgA
AAALAAAABnQADUJhbmR0dmFpbGFibGV0AAYyNjE2MjlOAAZVcHRpbWV0AAQ0NzQwdAANRGlza0F2
YWlsYWJsZXQACzE0MzU2Mzg1NzkydAANU3dhcEF2YWlsYWJsZXQACjIxNDc0ODM2NDd0AA5JZGxl
UGVyY2VudGFnZXQAAjk5dAAMJmFtOXZhaWxhYmxldAAKMjk2MTg3MDg0OHg=,10770,0x6f775f736563726574}]
Aug 7, 2009 4:00:51 PM ow.routing.impl.AbstractRoutingDriver invokeCallbacks
INFO: Invoke 1 callbacks. lastHop: null, onRootNode: false, on paraxos1.irisa.fr/131.254.201.16:3997
Aug 7, 2009 4:00:51 PM ow.dht.impl.BasicDHTImpl$2 process
INFO: A callback invoked: 0800313a49503133312e3235342e3230312e3136
Aug 7, 2009 4:00:51 PM ow.messaging.tcp.ConnectionPool get
INFO: A Socket found in the hash table:
Aug 7, 2009 4:00:51 PM ow.messaging.tcp.TCPMessageSender send
INFO: send(paraxos2.irisa.fr/131.254.201.17:3997, ROUTE_NONE)
Aug 7, 2009 4:00:51 PM ow.messaging.Message encode
INFO: tag:ROUTE_NONE # of contents:4
Aug 7, 2009 4:00:51 PM ow.messaging.timeoutcalc.RTTBasedTimeoutCalculator updateRTT
INFO: To paraxos2.irisa.fr/131.254.201.17:3997: RTT: 3, ave. RTT: 2, mdev: 1, mdev_max: 500, rttvar: 500, timeout: 2002
Aug 7, 2009 4:00:51 PM ow.messaging.tcp.ConnectionPool get
INFO: A Socket found in the hash table:
Aug 7, 2009 4:00:51 PM ow.messaging.tcp.TCPMessageSender send
INFO: send(paraxos2.irisa.fr/131.254.201.17:3997, ADJUST_LAST_HOP_REQ)
Aug 7, 2009 4:00:51 PM ow.messaging.Message encode
INFO: tag:ADJUST_LAST_HOP_REQ # of contents:1
Aug 7, 2009 4:00:51 PM ow.messaging.timeoutcalc.RTTBasedTimeoutCalculator updateRTT
INFO: To paraxos2.irisa.fr/131.254.201.17:3997: RTT: 2, ave. RTT: 2, mdev: 0, mdev_max: 500, rttvar: 500, timeout: 2002
Aug 7, 2009 4:00:51 PM ow.messaging.tcp.ConnectionPool get
INFO: A Socket found in the hash table:
Aug 7, 2009 4:00:51 PM ow.messaging.tcp.TCPMessageSender send
INFO: send((not resolved)/146.48.83.198:3997, TERMINATE_NONE)
Aug 7, 2009 4:00:51 PM ow.messaging.Message encode
INFO: tag:TERMINATE_NONE # of contents:2
Aug 7, 2009 4:00:51 PM ow.messaging.timeoutcalc.RTTBasedTimeoutCalculator updateRTT
INFO: To brunello.isti.cnr.it/146.48.83.198:3997: RTT: 34, ave. RTT: 35, mdev: 2, mdev_max: 500, rttvar: 500, timeout: 2035
Aug 7, 2009 4:00:51 PM ow.routing.impl.IterativeRoutingDriver$Querier call
INFO: Callback result: null
Aug 7, 2009 4:00:51 PM ow.messaging.tcp.ConnectionPool get
```

Figure 5: The log tail visualization page.

2.3 XML validation

This section describes in detail the validation process for the AEM resource queries encoded using the Job Description Submitting Language (JSDL). The AEM uses JSDL for submitting jobs specifications to SRDS, and the Grid Laboratory Uniform Environment (GLUE) XML-related language is used for sending replies to the AEM, describing the services/resources that match the related JSDL encoded query. Regarding JSDL we presents a set of extensions aimed at increasing the expressiveness of AEM requests.

Standard JSDL only allows to describe resources whose attributes are all **static** (we set forth this distinction at the beginning of section 2.1), in this being tied to a Scientific HPC Grid mentality, where applications are mostly large MPI batch jobs running on powerful clusters exclusively with each other. On the other hand, the XtreemOS platform has a different goal: sharing of computational resources on a grid must be invisible and efficient just like sharing a single machine among several applications is today. It is essential thus to be ble to specify constraints for the application execution which relate the application behaviour to the value of **dynamic** attributes.

Being able to specify that an application needs a certain amount of *free* memory or a certain amount of *available* computing power is paramount to avoid trashing effects on both that application and all those which happen to share some resource with it.

In the following we describe the currently enforced extension schema in Section 2.3.1, and then, in Section 2.3.2 the simpler and more useful extensions proposed for the next XtreemOS release.

2.3.1 JSDL validation

The JSDL standard exploits two different XML namespaces to describe different semantics for some element contained in the standard.

The first namespace (*http://schemas.ggf.org/jsdl/2005/11/jsdl*) describes all the elements that belong to the basic JSDL normative schema.

The second one (*http://schemas.ggf.org/jsdl/2005/11/jsdl-posix*) contains element that belongs to a particular extension of JSDL. The normative extension defines a schema describing an application executed on a POSIX compliant system. The namespace prefix used for this schema in the specification is *jsdl-posix*.

We set up a further extension with an additional schema describing dynamic resource attributes. As explained in section 2, we call dynamic those attributes that are dynamic-valued. In particular, our schema redefines some elements of the normative schema that can also have dynamic semantics, and it introduces

JSDL Name	RSS (static)	ADS (dynamic)	Dynamic Semantics
OperatingSystem	x		
CPUArchitecture	x		
IndividualCPUCount	x		
IndividualPhysicalMemory	x	x	Amount of free physical memory
IndividualVirtualMemory	x	x	Amount of free virtual memory
IndividualDiskSpace	x	x	Free space on local scratch disk
IndividualNetworkBandwith	x	x	Unallocated network bandwidth

Table 1: JSDL attributes used in SRDS

two new dynamicity-related elements. The new elements are placed within the *jsdl-srds* namespace.

Current JSDL Attributes Management (XtreemOS 2.0) Table 1 lists a subset of JSDL attributes used by RSS and ADS. An attribute may have both static and dynamic meaning or only the static one. That is, whenever an already existing JSDL attribute makes sense as a dynamic value, we assign it a double meaning. A requirement on memory can be interpreted in a physical sense (how much memory is installed on the node) or as a current availability (is there enough free memory right now on the node?). If an attribute has no meaning when interpreted as dynamically valued (e.g. the CPU architecture will not change that easily), no extension is made to the standard for that attribute.

The RSS module is in charge to manage static values of attributes, thus it will completely ignore dynamic-related JSDL extensions. Out of the static attributes, OperatingSystem, CPUArchitecture and IndividualCPUCount are exact values, while IndividualPhysicalMemory, IndividualVirtualMemory, IndividualDiskSpace and IndividualNetworkBandwith are range values.

Queries with Dynamic Constraints For each attribute handled by the ADS a value between 0 and 1 is used for exploiting dynamic query. This value is called **dynamic-epsilon** and it is mandatory for performs query with dynamic constraint. It enables the checking of dynamic values for that attribute (according to the meaning of the attribute listed in table 1), and defines the actual range of values that are acceptable for the dynamic value.

The threshold value for the value of the dynamic attribue is computed from the one of the static value, by keeping the same **UpperBound** (if present) and multiplying the **LowerBound** L of the corresponding static range by the dynamic-epsilon value, to get the dynamic value range. When a resource has a dynamic value lower than the threshold, the node is discarded. That allows the end-user to define queries of the form *"I need a machine with 4 GB installed, and at least 2 GB free"*. The epsilon value is expressed in the JSDL query by extending the JSDL syntax with a new XML attribute within the tag specifying the resource attribute.

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

This mechanism minimizes changes to the JSDL syntax, and allows the same query to address both the dynamic and the static attributes of a query. Since the RSS ignores non-standard XML attributes of the query XML tags, the RSS loks for resources that are within the static range; the ADS will then discard those which, according to dynamic information, do not belong to the range extended by the epsilon parameter.

The JSDL standard permits unlimited ranges to either negative or positive infinity. A valid dynamic constraint requires both an epsilon value and a L element, see table 2 for a full desription of the cases of the dynamic-epsilon extension.

Table 2: Semantics of dynamic attribute queries with the dynamic-epsilon JSDL extension.

Dynamic costraint specification	**Semantics**
`<jsdl-srds:Resource dynamic-epsilon=e>` `<L> xx </L> <U> yy </U>`	Queries resources with: xx≤static_attribute≤yy xx∗e≤dynamic_attribute
`<jsdl-srds:Resource dynamic-epsilon=e>` `<L> xx </L>`	Queries resources with: xx ≤ static_attribute xx∗e≤ dynamic_attribute
`<jsdl-srds:Resource dynamic-epsilon=e>` `<U> xx </U>`	Incorrect dynamic specification is ignored. Query processed w.r.t. static attributes only.
`<jsdl-srds:Resource>` `<L> xx </L> <U> yy </U>`	No dynamic specification given. Query processed w.r.t. static attributes only.

Extending JSDL with new Tags Table 3 presents the new *tags* for the XtreemOS JSDL. These tags convey inherently dynamic values, with no static counterpart. They are missing in the JSDL standard and are relevant in the XtreemOS settings, where we want to be able to deploy processes on machine that are only partially loaded.

JSDL tag	ADS internal name	ADS semantic
IdlePercentage	IdlePercentage	Percentage of idle CPU (%)
Uptime	Uptime	Time from last reboot (minutes)

Table 3: Dynamic attribute on JSDL

The schemas in the last part of this section, report the namespaces and the full tags description needed as extension to the JSDL standard schema Version 1.0 created by OGF.

The **IdlePercentage** element is described by IdlePercentage_Type, much like the other dynamic elements. It allows dynamic-epsilon attribute and contains RangeValue_Type subelements.

We decided to create another XML tag for Uptime since for semantic reasons it should not contain a tolerance attribute. In this case we mapped it with RangeValueNoDynamic_Type that does not allow dynamic-epsilon attribute but contains the same subelements of RangeValue_Type.

2.3.2 Improved Dynamic Attribute Management

In this section we present two major extension to the JSDL which have been already coded, but are not yet used in the current XtreemOS release. The aim at simplyfing and make more uniform the definition of constraints based on dynamic-valued resource attributes. It will substitute the ad-hoc dynamic-epsilon XML attribute with a more general mechanism which can be applied more uniformly. The changes in the SRDS version 3 (to be deployed with XtreemOS public releae 2.1) are

1. Improvement on the dynamic-epsilon system management
2. Introduction of two new Resource Tag

Along with the descriptions we enclose the validation schema for our proposals.

The old dynamic-epsilon mechanism is now linked to a tag named **tolerance**, since this name suits better with the semantic we intend and avoids ambiguity between actual dynamic value and desired dynamic value.

The tolerance attribute is applied to the already existing XML tags LowerBound, UpperBound, Exact, LowerBoundedRange and UpperBoundedRange.

The threshold value for a query about a dynamic value is computed multiplying the corresponding static lower bound, listed previously, by the tolerance, to get the lower bound value for the dynamic value range.

A resource is discarded when the dynamic value for an attribute is not above the threshold range computed using the tolerance. In case of *Exact* values a node is discarded if its dynamic value stays under the threshold (that is, we do not use the upper bound value we could derive from the Exact specification, as it better suits the intended use of the tolerance XML attribute).

In the following we show some usage examples of the **tolerance** attribute.

With the Range values:

```
<jsdl-srds:IndividualDiskSpace>
        <jsdl-srds:Range>
                <jsdl-srds:LowerBound tolerance="0.5">30000
                </jsdl-srds:LowerBound>
                <jsdl-srds:UpperBound tolerance="0.2">600000
                </jsdl-srds:UpperBound>
        </jsdl-srds:Range>
</jsdl-srds:IndividualDiskSpace>
```

With the Exact value:

```

<jsdl-srds:IndividualDiskSpace>
        <jsdl-srds:Exact tolerance="0.5">30000
        </jsdl-srds:Exact>
</jsdl-srds:IndividualDiskSpace>
```

Only the LowerBoundedRange element (the syntax is the same for the Upper-BoundedRange)

```
<jsdl-srds:IndividualDiskSpace>
        <jsdl-srds:LowerBoundedRange>2097152.0
        </jsdl-srds:LowerBoundedRange>
</jsdl-srds:IndividualDiskSpace>
```

Managing Vivaldi coordinates A new feature of the SRDS is to manage neighborhood queries on Vivaldi coordinates[1] inside resource queries. The JSDL request contains inside the Resource tag a new XML tag called **Vivaldi**:

```
 <jsdl:Resources>
 ...
<jsdl-srds:Vivaldi>
        <jsdl-srds:Coordinatex>34,567   </jsdl-srds:Coordinatex>
        <jsdl-srds:Coordinatey>2,589    </jsdl-srds:Coordinatey>
        <jsdl-srds:radius>10,456</jsdl-srds:radius>
</jsdl-srds:Vivaldi >
</jsdl:Resources>
```

[1]Vivaldi coordinates are pseudo coordinates iteratively computed from a set of network delay measurements among resource pairs. Vivaldi coordinates allow to estimate network proximity, i.e. expected communication delay, between arbitrary pairs of nodes.

A set of resulting machines should have vivaldi coordinates in the resulting GLUE XML answer, and those coordinates shall fit within the area delimited by the circle with origin in *(Coordinatex,Coordinatey)* and *radius* ray.

The Vivaldi constraint in the requests can be specified to be mandatory or not. If we want a set of machines that fits the resources constraints (IndividualPhysicalMemory, IdlePercentage, IndividualVirtualMemory etc.) and it is preferable (not mandatory) for them to lie within the circle defined by the Vivaldi constraint, we choose this behaviour by means of the same Vivaldi JSDL tags. The attribute of the Vivaldi tag expressesing best-effort or strict meaning for coordinate matching is named **mandatory** and must have a boolean value.

An example:

```
 <jsdl:Resources>
 ...
<jsdl-srds:Vivaldi mandatory="true">
        <jsdl-srds:Coordinatex>34,567   </jsdl-srds:Coordinatex>
        <jsdl-srds:Coordinatey>2,589    </jsdl-srds:Coordinatey>
        <jsdl-srds:radius>10,456</jsdl-srds:radius>
</jsdl-srds:Vivaldi >
</jsdl:Resources>
```

By default, matching is strict. In this case all the resulting machines should fits with all the resources tag.

```
 <jsdl:Resources>
 ...
<jsdl-srds:Vivaldi mandatory="false">
        <jsdl-srds:Coordinatex>34,567   </jsdl-srds:Coordinatex>
        <jsdl-srds:Coordinatey>2,589    </jsdl-srds:Coordinatey>
        <jsdl-srds:radius>10,456</jsdl-srds:radius>
</jsdl-srds:Vivaldi >
</jsdl:Resources>
```

A false *mandatory* flag specifies a simple preference for machines into the circle, but we will also accept machines outside it.

JSDL SRDS schema

```
 <?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://schemas.ggf.org/jsdl/2005/11/jsdl-srds"
        elementFormDefault="unqualified"
        xmlns:jsdl-srds="http://schemas.ggf.org/jsdl/2005/11/jsdl-srds">
        <!-- COMPLEX TYPES: Definitions for the RangeValueType -->
  <xsd:complexType name="Boundary_Type">
    <xsd:simpleContent>
      <xsd:extension base="xsd:double">
        <xsd:attribute name="exclusiveBound" type="xsd:boolean" use="optional"/>
          <xsd:anyAttribute namespace="##other" processContents="lax"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="Exact_Type">
    <xsd:simpleContent>
      <xsd:extension base="xsd:double">
```

```
        <xsd:anyAttribute namespace="##other" processContents="lax"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="Range_Type">
    <xsd:sequence>
      <xsd:element ref="jsdl-srds:LowerBound"/>
      <xsd:element ref="jsdl-srds:UpperBound"/>
    </xsd:sequence>
   <!-- xsd:anyAttribute namespace="##other" processContents="lax"/ -->
  </xsd:complexType>

  <xsd:complexType name="RangeValue_Type">
    <xsd:sequence>
      <xsd:element ref="jsdl-srds:UpperBoundedRange" minOccurs="0"/>
      <xsd:element ref="jsdl-srds:LowerBoundedRange" minOccurs="0"/>
      <xsd:element ref="jsdl-srds:Exact" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="jsdl-srds:Range" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="dynamic-epsilon" type="xsd:double" xmlns="default␣namespace" use="required"/>
    <!--xsd:anyAttribute namespace="##other" processContents="lax"/-->
  </xsd:complexType>
  <xsd:complexType name="RangeValueNoDynamic_Type">
    <xsd:sequence>
      <xsd:element ref="jsdl-srds:UpperBoundedRange" minOccurs="0"/>
      <xsd:element ref="jsdl-srds:LowerBoundedRange" minOccurs="0"/>
      <xsd:element ref="jsdl-srds:Exact" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="jsdl-srds:Range" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>

    <!--xsd:anyAttribute namespace="##other" processContents="lax"/-->
  </xsd:complexType>

  <!--  ==============Tag added by SRDS================== -->
     <!--  va definita la semantica esatta di IDLEPERCENTAGE -->
    <xsd:element name="IdlePercentage" type="jsdl-srds:RangeValue_Type"/>
    <xsd:element name="Uptime" type="jsdl-srds:RangeValueNoDynamic_Type"/>
    <!--  ============================================================ -->
        <xsd:element name="IndividualPhysicalMemory" type="jsdl-srds:RangeValue_Type"/>
        <xsd:element name="IndividualVirtualMemory" type="jsdl-srds:RangeValue_Type"/>
        <xsd:element name="IndividualNetworkBandwidth" type="jsdl-srds:RangeValue_Type"/>
        <xsd:element name="IndividualDiskSpace" type="jsdl-srds:RangeValue_Type"/>

        <xsd:element name="UpperBoundedRange" type="jsdl-srds:Boundary_Type" />
        <xsd:element name="LowerBoundedRange" type="jsdl-srds:Boundary_Type"/>
        <xsd:element name="Exact" type="jsdl-srds:Exact_Type" />
        <xsd:element name="Range" type="jsdl-srds:Range_Type" />
        <xsd:element name="LowerBound" type="jsdl-srds:Boundary_Type"/>
        <xsd:element name="UpperBound" type="jsdl-srds:Boundary_Type"/>

</xsd:schema>
```

JSDL SRDS schema with Tolerance attribute and tag for Vivaldi coordinates

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://schemas.ggf.org/jsdl/2005/11/jsdl-srds"
        elementFormDefault="unqualified"
        xmlns:jsdl-srds="http://schemas.ggf.org/jsdl/2005/11/jsdl-srds">
        <!-- COMPLEX TYPES: Definitions for the RangeValueType -->

  <xsd:complexType name="Boundary_Type">
    <xsd:simpleContent>
      <xsd:extension base="xsd:double">
        <xsd:attribute name="exclusiveBound" type="xsd:boolean" use="optional"/>
        <xsd:attribute name="tolerance" type="xsd:double" xmlns="default␣namespace" use="required"/>
          <xsd:anyAttribute namespace="##other" processContents="lax"/>
      </xsd:extension>
     </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="Exact_Type">
    <xsd:simpleContent>
      <xsd:extension base="xsd:double">
        <xsd:attribute name="tolerance" type="xsd:double" xmlns="default␣namespace" use="required"/>
         <xsd:anyAttribute namespace="##other" processContents="lax"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="DoubleValue_Type">
    <xsd:simpleContent>
```

```
      <xsd:extension base="xsd:double">
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="Range_Type">
    <xsd:sequence>
      <xsd:element ref="jsdl-srds:LowerBound"/>
      <xsd:element ref="jsdl-srds:UpperBound"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RangeValue_Type">
    <xsd:sequence>
      <xsd:element ref="jsdl-srds:UpperBoundedRange" minOccurs="0"/>
      <xsd:element ref="jsdl-srds:LowerBoundedRange" minOccurs="0"/>
      <xsd:element ref="jsdl-srds:Exact" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="jsdl-srds:Range" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RangeValueNoDynamic_Type">
    <xsd:sequence>
      <xsd:element ref="jsdl-srds:UpperBoundedRange" minOccurs="0"/>
      <xsd:element ref="jsdl-srds:LowerBoundedRange" minOccurs="0"/>
      <xsd:element ref="jsdl-srds:Exact" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="jsdl-srds:Range" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    </xsd:complexType>

   <xsd:complexType name="VivaldiValue_Type">
      <xsd:sequence>
        <xsd:element ref="jsdl-srds:Coordinatex" minOccurs="1" maxOccurs="1" />
        <xsd:element ref="jsdl-srds:Coordinatey" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="jsdl-srds:Radius" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="mandatory" type="xsd:boolean" xmlns="default␣namespace" use="required"/>
    </xsd:complexType>


  <!--  ==============Tag added by SRDS================== -->

    <xsd:element name="IdlePercentage" type="jsdl-srds:RangeValue_Type"/>
    <xsd:element name="Uptime" type="jsdl-srds:RangeValue_Type"/>
    <!--  ============================================================ -->
        <xsd:element name="IndividualPhysicalMemory" type="jsdl-srds:RangeValue_Type"/>
        <xsd:element name="IndividualVirtualMemory" type="jsdl-srds:RangeValue_Type"/>
        <xsd:element name="IndividualNetworkBandwidth" type="jsdl-srds:RangeValue_Type"/>
        <xsd:element name="IndividualDiskSpace" type="jsdl-srds:RangeValue_Type"/>
        <xsd:element name="Vivaldi" type="jsdl-srds:VivaldiValue_Type"/>
        <xsd:element name="Coordinatex" type="jsdl-srds:DoubleValue_Type"/>
        <xsd:element name="Coordinatey" type="jsdl-srds:DoubleValue_Type"/>
        <xsd:element name="Radius" type="jsdl-srds:DoubleValue_Type"/>

        <xsd:element name="UpperBoundedRange" type="jsdl-srds:Boundary_Type" />
        <xsd:element name="LowerBoundedRange" type="jsdl-srds:Boundary_Type"/>
        <xsd:element name="Exact" type="jsdl-srds:Exact_Type" />
        <xsd:element name="Range" type="jsdl-srds:Range_Type" />
        <xsd:element name="LowerBound" type="jsdl-srds:Boundary_Type"/>
        <xsd:element name="UpperBound" type="jsdl-srds:Boundary_Type"/>


</xsd:schema>
```

An example of JSDL that uses all namespaces

```
<?xml version="1.0" encoding="UTF-8"?>
<!--    this jsdl file contains all dynamic elements with related
        dynamic-epsilon values, further it contains POSIXApplication
        element.
-->
<jsdl:JobDefinition xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
        xmlns:jsdl-posix="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
        xmlns:jsdl-srds="http://schemas.ggf.org/jsdl/2005/11/jsdl-srds"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
        xsd:schemaLocation="http://schemas.ggf.org/jsdl/2005/11/jsdl␣Jsdl_Normative_OGF.xsd">
   <jsdl:JobDescription>
      <jsdl:JobIdentification>
         <jsdl:JobName>My Gnuplot invocation</jsdl:JobName>
         <jsdl:Description> Simple application invocation:
            User wants to run the application 'gnuplot' to
            produce a plotted graphical file based on some data
            shipped in from elsewhere(perhaps as part of a
            workflow). A front-end application will then build
            into an animation of spinning data.
            Front-end application knows URL for data file which
            must be staged-in. Front-end application wants to
            stage in a control file that it specifies directly
            which directs gnuplot to produce the output files.
            In case of error, messages should be produced on
            stderr (also to be staged on completion) and no
            images are to be transferred.
         </jsdl:Description>
      </jsdl:JobIdentification>
      <jsdl:Application>
         <jsdl:ApplicationName>gnuplot</jsdl:ApplicationName>
         <jsdl-posix:POSIXApplication>
            <jsdl-posix:Executable>
               /usr/local/bin/gnuplot
            </jsdl-posix:Executable>
            <jsdl-posix:Argument>control.txt</jsdl-posix:Argument>
            <jsdl-posix:Input>input.dat</jsdl-posix:Input>
            <jsdl-posix:Output>output1.png</jsdl-posix:Output>
         </jsdl-posix:POSIXApplication>
      </jsdl:Application>
      <jsdl:Resources>
         <jsdl-srds:IndividualPhysicalMemory dynamic-epsilon="0.3">
            <jsdl-srds:LowerBoundedRange>2097152.0</jsdl-srds:LowerBoundedRange>
         </jsdl-srds:IndividualPhysicalMemory>
         <jsdl-srds:IndividualVirtualMemory dynamic-epsilon="0.3">
            <jsdl-srds:LowerBoundedRange>2097152.0</jsdl-srds:LowerBoundedRange>
         </jsdl-srds:IndividualVirtualMemory>
         <jsdl-srds:IndividualNetworkBandwidth dynamic-epsilon="0.3">
            <jsdl-srds:LowerBoundedRange>20971.0</jsdl-srds:LowerBoundedRange>
         </jsdl-srds:IndividualNetworkBandwidth>
         <jsdl-srds:IndividualDiskSpace dynamic-epsilon="0.3">
         <jsdl-srds:Range>
            <jsdl-srds:LowerBound>0</jsdl-srds:LowerBound>
            <jsdl-srds:UpperBound>600000</jsdl-srds:UpperBound>
         </jsdl-srds:Range>
         </jsdl-srds:IndividualDiskSpace>
         <jsdl-srds:IdlePercentage dynamic-epsilon="0.3">
            <jsdl-srds:LowerBoundedRange>20971.0</jsdl-srds:LowerBoundedRange>
         </jsdl-srds:IdlePercentage>
          <jsdl-srds:Uptime>
           <jsdl-srds:Exact>10</jsdl-srds:Exact>
          </jsdl-srds:Uptime>
      </jsdl:Resources>
      <jsdl:DataStaging>
         <jsdl:FileName>control.txt</jsdl:FileName>
         <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
         <jsdl:DeleteOnTermination>true</jsdl:DeleteOnTermination>
         <jsdl:Source>
            <jsdl:URI>http://foo.bar.com/~me/control.txt</jsdl:URI>
         </jsdl:Source>
      </jsdl:DataStaging>
      <jsdl:DataStaging>
         <jsdl:FileName>input.dat</jsdl:FileName>
         <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
         <jsdl:DeleteOnTermination>true</jsdl:DeleteOnTermination>
         <jsdl:Source>
            <jsdl:URI>http://foo.bar.com/~me/input.dat</jsdl:URI>
         </jsdl:Source>
      </jsdl:DataStaging>
      <jsdl:DataStaging>
         <jsdl:FileName>output1.png</jsdl:FileName>
         <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
         <jsdl:DeleteOnTermination>true</jsdl:DeleteOnTermination>
         <jsdl:Target>
            <jsdl:URI>rsync://spoolmachine/userdir</jsdl:URI>
         </jsdl:Target>
      </jsdl:DataStaging>
   </jsdl:JobDescription>
</jsdl:JobDefinition>
```

2.4 Neighborhood Query Functionality

The latest release of SRDS provides a simple neighbourhood query functionality. The functionality has been designed using as a reference case the need to store Vivaldi network coordinates of several resources [6] and then query resources within a specified radius from a given coordinate pair. Vivaldi coordinates are pseudo coordinates iteratively computed by a distributed algorithm from a set of network delay measurements among resource pairs. Vivaldi coordinates allow to estimate the degree of network proximity, i.e. estimate the expected communication delay, between arbitrary pairs of nodes. A first concrete application of this approach is the storing of XFS servers and XtreemOS mobile devices with bidimensional Vivaldi coordinates, in order to match nearby file server to mobile resources and allow efficient access to XFS from mobile clients.

Relying on the sharing of a data structure distributed over a DHT, the mechanism shall support concurrent access, on the ground that updating the coordinate-resource relationship will certainly happen over time. We choose to exploit a DHT with transactional properties in order to ensure safe concurrent operation. The Scalaris DHT [21] provides a mechanism for enveloping a list of DHT operations into a transactional session with commit/abort functionality. The basic mechanism has been exploited within the concurrent-safe general query engine described in Section 2.1.

Client Interface There are two principal interfaces that are currently exported via DIXI/XATI.

- *pushingVivaldiCoordinates(Double x, Double y, InetAddress IP)*

  It is used to push machine's vivaldi coordinates into the DHT. We assume for now that these are two-dimensional coordinates with no negative values. The approach, as described below, is adopted just for the sake of simplifying the first implementation.

- *searchVivaldiNeighbors(Double originX, Double originY, Double rad, int wantedResult)*

  It is used to retrieve a set of at least *wantedResult* machines that reside in the area defined by the origin coordinates and the radius.

Implementation Current implementation assumes that the coordinate space can be divided into square blocks. To store the data concerning to a resource we will use a few namespaces. In the following we will simply refer to the data with the name of IP, which is the distinguishing trait of a resource, and we will use additional namespaces as indexes to speed up neighbourhood searches. The size of the squares used to partition the Vivaldi coordinate space is flexible.

The namespaces used in the implementation are stored within the Scalaris DHT, so where needed all the operation within a namespace is enclosed in a transaction, letting e.g. each get/update pair an atomic operation. The current implementation does not actually need more than that, although it would also be possible to support transactions over multiple namespace operations, as long as they are mapped to the same Scalaris instance.

We use the scalaris DHT with three different namespaces one for X coordinates, one for Y coordinates and another one for the IPs. Basically we discretize the X and Y values in set of delimited values X_1,X_2,...,X_n Y_1,Y_2,...,Y_n every Y_i and X_i representing a primary key in the associated namespace.

When choosing how to group data into data structures within the DHT, we avoided excessive fragmentation and split the data only along a single dimension. Each X_i key thus manages a set of values (x_j, y, IP) such that $X_i \leq \ x_j < X_{i+1}$, and the Y_i keys behave correspondingly. We have traded some selectivity in the search for a lower number of DHT accesses and transactions. The two X and Y namespaces are used only to optimize the search direction, allowing to reduce the search into a single namespace and then filter the results locally.

Of course, supporting a larger amount of data requires moving to a namespace where the (X_i, Y_i) pair is the key, and the search procedure that we describe in the following is actually performed along both axes.

When a new pushing request arrives the mechanism make an approximation of X,Y value and store it into the DHT. We assume each IP record is updated only by one entity (tipically, the resource itself), thus simplifying transaction management in Algorithm 1.

Algorithm 1 Coordinate Pushing Algorithm

```
1: if namespaceIP.get($IP$)$\neq \emptyset$ then
2:     dht.remove($IP$)
3: end if
4: namespaceIP.put($IP$,touple($X$,$Y$,$IP$))
5: namespaceX.update(square($X$), touple($X$,$Y$,$IP$))
6: namespaceY.update(square($Y$), touple($X$,$Y$,$IP$))
```

Clearly, two different machines may request to update the same X or Y auxiliary record, so the *update* operation in these namespaces are made atomic by wrapping each one inside a transactional session. No cross-namespace transaction handling is needed, as the values being updated in each X and Y auxiliary record are strictly disjoint for different resources, by the assumptions we made before, and thus secondary namespace updates do not need to be totally ordered

Algorithm 2 Retrieving algorithm

```
1: function searchVivaldiNeighbors
   (double $X_0$, double $Y_0$, double $r$, int howmanyres) {
2: Array results ← $\emptyset$
3: $X_{min} \leftarrow \text{getstrip}(X_0 - r)$
4: $X_{max} \leftarrow \text{getstrip}(X_0 + r)$
5: $X \leftarrow \text{getstrip}(X_o)$
6: $i \leftarrow 0$
7: explore($X$,results)
8: while $X - i \geq x_{min}$ && $X + i \leq x_{max}$ && results.size() $\leq$ howmanyres do
9:    explore($X - i$)
10:   explore($X + i$)
11:   $i + +$
12: end while
13: }

14: function explore(Int $X$, Array results) {
15: $res_{stripe}$ ← dht.getstripeeres($X$)
16: //function Intersect return results from stripe that intersect the circle
17: results.add(intersect($res_{stripe}$)
18: }
```

to ensure consistency (although, of course, a slow update on the auxiliary namespace may affect query results which concurrenlty explore the corresponding part of the coordinate space).

Whenever a new neighbourhood request arrives, the algorithm computes the circle centered at the point $P = (originX, originY)$ having radius *rad*. It then searches for matching IP records within the given circle area by exploring one of the two auxiliary spaces (in our case, the X namespace, but of course Y may be chosen if it leads to a lower number of operations).

The algorithm starts to seek for results from the stripe containing P and continues outward until all the stripes intersecting the circle have been explored, or the desired number of results has been found.

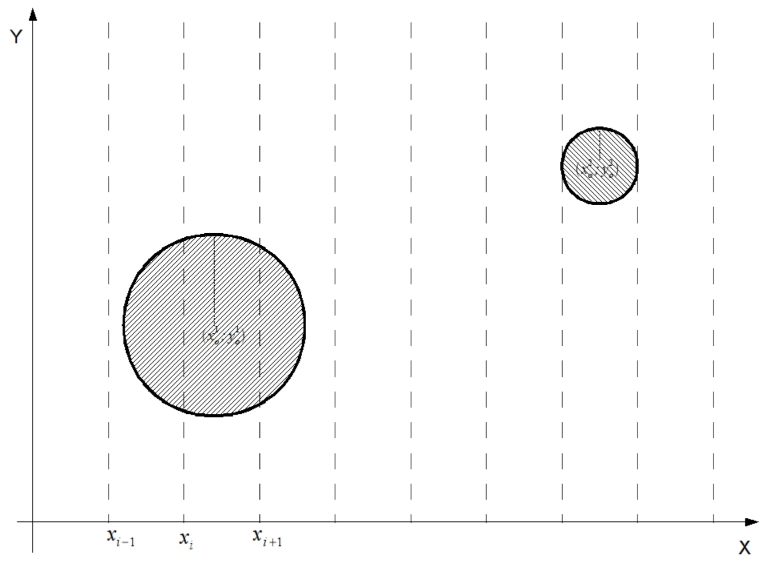


Figure 6: X-defined stripes and intersection with targeted neighbourhood

2.5 SRDS Performance Tests

This section describes the experiments performed to evaluate the scalability and the performance of the SRDS component. In particular we have tested the latency of two SRDS services whose implementations has been modified, that is the service exploited by Job Directory Service (JDS) and the service exploited by Application Execution Manager (AEM). In order to test the AEM link, we submit a simple JSDL query to simulate the request issued for a job execution. In the JDS test we focus on a number of operations to manage insertion, modification

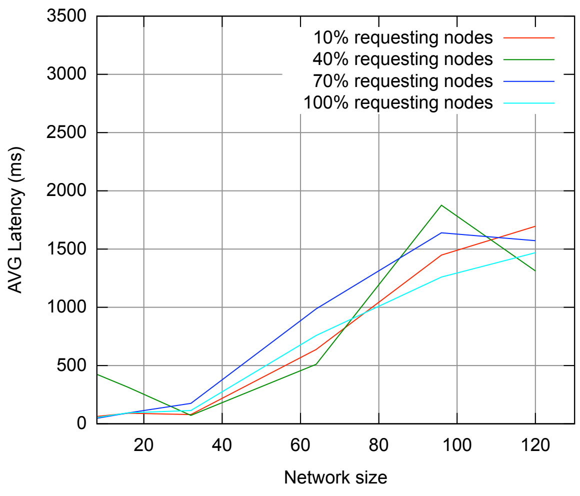


Figure 7: Average latency of AEM operations with increasing overlay size and varying percentage load.

and deletion of a job from the distribute Job Directory. Unless otherwise indicated, the overlay networks active during the tests of the SRDS are the RSS and the Overlay Weaver.

SRDS has been tested as a stand-alone component. The requests were forwarded to the node via the RMI interface we use for our everyday test, and not by the DIXI framework as in the complete XtreemOS platform. In the AEM test, since we measure latencies only after the request has been already received by the SRDS, the measurements are equivalent than using DIXI, but using RMI give us an additional flexibility to perform more exhaustive tests. The latency measures for JDS actually contains the RMI overhead which is actually negligible (about 10 milliseconds).

2.5.1 AEM performance tests

The AEM performance in locating resources on a large network according to a JSDL query has been tested by measuring the operation latency on a subset of the Grid5000/Aladdin computing platform[2]. We measured the performance of the SRDS prototype, with the OW and the RSS overlays active, scaling up the network size from 8 to 120 nodes. For these tests we used the RMI interfaces

[2]Experiments presented in this section were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

of SRDS, and we measured the overhead with the SRDS subtracting the RSS query resolution time. The SRDS version used in the test is the "production" one already distributed with the XtreemOS public release 2, not including latest additions which will be released later on, e.g. the REMED algorithm to speed up dynamic attribute matching [3].

As the testbed used cannot be allocated to this kind of XtreemOS testing for more than minutes, and the node reservations usually end up each one with a different set of machine belonging to several clusters of Grid5000, we had to specifically take care of overlay stabilization.

The test was structured as a double experiment, with two straws of requests coming from a subset of the nodes, separated by about a minute. This allows the overlay networks to stabilize thanks to the first straw and the pause, before we measure the actual system behaviour.

In the test, which is designed to evidence the limits of the system, all nodes provide their own information and a variable percentage of them also take part in the request activity. The percentage of nodes which perform queries varies from 10% to 100% of the testbed. The queries were crafted to avoid degeneration (i.e. results which are always empty, or always hit the whole platform), with found results limited to a small number (up to 16 in the largest overlay configuration) in order to reproduce a typical query traffic.

The small tests (up to 64 machines) where almost always on a single cluster, while larger testbeds where using machines from physically separate clusters.

As we can see from figure 7, the average latency of the resource discovery increases with the size of the platform, due both to the logarithmic complexity of the DHT routing, and to the linear increase (w.r.t. the size of the answer) related to checking dynamic attribute constraints in current implementation. The second cost term is going to be reduced by the adoption of more sophisticated techniques like REMED [3].

We also underline that several glitches in the network, in the form of sporadic very high query processing times, were observed during the experiment. While the average service time is good, we have a population of very fast latencies, well below 200ms, and another population of high latencies which offset the average and are often due to network timeouts during the DHT and RSS routing. Figure 8 show the standard deviation of the latencies previously shown.

This is most likely due to the high network load of the testbed, where the portion of resources which are not used in our tests are usually allocated to HPC job with high computation and communication demand. This interpretation is supported by the observation that using larger platforms (i.e. reserving almost all the machines in the clusters we are using as testbed) actually seems to reduce the average and the standard deviation of the measured latencies.

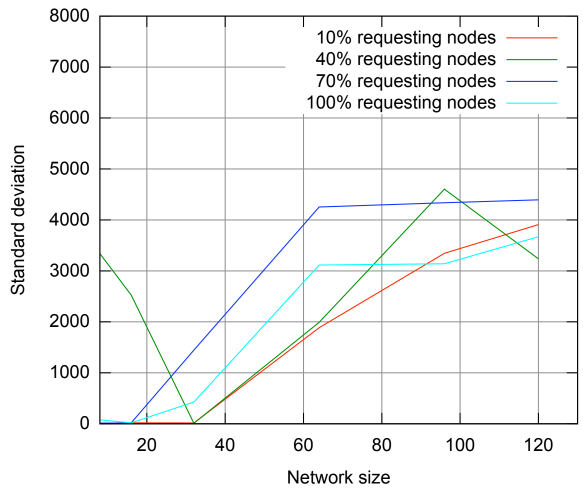


Figure 8: Standard deviation of the latency of AEM operations with increasing overlay size and varying percentage load.

Table 4: JDS operations with the cost in terms on DHT accesses

Function	Cost in term of DHT primitives
AddJob	a number of put/get
RemoveJob	a number of put/get and a single remove
{Remove, Add, Update}Attribute	a single DHT get and put
GetJob, GetAttribute	a single DHT get

2.5.2 JDS Performance

To test the scalability of the SRDS we measured the latency of a number of JDS operations. Table 4 shows the complete list of measured operations with the relative cost in term of primitive message over the DHT overlay. The complexity of the operations is variable, but is rather homogeneous, e.g. the *GetJob* is a single DHT get, while the *AddJob* requires a sequence of put/get primitives. Here, "a number" of put and gets depends on the number of *strict* attributes as defined in Section 2.1, and it is roughly $2k + 1$ put/get operations if k is the number of strict attributes.

The tests have been repeated with different network sizes (8, 16, 32, 64 and 128 nodes). All the nodes belong to the *bordeaux* site of Grid5000, except for those tests with 64 and 128 nodes, for which we used nodes from three different clusters (bordereau, bordemer, bordeplage).

The test has been structured in the following way. First, all the nodes start SRDS and wait 60 seconds in order to make the RSS and Overlay Weaver stabilize. The resource monitor is kept active during the whole test, so all nodes perform a put primitive with the locally measured data at a fixed rate (every 20 seconds).

After this first stabilization phase, we launch two test runs. Each test consists in three request for each JDS operations, separated by a time interval of 200 milliseconds. The latencies were computed considering only the second straw of requests, and averaging all results. The measurements also take in account the delay of the RMI interface used to deliver requests to single nodes.

The results, presented in Figure 9, show that latencies scale with an increasing number of nodes in the network. Furthermore it is also clearly visible that JDS operations with similar requirements in terms of DHT primitives show a similar behaviour among themselves.

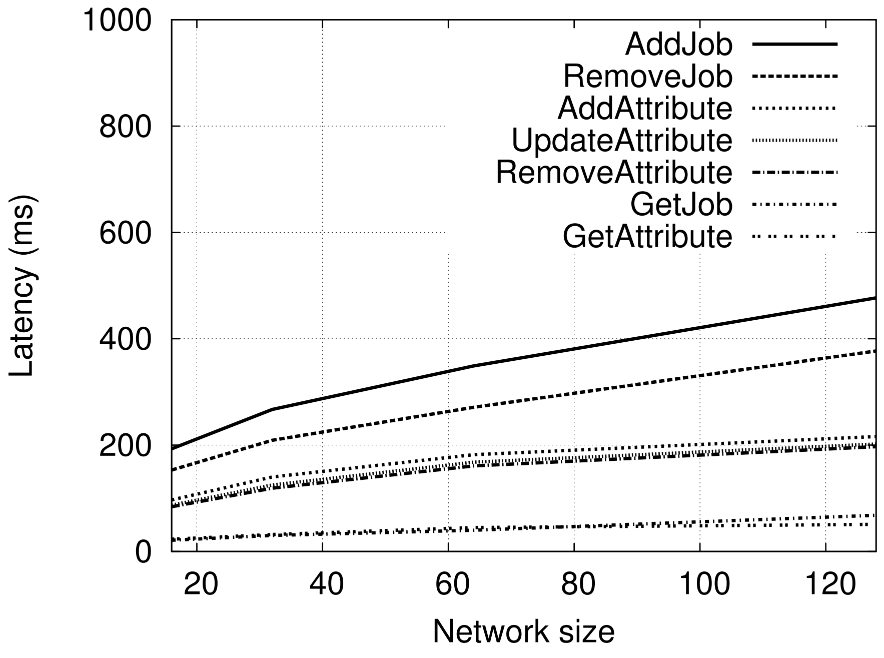


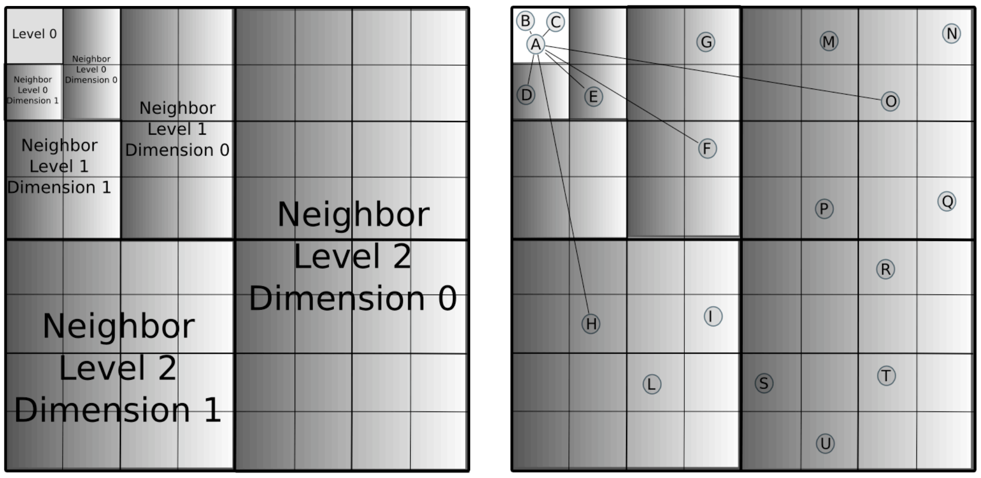
Figure 9: Latency of JDS operations with increasing overlay size.

2.6 Conclusions

In this first part of the deliverable we have shown the improvements and changes of the SRDS addressing network fault tolerance, extended configurability, and stronger checking of system interaction, which were designed in order to strengthen the integration within the XtreemOS system, and make XtreemOS itself more robust. Besides, we described an integrated overlay monitoring interface, improving system management.

The XML validation mechanism put in place within the SRDS will be used to enforce strict checking of the extensions to the JSDL standard which are being developed within the project. Those extensions directly concerning the resource discovery mechanisms were also documented in this section.

Finally, test results of the latency of AEM and JDS operations had been analyzed, leading to the conclusion that the SRDS is a scalable system for the XtreemOS purposes, and that integrating improved algorithms for dealing with dynamically-updated values can improve the overall SRDS performance when large resource requests are issued.


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

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

Figure 10: Sample 2-dimensional attribute space divided into cells.

3 RSS

In the past year, the work on the Resource Selection Service has focused on improvements targeted at making the administration of RSS easier.

As described in Deliverables D3.2.4 and D3.2.8 and [5], the RSS nodes build a P2P overlay using a multi-dimensional virtual coordinate space. Each node is mapped onto this virtual space based on its static attribute values and is represented as a point. The virtual space is divided into a set of *cells* using a recursive algorithm shown in Figure 10. Specifically, the space is partitioned along each dimension (similar to a mesh) using a set of cell *boundaries*. As shown in Figure 10(b), to allow query routing, each node maintains a number of links to other nodes located in specific other cells of the overlay. In addition, each node is supposed to maintain a full list of other nodes located in the same lowest-level cell. We call these other neighbors the *order-zero neighbors*.

The work on RSS management is organized into two parts. First, we need to allow system administrators to control the RSS better, especially to maintain the set of attributes handled by the RSS. This means, we must be able to dynamically add or remove dimensions to/from the RSS overlay, and update the overlay links to allow queries to be routed in the new overlay. Second, the RSS needs a number of internal configuration parameters such as the boundaries between cells and the nesting level. We developed mechanisms to allow the RSS to tune these parameters automatically through the use of self-managing algorithms.

3.1 RSS manual management

There are several situations in which the grid administrators may need to modify the set of attributes that describe the computing resources. For example, a new attribute has to be added if the users need a specific library and the resource selection service has to distinguish between the nodes that have the library and those that don't (or even more, to distinguish between different versions of the library). In case a certain library or feature is not needed anymore, we need to be able to remove the corresponding attribute from the computing nodes. Another possible case is the one in which the method to compute the attribute values has to be modified. For example, there could be changes in the measurement unit for an attribute, or in the system files from where a value is read, or in a benchmarking method (if the value represents a benchmark score).

It would not feasible to shutdown and restart the resource selection service in the whole grid when we need to apply such changes - especially since it is practically impossible to synchronize all the machines to start with the new version in the same time. This would also introduce a quite significant downtime in which the users' requests cannot be processed. Thus, we need to add dynamic adaptation support in the RSS in order to allow the runtime modification of the attribute set. In essence this means adding and removing dimensions in the multi-dimensional space used to model the RSS overlay. Another issue that needs to be considered in this context is that during the reconfiguration we must support co-existing nodes that have different attribute sets.

We have designed a protocol for modifying dynamically the dimension sets and tested it through simulation, using PeerSim. In the remainder of this section we shall describe the protocol and the simulation results.

3.1.1 Requirements for updating the dimension set

In XtreemOS, we assume that the changes in the dimension set will be introduced only by the VO administrator; that is, there will be a single node from the overlay that will inject these changes into the system.

The Resource Selection Service will support three types of dimension set updates:

- **adding a dimension:** this requires the administrator to specify the name of the new attribute, and a method to calculate its value;
- **removing a dimension:** the administrator has to specify the name of the dimension to be removed;
- **updating a dimension:** this refers to changing the method by which an attribute value is computed; the administrator will provide the new value.

In our implementation, the **update** operation is represented as an **add** operation, that specifies the name of the dimension to be updated and the new description of the dimension.

In order to support updates of the dimension set, there are two issues that the RSS needs to address: propagating the update to all the nodes in a short time, and handling the user queries while the update is being done. The following two subsections explain how we addressed these issues.

3.1.2 The update protocol

The problem of propagating the dimension set updates has similarities with the problem of propagating updates in a distributed data store. This problem has been extensively studied [20], and the two most popular approaches to it are:

- propagating the state (i.e., the whole set of data that has been updated)
- propagating only the update operations

The second method is preferred in most of the cases, because it incurs a lower overhead, and also because it allows for the merging of write operations done by multiple processes.

However, in our case the first method is more convenient because the problem is simpler than the general case of updating a distributed data store. Specifically, here are the significant aspects that differ from the general case:

- the data set (i.e., the set of dimensions) is small, and transmitting the whole set from one node to another does not bring a significant overhead
- there is a single administrator that injects changes into the system, so we do not have to merge modifications that come from different sources

The new dimension sets are disseminated through a push-pull gossip protocol, which allows for a rapid propagation (exponential rate) and has the advantage of a low overhead. Specifically, we use the same Cyclon protocol [25] that stays at the base of the RSS overlay. In order to minimize the communication overhead, the nodes exchange the full information only for the dimensions that actually need to be updated.

The administrator associates each update with a timestamp; also, each dimension is associated with a timestamp that represents the time of the last modification done to that dimension (or the time when the dimension was introduced, if there are no modifications). Thus, we define the signature of a dimension set as a set of pairs ($dimension\_name_i$, $timestamp_i$) and a of the timestamp for the whole set.

```
UpdateDimensionSet($node$):
$neighbor \leftarrow$ select random neighbour
send (GossipRequest: my_timestamp) to $neighbor$
receive (GossipResponse: neighbor_timestamp) from $neighbor$
if $my\_timestamp < neighbor\_timestamp$ then
    receive(dimension_set_signature) from $neighbor$
    $dimensions\_list \leftarrow$ determine set of dimensions that need updates
    send (DimensionsRequest: $dimensions\_list$) to $neighbor$
    receive (DimensionsResponse: $updated\_dimensions$) from $neighbor$
    update dimension set
else
    send(dimension_set_signature) to $neighbor$
    receive (DimensionsRequest: $dimensions\_list$) from $neighbor$
    send (DimensionsResponse: $updated\_dimensions$) to $neighbor$
end if
```

Figure 11: Pseudocode for a gossip exchange during which a node updates its dimension set.

Each node periodically gossips with one of its Cyclon neighbors in order to update the dimension set. We are using a push-pull gossip model, meaning that whichever of the two nodes has an older dimension set, it will obtain the update from the other one. Specifically, when two nodes gossip, they first exchange the timestamps of their dimension sets. If these are different, the node with the older version of the set will get the signature for the other node's dimension set. Then, it will compare the newer signature with its own and will request from the other node the full information only for the dimensions that were actually modified. The pseudocode for the gossip exchange is shown in Figure 11.

3.1.3 Query handling during updates

Although the system converges very fast to recreate internal overlay links after making a dimension set update, there is a time interval during which some nodes have different dimension sets than the others. The main difficulty in handling queries during these intervals is that some nodes have incomplete neighbor sets: either because they have just switched to the new dimension set and have not found enough neighbors yet, or because they still have the old dimension set and their former neighbors have already switched to the new one.

There are two possible directions for approaching this problem:

1. Maximizing the delivery rate: the delivery rate can be kept almost unmodified if each node will simultaneously keep separate lists of neighbors, ac-

cording to the new and the old dimension sets. Unfortunately this approach incurs a higher overhead and also introduces new issues (e.g., for how long to keep the old list, or how to decide which neighbors list to use for routing queries).

2. Minimizing the overhead: another possibility is to delay the queries received during a system reconfiguration until the overlay becomes stable again (or even to reject queries during reconfiguration). This is a simpler solution and does not introduce the overhead of storing and managing additional neighbors list. However, it would affect the availability of the system, especially in the case of two or more subsequent reconfigurations.

We opted for an approach that represents a compromise between these two directions, in terms of overhead and availability. Specifically, the current version of the protocol can handle queries during reconfiguration, but only routes them among neighbors that have the same version of the dimension set. The query will be interpreted according to the dimension set of the node that initially introduced it. The nodes only keep neighbors lists for the current dimension set, and thus the query will only be routed through nodes that have the same set as its initiator node.

The solution that we have chosen is simple to implement and does not add any overhead to the base protocol. However, it causes a temporary drop in the query delivery rate during reconfiguration. We performed a set of simulation experiments in order to evaluate this effect and to estimate whether the delivery rate decrease remains at an acceptable level. We also proposed an improvement in the RSS base protocol, in order to reduce the time interval during which the query delivery rate drops. The simulation results and the protocol improvement are presented in the following two sections.

3.1.4 Simulation results

The first set of simulation experiments that we have performed aimed to evaluate the speed at which the dimension set updates are propagated through the system. Our experiments showed that the updates are spread at an exponential rate among the nodes, which is consistent with the theoretical results proved by the existing litterature on push-pull gossiping [15].

Figure 12 presents the results that we have obtained for 100,000 nodes, injecting three consecutive dimensions set changes into the system. On the y axis we represented the total number of nodes that have received an update (each curve corresponding to one of the three updates). The x axis represents the time, measured in gossip cycles. A gossip cycle is the time interval during which a node

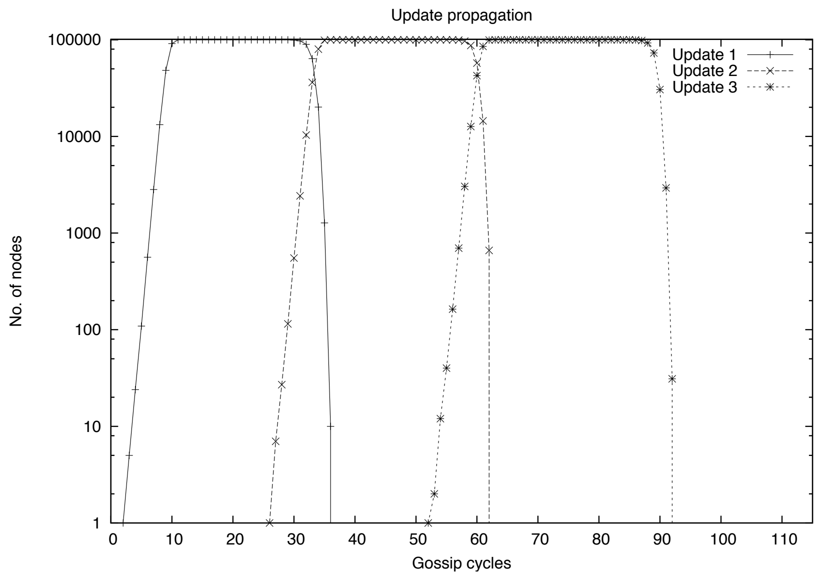


Figure 12: Propagation of the dimension set updates among nodes.

initiates an information exchange (gossips) with one of its neighbors. All the nodes gossip periodically with the same frequency, and usually the length of a gossip cycle is of a few seconds.

After estimating the speed for spreading updates in the system, we also performed simulation tests to evaluate the ability of the RSS to respond to queries during reconfiguration. We simulated a set of reconfiguration operations (adding or removing dimensions), while continuously launching queries into the system. These tests were made with 1000 simulated nodes, each having a gossip cache size of 30 and a gossip length of 5; the gossip length represents the number of items that two nodes exchange when gossiping. Figure 13 shows the delivery rate for an experiment in which a dimension was added to a dimension set with 2 initial dimensions; the reconfiguration was done at 90 gossip cycles from the system startup. Figure 14 shows the delivery rate for an experiment in which we initiated two reconfiguration operations: removing a dimension and then adding a new one (at 80 and 160 cycles from the system startup).

From the results shown in these figures, and also from other similar simulation experiments, we observe that:

- The RSS overlay needs some time to converge, as the nodes need to build their lists of neighbors. During this time, the service will only return partial respones to queries, as it is not able to find all the nodes from the network that match the given criteria. We observed that it takes approximatively 30 gossip cycles until the query delivery rate raises above 80%. However we

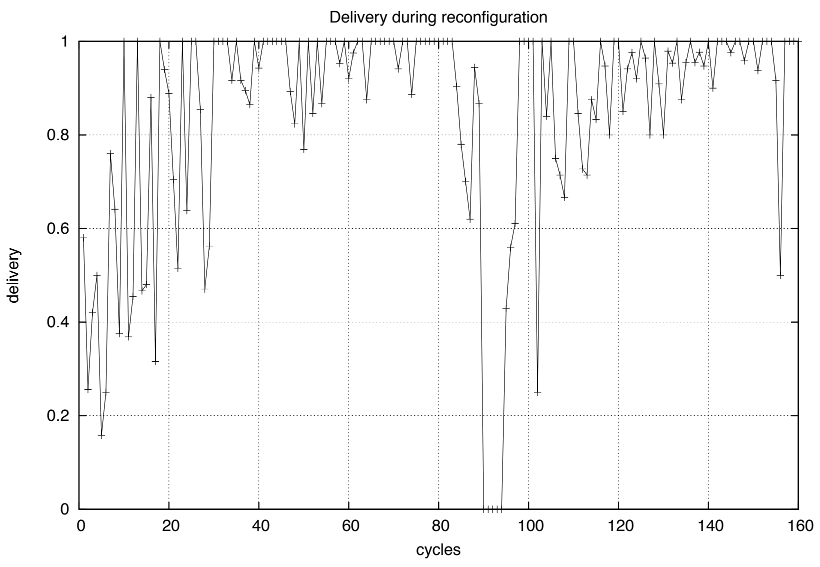


Figure 13: Query delivery rate when the RSS is reconfigured once.

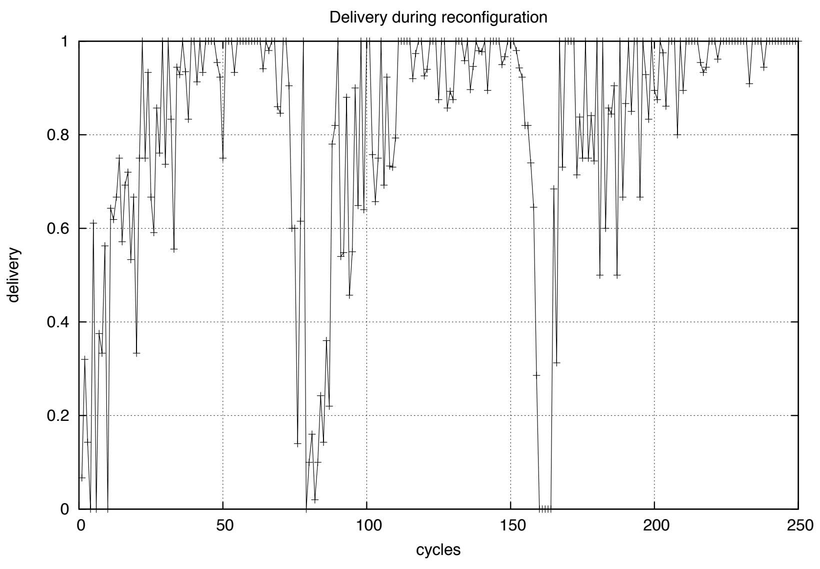


Figure 14: Query delivery rate when the RSS is reconfigured twice.

note that in the experiments we performed, the system was configured to search for *all* the nodes from the system that match the given criteria. In real-life situations this happens only rarely, as most of the queries request only a limited number of nodes; for this type of queries, the RSS is able to provide 100% delivery rate within a very short time after startup.

- After a reconfiguration of the dimension set, the overlay also needs time to rebuild itself. Although the nodes reuse items from their previous lists of neighbors when they are creating the new lists, the structure of the multidimensional space is changed and the nodes will have to find new neighbors that they weren't previously connected to. For this reason, the number of cycles needed to rebuild the overlay after a reconfiguration is similar with the number of cycles needed to build the overlay at startup.

3.1.5 Convergence optimizations

As we have seen in the previous section, after a reconfiguration the RSS overlay needs several gossip cycles to rebuild itself, and the number of cycles is of the same order of magnitude with the number of cycles needed to build the overlay after a system startup – as these two processes are very similar.

The time needed by the overlay to converge at system startup usually does not represent a significant problem, but as we introduced the possibility to reconfigure the system this issue becomes more important. As the overlay must be rebuilt at each system reconfiguration, we need to ensure a low convergence time. For this reason, we improved the original RSS protocol in order to reduce the convergence time; this will have an impact both on system startup and on reconfiguration.

Additional simulations (not discussed here in details) show that the delay in the overlay's convergence is mainly due to the longer time needed by the nodes to find the full list of other nodes located in their lowest-level cell (called the zero-order neighbors). This is caused by the fact that the Vicinity protocol used by RSS is designed to build neighbors lists that cover the multidimensional space as much as possible (i.e., the protocol running on a node aims to find at least one neighbor for each level and each dimension). While the protocol succeeds in rapidly finding sets of neighbors that cover the space, it is less efficient in regard to zero-level neighbors. This is because the query routing algorithm has different requirements for the connectivity in the zero-level cell. For levels greater than 0, the algorithm needs only one neighbor for each level and dimension. However in a zero-level cell the queries are broadcasted and the algorithm requires the nodes to form a connected graph in order to achieve 100% delivery rate.

Based on these observations and on the simulation results that show a slower convergence for the number of zero-level neighbors, we decided to add to the

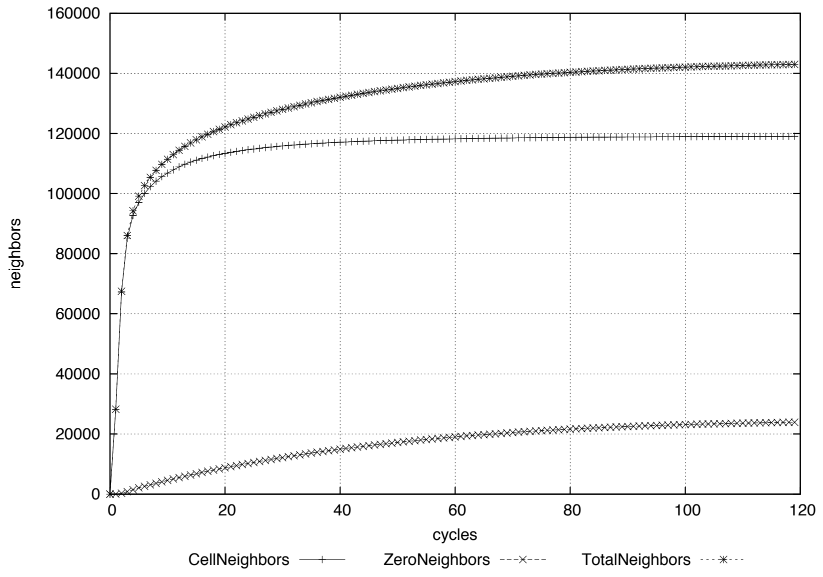


Figure 15: Total number of neighbors discovered by the nodes with the original RSS protocol.

RSS a new Vicinity-based protocol layer that is dedicated to finding zero-level neighbors. This protocol aims to bring into a node's gossip cache other nodes that are as close as possible in the multidimensional space, giving preference to nodes that are in the same zero-level cell.

In order to evaluate the improvements brought by the new protocol, we have performed some simulation tests with various system sizes. We present as follows the reconfiguration performance of a system with 10,000 nodes, in a 6-dimensions space.

Figure 15 shows the total number of neighbors discovered in time by the nodes with the original version of RSS, using a gossip cache of 100 items and a gossip lengthe of 50 items. These values are greater than the typical values used in the RSS (gossip cache 30 and gossip length 5), as we intended to evaluate whether a larger cache size and gossip length can improve the convergence. Although we obtained a better convergence with these values, the improvement was still not satisfactory: it took more than 80 gossip cycles to discover a number of zero-level neighbors that is close to the final value. Figure 16 shows the total number of neighbors discovered with the new Vicinity protocol added, using a cache size of 30 and a gossip length of 15. We can observe that the number of zero-level neighbors converges in less than 20 cycles, which is a significant improvement compared to the original version of the RSS.

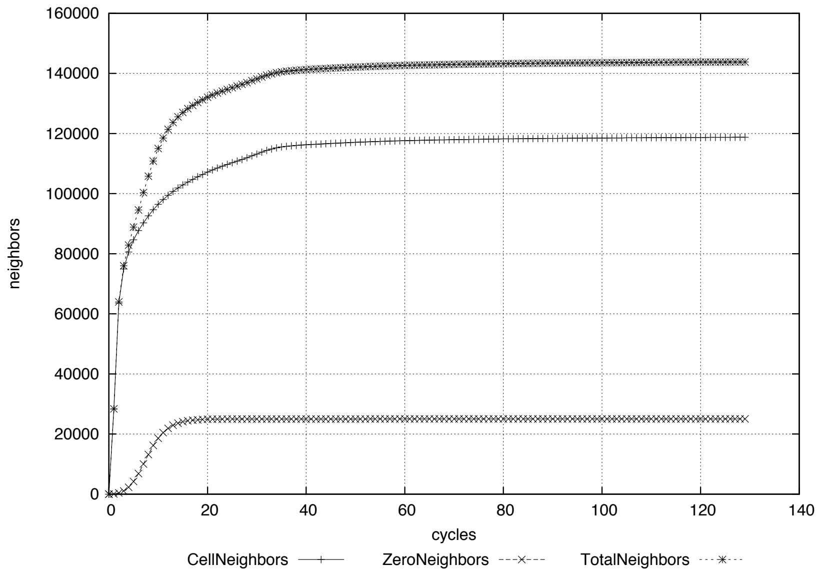


Figure 16: Total number of neighbors discovered by the nodes with the improved RSS protocol.

Although adding a new gossip protocol to the RSS increases the communication and the storage overhead, this does not have a significant impact since the RSS's overhead can be considered as negligible [5].

Figures and show the query delivery rate in the two experiments, and the improvement brought by the new protocol is again clearly visible.

In the near future we intend to perform more simulation experiments to evaluate the behavior of the new protocol during system reconfiguration, and integrate these improvements in the RSS implementation.

3.2 RSS Self-Management

The division of the RSS space into cells has a very strong impact on the overall RSS performance. In particular, the layout of cells strongly affects the structure of peer connections in the RSS overlay and determines the efficiency of query routing. For example, the RSS suffers a suboptimal performance if some of the cells contain a disproportionate number of nodes, since the search algorithm within level-zero cells is based on flooding. On the other hand, the RSS is very efficient at handling queries if nodes are evenly balanced between all cells in the system. Furthermore, the query routing overhead is reduced if query ranges overlap with cell boundaries.

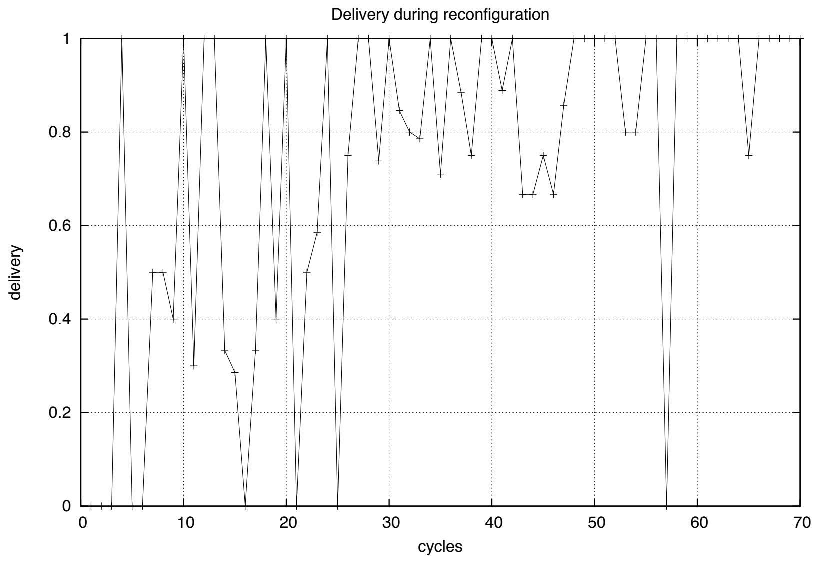


Figure 17: Delivery rate at startup with the original RSS protocol.

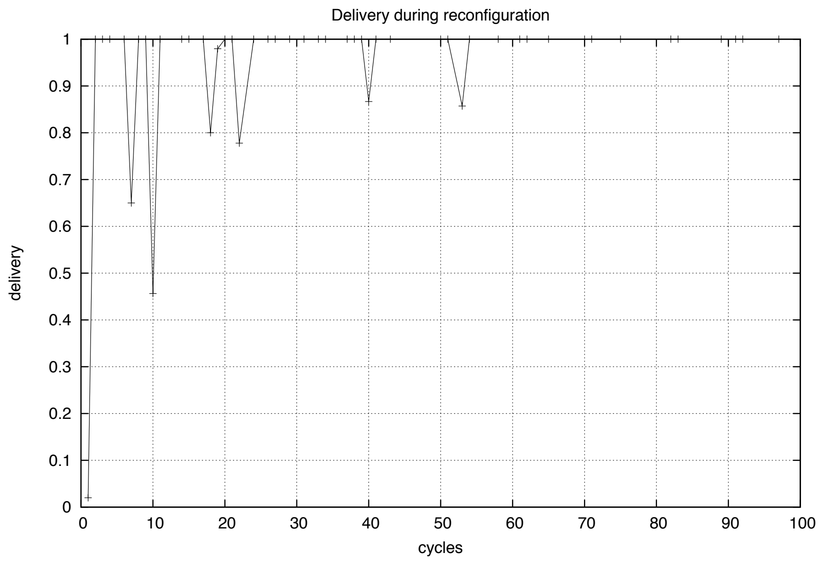


Figure 18: Delivery rate at startup with the improved RSS protocol.

In the current RSS implementation, cell boundaries for each dimension are manually defined in the configuration file by the system administrator. This has a number of drawbacks. First of all, the administrator must have a global knowledge about node attribute values and query ranges in order to define optimum cell boundaries. This may be very challenging if custom dimensions are introduced to the RSS, for example corresponding to software versions or some self-measured metrics. Moreover, since the population of nodes in the RSS is dynamic and dimensions are added and removed at runtime, the administrator must continuously update the cell boundary definitions to maintain an efficient system configuration.

These shortcomings can be addressed by adding a new RSS component that will automatically calculate and adjust cell boundaries at runtime. In principle, such a component should work autonomously and should not require any manual input from the administrators. To reach this goal, the RSS nodes should monitor the *distribution* of node attribute values per each dimension and the distribution of query ranges per each dimension. Using these distributions, a privileged node (such as a core node) can then calculate cell boundaries that evenly balance nodes between cells and overlap with the most common query ranges. The reconfiguration mechanism described in Section 3.1 can be used to spread new boundary definitions from the priviledged node to all nodes in the system.

The following sections describe our protocol for RSS nodes that approximates system-wide distributions of node attribute values.

3.2.1 System Model

The goal of our algorithm is to estimate, in a scalable fashion, the cumulative distribution of an attribute A at every node in the system. Nodes are called *peers* because we assume no central point of control and all nodes participate equally in the algorithm. We assume the peers are organised in a P2P overlay where each peer maintains links to a small number of other nodes in the systems called its *neighbours*. The set of neighbours of a peer changes over time, as peers exchange neighbour lists to obtain robust connectivity [11, 25].

The cumulative distribution function (CDF) for an attribute A is defined as a function $F : \mathbb{R} \rightarrow \mathbb{R}$ such that $F(x)$ is equal to the fraction of nodes that have a value for A at or below x

$$F(x) = \frac{1}{N}\Big|\{p : A(p) \leq x\}\Big|$$

where N denotes the system size.

In order to approximate F, we estimate the function at a set of discrete points, keeping the results in a data structure that is similar to a cumulative histogram. Specifically, we define a sequence of λ elements, called H, where the i'th element,

$H(i)$, contains a pair (t_i, f_i) representing the fraction of peers f_i that have a value for A at or below the *threshold* t_i

$$f_i = \frac{1}{N}\Big|\{p : A(p) \le t_i\}\Big|$$

The thresholds can be chosen arbitrarily within the attribute domain. Each element corresponds to a single CDF value, since $F(t_i) = f_i$ for $i \in [0, \lambda)$. Hence, the CDF function can be approximated by interpolating the points of H. We discuss in the next section how to efficiently and accurately obtain the value of F at the points in H.

We measure the CDF approximation accuracy using two classical metrics. The Kolmogorov-Smirnoff (or maximum error) metric defines the distance between function F and its approximation F_p at node p as

$$\sup_x |F(x) - F_p(x)|$$

Given that the attribute space in our system is discrete, we define the error of F_p as

$$\text{Err}_m(p) = \max_{min \le i \le max} |F(i) - F_p(i)|$$

Since different peers in the overlay can generate slightly different distribution estimations, we calculate the corresponding aggregate of these metrics over all peers

$$Err_m = \max_{1 \le p \le N} \text{Err}_m(p)$$

This error metric provides an upper bound on the approximation error of any peer in the system.

While the maximum error metric is useful to bound the error that any peer observes, this bound is determined by a single point discrepancy between F and F_p. Hence, it is quite sensitive to noise. A common approach to summarise the error contributed by all points calculates the area between the two curves

$$\int_x |F(x) - F_p(x)|\, dx$$

In the discrete case, this metric corresponds to a sum of $|F(x) - F_p(x)|$ over all attribute values. We use the average vertical distance between F and F_p to create a comparable error measure to $\text{Err}_m(p)$

$$\text{Err}_a(p) = \sum_{x=min}^{max} \frac{|F(x) - F_p(x)|}{max - min}$$

Again, we calculate an aggregate of these metrics across all peers

$$Err_a = \operatorname*{avg}_{1 \le p \le N} \text{Err}_a(p)$$

3.2.2 Calculating Cell Boundaries

In order to calculate cell boundaries, RSS nodes need to estimate attribute CDFs for each dimension. The number of cells per dimension is equal to 2^l, where l is the nesting level, i.e., the number of recursive divisions of the attribute space. Given an attribute CDF F for dimension d, cell boundary i is defined as value $Bound(d, i)$ such that

$$F(Bound(d, i)) = \frac{i}{2^l}$$

where $1 < i < 2^l$. This way, the boundaries divide the attribute space into 2^l cells with equal numbers of attribute values.

3.2.3 CDF Approximation Algorithm

The CDF approximation algorithm is based on periodic gossip rounds, executed at the same rate by all nodes, where neighbouring nodes exchange information. A sequence of several gossip rounds, called an *aggregation instance,* generates a new CDF approximation at all nodes in the system. Nodes occasionally initiate new aggregation instances in order to improve the CDF estimation accuracy and to handle system churn.

Each aggregation instance is started by a probabilistically chosen node, which selects a set of t_i thresholds and epidemically spreads the information about the new instance and the thresholds to other nodes using periodic gossip. Next, the nodes run an averaging protocol [13] which estimates the corresponding CDF values. In order to calculate the fraction f_i of nodes that have attribute values below (or at) t_i, a peer p enters the averaging protocol with a value of 1 if $A(p) \leq t_i$, and 0 otherwise. Through a sequence of gossip exchanges, the nodes estimate the mean of all the introduced values, which is by definition is equal to f_i. The estimation accuracy increases exponentially with time, and after a fixed number of rounds, all nodes update their CDF estimations and terminate the aggregation instance.

Figure 19 shows the pseudocode for our algorithm.

Starting an Aggregation Instance We associate each aggregation instance with a unique instance identifier *id*. The instances may overlap in time, and thus a peer may participate in multiple independent instances simultaneously. Since the instances are executed in isolation from each other, we simplify the algorithm description and assume only one running aggregation instance.

Any peer in the system may start a new aggregation instance. To prevent the system from being overwhelmed by new instances, a peer should start a new instance with probability P_s per round calculated as $\frac{1}{N_p R}$. The value of N_p is the

```
// Executed by a probabilistically selected node at the beginning of an instance
StartInstance($p$):
$\{t_i\} \leftarrow$ select $\lambda$ interpolation points
$H_p \leftarrow \{(t_i, f_i) \mid f_i = 1 \text{ iff } A(p) \leq t_i\}$

// Run by each node in each round
Round($p$):
$q \leftarrow$ select random neighbour
send ($Req$, $H_p$) to $q$
receive ($Resp$, $H_q$) from $q$
Merge($H_q$)
while round has not finished do
    receive ($Req$, $H_n$) from $n$
    send ($Resp$, $H_n$) to $n$
    Merge($H_n$)
end while

Merge($H_q$):
if $H_q \neq \emptyset$ then
    let $H_q = \{(t_i, f_i)\}$
    if $H_p = \emptyset$ then
        $H_p \leftarrow \{(t_i, f'_i) \mid f'_i = 1 \text{ iff } A(p) \leq t_i\}$
    end if
    $H_p \leftarrow \{(t_i, \frac{f_i + f'_i}{2})\}$
end if
```

Figure 19: Aggregation algorithm at peer p. For simplicity, the system size estimation and time-to-live termination mechanisms are not shown. The H_p variable is initialised with $\emptyset$ at all peers.

current estimation of N at peer p generated in a previous aggregation instance (nodes joining the system are bootstrapped by their initial neighbours), and R is the system constant that regulates the frequency of new aggregation instances. In a stable state, with a steady number of peers in the system, a new aggregation instance should be created on average with frequency $\frac{1}{R}$ rounds.

For each aggregation instance, peer p stores a set of interpolation points H_p, described previously, and a *weight* variable w_p, which it uses to estimate the system size. In order to start a new instance, peer p selects a set of threshold values t_i using the SELECTPOINTS procedure described later, generates an initial set of interpolation points $H_p = \{(t_i, f_i) \,|\, i \in [0, \lambda),\ f_i = 1 \text{ if } A(p) \leq t_i;\ 0 \text{ otherwise}\}$ and sets its weight w_p to one.

Joining an Aggregation Instance In every round, each peer contacts one of its neighbours for a gossip exchange in the ROUND procedure. A peer also accepts any connections from other peers (not necessarily its neighbours) for exchanges. During an exchange between peers p and q, peer p sends H_p and w_p to q and peer q replies with H_q and w_q. Both peers then merge the received values in the MERGEGOSSIP procedure.

The MERGEGOSSIP procedure at peer p has three cases. If q does not participate in the aggregation instance yet, it sends an empty H_q to p, and p simply ignores the gossip exchange. Otherwise, if p has an empty H_p set, it joins the instance by setting its weight w_p to zero and creating an its initial set of interpolation points $H_p = \{(t_i, f_i) \,|\, i \in [0, \lambda),\ f_i = 1 \text{ if } A(p) \leq t_i;\ 0 \text{ otherwise}\}$. Note that p uses the thresholds t_i obtained from q to initialise H_p so that all nodes participating in this aggregation instance use identical interpolation point definitions as assigned by the peer that started the instance. Finally, peer p averages the w_p and w_q weights and merges H_p and H_q by averaging the corresponding f_i values.

In a push-pull gossip protocol – where both peers exchange information – every peer in the system joins an aggregation instance with very high probability in just a few rounds. Note that during any merge, the sum of all weights for the instance remains one, while the variance between peers is reduced. Similarly, the f_i values in the CDF approximations H approach $1/n_i$ where n_i is the number of peers with $A(p) \leq t_i$. As the peers exchange and merge their H sets and w weights, they quickly approximate the desired CDF and the system size.

Terminating an Aggregation Instance Every instance is associated with a time-to-live counter, which is reduced by one per round at each peer. For simplicity, this mechanism is not shown in Figure 19. When an instance ends, each peer p updates its estimation of the number of nodes in the system $N_p = \frac{1}{w_p}$ and approximates the whole attribute CDF by interpolating the points of H_p. We use simple linear regression between each consecutive pair of points to obtain F_p, but more complex approaches are possible. Finally, each peer deletes its H_p set and stops participating in the aggregation instance.

Extreme CDF Values So far, for simplicity we have ignored two special points in any approximation: the first and last. Our algorithm finds the minimum and maximum attribute values to use in later aggregation instances. In order to discover these values, both are added to H and treated specially. When merging tuples, the corresponding minimum or maximum is chosen. With this simple addition, all nodes quickly converge on both values.

Multiple Attribute Values per Node The aggregation algorithm can be easily extended to handle cases where individual nodes are allowed to have multiple attribute values. For example, to estimate the distribution of file sizes at all nodes in the system each node contributes its set of file sizes. In this case, we define $A(p) \subset A$ as the set of values for attribute A at peer p and A as the set of all attribute values at all nodes in the system. The CDF for attribute A is defined as function $F : \mathbb{R} \to \mathbb{R}$ such that

$$F(x) = \frac{\Big|\{a \in A :\ a \leq x\}\Big|}{|A|}$$

As previously, the CDF is approximated by calculating the value for F in a set of discrete points (t_i, f_i) where $F(t_i) = f_i$. The t_i thresholds are generated by MinMax (or one of the other heuristics) and disseminated to other nodes by gossipping. In order to calculate f_i, nodes generate two values using the averaging algorithm. First, each node p calculates avg_i – the average number of attribute values below t_i per node – by contributing $|\{a \in A(p) :\ a \leq t_i\}|$ to the averaging algorithm. Second, each node p calculates avg – the average number of attributes per node – by contributing $|A(p)|$ to the averaging algorithm. Note that avg is independent of i and can be calculated once for all the CDF points. The f_i value is then given as $f_i = \frac{avg_i}{avg}$.

3.2.4 Interpolation Point Selection

When starting a new aggregation instance, each peer needs to decide on the placement of the interpolations points in H. Initially a node may have no prior knowledge about the attribute distribution. The simplest approach in this case is to spread the interpolation points at uniform intervals within the attribute domain.

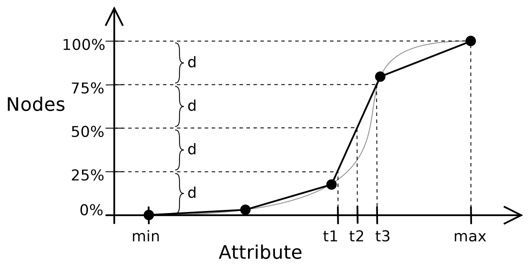


Figure 20: Interpolation point selection using HCut. The gray curve represents F– the true CDF. The black line represents H_p – the previous CDF interpolation at p.

However, the distributions of node characteristics in large-scale distributed systems are often highly skewed [2], resulting in a poor approximation using uniform intervals. An alternative consists of using attribute values found in a subset of neighbours of the initiating node.

Once the system has a rough estimate of the attribute distribution, it can use this knowledge to further refine the selection of interpolation points in future aggregation instances and to reduce the approximation error in the estimated CDF. Different selection algorithms may be used depending on the evaluation metric that the application tries to optimise. We discuss different refinement techniques in the rest of this section.

Minimising the Maximum Error One of the simplest bin selection heuristics to reduce $\mathrm{Err}_m(p)$ which we call ***HCut***, chooses the interpolation points for a new aggregation instance such that they divide the image of H_p into $(\lambda+1)$ equal size quantiles. Since $\mathrm{Err}_m(p)$ is determined by the maximum vertical distance between interpolation points, this heuristic attempts to limit the maximum error to $\frac{1}{\lambda+1}$, assuming the CDF does not change significantly between aggregation instances. Figure 20 illustrates the execution of the HCut algorithm. The interpolation points for the next aggregation instance (t_1, t_2, t_3) correspond to 25%, 50%, and 75% quantiles.

The HCut algorithm is quite efficient at approximating continuous, smooth CDFs. However, in many systems the number of existing attribute values is small. Moreover, for many common node properties, large numbers of nodes have identical or very similar attribute values. For example, many PCs have 512 MB, 1 GB, or 2 GB of RAM, but relatively few current machines have an amount of RAM that is between these three values. The CDFs of such attributes are step functions that are poorly approximated by HCut.

```
1: SelectPoints($H$):
2: $H_{old} \leftarrow H$
3: loop
4:   find $n$ that maximises $|f_n - f_{n-1}|$ in $H$
5:   find $m$ that minimises $|f_{m+1} - f_{m-1}|$ in $H_{old}$
6:   if $|f_n - f_{n-1}| > |f_{m+1} - f_{m-1}|$ then
7:     remove point $(t_m, f_m)$ from $H$ and $H_{old}$
8:     add point $(\frac{t_n+t_{n-1}}{2}, \frac{f_n+f_{n-1}}{2})$ to $H$
9:   else
10:    return $H$
11:  end if
12: end loop
```

Figure 21: MinMax interpolation point selection algorithm. The algorithm iteratively attempts to split the widest gap while removing the midpoint from the narrowest cluster of three points.

To approximate discontinuous CDFs, we propose *MinMax* – a heuristic that attempts to identify and approximate the steps in these CDFs. Figure 21 shows the pseudocode for MinMax. Instead of dividing the space into even quantiles like HCut, MinMax iteratively finds the farthest two consecutive interpolation points (by vertical distance) in the previous set of interpolation points H_{old}, denoted n and $n-1$, and the closest three interpolation points (again, by vertical distance) in H, denoted $m-1$, m, and $m+1$. If the two farthest points are farther apart than the closest three, point midpoint m of the closest three is removed from both H and H_{old}, and a new point is added to H at the new, interpolated midpoint between n and $n-1$. When no points satisfy the condition, the thresholds in H are returned as the output of the algorithm.

A sample MinMax step is graphically illustrated in Figure 22. MinMax changes the interpolation points only if it is expecting to reduce the interpolation error. By iteratively splitting the steepest fragments in the interpolated curve over multiple aggregation instances, MinMax efficiently identifies steps in the CDF.

Minimising the Average Error The HCut and MinMax heuristics attempt to minimise the maximum vertical distance measured by $\mathrm{Err}_m(p)$. However, $\mathrm{Err}_a(p)$ depends upon the area between the CDF and the interpolation. To reduce the area, we consider the *LCut* heuristic that selects the interpolation points based on their Euclidean distance instead of vertical distance.

Figure 23 shows a sample execution of the LCut heuristic. In order to decide on the interpolation point placement, peer p first calculates the length of the H_p

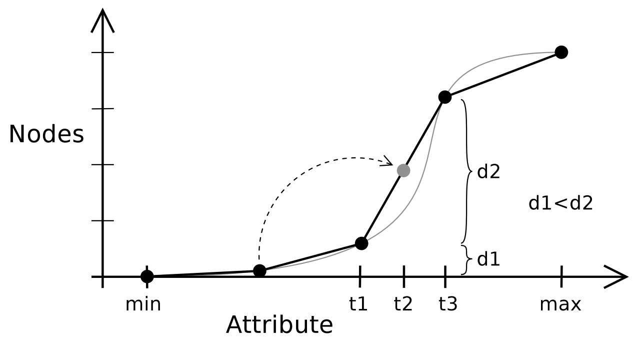


Figure 22: Interpolation point selection using MinMax. The distance between min and t_1 is less than t_1 and t_3. We thus assume that the first segment contributes more to the average error than the second. Moving the midpoint of the first segment to the second at point t_2 is then likely to reduce the average error $Err_a(p)$.

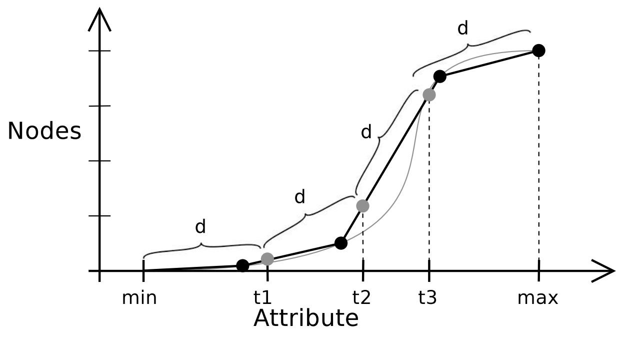


Figure 23: Interpolation point selection using LCut. Points $\{t_1, t_2, t_3\}$ are chosen to divide equally the Euclidean distance along the previous approximation H.

linear interpolation curve for the previous aggregation instance. Then, p divides the H_p curve into λ equal length (by Euclidean distance) segments to determine the placement of new points. Note that the horizontal axis is scaled by $max - min$ in order to equalise the horizontal and vertical coordinate ranges. As shown later in Section 3.2.6, compared with HCut and MinMax, LCut achieves lower average interpolation error Err_a, but suffers from higher maximum error Err_m.

3.2.5 Dynamic Confidence Estimation

Our algorithm has the additional property that it allows each node to estimate its own CDF approximation accuracy. This can be used to dynamically tune the algorithm parameters – such as the number of interpolation points and the number of executed instances – according to application-specific accuracy requirements.

The accuracy estimation is based on the fact that nodes are able to estimate the CDF value very accurately at the points of H. To estimate the accuracy of the approximation, each peer p generates an additional set of *verification points* V_p similar to the interpolation points H_p, where each element in V_p is a pair (t_i, f_i) such that $F(t_i) = f_i$. The extra V_p points are added to aggregation algorithm to be gossipped and merged along with the original H_p points.

The t_i thresholds for the verification points are independently chosen by each node that initiates a new aggregation instance. Their selection depends on the selected error metric. For example, in order to estimate the average CDF approximation accuracy (defined by the $Err_a(p)$ metric), the t_i thresholds are selected uniformly between the attribute minimum and maximum. At the end of an instance, each peer p estimates the accuracy of its CDF approximation F_p as

$$Err'_a(p) = \operatorname*{avg}_{i<|V_p|} |F_p(t_i) - f_i|$$

The maximum approximation error $Err_m(p)$ is generally more difficult to estimate compared to $Err_a(p)$ since it is determined by a single point in the CDF. In order to estimate $Err_m(p)$, a peer q that starts a new aggregation instance selects the verification points V_q based on its current CDF interpolation. Specifically, the V_q points are inserted between the H_q points by iteratively dividing the farthest two points in H_q by vertical distance. This way, peer q attempts to find the attribute values at which the true CDF and the interpolated curve most differ. When an instance ends, each peer p estimates its approximation accuracy as

$$Err'_m(p) = \max_{i<|V_p|} |F_p(t_i) - f_i|$$

3.2.6 Performance Evaluation

We evaluate our algorithms in PeerSim, a simulator for peer-to-peer systems [14]. Using PeerSim allows us to evaluate systems with 100,000 nodes, which would be infeasible using a real-world deployment. Unless specified otherwise, all evaluations are based on 100,000 nodes and $\lambda = 50$ interpolation points.

We did not use a synthetic distribution of attribute values. Synthetic distributions are typically smooth and therefore easier to approximate. Our evaluation instead uses real-world data where skew and discontinuities occur. Specifically, we use host traces from the BOINC volunteer computing project [2]. For each machine that participated in BOINC in 2008, we extract the following four attributes: measured CPU performance in FLOPS, measured downstream bandwidth, amount of installed memory, and amount of installed disk space. We filtered out samples from the trace that result from obviously faulty readings (for

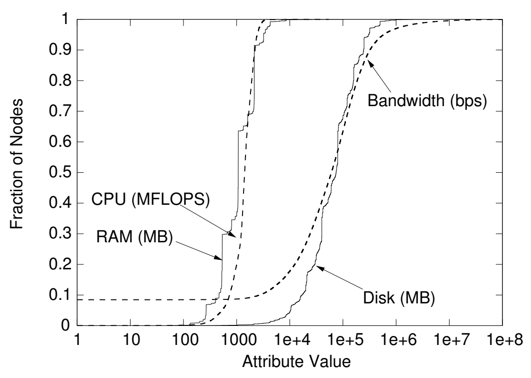


Figure 24: Actual attribute distributions F

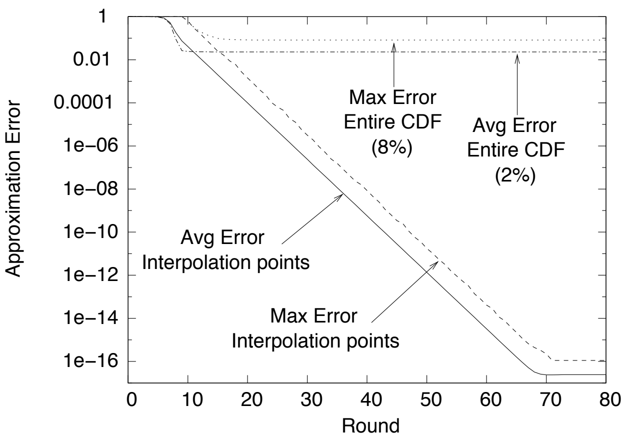


Figure 25: Approximation accuracy over one aggregation instance (RAM).

example, a machine with a bandwidth capacity above 10^{31} bps or one with a negative amount of memory). Figure 24 shows the actual CDFs of these four attributes. The CPU and bandwidth attributes have smooth distributions, while the RAM and available disk attributes have much more skewed distributions. Our evaluation demonstrates that skewed distributions are harder to estimate accurately.

We compare our aggregation algorithm with two other CDF estimation approaches: the histogram-based EquiDepth heuristic [10] and random sampling [9]. In the latter approach we construct an attribute CDF based on a random subset of attribute values drawn from the system. For each algorithm, we measure the maximum approximation error Err_m and the average approximation error Err_a.

CDF estimation accuracy Our aggregation algorithm achieves a very accurate approximation of the CDF at the interpolation points. Figure 25 shows the maximum and average approximation error over all peers measured at each protocol

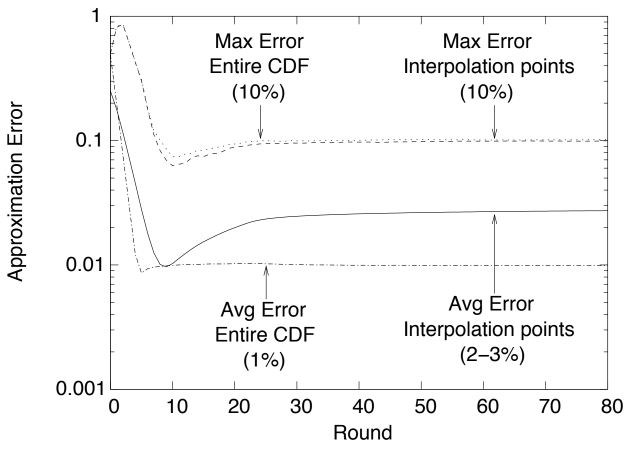


Figure 26: Approximation accuracy in EquiDepth (RAM).

round. We compare the maximum and average error at the H interpolation points with the Err_m maximum and Err_a average error over the entire CDF domain. For clarity, only the RAM attribute is displayed – the algorithm generates consistent results for all other attributes. During the first few rounds of the experiment, not all nodes have joined the aggregation instance and the error is equal to the maximum value of one. However, starting from round 10, the error at the interpolation points decreases at an almost perfectly exponential rate and quickly becomes negligible. After 70 rounds it reaches the level of hardware rounding errors. Since all attributes show similar results in our evaluation, we consider 25 rounds sufficient to accurately calculate the CDF at the interpolation points. The standard deviation of our error metrics across all system nodes remains below 10^{-5}, and hence, in a single aggregation instance all peers generate nearly identical CDF approximations. At the same time, the Err_m and Err_a error over the entire CDF domain does not decrease below a few percent due to the interpolation error at points outside of H. In order to reduce the interpolation error, nodes need to either add new points to H or select a new set of interpolation points that better fits the CDF curve.

For comparison, we implemented the EquiDepth approach and show the different trends in error measurements in Figure 26. For error measurements across the entire CDF, both algorithms have errors in the same magnitude (8% max for ours and 10% max for EquiDepth). However, the approximation error over time at the selected bins does not improve in the EquiDepth approach. This approach suffers a significant approximation error even at selected histogram bins due to sample duplication [10]. Our algorithm instead leverages the decreasing error over time at the selected points both to refine the selection of points in new ag-

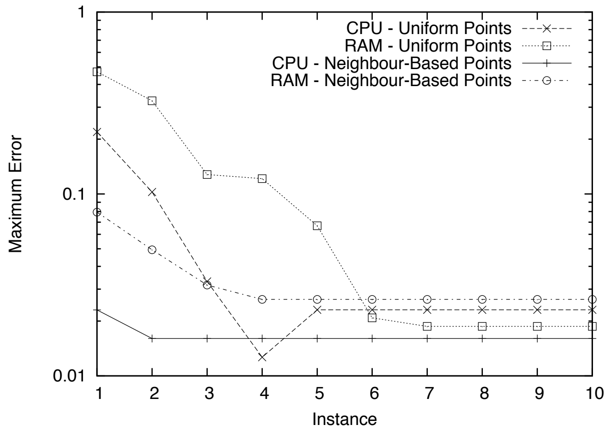


Figure 27: Approximation accuracy of MinMax using different bootstrap approaches.

gregation instances to reduce maximum and average errors across the entire CDF and to dynamically gauge the accuracy of own CDF estimation.

Initial interpolation point selection While a single instance of our aggregation protocol very accurately estimates the attribute CDF at the interpolation points, we present several techniques to refine the set of interpolation points over multiple aggregation instances. The selection of the interpolation points significantly affects the interpolation error over the entire CDF, and different refinement techniques are presented to minimise the different measures of error that we present. However, all of the refinement techniques use a previous estimate of the CDF to generate a new, refined set of interpolation points. This subsection describes how the algorithms choose the initial set of points.

As an example, in Figure 27 we consider two approaches to bootstrapping the first aggregation instance of the MinMax algorithm: assigning interpolation points uniformly between the minimum and maximum attribute value calculated using an aggregation instance (labelled “Uniform Points”) and using a random subset of the attribute values of the peer’s neighbours in the P2P overlay (“Neighbour-Based Points”). The results clearly demonstrate that using neighbour attribute values significantly improves the algorithm’s convergence. We believe that since the MinMax algorithm is trying to spread the interpolation points according to the distribution of the values, taking the initial interpolation points from neighbours bootstraps the algorithm with points already from the desired distribution. Further, we also see that MinMax converges much faster for smoother CDFs (CPU and bandwidth) than for the heavily-skewed CDFs (RAM and disk size) where the precise selection of interpolation points is crucial for overall accuracy. Since similar results hold for the other refinement algorithms, we bootstrap the first aggregation instance in the rest of the evaluation using the attribute values of a node's neighbours.

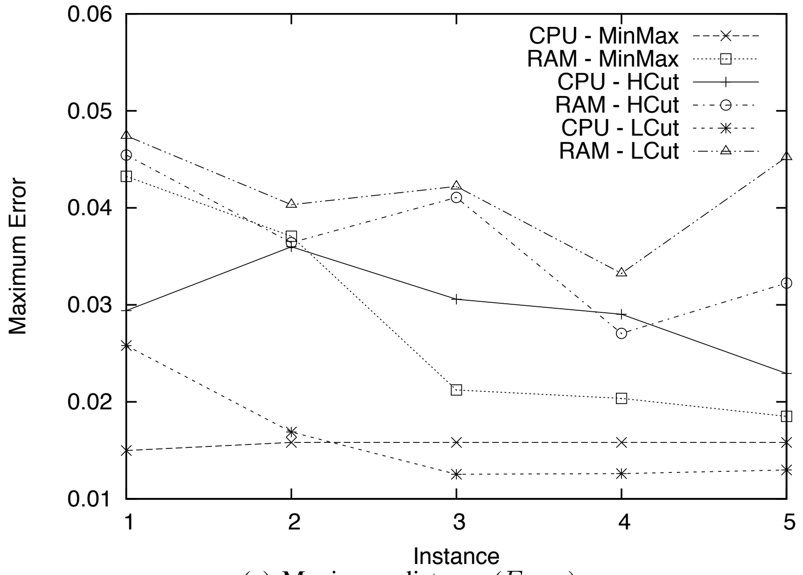
Convergence over multiple instances In this subsection, we compare our approximation techniques with EquiDepth and random sampling over multiple aggregation instances. After only three instances, our algorithms can obtain an order of magnitude better average error and several times better maximum error than EquiDepth.

All our refinement algorithms manage to effectively reduce the maximum and average approximation error in Figure 28. For the Err_m measure, all algorithms achieve good results for smooth distributions (CPU), but for heavily-skewed CDFs (RAM), MinMax significantly outperforms the others. We thus focus on the MinMax algorithm when minimising Err_m in the remaining experiments. For the Err_a metric, LCut achieves significantly lower error after 3 instances than any other algorithm by an order of magnitude. We similarly focus on LCut when minimising Err_a in later experiments.

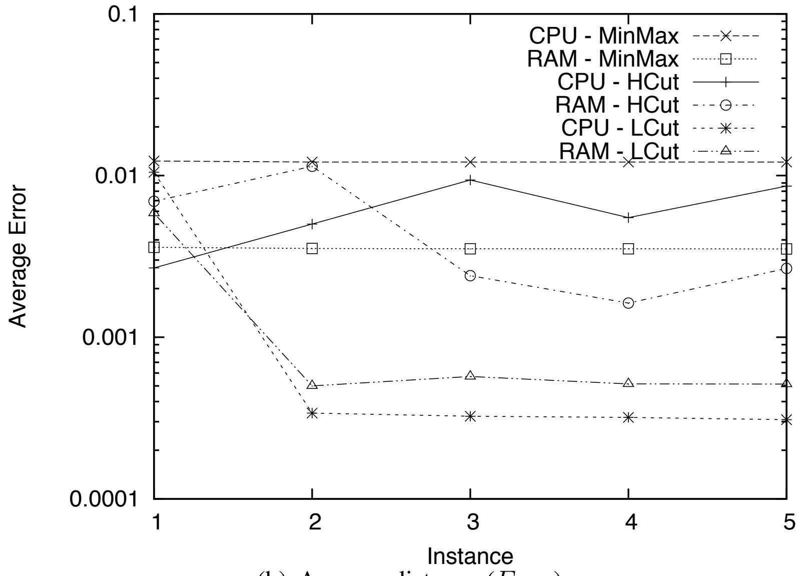
The performance differences of MinMax and LCut for each error metric demonstrates the difficulty of optimising for both metrics simultaneously. Although LCut performs best for Err_a and indeed for Err_m for smooth CDFs, it has the worst performance for Err_m for skewed CDFs where precise interpolation selection is most important. It remains as future work whether a single heuristic can significantly reduce both error metrics for diverse CDF curves.

The results for our algorithms are compared with the approximation error generated by EquiDepth and random sampling in figures 29 and 30, respectively. We execute the EquiDepth phases with the same frequency and duration as aggregation instances to make as fair a comparison as possible. Since EquiDepth does not refine its histogram bins based on CDF approximations from previous phases, it generates the same approximation error in every phase. Consequently, EquiDepth suffers a few times higher maximum error compared to MinMax (particularly for highly-skewed distributions), and at least an order of magnitude higher average error compared to LCut.

The accuracy of CDF approximation using random sampling depends on the generated sample size. In the 100,000-node system we use in our evaluations, about 1,000 to 10,000 random samples are required to achieve an accuracy similar to that of MinMax or LCut. As discussed later in this paper, the random sampling approach in [9] would generate in this case between 1,000 to 10,000 messages per node – a prohibitive cost compared to our approach. Finally, we note that the error measurements for random sampling are higher for heavily-skewed CDFs compared to smooth CDFs.

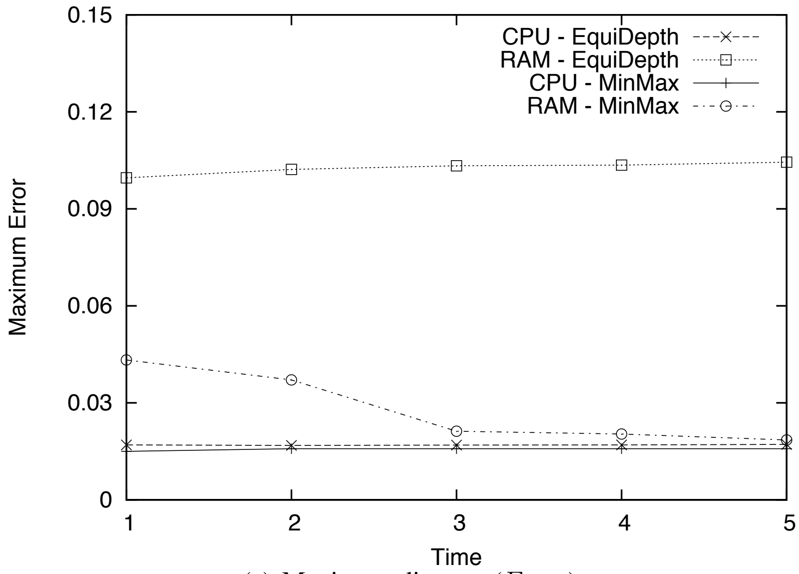


(a) Maximum distance (Err_m)

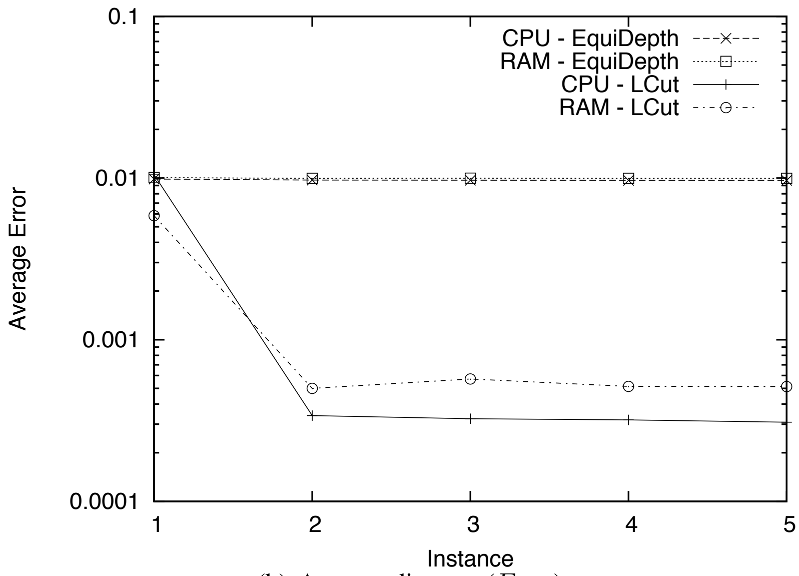


(b) Average distance (Err_a)

Figure 28: Comparison between HCut, MinMax, and LCut.

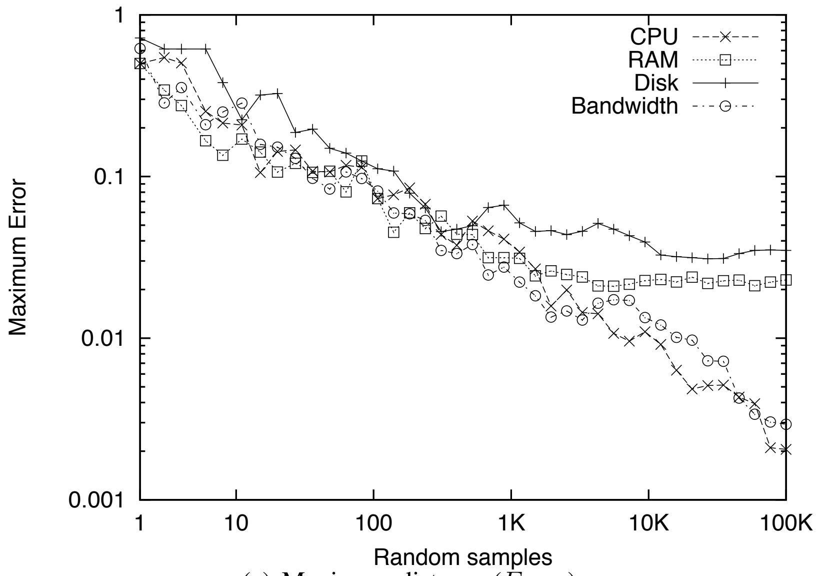


(a) Maximum distance (Err_m)

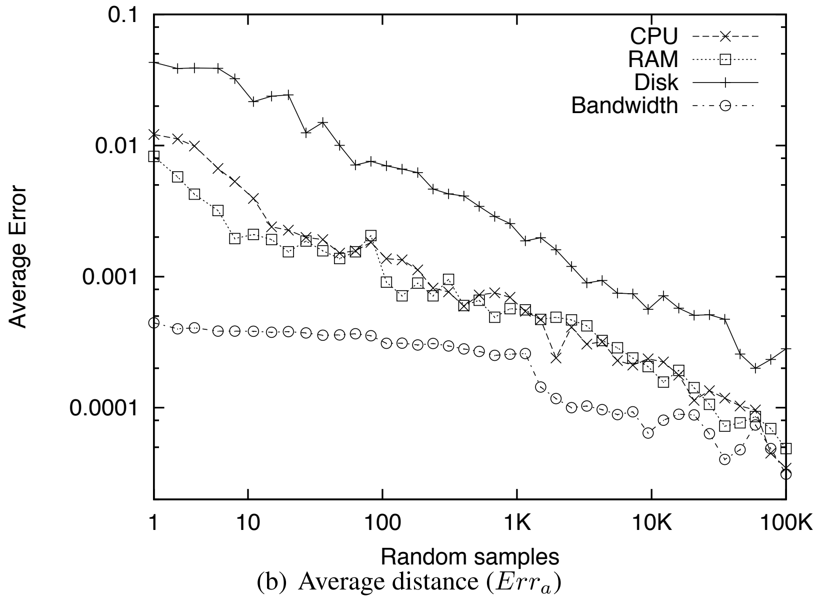


(b) Average distance (Err_a)

Figure 29: Approximation error in EquiDepth over multiple phases.

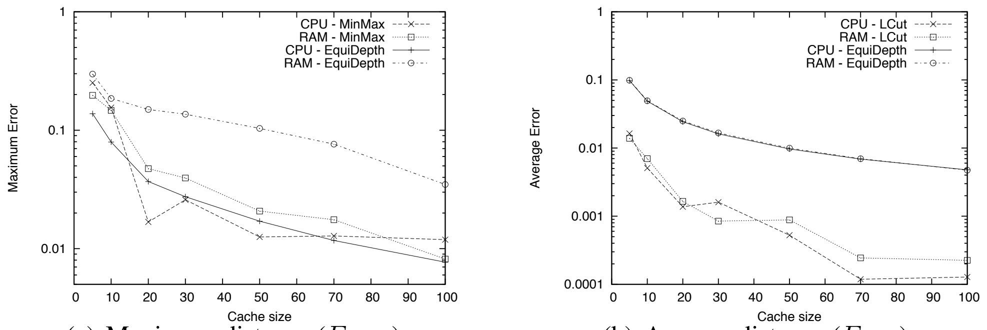


(a) Maximum distance (Err_m)



(b) Average distance (Err_a)

Figure 30: Approximation error for random sampling.



(a) Maximum distance (Err_m) (b) Average distance (Err_a)

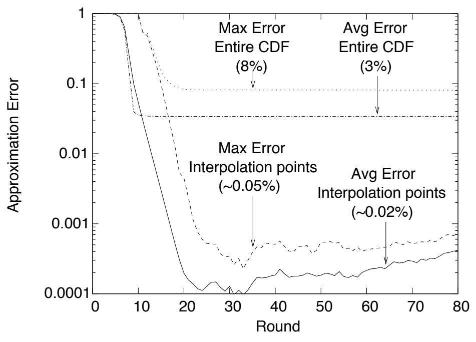
Figure 31: Influence of the number of interpolation points on aggregation accuracy.

Influence of the number of interpolation points One way to improve distribution estimation accuracy is to increase the number of interpolation points. This section explores the tradeoff between the increased accuracy one can derive from high number of interpolation points and the implied communication costs.

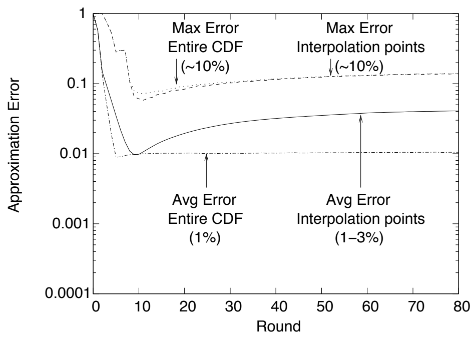
Figure 31 shows the Err_m and Err_a accuracy after 4 instances (phases) in our aggregation algorithm and the EquiDepth algorithm when using between 10 and 100 interpolation points (bins). As one would expect, more interpolation points bring better estimation accuracy. The slight variations in the graph (in a few cases the error increases when more interpolation points are used) can be explained by the random component in our heuristic. As previously, EquiDepth is outperformed by MinMax with the Err_m metric and LCut with the Err_a metric.

A number of 50 points provides an acceptable accuracy for many possible applications: an Err_m maximum distance of 2%, as obtained with MinMax, or an Err_a average error 0.1%, as obtained with LCut. However, for the applications that need higher accuracy, increasing the number of points does not incur a large performance penalty: when adding 10 extra points, the size of the messages exchanged among the peers increases by approximately 160 bytes; for the current capacities of the usual network links, this is almost negligible. Furthermore, if the CDF does not change significantly over time, nodes can combine interpolation points obtained in multiple aggregation instances in order to reduce the overall interpolation error.

Impact of churn We model churn by randomly replacing a fixed fraction of nodes in the overlay with new nodes at each simulation round. Since changes in the attribute distribution are entirely application specific, we maintain a constant attribute CDF over the course of each experiment. We set a churn rate based on measurements on existing P2P systems [23]. Assuming a gossip periodicity of



(a) Aggregation



(b) EquiDepth

Figure 32: Approximation accuracy in the presence of churn, for a single instance (RAM).

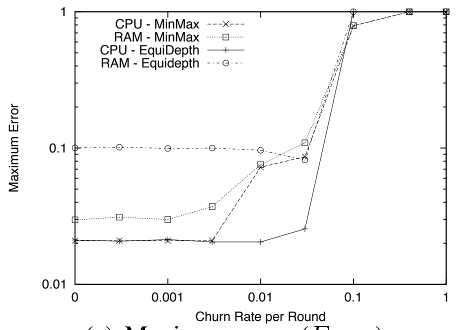
one second and a mean session duration of 15 minutes, approximately 0.1% of nodes leave the system per round and rejoin with a different attribute value drawn from the same distribution.

Figure 32(a) shows the maximum and average CDF approximation error in one instance of our algorithm in a system with churn. The evaluation metrics do not include nodes that join the system during the instance execution, since their CDF approximations are undefined. After an initial phase, when the instance is propagated to all nodes, the approximation error starts to gradually decrease. Since some nodes leave the system before their f_i values are disseminated and averaged, the CDF approximation error at interpolation points does not converge to zero. However, the obtained accuracy is on the order of 0.01% and is clearly sufficient to approximate the CDF through interpolation.

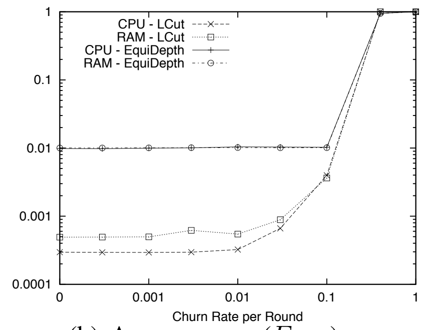
For completeness, figure 32(b) shows the approximation error produced by an EquiDepth phase in the same system setup. EquiDepth is not significantly affected by churn, but as previously, the heuristic is not able to reduce the maximum approximation error below 10% (and 1% for the average error) even at the selected histogram bins.

Figure 33 shows the maximum and average approximation error incurred by our aggregation algorithm and EquiDepth after 8 protocol instances (phases). In this experiment, joining nodes are included in the evaluation metrics, since they receive initial CDF approximations – generated in the previous aggregation instances – from their neighbours. Moreover, joining nodes ignore aggregation instances (phases in EquiDepth) that had started before these nodes entered the system in order not to distort the results from already running aggregation instances.

All algorithms show a very high resilience to churn, which starts to significantly decrease the approximation accuracy only at rates of 1% nodes per gossip

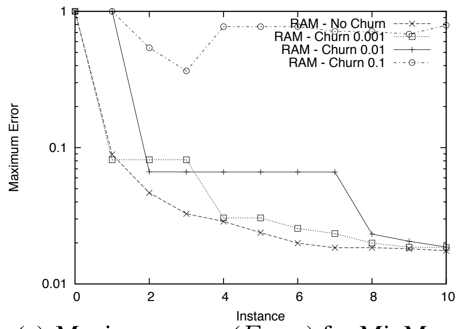


(a) Maximum error (Err_m)

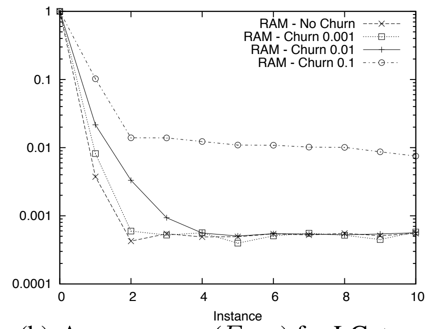


(b) Average error (Err_a)

Figure 33: Impact of churn on approximation accuracy.



(a) Maximum error (Err_m) for MinMax



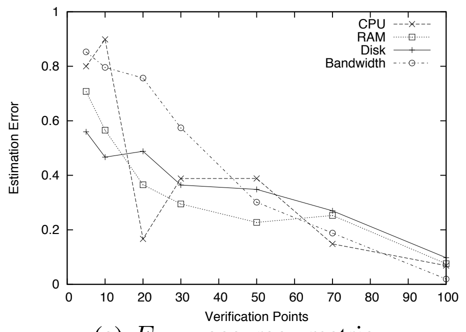
(b) Average error (Err_a) for LCut

Figure 34: RAM distribution approximation error in systems with churn.

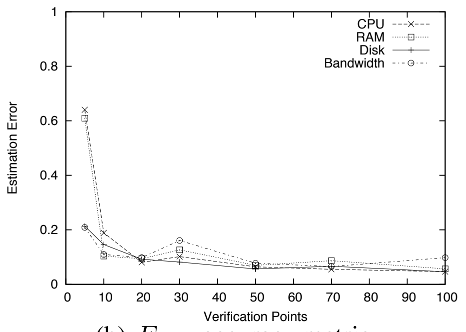
round (i.e., 1% per second). This rate is 10 times higher than the levels observed in [23].

We also observe that churn reduces the algorithm's convergence speed. This is caused by fact that nodes joining the system do not participate in current aggregation instances, but instead copy the CDF estimations obtained by other nodes in previous instances. For extreme churn rates, very few nodes (if any) complete a full aggregation instance and the results generated by aggregation become inaccurate, as shown in Figure 34.

Dynamic confidence estimation As described in section 3.2.5, nodes can use the aggregation framework to assess the accuracy of their own CDF approximations. We evaluate the accuracy estimation algorithms by computing the average difference between the nodes' assessment of an error metric and the actual value for that error metric. Given the true CDF approximation accuracy $Err_a(p)$ at node p, and p's own estimation of its accuracy $Err'_a(p)$, we define the error in accuracy



(a) Err_m accuracy metric



(b) Err_a accuracy metric

Figure 35: Accuracy estimation error for MinMax.

estimation at node p as

$$\frac{|Err_a(p) - Err'_a(p)|}{Err_a(p)}$$

Similarly, the error in $Err_m(p)$ estimation at node p is defined as

$$\frac{|Err_m(p) - Err'_m(p)|}{Err_m(p)}$$

Figure 35 shows the accuracy estimation results for the two considered metrics. Using 20 verification points, nodes can estimate their own average approximation accuracy with a 10% error. This adds a 40% traffic overhead to our CDF approximation algorithm. The error drops under 5% if a sufficient number of interpolation points is used. As expected, more verification points are required to obtain an accurate estimation of the Err_m maximum error. However, the experiment shows that even this difficult metric can be roughly estimated using our aggregation framework.

Cost evaluation An important objective in designing the aggregation algorithm is to achieve a low communication cost. The network traffic exchanged by any node is proportional to the number of interpolation points (λ) and the number of gossip rounds. For 50 interpolation points, the size of a gossip message is approximately 800 bytes. At each gossip cycle each node issues exactly one gossip and receives on average one gossip. Each gossip requires sending and receiving one message. Therefore, to estimate an attribute CDF with $\lambda = 50$ and 25 rounds, each peer will send, on average, about 40 kB of data (50 messages), and receive another 40 kB (50 messages). Since three aggregation instances are sufficient for MinMax and LCut to converge, an accurate CDF approximation can be obtained by sending 120 kB of data (150 messages) per node. This cost does not depend on the system size.

The time required to generate a CDF estimation depends on the gossip periodicity. If we consider a periodicity of 1 second, then a CDF can be obtained in about 25 seconds using an average upstream bandwidth of about 1.6 kB/s, and a downstream bandwidth with a similar value. The CPU, memory, and topology maintenance costs are negligible.

The costs of EquiDepth and our aggregation algorithm are very similar. Both algorithms are based on a push-pull gossip exchange over a random overlay and hence generate the same number of messages. Moreover, since we use 50 histogram bins in EquiDepth and 50 interpolation points in aggregation, the message sizes are almost identical in both approaches. The only advantage of EquiDepth over our algorithm is a better convergence speed, since EquiDepth does not improve its results between consecutive phases.

In random sampling, about 1,000 to 10,000 samples must be obtained by a node in a 100,000-node system in order to achieve a CDF approximation accuracy comparable to that of MinMax or LCut. Using random walks [9], this requires generating between 1,000 and 10,000 messages per node – an order of magnitude more compared to our approach.

3.2.7 Related Work

The task of data aggregation, or synopsis construction, has been well-studied in the past in the areas of sensor networks [1, 18] and distributed databases [24]. However, most of the proposed algorithms are reactive. Each time a node requests aggregation, a dissemination tree (or weighted graph) is constructed between nodes in order to collect the required data from the system. Such graphs are neither robust to failures of nodes near the sink nor do they efficiently disseminate the result to all nodes. The focus of this paper is instead to collect aggregation information at all nodes while evenly distributing the overhead using a symmetric algorithm.

Our approach is based on gossip protocols, which are renowned for their scalability, robustness, and low cost [16, 13]. These protocols have been previously used to approximate simple system properties such as minimum, maximum, and mean values of an attribute. We extend these algorithms by adding mechanisms that allow nodes to approximate system-wide distributions and to assess and improve the accuracy of these approximations.

Several existing algorithms allow nodes to estimate their own ranks and slices [17, 12, 8]. While these solutions incur less overhead, they provide more limited information than a distribution estimation. For example, they do not enable nodes to estimate whether an attribute distribution is skewed, imbalanced, or contains outliers: node ranks by definition are always assigned between 1 and N (system size), regardless of the actual attribute distribution. Such algorithms are not sufficient to determine important RSS properties.

The problem of outlier detection is addressed using gossip by Eyal et al. [7]. The algorithm gossips synopses of clusters and outliers to enable both the removal of the outliers and the discovery of cluster formation. However, the cluster synopses do not estimate the full distribution of node parameters, and our algorithm is also well-suited to other distributions without clusters.

A simple way to estimate an attribute distribution is to generate a random sample of attribute values [9, 11]. However, as we have shown in the evaluation section, such an approach is extremely inefficient compared to our algorithm.

Haridasan et al. estimate the distribution of an attribute value by gossipping synopses of equi-depth histograms [10]. Using equi-depth bins, the system converges towards an estimation accuracy around 7% in the absence of churn, while under the same conditions our system obtains an order of magnitude improvement. Furthermore, our algorithm also provides a useful estimation of its own accuracy to enable a tradeoff of accuracy for lower communication overhead.

3.2.8 Conclusions

This section shows how to efficiently and accurately estimate the statistical distribution of an attribute belonging to nodes in a peer-to-peer overlay. Our algorithm has a low cost in the order of 1.6 kB/s traffic over 75 seconds, and generates approximations within an average error of 0.05% and a maximum error of 2%. Further, the algorithm can estimate its own accuracy, and due to its use of gossip techniques, is quite resilient to churn – obtaining roughly the same average error for churn rates up to 1% per second.

In the remaining duration of the XtreemOS project, we will apply this technique to the dynamic estimation of node attribute distribution within the RSS. This will allow the RSS to self-manage internal configuration parameters such as the definition of cell boundaries, and will result in better overall performance.

4 Conclusions

In this document we have described the improvements made to the SRDS system and its RSS component in order to increase the overall usability, reliability, performance and scalability of the XtreemOS platform. We have discussed architectural changes of the SRDS service, and evaluated those changes with experiments running on platforms of up to 1 thousand real nodes and 100 thousand simulated nodes. Tests show that the added configurability and the option to change the configuration dynamically at run-time have been implemented within the RSS without harming the scalability and performance of the service, and that a self-managing behaviour of the RSS is feasible and will improve the efficiency and reliability of the XtreemOS resource location.

Similarly, the architectural changes in the SRDS have been evaluated by simulation on top of the Grid'5000/Aladdin platform. Accordingly, new features that have been added to the SRDS interface in order to allow a better exploitation of the discovery service by other XtreemOS modules, they do not impair the SRDS performance on large networks. This has been experimentally verified for the AEM and JDS services, measuring the service time of resource and Job Directory queries in large overlays. New functionalities like the generic query engine and the neighborhood query support will increase the integration of the SRDS and the highly available services within the XtreemOS platform.

Several of the documented improvements concerning both the SRDS and the RSS are also relevant as academic results, some of them having been already published. As a matter of fact, changes and additions described in the document are already part of the the current XtreemOS public release, or are currently being integrated in the upcoming public release.

References

[1] Charu C. Aggarwal and Philip S. Yu. *A Survey of Synopsis Construction in Data Streams*, chapter 9. Springer-Verlag New York, LLC, 2006.

[2] David P. Anderson and Kevin Reed. Celebrating diversity in volunteer computing. In *Proc. 42nd Intl. Conf. on Systems Science*, 2009.

[3] E. Carlini, M. Coppola, D. Laforenza, and L. Ricci. Reducing traffic in dht-based discovery protocols for dynamic resources. In *TODO*, 2009.

[4] Massimo Coppola, Guillame Pierre, Jeff Napper, Emanuele Carlini, Susanna Martinelli, Laura Ricci, and Patrizio Dazzi. XtreemOS Research Project

Deliverable D3.2.8 Reproducible evaluation of a service/resource discovery system, November 2008.

[5] Paolo Costa, Jeff Napper, Guillaume Pierre, and Maarten van Steen. Autonomous resource selection for decentralized utility computing. In *Proceedings of the 29th International Conference on Distributed Computing Systems (ICDCS)*, Montreal, Canada, June 2009.

[6] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, October 2004.

[7] Ittay Eyal, Idit Keidar, and Raphael Rom. Distributed clustering for robust aggregation in large networks. In *Proc. 5th Workshop on Hot Topics in System Dependability*, 2009. To appear.

[8] Antonio Fernández, Vincent Gramoli, Ernesto Jiménez, Anne-Marie Kermarrec, and Michel Raynal. Distributed slicing in dynamic systems. In *Proc. ICDCS*, 2007.

[9] Cyrus Hall and Antonio Carzaniga. Uniform sampling for directed p2p networks. In *Proceedings of the 15th Euro-Par Conference*, pages 511–522. Springer, August 2009.

[10] Maya Haridasan and Robbert van Renesse. Gossip-based distribution estimation in peer-to-peer networks. In *Proc. 7th Intl. Workshop on Peer-to-Peer Systems*, 2008.

[11] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Middleware*, volume 3231 of *Lecture Notes in Computer Science*, pages 79–98. Springer, 2004.

[12] Márk Jelasity and Anne-Marie Kermarrec. Ordered slicing of very large-scale overlay networks. In *Proc. 6th Intl. Conf. on Peer-to-Peer Computing*, 2006.

[13] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[14] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[15] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), August 2007.

[16] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proc. 44th IEEE Symposium on Foundations of Computer Science*, 2003.

[17] Alberto Montresor, Mark Jelasity, and Ozalp Babaoglu. Decentralized ranking in large-scale overlay networks. In *Proc. Self-Adaptive and Self-Organizing Systems Workshops*, 2008.

[18] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. *ACM Transactions on Sensor Networks*, 4(2), 2008.

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

[20] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[21] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA, 2008. ACM.

[22] K. Shudo, Y. Tanaka, and S. Sekiguchi. Overlay weaver: An overlay construction toolkit. *Computer Communications*, 31(2):402–412, February 2008.

[23] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proc. 6th Conf. on Internet Measurement*, 2006.

[24] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[25] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.