Master Thesis

Integration of COMPSs programming model with a new generation of Object Storage Platforms

Polytechnic University of Catalonia
Master in Innovation and Research in Informatics (MIRI)
High Performance Computing (HPC)

Carlos Díaz Suárez

04-2016
This thesis is submitted to the Barcelona School of Informatics (FIB) at the Polytechnic University of Catalonia (UPC) in partial fulfillment of the requirements for the degree of Master in Innovation and Research in Informatics (MIRI) with specialization in High Performance Computing (HPC).

Contact Information:

Author(s):

Carlos Díaz Suárez
E-mail: carlos.diaz.suarez@est.fib.upc.edu
LinkedIn: https://es.linkedin.com/in/carlosdiazsuarez

University advisor:

Prof. Rosa M. Badia Sala
E-mail: rosab@ac.upc.edu
Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya
Abstract

New challenges in science require handling and processing huge amounts of data, that is, Big-Data. This demand appears in areas like biology (molecular dynamics, genomic and protein docking analysis, neuroscience analytics, physiological human simulation, etc.), geosciences (air quality control, climate prediction, oil exploration, etc.), social sciences (social graph analytics, smart cities, etc.) among others.

Nowadays, researchers on each of these areas face the problem from its own perspective, but to have high performance applications, a unification of efforts with the research in computer science is key. Computer science research on programming models, middleware, and infrastructures is necessary to provide a unified, productive, and efficient environment for Big-Data applications.

On the other hand, the advances in data storage technology with Non-Volatile RAM (NVRAM) or Storage-Class Memories (SCM) devices, with low latencies, high bandwidth, and byte-addressable interface, are promising for Big-Data applications that need high IO performance.

A new generation of Object Storage Platforms is emerging to take full advantage of this high-performance and byte-addressable storage devices. This new object storage platforms store data as objects as understood in the object-oriented programming paradigm and opposed to traditional data management systems that store data as files or in databases. A single data model will work for both, in-memory and for stored data, enabling a tight integration with programming models.

Programming models and its corresponding middleware provide to developers a simple way for writing their parallel and distributed applications unaware of the underlying infrastructure. Parallelism and distributed computing are crucial nowadays for developing high-performance Exascale applications. Programming models also allow to pass information or hints to the middleware responsible of the application execution. This information or hints could be dynamically used to optimize the actual mapping of computation and data access/movement to resources.
In this master thesis it will be shown that this marriage of programming models and object storage platforms is a powerful framework to deal with the Exascale and Big-Data issues. Experiments run in the Marenostrum 3 Supercomputer, at the Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS), show that the integration of both technologies, programming models and object-storage platforms, could reduce considerably the execution time of scientific applications.

The COMPSs programming model, which is the main research line of the Workflows and Distributed Computing at BSC-CNS, has been integrated with two objects storage platforms, dataClay and Hecuba, that are the current projects of the Storage System group and the Autonomic Systems and e-Business Platforms group at BSC-CNS respectively. The collaboration and coordination of all the groups and the support team of Marenostrum has been crucial to achieve the objectives of this project.

The work done for this master thesis is a first step in the integration of these technologies and aims to be a guide for further research giving hints on what is important to consider in order to have the expected performance. The reader of this document is encouraged to discover the details of the integration of COMPSs with this new generation of object storage platforms in the following pages.

Keywords: Exascale, Big-Data, Programming Models, Object Storage Platforms
## Contents

Abstract i

1 Introduction 1
   1.1 Motivation ............................................. 1
   1.2 Objectives ............................................. 3
   1.3 Methodology .......................................... 4
   1.4 Planning .............................................. 5
   1.5 People ................................................. 7
      1.5.1 Workflows and Distributed Computing .......... 7
      1.5.2 Storage System .................................. 7
      1.5.3 Autonomic Systems and e-Business Platforms .. 7
   1.6 Document Organization ................................. 8

2 Related Work 9
   2.1 Programming Models .................................... 9
      2.1.1 STARSs ............................................ 9
      2.1.2 COMPSs ........................................... 10
         2.1.2.1 Programming Model ......................... 10
         2.1.2.2 Runtime System ............................. 11
      2.1.3 Java Example ..................................... 12
         2.1.3.1 Task Selection Interface ............. 13
         2.1.3.2 Task Dependency Graph .................. 14
      2.1.4 PyCOMPSs .......................................... 15
      2.1.5 Python Example ................................... 16
         2.1.5.1 Task Decorators ........................... 17
   2.2 New Object Storage Platforms ........................ 18
      2.2.1 Persistent Objects ............................... 18
      2.2.2 Self-Contained Objects ......................... 21
      2.2.3 Traditional Databases ........................... 22
      2.2.4 dataClay ......................................... 24
      2.2.5 Hecuba ........................................... 28
3 Storage API

3.1 COMPSs and the New Storage Platforms ........................................... 30
3.2 Storage API Specification ............................................................... 31
   3.2.1 SCO Interface ................................................................. 31
      3.2.1.1 Make Persistent ....................................................... 32
      3.2.1.2 Get Identifier ....................................................... 32
      3.2.1.3 Delete Persistent .................................................. 32
      3.2.1.4 Wrappers ............................................................. 33
   3.2.2 Storage Interface ............................................................... 34
      3.2.2.1 Init ................................................................. 34
      3.2.2.2 Get by Identifier .................................................. 34
      3.2.2.3 New Replica ......................................................... 35
      3.2.2.4 Get Locations ....................................................... 35
      3.2.2.5 New Version ........................................................ 36
      3.2.2.6 Consolidate Version ............................................... 37
      3.2.2.7 Execute Task ......................................................... 37
      3.2.2.8 Get Result ........................................................... 38
      3.2.2.9 Delete ................................................................. 38
      3.2.2.10 Finish ............................................................... 38
   3.3 Unit Tests for the Storage API ..................................................... 38

4 Integration

4.1 COMPSs 1.3 ................................................................................... 40
   4.1.1 Release Features ...................................................................... 40
      4.1.1.1 Persistent Workers .................................................... 40
      4.1.1.2 Communication Library ............................................. 41
   4.1.2 Software Packages ................................................................ 41
      4.1.2.1 Programming Model Package .................................... 42
      4.1.2.2 Runtime Package ....................................................... 43
      4.1.2.3 Engine Package ......................................................... 46
      4.1.2.4 Other Packages ......................................................... 49
   4.2 Integration with COMPSs 1.3 ........................................................ 51
      4.2.1 Loader Integration ................................................................ 51
         4.2.1.1 The MakePersistent method .................................... 52
         4.2.1.2 The checkSCOType method ...................................... 53
         4.2.1.3 The checkSCOPersistent method ............................ 53
         4.2.1.4 The PSCOId object .................................................. 54
      4.2.2 API Integration ................................................................... 56
      4.2.3 Engine Integration: Task Scheduler ...................................... 57
      4.2.4 Adaptors Integration: Worker .............................................. 57
   4.3 Integration with PyCOMPSs ............................................................. 58
5 Infrastructure

5.0.1 Development Infrastructure: BSCGrid ................. 60
5.0.2 Development Infrastructure: Minerva ................. 61
5.0.3 Production Infrastructure: Marenostrum III ......... 61

6 Compilation

6.1 COMPSs dataClay Application .......................... 63
6.1.1 Step 1: Application Class Registry .................. 63
6.1.2 Step 2: SCO Class Generation ....................... 64
6.1.3 Step 3: Final Application Compilation ............... 65
6.2 PyCOMPSs & Hecuba Application ........................ 66

7 Deployment

7.1 dataClay Deployment Components ....................... 68
7.1.1 dataClay Logic Module ............................. 68
7.1.2 dataClay Data Service ................................ 68
7.1.3 PostgreSQL ........................................ 69
7.2 COMPSs Deployment Components ....................... 69
7.2.1 COMPSs Master .................................... 69
7.2.2 COMPSs Worker .................................... 70
7.3 COMPSs & dataClay deployment in MN3 .................. 70
7.4 Hecuba Deployment Components ........................ 71

8 Analysis

8.1 Benchmark ............................................ 73
8.2 Previous Considerations ................................ 74
8.2.1 When persist a SCO with dataClay? .................. 74
8.2.2 Are both benchmark versions equivalent? ............ 75
8.2.3 What happen in the Compute phase? ................ 76
8.2.4 What happen in the Results phase? .................. 76
8.3 Experimental Analysis ................................ 77
8.3.1 Memory Issues ..................................... 77
8.3.2 Object Size ........................................ 79
8.3.3 Scalability ........................................ 85
8.3.4 Speedup ........................................... 87
8.3.5 Locality ........................................... 88
8.3.6 Sub-linear Speedup ................................ 89
8.3.6.1 Management Overhead ............................ 90
8.3.6.2 Locality misses ................................ 91
8.3.7 Traces .............................................. 93

9 Conclusions ............................................. 97
10 Future Work

10.1 Other Benchmarks .................................................. 100
10.2 COMPSs & Hecuba Analysis ....................................... 101
10.3 The Scheduler .......................................................... 101
10.4 Consumer ............................................................... 102

11 Annexe

11.1 Benchmark Source Code ............................................. 103
    11.1.1 Molecule.java .................................................. 103
    11.1.2 ProducerFile.java ............................................. 107
    11.1.3 ProducerFileItf.java ......................................... 109
    11.1.4 ProducerDataClay.java ....................................... 110
    11.1.5 ProducerDataClayItf.java ..................................... 112

11.2 Storage API ............................................................ 112
    11.2.1 SCO Interface ................................................ 112
    11.2.2 Storage Interface ............................................. 113
    11.2.3 Storage Exception ............................................ 116
    11.2.4 Callback Handler ............................................. 116
    11.2.5 Callback Event ............................................... 117
List of Figures

1.1 Master Thesis Gant Diagram. ........................................... 6

2.1 COMPSs programming model. As the main application thread is executed tasks are generated and executed in parallel. The main thread doesn’t end until all tasks have finished and a data synchronization have been done. ........................................... 11

2.2 COMPSs supported infrastructures. COMPSs runtime make the programmer unaware of the application parallelization among different infrastructures such as grids, clusters and clouds. .............. 11

2.3 Example of a Task Dependency Graph generated by COMPSs. .... 14

2.4 Task Dependency Graph of Simple application example. ......... 15

2.5 Detail of PyCOMPSs. The Python binding uses a C++ library to connect through JNI to the COMPSs Runtime that is a Java native application. ......................................................... 16

2.6 Disk structure: A) Track, B) Geometrical sector, C) Track sector, D) Cluster. ........................................................... 19

2.7 Storage Primitives. A possible set of primitives an application could use to access files, objects, or persistent objects. Simplifying, the programming language and operative system could store and provide access to objects in primary (main) memory or persistent objects and files in secondary memories. ................................. 21

2.8 Self-Contained Object abstraction. In object oriented programming languages normally there is a File type (class) to deal with files and an Object type (class) to deal with objects. In the same way, the Self-Contained Object type (class) allows to interact with Persistent Objects. ..................................................... 22

2.9 Traditional data models for databases. ............................... 23

2.10 Example of a Select SQL statement. ................................. 24

2.11 Basic data structures in memory. Data items are adjacent or linked. 25

2.12 Self-Contained Objects in dataClay. Object data, code and behaviour policies: security, privacy, integrity, etc. is provided by a SCO. ................................................................. 26

2.13 3rd-party Enrichment in dataClay. .................................. 27

2.14 Self-Contained Objects can be off-loaded to other infrastructures. 28
2.15 On the left, Hecuba is deployed as a ring cluster. On the right, Persistent Objects are organized in blocks within a key-value table. Elements on each block can be accessed with a key iterator.

3.1 Storage API. Provide interfaces for both, application and middleware, with the storage layer.

3.2 Self-Contained Object and Storage Interfaces.

3.3 File descriptors. In the figure the operative system uses a file descriptor to index a table with file permissions and later a table with extended file information. The file descriptor is a number in this example, the filename is stored in a secondary table with other file information.

3.4 A Self-Contained Object with wrappers to class properties and methods stored in a Persistent Object.

3.5 New Replica scenario. 3 Self-Contained Objects, each on a different resource, give access to a Persistent Object and its 2 replicas respectively.

3.6 New Version scenario. Self-Contained Objects on different resources where each gives access to a version or a replica of a Persistent Object.

4.1 COMPSs 1.3 Software Packages

4.2 JavaGAT structure.

4.3 Java NIO channels. Channels read data into buffers, and buffers write data into channels.

4.4 Java NIO threads. A Thread uses a Selector to handle 3 Channel’s

4.5 COMPSs Software Packages. Engine Package.

4.6 Make Persistent SCO interface method. The \texttt{<makePersistent>} method of a SCO can be called from the main application but also from a task.

4.7 The \texttt{<PSCOId>} class.

4.8 Execution flow diagram of a COMPSs application.

5.1 Development infrastructure. Hosts used from BSCGrid cluster.

8.1 Benchmark phases of the File and dataClay versions.

8.2 Task Dependency Graph (TDG) of the File and dataClay benchmark versions with 16 objects and 16 SCOs respectively.

8.3 Memory profile of a Job that exceeds the node memory limit.

8.4 Memory profile of a Job that fits the node memory limit.

8.5 Molecule class.

8.6 Memory profile for 1 molecule. Third column shows the number of allocated bytes and fourth column shows the number of allocated objects.
List of Tables

2.1 STARSs programming models family. OMPSs/COMPSs application scope. ........................................... 10
2.2 Arguments of the `<@task>` decorator ................................................................. 17
2.3 Memory & Storage Hierarchy in Large Systems. Left devices are closer to CPU. ........................................... 20
8.1 Number of atoms, File and Memory size. ....................................................... 81
8.2 Number of transfers due to a data locality miss with high cost. ... 89
Chapter 1

Introduction

1.1 Motivation

Challenges in science for the next 2020 horizon in areas like: Medicine, with advances in personalised medicine to provide better diagnosis and treatments to each patient. Biology, with modeling and simulation of biological systems at multiple scales from atomic, through genomic and cellular to ecosystems, for better understanding of processes such as cell growth, metabolism, locomotion or sensing. Aerospace, with high fidelity simulations to reduce the number of expensive real-life experiments and save costs. Astrophysics, with direct comparisons between complex simulations and large-scale astronomical surveys of distant galaxies and quasars that will help to improve our understanding about the Universe. Climate, with models that synthesize observations and theories of the Earth system as accurately as possible for a better research on climate change and weather prediction. Social sciences, with smart cities that aims to improve citizen lives at different levels: transport information, formalities, different indicators e.g. for air pollution, etc. Which are only some examples, will not only require large-scale systems that break the exaflop barrier, called Exascale, that is, $10^{18}$ floating point operations per second, but also able to process big amounts of data, that is Big-Data, from sensors, surveys or other data sources.

The emerging Exascale computing architecture will not be simply 1000 x today’s petascale architecture. All proposed Exascale computer systems designs will share some of the following challenges: Processor architecture is still unknown. System power is the primary constraint for the Exascale system: simply scaling up from today’s requirements for a petaflop computer, the exaflop computer in 2020 would require 200 MW, which is untenable. The target is 20-40 MW in 2020 for 1 exaflop. Memory bandwidth and capacity are not keeping pace with the increase in flops: technology trends against a constant or increasing memory per core. Although the memory per flop may be acceptable to applications, memory per processor will fall dramatically, thus rendering some of the current scaling approaches useless. Clock frequencies are expected to decrease to conserve power; as a result, the number of processing units on a single chip will have to increase –
Chapter 1. Introduction

this means the Exascale architecture will likely be high-concurrency – billion-way concurrency is expected.

Cost of data movement, both in energy consumed and in performance, is not expected to improve as much as that of floating point operations, thus algorithms need to minimize data movement, not flops. Programming model will be necessary: heroic compilers will not be able to hide the level of concurrency from applications. The I/O system at all levels – chip to memory, memory to I/O node, I/O node to disk—will be much harder to manage, as I/O bandwidth is unlikely to keep pace with machine speed. Reliability and resiliency will be critical at the scale of billion-way concurrency: “silent errors,” caused by the failure of components and manufacturing variability, will more drastically affect the results of computations on Exascale computers than today’s petascale computers. These challenges represent a change in the computing cost model, from expensive flops coupled with almost free data movement, to free flops coupled with expensive data movement.

On one hand, programming models try to bridge the gap between the underlying hardware architecture and the supporting layers of software available to applications. Programming models are different from both programming languages and application programming interfaces (APIs). Specifically, a programming model is an abstraction of the underlying computer system that allows for the expression of both algorithms and data structures. In comparison, languages and APIs provide implementations of these abstractions and allow the algorithms and data structures to be put into practice—a programming model exists independently of the choice of both the programming language and the supporting APIs. The most obvious design goal for programming models is to support massive degrees of parallelism. Although some aspects of future programming models are likely to resemble those used on today’s large-scale systems, those models and their implementations must support the expression and management of a significantly greater level of concurrency within a system that is composed of nodes consisting of processors with hundreds to thousands of cores.

On the other hand, the recent advances in data storage technology with Non-Volatile RAM (NVRAM) or Storage-Class Memories (SCM) devices, with low latencies, high bandwidth and byte-addressable interface, are promising for Big-Data applications that need high IO performance. A new generation of Object Storage Platforms is emerging to take full advantage of this high-performance and byte-addressable storage devices. This new object storage platforms store data as objects as understood in the object-oriented programming paradigm and opposed to traditional data management systems that store data as files or in databases. A single data model will work for both, in-memory and for stored data, enabling a tight integration with programming models.
In this master thesis it will be shown that the marriage of programming models and object storage platforms is a powerful framework to deal with the Exascale and Big-Data issues. Experiments run in the Marenostrum 3 supercomputer, at the Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS), show that the integration of both technologies, programming models and object-storage platforms, could reduce considerably the execution time of scientific applications.

The COMPSs programming model, which is the main research line of the Workflows and Distributed Computing at BSC-CNS, has been integrated with two objects storage platforms, dataClay and Hecuba, that are the current projects of the Storage System group and the Autonomic Systems and e-Business Platforms group at BSC-CNS respectively. The collaboration and coordination of all the groups and the support team of Marenostrum has been crucial to achieve the objectives of this project.

The work described here has been focused mainly in the integration of COMPSs with dataClay. The integration with Hecuba was also done and it is commented in this document, but its installation in the Marenostrum supercomputer is currently in progress, therefore the experiments and the results analysis have been done based on the integration of COMPSs with dataClay.

## 1.2 Objectives

Objectives in this master thesis are:

- **Storage Concepts Definition**: What is a Persistent Object and the Self-Contained Object abstraction, and other concepts related to new object storage platforms, are defined in this document.

- **Storage API Specification**: The specification of a unique API for object storage platforms that allows the integration with programming models, like COMPSs. Each object storage platform must provide an implementation of this API.

- **Integration Performance Study**: Different experiments have been run in Marenostrum supercomputer to study the actual performance of the integration in a production system.
1.3 Methodology

The steps to achieve the previous objectives have been:

1. A study of concepts like Persistent Objects or the Self-Contained Object abstraction and new features in object storage platforms, like dataClay and Hecuba, have been done to understand how they work.

2. A reverse engineering study to understand the programming model and runtime (middleware) of COMPSs and identify the requirements to integrate it with the new object storage platforms.

3. Based on previous requirements, a Storage API specification have been designed where each object storage platform must provide an implementation of this API.

4. Unit tests have been developed to validate the Storage API implementation, provided by an object storage platform, independently of COMPSs. This tests simulate a basic and complex use of the Storage API. These tests have been used to do a first performance tuning of the Storage API implementation.

5. The last release of COMPSs (release 1.3) has been extended to support Persistent Objects through the Storage API.

6. The integration of COMPSs with the object storage platforms have been validated first on a development infrastructure using a synthetic benchmark. At this point several concurrency and communication problems were detected and a tuning process of the integration and Storage API implementation was done until a good performance was obtained.

7. After this, the porting of the integration to a final production infrastructure like the Marenostrum supercomputer was done. At this point different issues related to the execution in a supercomputer were solved.

8. Finally, different experiments have been run in Marenostrum to study the actual performance of the integration in a production system. A result analysis have been done and reported in this document.
1.4 Planning

The Master Thesis is worth 30 ECTS credits. ECTS stands for European Credit Transfer System and is the standard definition of credit adopted by all the universities of the European Higher Education Area (EHEA) to guarantee the homogeneity and the quality of the studies on offer.

ECTS is based on the principle that 60 credits measure the workload of a full-time student during one academic year. The student workload of a full-time study programme in Europe amounts in most cases to around 1500-1800 hours per year and in those cases 1 credit stands for around 25 to 30 working hours. An amount of $30 \times 30 = 900$ hours was initially planned for this Master Thesis.

This master thesis have been done as a partial time work together with other projects at BSC-CNS with an average of 4 hours per day dedicated to it. Therefore, $900 / 4 = 225$ labour days were estimated to its completion. This estimation have been enough as shown in figure 1.1 as an average of 210 labour days were spent.
<table>
<thead>
<tr>
<th>Task Name</th>
<th>Start Date</th>
<th>End Date</th>
<th>Duration</th>
<th>Q3</th>
<th>Q4</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Thesis</td>
<td>07/01/15</td>
<td>04/22/16</td>
<td>203d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proposal Registration</td>
<td>07/09/15</td>
<td>07/16/15</td>
<td>1d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object Storage Platforms Study</td>
<td>07/13/15</td>
<td>07/19/15</td>
<td>5d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPS+ 1.3 Study</td>
<td>07/20/15</td>
<td>07/31/15</td>
<td>10d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage API Specification</td>
<td>08/03/15</td>
<td>08/07/15</td>
<td>5d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPS+ 1.3 Extension</td>
<td>08/10/15</td>
<td>08/14/15</td>
<td>30d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage API Implementation</td>
<td>08/10/15</td>
<td>08/31/15</td>
<td>30d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summer Holidays</td>
<td>08/17/15</td>
<td>08/28/15</td>
<td>6d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage API Unit Testing</td>
<td>09/02/15</td>
<td>10/06/15</td>
<td>15d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integration Test</td>
<td>10/12/15</td>
<td>12/21/15</td>
<td>51d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Christmas Holidays</td>
<td>12/22/15</td>
<td>01/01/16</td>
<td>15d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Posting to MNO Supercomputer</td>
<td>01/12/16</td>
<td>02/05/16</td>
<td>34d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MNO Maintenance</td>
<td>02/29/16</td>
<td>03/11/16</td>
<td>10d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MT Document</td>
<td>02/29/16</td>
<td>04/04/16</td>
<td>30d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experimental Analysis</td>
<td>04/14/16</td>
<td>04/25/16</td>
<td>20d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easter Holidays</td>
<td>03/27/16</td>
<td>03/28/16</td>
<td>1d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MT Presentation</td>
<td>03/27/16</td>
<td>04/08/16</td>
<td>11d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MT Defense Week</td>
<td>04/10/16</td>
<td>04/22/16</td>
<td>5d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.5 People

This Master Thesis has been a collaborative work between different research groups at Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS).

Main research objectives for each of them are:

1.5.1 Workflows and Distributed Computing

The Workflows and Distributed Computing group at BSC-CNS does research and development in programming environments for grid, clusters and clouds. Specifically, the objective is to offer tools and environments that enable to develop applications that are transparent to the underlying computational platform. What is more, techniques as concurrency exploitation will be applied to efficiently execute the application in the distributed resources.

1.5.2 Storage System

The Storage System group at BSC-CNS researches new storage abstractions to organize and store data better suited for Exascale and Big-Data challenges, implements production-quality middleware solutions based on the proposed new abstractions to mitigate the Exascale and Big-Data problems and evaluates the potential of virtualization to improve both efficiency and usability of storage systems.

1.5.3 Autonomic Systems and e-Business Platforms

The Autonomic System and e-Business Platforms group at BSC-CNS researches in Data-driven Scientific Computing. The goal of this area is to design resource management strategies for Big-Data applications, defining policies that enable distributed data stores to meet high-level performance goals. Proposes novelty resource management strategies as query-driven data model, which focus on adapting the data model to the particular type of accesses implemented by the applications.
Chapter 1. Introduction

1.6 Document Organization

This Master Thesis document is organized in the following chapters:

1. **Introduction**: Master thesis motivation, objectives, methodology, planning, people and document organization are described here.

2. **Related Work**: Current research in parallel programming models at BSC-CNS is introduced in this chapter. The COMPSs and PyCOMPSs programming models are explained with Java and Python examples respectively. Persistent Objects and Self-Contained Object concepts are explained. Two research projects at BSC-CNS in object storage platform, dataClay and Hecuba, are also described in this chapter.

3. **Storage API**: This chapter contains the Storage API Specification. Storage API interfaces and methods are described with detail.

4. **Integration**: Describes the integration process of the last release of COMPSs and PyCOMPSs programming models with the Storage API.

5. **Infrastructure**: The infrastructures used for testing during development and for the experimental analysis are described in this chapter.

6. **Compilation**: This chapter explain the compilation process of an application that will be executed with COMPSs and use an object storage platform for object persistence.

7. **Deployment**: Describes the deployment components of COMPSs, dataClay and Hecuba.

8. **Analysis**: The benchmark used for the experiments is described here. Some previous consideration about the experiments are also done. Finally, a result analysis is done from different point of view.

9. **Conclusions**: Final remarks about the work done for this Master Thesis are presented here.

10. **Future Work**: Some proposals of future work can be found in this chapter.

11. **Annexe**: Storage API and Benchmark source code.

12. **References**
Chapter 2

Related Work

2.1 Programming Models

2.1.1 STARSs

Nowadays, there are many heterogeneous systems, symmetric multiprocessing systems (SMPs), graphic processor units (GPUs), clusters, clouds, and more to come in the next years. Getting productivity in application development and execution performance for such systems is really a challenge.

STARSs (* Superscalar) is a family of programming models that provides programmers an interface to run the “same” source code on “any” target architecture.

Main characteristics of the STARSs programming models family are:

- Sequential code with annotations: The programmer don’t need to change the already written source code to run the application in parallel. The programmer only has to add some annotations to the already written sequential code.

- Task-Based: Annotations are use mainly to specify which functions or methods in the source code are tasks. A task can be seen as the parallel execution unit of the programming model. Each time the desired function or method is called a task is created in the system to be executed asynchronously and in parallel with other existing tasks.

- Single logical address space: Task parameters are variables accessed through a reference or address. Such references are used to detect dependencies between tasks. Although tasks are executed in non-shared memory machines, a single logical address space is managed by the programming model.

- Different target architectures: The programming model allows the application to run on different architectures and the programmer doesn’t need to be aware about the details of the underlying execution architecture or infrastructure.
Chapter 2. Related Work

There are two main approaches in the STARSs programming model family. One oriented to shared-memory systems called OMPSs. Another mainly for distributed systems with name COMPSs. A task in OMPSs can last from microseconds to milliseconds while in COMPSs they can last from seconds to a day. In OMPSs the logical address space are memory addresses, instead in COMPSs the address space are filenames or object references. OMPSs is oriented to scientific languages like C/C++ or Fortran and architectures like SMPs or GPUs. COMPSs is oriented to newer languages like Java or Python, but also C/C++ is supported. COMPSs usual targets are clusters or clouds.

<table>
<thead>
<tr>
<th></th>
<th>STARSs</th>
<th>COMPSs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average task granularity</strong></td>
<td>100us - 10ms</td>
<td>1s - 1day</td>
</tr>
<tr>
<td><strong>Domain to compute dependencies</strong></td>
<td>Memory addresses</td>
<td>Files, Objects</td>
</tr>
<tr>
<td><strong>Languages supported</strong></td>
<td>C, C++, Fortran</td>
<td>Java, C, C++, Python</td>
</tr>
<tr>
<td><strong>Target infrastructure</strong></td>
<td>SMPs, GPUs, Clusters</td>
<td>Clusters, Clouds</td>
</tr>
</tbody>
</table>

Table 2.1: STARSs programming models family. OMPSs/COMPSs application scope.

2.1.2 COMPSs

COMP Superscalar[2][1][5] (COMPSs) is a programming model for development and also a runtime system for parallel execution of applications in a set of distributed resources.

2.1.2.1 Programming Model

The native programming language of COMPSs programming model is Java, but also can be use with C/C++ and Python applications.

The most important feature of COMPSs programming model is that allows to easily convert a sequential code into a parallel code just adding some annotations in the code. In the case of Java and C/C++ programming languages such annotations are written in a new file, called the Task Selection Interface file, and there is no need to modify the already-written sequential code of the application. In the case of Python such annotations are written as Python decorators in the sequential code. Each annotation indicates a method or function in the sequential code to be executed parallel by the COMPSs runtime.
Figure 2.1: COMPSs programming model. As the main application thread is executed tasks are generated and executed in parallel. The main thread doesn’t end until all tasks have finished and a data synchronization have been done.

Therefore, with COMPSs, a programmer can write its application as is going to be executed sequentially and later can easily make it parallel just adding some annotations.

2.1.2.2 Runtime System

The parallel and distributed execution of an application needs to deal with issues like thread creation, thread synchronization, data distribution, messaging or fault tolerance, among others things. The COMPSs runtime manages all these issues and the programmer doesn’t need to be aware of them when writing the application.

Figure 2.2: COMPSs supported infrastructures. COMPSs runtime make the programmer unaware of the application parallelization among different infrastructures such as grids, clusters and clouds.
Chapter 2. Related Work

The Runtime system also abstracts the application from the desired underlying infrastructure. With a little effort the programmer can specify and/or change the infrastructure used. COMPSs runtime supports distributed infrastructures such as Grids, Clusters and Clouds.

2.1.3 Java Example

COMPSs has a premise that is not to modify if possible the code of an existing application to run it in parallel. In case of the Java language this can be achieved by means of the Task Selection Interface. The Task Selection Interface, which is explained later, is a new file created to declare which are the methods of the application that are going to be tasks in the parallel execution. The application code is not modified at all.

The following Java example is the Simple application. In this application three file names are passed as arguments. Each file is a counter. Counters are initialized. The three counters are incremented in a loop. The loop has three iterations. On each iteration the three counters are incremented by one unit. The main program is written in the <Simple> java class.

```java
public class Simple {
    /* Main Program */
    public static void main(String[] args) {
        String counter1 = args[0], counter2 = args[1], counter3 = args[2];
        initializeCounters(counter1, counter2, counter3);
        for (i = 0; i < 3; i++) {
            increment(counter1);
            increment(counter2);
            increment(counter3);
        }
    }
}
```

The <increment> method is implemented on another class called <SimpleImpl>. It reads a counter file, increment the counter value and writes it back into the counter file.

```java
public class SimpleImpl {
    /* Subroutine */
    public static void increment(String counterFile) {
        int value = readCounter(counterFile);
        value++;
    }
}
```
Chapter 2. Related Work

writeCounter(counterFile, value);}
}

2.1.3.1 Task Selection Interface

The Task Selection Interface is a Java annotated interface used to declare the application methods to be executed as tasks. The annotations specify the necessary metadata about a task. The metadata can be of three different types:

- For each parameter of a method, the datatype (currently File type, primitive types and the String type are supported) and its direction (IN, OUT or INOUT).
- The Java class that contains the code of the method.
- The constraints that a given resource must fulfill to execute the method, such as the number of processors or the main memory size.

To run the Simple application parallel the Task Selection Interface is written on a Java interface called <SimpleItf>. In this interface, the method <increment> has been declared as a task.

/* Task Selection Interface */
public interface SimpleItf {

    @Method(declaringClass = "SimpleImpl")
    @Constraints(processorCoreCount = 4, memoryPhysicalSize = 1.5f)
    void increment(
        @Parameter(type = FILE, direction = INOUT)
        String counterFile
    );
}

The annotation <@Method> tells COMPSs that the task is a Java method. The argument <declaringClass> indicates the class the method is implemented.

The annotation <@Constraints> is used to specify requirements on the resources where the task can be executed. For example, the argument <processorCoreCount> indicates the number of cores and the argument <memoryPhysicalSize> indicates the minimum amount of memory the resource must have.

For each method parameter, the annotation <@Parameter> is used to indicate the parameter type and direction. The direction is used by COMPSs to find out data dependencies and create a Task Dependency Graph.
2.1.3.2 Task Dependency Graph

In mathematics and computer science, a directed acyclic graph (DAG), is a directed graph with no directed cycles. That is, it is formed by a collection of vertices and directed edges, each edge connecting one vertex to another, such that there is no way to start at some vertex $v$ and follow a sequence of edges that eventually loops back to $v$ again.

DAGs may be used to model many different kinds of information. A collection of tasks that must be ordered into a sequence, subject to dependencies that certain tasks must be performed earlier than others, may be represented as a DAG with a vertex for each task and an edge for each dependency.

COMPSs runtime manages internally a DAG of task dependencies called Task Dependency Graph. Dependencies of a task are those task parameters that are an output of a previous task. Once dependencies have been resolved, that is, task parameters are available, the task is submitted to execution. COMPSs allows to see the complete Task Dependency Graph at the end of the parallel execution with the following command line option.

```
--graph=<bool>, --graph, -g
```

Generation of the complete graph (true/false)
When no value is provided it is set to true
Default: false

For the Simple application example, the Task Dependency Graph is shown in Figure 2.4. On each iteration, three $<\text{increment}>$ tasks are executed in parallel, each of them works on a different counter file. The arrows show the dependencies.
between tasks of different iterations that work on the same counter file.

![Task Dependency Graph of Simple application example.](image)

Figure 2.4: Task Dependency Graph of Simple application example.

### 2.1.4 PyCOMPSs

Python is a powerful object-oriented programming language comparable to Perl, Ruby or Java.

Python is an easy-to-use language that makes it simple to get your program working. It is not only ideal for prototyping but also for bigger projects, as it comes with libraries that supports many common programming tasks and it can be easily extended by adding other libraries written in languages such as C or C++. It is a free software and it can run on many computers architectures and operating systems.

Despite all these advantages, big projects in the scientific community need to be parallelisable. Current approaches to run parallel a Python application spawn multiple Python processes and requires the programmer to include infrastructure related details in the application.

PyCOMPSs\[4\]|\[3\], built on top of COMPSs, is a parallel programming framework for Python applications that overcomes the aforementioned limitation of other approaches as the programmer keeps unaware of the underlying infrastructure.
Figure 2.5: Detail of PyCOMPSs. The Python binding uses a C++ library to connect through JNI to the COMPSs Runtime that is a Java native application.

Figure 2.5 shows how the Python language has been integrated in COMPSs. A Python module called here Python binding offers the COMPSs features to the Python application. For that, the Python binding uses a C++ library that communicates through the Java Native Interface (JNI) with the COMPSs runtime.

One of the contributions of this master thesis has been the development and upgrade of such C++ library. The C++ library, also called binding common library, allows the current C/C++ and Python bindings to invoke the features offered by the COMPSs runtime, such as asynchronous task execution. Moreover, the common binding library is key to support in the future other programming languages, like for example, the R programming language.

2.1.5 Python Example

As example for PyCOMPSs, let’s use an equivalent Simple Python application to the previous one written in Java for COMPSs. Here is presented the main code:

```python
def main_program():

    # Check and get parameters
    if len(sys.argv) != 5:
        usage()
        exit(-1)

    N = int(sys.argv[1])
    counter1 = int(sys.argv[2])
    counter2 = int(sys.argv[3])
    counter3 = int(sys.argv[4])
```
# Initialize counter files
initializeCounters(counter1, counter2, counter3)
print "Initial counter values:"
printCounterValues()

# Execute increment
for i in range(N):
    increment(FILENAME1)
    increment(FILENAME2)
    increment(FILENAME3)

# Write final counters state (sync)
print "Final counter values:"
printCounterValues()

## 2.1.5.1 Task Decorators

Task declaration in PyCOMPSs is done by means of Python decorators, which are part of the standard Python syntax and permit to wrap calls to functions and add some additional code.

In particular, the programmer needs to add, before the definition of the function, a `@task` decorator that describes the task. In PyCOMPSs, the user can define as a task:

- Functions
- Instance methods: methods invoked on objects.
- Class methods: static methods belonging to a class.

Task description is done through a set of arguments summarized in Table 2.2.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>returns</td>
<td>int (for integer and boolean), long, float, str, dict, list, tuple, user-defined classes.</td>
</tr>
<tr>
<td>isModifier</td>
<td>True (default) or False</td>
</tr>
<tr>
<td>priority</td>
<td>True or False (default)</td>
</tr>
</tbody>
</table>

Table 2.2: Arguments of the `@task` decorator
There is no need to include an argument in the `<@task>` decorator for a given parameter if:

- It is a primitive type (integer, long, float, boolean) or a string, its type can be automatically inferred and its direction is always IN.
- It is a read-only object, its type is automatically inferred and the direction defaults to IN.

Similar to the Java example, in the Python Simple example the `<increment>` function is declared as a task and its unique argument is described as an in-out file.

```python
@task(filePath = FILE_INOUT)
def increment(filePath):
    # Read value
    fis = open(filePath, 'r')
    value = fis.read()
    fis.close()

    # Write value
    fos = open(filePath, 'w')
    fos.write(str(int(value) + 1))
    fos.close()
```

The graph of task dependencies generated in the case of the Python Simple example is exactly the same as the Java Simple one.

For more information about PyCOMPSs refer to the COMPSs User Manual for Application Development[2].

### 2.2 New Object Storage Platforms

As mentioned before, new object storage platforms are emerging to get benefit from new byte-addressable non-volatile memory devices. In this section, concepts and features of this kind of storage technology are explained.

#### 2.2.1 Persistent Objects

Since ever, data has been stored in files or databases. Data has been treated as a 2nd class citizen where the programming code has been 1st class one. Data has its own primitives to be accessed and they are different from the ones used to
access objects in commonly used object oriented programming languages.

When hard disk drives (HDDs) are used, the data must be stored in terms of blocks. To access the data stored in this way from an application, the file abstraction is used by operative systems and programming languages. The sector is the minimum storage unit of a HDD. All files must occupy at least one sector, regardless of the file’s actual size. However, operating systems typically operate on blocks of data, which may span multiple sectors.

![Figure 2.6: Disk structure: A) Track, B) Geometrical sector, C) Track sector, D) Cluster.](image)

A file needs its own primitives (open, read, write, close, delete) based on blocks to be accessed. The file’s access primitives are different from the ones used to access objects in the main (RAM) memory (new, get, set, delete).

The access to data stored in files becomes pretty inefficient as the application data grows. Then databases appear in this scenario as a more efficient abstraction to access data stored in HDDs. Each Database offer a query language to access the data stored on it. But again the primitives offered by the query language (select, insert, update, delete) are different from those used to access objects in main (RAM) memory.

Anyway, if hard disk drives (HDDs) are used to finally stored data, the access to data provided by this kind of technology is always slower than the access to RAM memories, see table 2.3.
Chapter 2. Related Work

Memory type | SRAM | DRAM | NVRAM | SSD | HDD | TAPE
---|---|---|---|---|---|---
Use | Registers, caches | Main memory | Storage | Storage | Storage | Long-term storage
Byte-addressable | Yes | Yes | Yes | No | No | No
Access time | 1ns / 5ns | <10ns | 10ns / 50ns | 20us | 5ms | 40s
Volatile | Yes | Yes | No | No | No | No
Cost | High | High | High | Low | Low | Low
Integrated-on-chip | Yes | Yes | Both | Both | No | No
Failure | Low | Low | Low | High | Low | Low

Table 2.3: Memory & Storage Hierarchy in Large Systems. Left devices are closer to CPU.

New storage devices, such as non volatile memories (NVRAMs), are byte addressable. This means that objects could be stored in such devices as they are in the main (RAM) memory, and therefore, they could be accessed from the application with the same primitives (new, get, set, delete).

The technology used in NVRAM memories provides an access time that is closer to the RAM memories than HDDs or solid state drives (SSDs) access time, see again table 2.3.

With this new Storage Class Memory (SCM) devices, objects could be stored in secondary memory, non-volatile memory or also called persistent memory, as they are in main memory. And, as files are for HDDs, for persistent memories a new abstraction appears called Persistent Objects.

A Persistent Object is a regular object, previously in RAM, memory that has been stored in a persistent memory.
2.2.2 Self-Contained Objects

Object oriented programming languages commonly provide (a) type(s), or (a) class(es), to deal with files stored in secondary memories and bring them to the main memory so the application could open, read, write, close or delete a file. Normally, files stored in secondary memories are big, and cannot be full loaded into the main memory, therefore only the part currently needed by an application is brought to main memory for it use.

In the same way as files, a type (class) is needed to deal with persistent objects stored in secondary memories. This type (class), that could be provided by a programming library, is called here Self-Contained Object type (class). A Self-Contained Object (SCO) brings to main memory, not the full Persistent Object (PO) but only the data currently needed by the application.

In an application, as done with a File object, we could create a new SCO in main memory, write on it, and store it as a PO in secondary memory. Later, the SCO allows to access the PO stored in secondary memory by its identifier, and will bring to main memory only the requested data.
Figure 2.8: Self-Contained Object abstraction. In object oriented programming languages normally there is a File type (class) to deal with files and an Object type (class) to deal with objects. In the same way, the Self-Contained Object type (class) allows to interact with Persistent Objects.

As the File type (class) offers properties and methods to deal with files, like a method for read til the next carrier return in the file or a method to ask for the end of the file, the Self-Contained Object type (class) also could offer properties and methods to work with persistent objects as we will see later.

2.2.3 Traditional Databases

Data is typically organized to model aspects of reality in a way that supports processes requiring information. Commonly, such model is called Data Model and the organized data is called Database. That is, a data model describes the structures and relationships in a database and how the database is to be used.

Several data models have been suggested:

- Flat model: This may not strictly qualify as a data model. The flat (or table) model consists of a single, two-dimensional array of data elements, where all members of a given column are assumed to be similar values, and all members of a row are assumed to be related to one another.

- Hierarchical model: In this model data is organized into a tree-like structure, implying a single upward link in each record to describe the nesting, and a
sort field to keep the records in a particular order in each same-level list.

- Network model: This model organizes data using two fundamental constructs, called records and sets. Records contain fields, and sets define one-to-many relationships between records: one owner, many members.

- Relational model: It is a database model based on first-order predicate logic. Its core idea is to describe a database as a collection of predicates over a finite set of predicate variables, describing constraints on the possible values and combinations of values.

- Object-relational model: Similar to a relational database model, but objects, classes and inheritance are directly supported in database schemas and in the query language.

- Star schema: The simplest style of data warehouse schema. The star schema consists of a few "fact tables" (possibly only one, justifying the name) referencing any number of "dimension tables". The star schema is considered an important special case of the snowflake schema.

Data in a database is managed by means of a Query Language. For example, SQL (Structured Query Language) is a standard interactive and programming language for getting information from and updating a database.
Finally, a database management system (DBMS) is a computer software application that interacts with the user, other applications, and the database itself to capture and analyze data. A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases.

### 2.2.4 dataClay

dataClay\[6\][7] is a new storage platform for Persistent Objects. dataClay has the following pillars:

**A unique data model**

Nowadays, there is a data model for volatile data, objects and their relationships in primary memory, and another different data model for non-volatile data, persistent data or data stored in secondary memory.

This means waste time doing things twice. Data should be stored in persistent memory in the same way as when it is in volatile memory, that is as objects and relationships between them.

**Query as in (volatile) memory**

In primary memory, data is never queried like in a database. The data in primary memory has been linked according to the needs of programs. Normally, the next data item is found following a memory link or reference, not using a query as in a database. Following links is faster than a query over a whole database.

![Example of a Select SQL statement.](image)
When querying data, programs should not make any difference whether data is in memory or in persistent storage.

**Ready for new storage devices**

As mentioned in section 2.3, new storage class memories (SCMs) are non-volatile memories (NVRAMs) with a performance between main memory (RAM) and solid state drives (SSDs).

File systems or table-based databases are not the right abstraction as they both were designed to use block devices. They can be used also on storage class memories (SCMs) but it would be an important loss of potential as these memories are byte-addressable what means that objects could be stored in SCMs as they are in main memory.

**Coupling data and computation**

dataClay stores object data, code and behaviour policies that define issues as security, privacy, integrity and life cycle among others. This differs from other approaches that offer objects as the persistent abstraction, such as object-relational mapping libraries or OODB that only stored the object data.

This integration of data and code enables, in a very simple way, to perform computation close to the data, thus enabling the programmer or the middleware to decide where a given computation has to be executed. As object data, code and behaviour policies are stored together, the object can be moved to different infrastructures an still behave as it was designed.
Self-Contained Objects

The implementation of Self-Contained Objects done in dataClay push the idea of data services to the limit.

![Diagram of Self-Contained Objects in dataClay](image)

Figure 2.12: Self-Contained Objects in dataClay. Object data, code and behaviour policies: security, privacy, integrity, etc. is provided by a SCO.

In a traditional schema, there is a centralised data service which not only provide access to a client application to data persisted in a data store but also deal with issues like code execution, data security, data integrity, etc. dataClay push the idea of such data service to the limit, based on the Object-Oriented paradigm, as now each Self-Contained Object (SCO) provide not only the access to the persisted data (or Persistent Object) but also to code execution, security and integrity issues related to a Persistent Object.

Data sharing

The value of data increases when it is shared. Unfortunately, today’s methods to share data are not optimal. Today we have three basic mechanisms to share data.

- We can share the data by giving access to the different actors to the shared data; this option is very flexible but also dangerous because the owner of the data cannot limit what can be done to the data in a fine grain way.

- We can copy the data from the owner infrastructure to the infrastructure of the consumer. This option is also very flexible but incurs in a few problems such as unnecessary data copies, the need of keeping the data up to date, and most importantly the loss of control of the data owner over its data.
• We can encapsulate the data, and the rest of actors interact with the data using the existing options. This last approach has the problem that new operations on data need to be implemented by the data owner, who may not have the resources or interest to implement it.

3rd-party Enrichment

In order to improve data sharing, dataClay offers a mechanism to enrich data by third parties (not the data owner) without removing the control from the data owner. By enrichment we understand the modification of the data model (adding new classes, or fields, to exiting classes), the attachment of new code to the data (adding new methods, new versions of exiting methods, or even new behaviour policies), and of course, the addition of new data objects to the data set. For instance, this functionality enables researchers to adapt data from other research groups without unnecessary duplications and without taking the control of the data from the data owner.

Figure 2.13: 3rd-party Enrichment in dataClay.

This enrichment should:

• Be possible during the life of data
• Not be limited to the data owner
• Enable different views of the data to different users/clients.

Several enrichments should be available concurrently, and not everybody should see the same enrichments.

Distributed Infrastructure
Using a single infrastructure may become a bottleneck. A distributed infrastructure, a federation of multiple infrastructures, is commonly needed for Big-Data applications. Data need to be offloaded to other infrastructures without breaking the data policies, therefore, security and privacy policies should be part of the data.

With dataClay enabling 3rd-party enrichment the data owner does not lose the control of the data. Policies are defined using a declarative language and are enforced as part of the Self-Contained Object. Self-Contained Objects can be offloaded then to resources not accessible by the data provider.

![Figure 2.14: Self-Contained Objects can be off-loaded to other infrastructures.](image)

### 2.2.5 Hecuba

Hecuba is a Python middleware that aims to facilitate programmers an efficient and easy interaction with non-relational databases.

Hecuba provides the implementation of the interface necessary for PyCOMPSs to work with a storage backend suitable to support Big-Data applications.

The storage backend used in Hecuba is Cassandra. Cassandra implements a non-centralized architecture, based on peer-to-peer communication, in which all nodes of the cluster are able to receive and serve queries. Each node of the cluster is assigned a token. A partitioner function uses this token to decide how data is distributed among the nodes in the shape of a ring.

Data in Cassandra is stored on tables by rows, which are identified by a key chosen by the user. Cassandra stores data by rows, and one node is responsible for hosting a specific row. The target node is chosen based on the key of the row.
and of the token of each node by the partitioner algorithm. In order to guarantee data availability, Cassandra can be configured to keep several replicas for all data.

![Diagram of Hecuba deployment]

Figure 2.15: On the left, Hecuba is deployed as a ring cluster. On the right, Persistent Objects are organized in blocks within a key-value table. Elements on each block can be accessed with a key iterator.

The mapping of a Python dictionary on a data model in Cassandra is straightforward as both consist on values indexed by keys.

Each class contains one or more Persistent Dictionary that is mapped on a Cassandra table.

To implement a class backed up by Hecuba it is just necessary to indicate that the class is a subclass of `<StorageObj>`, which is a Hecuba class that implements the Self-Contained Object abstraction, and that contains all the methods that are necessary to access and manipulate data backed up by Cassandra, that is, Persistent Objects.

Thus, the code of the application is mostly independent of the data backend used. Notice that programmers can add their own methods to the class definition.

For more information about Apache Cassandra please refer to its website: [http://cassandra.apache.org/]
Chapter 3

Storage API

3.1 COMPSs and the New Storage Platforms

COMPSs is a middleware for parallel execution of applications on a set of distributed resources. The computation is distributed among the different resources based on data locality, this means, COMPSs tries the code to be executed in the resource where the data is stored.

The New Storage Platforms use the Self-Contained Object (SCO) abstraction to manage Persistent Objects (POs) in secondary memory.

To let an Application, parallelised with COMPSs, to work with Self-Contained Objects and therefore with the underlying Persistent Objects, an interface with these New Storage Platforms is needed at programming language level. Such interface is presented here as the Storage API. The Storage API is the main contribution in this master thesis.

Figure 3.1: Storage API. Provide interfaces for both, application and middleware, with the storage layer.
3.2 Storage API Specification

The Storage API is a software interface that allows both, application and COMPSs, to work with Self-Contained Objects (SCOs), and therefore with Persistent Objects (POs).

The storage platforms that store Persistent Objects have to implement this interface as part of their software stack, as the details on how Persistent Objects are stored depends on the storage platform itself.

In fact, the Storage API implies two interfaces, one for the application, the SCO Interface, and the other for the COMPSs runtime, the Storage Interface.

![Figure 3.2: Self-Contained Object and Storage Interfaces.](image)

### 3.2.1 SCO Interface

The SCO Interface allows an application to work with Persistent Objects as done with regular Objects. Those classes in the application code which objects the programmer wants to be persisted must extend the SCO Interface.

Objects that extend the SCO Interface become Self-Contained Objects. A Self-Contained Object instance is equivalent to a java.io.File instance in Java and a Persistent Object is like a regular file stored in secondary memory. A file in secondary memory is rarely accessed directly from an application, normally a class like the java.io.File in Java is used to open, close or delete the file or a class
like java.io.FileReader to read or java.io.FileWriter to write the data needed for the application on each moment, as normally the whole file doesn’t fit in main memory.

The same idea applies between Self-Contained Objects and Persistent Objects. The Self-Contained Object gives access only to the required property or lets invoke only the required method of the Persistent Object that the application needs on each moment.

The SCO Interface defines the following methods:

### 3.2.1.1 Make Persistent

This method is specified as `<void makePersistent(String name)>`, using a pseudo-code notation. It indicates to the Storage Platform that the Object in main memory must be stored as Persistent Object in secondary memory. As files have a name in secondary memory, a name for the Persistent Object is passed here as argument.

### 3.2.1.2 Get Identifier

It is Specified as `<String getID()>`, this method returns the identifier assigned to a Persistent Object by the Storage Platform. The identifier is alphanumeric and can be seen as a file descriptor different from the file name.

![File descriptors](image)

Figure 3.3: File descriptors. In the figure the operative system uses a file descriptor to index a table with file permissions and later a table with extended file information. The file descriptor is a number in this example, the filename is stored in a secondary table with other file information.

### 3.2.1.3 Delete Persistent

It is specified as `<void deletePersistent()>`, this method indicates to the Storage Platform that the object is not longer linked to a Persistent Object. The Persistent Object and its identifier in the Storage Platform are removed but the Self-Contained Object is still in main memory. At this point the programmer can
delete the Self-Contained Object or keep it in main memory to make it persistent again later on.

### 3.2.1.4 Wrappers

And the last but very important, the implementation of the SCO Interface must allow a mechanism to override or intercept the access to the original properties or methods declared in the class that extends the SCO Interface. Those properties or methods must appear also availables in the Self-Contained Object instance in main memory, but the real properties and methods are stored within the Persistent Object in secondary memory therefore the properties and methods in the Self-Contained Object are wrappers to the real ones.

![Figure 3.4: A Self-Contained Object with wrappers to class properties and methods stored in a Persistent Object.](image)

How a Persistent Object is stored in secondary memory depends on each Storage Platform like different operative systems store a file in different ways.
3.2.2 Storage Interface

The Storage Interface allows a middleware for distributed computing, like COMPSs, to work with Self-Contained Objects.

As seen COMPSs allows the programmer to divide the computation into tasks that are executed in parallel in a set of distributed resources. To be efficient COMPSs tries to send a task to the resource that has the data needed by the task, that is, the execution of tasks is driven by a data-locality policy.

In COMPSs each task is a selected application method (or function). COMPSs considers as data of a task the parameters and return value of the method. COMPSs will send the task to the resource that has more parameters stored in.

When any of the parameters or return value is a Self-Contained Object, COMPSs must use the implementation of the Storage Interface provided by the storage platform to work with them, for example, to know in which resource the Persistent Object is stored in order to apply its data-locality policy.

The Storage Interface specifies the following methods:

3.2.2.1 Init

In pseudo-code notation is defined as \(<\text{void init(String cfgfile)}\>). This method is used to pass a configuration file to the storage platform. The content and format of the configuration file depends on each storage platform.

These configuration parameters are used by the storage platform to create the necessary environment, for example, session variables, a connection pool or any other internal structure. Such environment is private for the storage platform and it is never exposed to the programmer.

If this initialization process done by the storage platform fails, for example, the configuration file is incorrect or incomplete, or any kind of connection problem happens, the storage platform must throw an exception with a useful description of the error.

3.2.2.2 Get by Identifier

Defined as \(<\text{Object getID(String id)}\rangle\), this method returns a Self-Contained Object that points to the Persistent Object with the identifier passed as argument.
If the identifier doesn’t correspond to a Persistent Object an storage exception must be thrown, like the "No such file or directory" exception for files.

3.2.2.3 New Replica

Defined as `<void newReplica(String id, String host)>`, this method tells to the storage platform to do a replica of the persistent object referenced by the identifier passed as argument, in the specified host.

A replica is a persistent object created in a different host, but has the same identifier as the original persistent object. A replica only can be done if the original persistent object is already in read-only state. The replica is created in read-only state also. Once a persistent object is in read-only state, it cannot be modified and the state cannot be change.

The replicas are managed by the storage platform and if a replica already exists in a host there is no need to replicate again on that host.

An exception must be thrown if an error happens when doing the replica, for example, there is no persistent object with the identifier passed, or the host doesn’t exists or cannot be reached, etc. The exception message should provide a useful description of the error.

![Figure 3.5: New Replica scenario. 3 Self-Contained Objects, each on a different resource, give access to a Persistent Object and its 2 replicas respectively.](image)

Notice that, on each resource, a replica only can be accessed throw a Self-Contained Object.

3.2.2.4 Get Locations

Defined as `<List<String> getLocations(String id)>`, this method returns a list with the locations, resource hostname or resource IP, of the persistent object with the identifier passed as argument. If the persistent object has replicas in different
resources, the location of all the replicas is also returned in the list.

At least one location, the location of the original persistent object, must be in the list. If any error happens a descriptive exception must be thrown.

3.2.2.5 New Version

Defined as \(<\text{String newVersion(String id, String host)>}\), this method tells the storage platform to create a copy of the persistent object referenced by the identifier passed in the specified host.

The persistent object to be copied must be in read-only state. If it has replicas it will be already in read-only state, but if it has no replica therefore must be set as read-only before doing the new version. Once a persistent object is in read-only it cannot be modified and its state cannot change.

The new persistent object, result of the copy, will have a new identifier different from the previous one. Initially, it will be in read-write state and have no replica, therefore it can be modified.

We use the term version, because, once the copy is modified it becomes a new version of the persistent object, that is, the data has changed. Also notice that, each version of the persistent object could have its own number of replicas.

If and error happens when doing a new version and exception with a descriptive message must be thrown.

---

Figure 3.6: New Version scenario. Self-Contained Objects on different resources where each gives access to a version or a replica of a Persistent Object.
3.2.2.6 Consolidate Version

This method is defined as `<void consolidateVersion(String id)>`. The term "consolidate version" indicates to the storage platform that the changes in the last version of a persistent object have to be merge into the original persistent object.

But, their identifiers are never merged, the original persistent object keeps its identifier unchanged. The identifier of the last version is passed as argument and the storage platform must know which is the original persistent object identifier. After the merging process, any previous version or replica is invalid and must be deleted.

If and error happens when doing the consolidate version process and exception with a descriptive message must be thrown.

3.2.2.7 Execute Task

This interface method indicates to the storage platform that a certain persistent object method must be executed by the storage platform.

It is defined as:

`<String executeTask(String id, String method, Object[] params, String host, CallbackHandler callback)>`

where, the argument `<id>` is the identifier of a persistent object, the argument `<method>` is the signature of the persistent object method to be executed, the argument `<params>` is an array with the parameters to be passed to such method, the `<host>` argument specifies the hostname or IP of the host where the method is going to be executed.

The execution of the persistent object method is done asynchronously by the storage platform. To know if the execution has been successful or not, a callback handler, `<callback>`, is passed as last argument to the `<executeTask>` call.

The callback handler will receive an callback event from the storage platform when the persistent object method execution ends with information about the execution, for example, if it has been successful or an error happened. On the other hand, the middleware could use the callback handler to wait for the returned object of the execution if needed.
3.2.2.8 Get Result

When a persistent object’s method is executed by the storage platform and such method returns an object, this object remains in the storage platform. The interface method \texttt{getResult()} allows the middleware to get the returned object from the storage layer if needed.

It is defined as \texttt{Object getResult(CallbackEvent event)}, where \texttt{Object} is the object that the persistent object method returns and \texttt{event} is the callback event sent by the storage platform to the callback handler when the persistent object method ends with information about the method execution.

3.2.2.9 Delete

This method is defined as \texttt{void delete(String id)}. This method tells to the storage platform to delete the Persistent Object with the identifier passed as argument. It is analogous to use a java.io.File object to delete a file in secondary memory. Notice that although the file has been deleted the java.io.File object is still in main memory, and the same applies between a Self-Contained Object in main memory that is used to delete a Persistent Object in secondary memory. If an error happens when deleting the Persistent Object a descriptive exception must be thrown.

3.2.2.10 Finish

Defined as \texttt{void finish()} this method indicates to the storage platform that no more accesses will be done to Persistent Objects, therefore, the storage platform can do any finalization task like closing connections or delete the environment variables, etc.

3.3 Unit Tests for the Storage API

A battery of Unit Tests have been developed to validate a Java implementation of the Storage API. Therefore this tests have been used to validate the dataClay Storage API implementation. There is a Unit Test for each method of the Storage API.

SCO Interface Unit Tests:

- TestConstructor
- TestMakePersistent
- TestDeletePersistent
Storage Interface Unit Tests:

- TestNewReplica
- TestNewVersion
- TestConsolidateVersion
- TestGetLocations
- TestDelete
- TestExecuteTask
- TestGetResult

There are also tests that simulate different application behaviours:

- TestSequentialApp: It simulates a sequential but iterative application where SCOs are created, replicated, versioned and finally consolidated.

- TestConcurrentApp: It simulates a concurrent application. Many threads are started at the same time. Each thread runs a sequential application. Therefore many independent applications are calling concurrently the Storage API implementation.

And a test to simulate a system under a big stress:

- TestStressedSystem: It is a concurrent test where not only many threads are started at the same time but also the number of SCOs to process is increased on different iterations trying to stress the system.
4.1 COMPSs 1.3

An important part of the work in this project has been the reverse engineering and analysis done with COMPSs version 1.3 in order to understand its internal structure and how it works. Understanding COMPSs 1.3 has been a necessary first step to later integrate it with Self-Contained Objects.

In this section COMPSs 1.3 structure and main features are briefly described.

4.1.1 Release Features

COMPSs 1.3 has been released recently in November 2015. This release has a complete different approach on its implementation from the previous one COMPSs 1.2. Two new features must be highlighted: Persistent Workers and a new Communication Library. This new features represent a big change in operation and performance respect to COMPSs 1.2.

4.1.1.1 Persistent Workers

COMPSs has a master-slave design, with only one master and multiple slaves called "workers". In COMPSs 1.2 each time the master sends a task to be executed in a resource a worker is created on that resource and it dies after it executes the assigned task. In this sense it can be said that workers are not "persistent" as they die after the task execution. A worker in COMPSs 1.2 is a java application that receives the task to execute on its application arguments.

COMPSs 1.3 introduces persistent workers. Master and persistent workers are started before the application execution, one per resource, and remain alive till the end of the application execution. A connection is established between each persistent worker and the master and they send commands to each other over this connection. As an example, a command sent to a worker can be for executing a task and a command sent to the master can be for notifying the end
of a task execution. Workers can communicate sending commands to each other for example to transfer data between them.

The main benefit is that with persistent workers the overhead of creating a Java application (process) each time a task need to be executed on a resource disappears.

### 4.1.1.2 Communication Library

As seen in the previous section, the use of persistent workers relies on the communication with the master and other workers.

In COMPSs 1.2 the only communication library used is JavaGAT, and commonly the communication is done through SSH connections. SSH connections are used to start a worker (a java process) on a remote resource each time a task needs to be executed or to transfer data to a remote resource.

COMPSs 1.3 comes with a new communication library based on Java NIO. Java NIO (New IO) is an alternative IO API for Java (from Java 1.4), meaning alternative to the standard Java IO and Java Networking API. Java NIO offers a different way of working with IO than the standard IO API. Java NIO is buffer oriented and non-blocking IO, while Java IO is stream oriented and blocking IO.

The use of the new NIO communication library in COMPSs 1.3 instead of creating SSH connections with JavaGAT, represents a big increment on the overall application execution performance, as establish SSH connections has a high cost on any system.

### 4.1.2 Software Packages

To integrate COMPSs 1.3 with Self-Contained Objects (SCOs) some of its software packages have been modified. Therefore it is good to briefly describe here the COMPSs 1.3 software packages and main features implemented on each of them in order to better understanding the integration with SCOs.
Figure 4.1: COMPSs 1.3 Software Packages

There are two main software packages, the Programming Model package and the Runtime package.

4.1.2.1 Programming Model Package

The \textit{Programming Model} package contains subpackages that make able to write a COMPSs application in Java, the native language of COMPSs, or any other supported language like Python or C/C++.

- \textit{Annotations package}: this package contains the annotation types used to write the Task Selection Interface of a COMPSs application written in Java, for example, method, parameter and constraint annotation types.

- \textit{Bindings}: it contains software modules that allows writing a COMPSs application in Python or C/C++ languages.
  
  - \textit{Bindings Common}: it is a C++ module that makes able a Python or a C/C++ application to communicate with the COMPSs Runtime written in Java through the Java Native Interface (JNI).

The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and
be called by applications and libraries written in other languages such as C, C++ and Python.

- **<Python>**: a Python module for writing a PyCOMPSs application. The programmer only need to import this module in the Python application as any other existing Python module. The **<Python>** module uses the **<Bindings Common>** module to communicate with the COMPSs Runtime.

- **<C/C++>**: a C++ module that allows a C or C++ application to use COMPSs. The programmer needs to compile its C or C++ application together with this module in order to have an executable that runs with COMPSs. The **<C/C++>** module also uses the **<Bindings Common>** module to communicate with the COMPSs Runtime.

### 4.1.2.2 Runtime Package

The Runtime package contains all the subpackages that makes COMPSs able to run parallel an application on a set of distributed resources. These subpackages are:

- **<Config>**: This package contains the configuration files for the logging library (Apache Log4j) and the default configuration files used to specify the set or resources availables (resources.xml), the set of resources used by the application (project.xml) and the runtime monitor configuration file.

- **<Loader>**: This package manipulates the bytecode of a Java application. It can be seen as the entry point of COMPSs where the methods specified in the Task Selection Interface are intercepted at execution time and redefined properly to run as a parallel task.

    Javassist (Java programming assistant) is the Java library used by this package to manipulate the Java bytecode. In this sense Javassist provides the support for structural reflection, i.e. the ability to change the implementation of a class at run time. Bytecode manipulation is performed at load-time through a provided class loader.

- **<Commons>**: This package contains interfaces and classes used in common by the rest of packages. For example, classes for logging, classes that define COMPSs data types (e.g. for runtime exceptions, constants, requests, tasks, parameters, resources...), object access registry, object serialization or Java threads.
• **<Adaptors>:** This package contains adaptors for the communication libraries supported in COMPSs, JavaGAT and NIO. The communication happens between COMPSs master and workers, therefore each adaptor has two parts, the implementation for the master and the implementation for the workers. COMPSs can communicate also with Web Services, the adaptor used in this case only need to have the master’s implementation as the web service does the worker part.

  – **<Java Grid Application Toolkit (JavaGAT)>:** offers a set of coordinated, generic and flexible APIs for accessing grid services from application codes, portals, data managements systems, etc. JavaGAT sits between grid applications and numerous types of grid middleware, such as Globus, Glite, SGE, SSH or Zorilla. JavaGAT lifts the burden of grid application programmers by providing them with a uniform interface that provides: file operations, file stream operations, job submission, monitoring and access to information services. As a result, grid application programmers need only to learn a single API to obtain access to the entire grid. Due to its modular design, JavaGAT can easily be extended with support for other grid middleware layers.

![JavaGAT structure](image)

  Figure 4.2: JavaGAT structure.

  – **<NIO>:** Non-blocking I/O (usually called NIO, and sometimes called "New I/O") is a collection of Java programming language APIs that offer features for intensive I/O operations. It was introduced with the J2SE 1.4 release of Java by Sun Microsystems to complement the existing standard I/O. Java NIO consist of the following core components: Channels, Buffers and Selectors. Java NIO has more classes and components than these, but the Channel, Buffer and Selector forms the core of the API.

  Typically, all IO in NIO starts with a Channel. A Channel is a bit like a stream. From the Channel data can be read into a Buffer. Data can also be written from a Buffer into a Channel. Here is an illustration of that:
There are several Channel and Buffer types. Here is a list of the primary Channel implementations in Java NIO: FileChannel, DatagramChannel, SocketChannel and ServerSocketChannel. As you can see, these channels cover UDP + TCP network IO, and file IO.

Here is a list of the core Buffer implementations in Java NIO: ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer and ShortBuffer. These Buffers cover the basic data types that you can send via IO: byte, short, int, long, float, double and characters. Java NIO also has a MappedByteBuffer which is used in conjunction with memory mapped files.

A Selector allows a single thread to handle multiple Channel’s. This is handy if your application has many connections (Channels) open, but only has low traffic on each connection. For instance, in a chat server. Here is an illustration of a thread using a Selector to handle 3 Channel’s:
To use a Selector you register the Channel’s with it. Then you call it’s select() method. This method will block until there is an event ready for one of the registered channels. Once the method returns, the thread can then process these events. Examples of events are incoming connection, data received etc.

- <Resources>: A cloud can be specified in COMPSs as a resource used for tasks execution. This package contains connectors for different cloud providers. There are connectors specific for Amazon or Azure, and more standard connectors based on JClouds, Emotive, One, rOCCI or VMM middleware.

- <Scripts>: It is a folder that contains the bash shell scripts used in COMPSs. For example, to start running COMPSs, to submit a COMPSs application in a supercomputer like Marenostrum, to start running a Persistent Worker in a remote node, or to generate a trace file.

- <Engine>: This package is described with detail in the next section.

### 4.1.2.3 Engine Package

The <Engine> package can be seen as the core package of the COMPSs Runtime. It contains the following subpackages:

![COMPSs Software Packages. Engine Package.](image)

- <API>: In this package is implemented the COMPSs Application Interface, COMPSs API. This API expose to other applications a set of methods
to interact with the COMPSs runtime. For example, to start/stop the runtime, to execute a task, to get an object or a file, among other methods.

As instance, when in the <Loader> package is detected the main method of the Java application a call to the start runtime method (<startIT>) defined in the COMPSs API is added before at execution time.

Or the Python or C/C++ bindings can execute a task in COMPSs calling, through JNI, the correspondent method (<executeTask>) in the COMPSs API.

• <Components>: It is informally said that COMPSs receives its name due to this package. That is Components Superscalar, COMPSs. In fact, it is where is implemented the logic of COMPSs. It has the following subpackages or components:

  – <Access Processor>: This component detects the access to any method that is included in the Task Selection Interface. When this happens, it creates a new task to execute such method and send the task to the <Task Analyser> component.

    The <Access Processor> also registers the access to objects or files from the application, those which are a task parameter. When an already registered object is accessed the <Access Processor> wait till the last writer task on such object has finished and later returns the object updated to the application.

  – <Task Analyser>: This component processes each task and tells to the <Data Manager> component to register all the task’s parameters and the access mode (read, write or read-write) of each of them. Registering the parameters access mode is necessary to be able to detect dependencies between tasks. Then the <Access Processor> checks the dependencies between the current task and previous ones. If there is no dependency the current task is enqueue to the <Task Dispatcher>.

  – <Task Dispatcher>: On one hand, this component registers all the available resources into the <Resource Manager> module, parses the constraints defined in the Task Selection Interface, and instantiate the scheduler, the default scheduler if no other is specified.

    On the other hand, it processes a queue of requests. A request can be to schedule or reschedule a task. Such task will be sent to the <Task
Scheduler. But also, a request can be a "task finished" notification, then the <Task Dispatcher> does the necessary actions like notify the task finalization to the <Task Scheduler> and release the resource assigned to the task.

- <Task Scheduler>: In this component different policies to schedule the execution of tasks among the available resources can be defined. There is a default schedule policy implemented on it, but other policies could be added.

When the selected schedule policy assigns a task to a resource, the <Task Scheduler> creates a job that links the task to the resource, and sends the job to the <Job Manager> component.

- <Job Manager>: This component processes a job. First it checks whether the necessary data transfers have been done. If not, transfers are ordered and the job waits till they are done. Other jobs can be processed meanwhile.

Transfers can fail and if so they are retried, but if still failing the job must be rescheduled. If transfers fail and the job has been already rescheduled, the job status is set to failed.

When transfers are ready the job is submitted to the assigned worker. The worker will execute the task described in the job and the <Job Manager> keeps listening for the job to finish or fail. If a job fails it tries to resubmit the job to the same worker, if this has been already done it sends a request to reschedule the job to the <Task Dispatcher>, if this has been already done and the job still fails, the job status is set to failed.

Whenever a job ends a notification is sent to the <Task Dispatcher> with the Job status, finished or failed.

- <Data Manager>: Also known as <Data Info Provider>, this module is used to register not only object or file accesses but also object or file versions when they are modified by a task. The <Data Manager> allows to bring the last version of an object or a file present in a worker to the master node; in COMPSs this is called a "synchronization".

- <Resource Manager>: This module is used not only to register all
the available resources specified in the configuration files, but also it starts or stops a worker on a resource and manages the pool of workers.

- `<Scheduler>`: This package contains an abstract class for the scheduler (that extends the "Task Scheduler" class) and its scheduler policies. A default implementation is provided, but any other scheduler or policy can be implemented and used instead.

- `<Runtime Monitor>`: This component monitors the overall execution and generates two outputs: a Task Dependency Graph and an XML Task State File. The Task Dependency Graph is a directed graph where the nodes are tasks and the directed links are data dependencies between two tasks. The XML Task State File is used to give post-mortem statistic information, for example, average task duration per task type, number of finished or failed tasks, workload per resource, etc.

- `<Types>`: This package contains classes that define types used in other packages. These types are related to parameters (FileParameter, ObjectParameter...), requests (TransferObjectRequest, TransferFileRequest, NotifyTaskEndRequest...), jobs (JobStatusListener), file info (FileInfo), object info (ObjectInfo), accesses (AccessParams), locations (Location), etc.

- `<Util>`: This package contains classes that implement some utility like a parser (CEIParser class) to read the Task Selection Interface file in order to register the methods, parameters and constraints specified in the file. Or a class to implement the Task Dependency Graph (Graph class).

### 4.1.2.4 Other Packages

In this section are mentioned other relevant packages used in COMPSs:

- `<Tools>`: This package contains different tools, like the COMPSs Monitor. The COMPSs Monitor has the following features available through a graphical interface:

  - Resources information: Provides information about the resources used by the application.
  - Tasks information: Provides information about the tasks definition used by the application.
  - Current tasks graph: Shows the tasks dependency graph currently stored into the COMPSs Runtime.
– Complete tasks graph: Shows the complete tasks dependency graph of the application.
– Load chart: Shows different dynamic charts representing the evolution over time of the resources load and the tasks load.
– Runtime log: Shows the runtime log.
– Execution Information: Shows specific job information allowing users to easily select failed or uncompleted jobs.
– Statistics: Shows application statistics such as the accumulated cloud cost.


• \textless Dependencies\textgreater: This package contains third-party software dependencies like:
  – Extrae developed at the Barcelona Supercomputing Center. Extrae is a software for application instrumentation and tracking. COMPSs uses Extrae to generate a trace file of the execution.
  – Paraver is another tool from BSC that is used to generate views of a trace file based on a configuration file. Traces shown in this document have been generated with this powerful tool.


• \textless Tests\textgreater: This package contains a set of tests used to verify the main features of COMPSs. This test can be executed in a Jenkins service or locally on the developer’s machine.
4.2 Integration with COMPSs 1.3

After a previous analysis of the structure and operative of COMPSs 1.3, some of its packages have been modified in order to integrate with Self-Contained Objects.

Those packages are:

- The `<Loader>` package
- The `<API>` package
- The `<Engine>` package, mainly the `<Task Scheduler>` component
- The `<Adaptors>` package, mainly the `<Worker>` component.

The changes made on this packages are described in this section.

4.2.1 Loader Integration

When running a Java application with COMPSs, the first thing the runtime does is to instrument the application main method. The `<Loader>` component reads the main method and when a call to a method declared in the Task Selection Interface is found, the `<Loader>` change the call to such method by a call to the `<executeTask>` method in the runtime API to execute the method remotely as a task.

The `<executeTask>` method has the following signature:

```java
public int executeTask(Long appId, String methodClass,
                      String methodName, boolean priority, boolean hasTarget,
                      int parameterCount, Object... parameters)
```

where:

- appId: it is the application identifier.
- methodClass: it is the class where the method is implemented.
- methodName: it is the name of the method to be executed as a task.
- priority: it indicates if the task has priority or not.
- hasTarget: it is true for object methods (e.g. mercedes = new Auto(); mercedes.run();), and false for class methods (e.g. Auto.wash();).
- parameterCount: number of task parameters.
parameters: task parameters.

Each parameter in the original method produces three task parameters:

- Task-Parameter Type: indicates the datatype of the original method parameter. Whether it is a file, `<FILE_T>`, or a primitive type, `<BOOLEAN_T>`, `<CHAR_T>`, `<STRING_T>`, `<BYTE_T>`, `<SHORT_T>`, `<INT_T>`, `<LONG_T>`, `<FLOAT_T>`, `<DOUBLE_T>`, or an object, `<OBJECT_T>`.

Two new types have been added. One used when a method parameter is a Self-Contained Object not persisted yet, `<SCO_T>`, and another if it is a Self-Contained Object already persisted, `<PSCO_T>`.

- Task-Parameter Direction: indicates if the original method parameter is an input `<IN>`, output `<OUT>`, or input-output `<IN_OUT>` parameter as defined in the task-selection interface.

- Task-Parameter Value: the task parameter value. It can be the method parameter itself or other value. A value can be an object or a datatype.

### 4.2.1.1 The MakePersistent method

COMPSs manages a Self-Contained Object (SCO) like any other Object until its `<makePersistent>` method is invoked and its data and code are stored in a Persistent Object (PO). The `<makePersistent>` method of a SCO can be invoked from the master in the main code of the application or from a worker if it has been selected as a task.

It is important to notice that before the `<makePersistent>` method call, the SCO data is all in main memory and it could be a big object that needs a considerable amount of memory. But after the `<makePersistent>` method call, the data and methods are not longer in main memory, they are in secondary memory stored in a PO.

The persisted SCO in main memory acts now like a wrapper object that gives access to the data or methods stored in secondary memory in a PO. The persisted SCO itself requires now few memory. At each time, only those SCO attributes that are accessed by the program are retrieved from secondary memory and kept in main memory.
Figure 4.6: Make Persistent SCO interface method. The \texttt{makePersistent()} method of a SCO can be called from the main application but also from a task.

### 4.2.1.2 The checkSCOType method

To know if a method parameter is just an Object or a Self-Contained Object not persisted yet or a Self-Contained Object already persisted is only possible at execution time. Therefore a mechanism to check the type of a method parameter at execution time is needed. This mechanism is implemented by a loader method called \texttt{checkSCOType()}, defined as \texttt{public static ParamType checkSCOType(Object o)}.

The \texttt{checkSCOType()} loader method inspects the method parameter at execution time and returns \texttt{OBJECT\_T} type for those which are only Objects, \texttt{SCO\_T} type for those which are SCOs not persisted yet and \texttt{PSCO\_T} type for those which are SCOs already persisted, that is, those SCOs that its \texttt{getID()} method returns a not null Persistent Object identifier.

### 4.2.1.3 The checkSCOPersistent method

When a method parameter has type \texttt{OBJECT\_T} or \texttt{SCO\_T}, the task parameter value passed to the \texttt{executeTask()} method in the runtime API is the original method parameter. But when it has the \texttt{PSCO\_T} type a \texttt{PSCOId} object is created and passed as task parameter value instead. This is done using another loader method called \texttt{checkSCOPersistent()}, defined as \texttt{public static}
Object checkSCOPersistent(Object o), that inspects the current method parameter and returns the method parameter itself or a <PSCOId> object when its type is <PSCO_T>. Why the use of the <PSCOId> objects is explained in the next section.

As resume, let’s see an example. Suppose that in the main code of the application there is a call to the method <foo()> of the object <a> with parameters <b> and <c>.

```java
public static void main(String[] args) throws Exception {
    AClass a = new AClass();
    BClass b = new BClass();
    CClass c = new CClass();
    a.foo(b, c);
    System.out.println(a);
}
```

The <Loader> component will rewrite the call as:

```java
public static void main(String[] args) throws Exception {
    AClass a = new AClass();
    BClass b = new BClass();
    CClass c = new CClass();
    executeTask( 0, "AClass", "foo", false, true, 9,
                checkSCOType(b), ParamDirection.IN, checkSCOPersistent(b),
                checkSCOType(c), ParamDirection.IN, checkSCOPersistent(c),
                checkSCOType(a), ParamDirection.IN_OUT, checkSCOPersistent(a));
    System.out.println(a);
}
```

So at execution time, thanks to the <checkSCOType> and <checkSCOPersistent> loader methods, the task parameter type, direction and value for each method parameter are passed to the <executeTask> method. Notice that the <a> object itself is passed also as an in-out task parameter.

### 4.2.1.4 The PSCOId object

Each Persistent Object (PO) has an alfa-numeric identifier assigned by the underlying storage platform. This identifier can be accessed by the <getID> method defined in the SCO Interface. A PO identifier looks like:
For any SCO not persisted yet, COMPSs internally manages the SCO as any other object in the runtime. For any persisted SCO, COMPSs substitutes in the runtime the persisted SCO by a $<PSCOId>$ object.

During the application execution, COMPSs keeps objects (task parameters) in its internal structures and this requires memory. But with persisted SCOs there is no need to keep the SCOs internally in the runtime structures. A smaller object that links the persisted SCO in main memory and the PO in secondary memory can be used instead, a $<PSCOId>$ object.

Notice that the use of $<PSCOId>$ objects allows to work with persisted SCOs from a language different from Java, like Python or C++, as SCOs from other languages cannot be kept, as they are in memory, in the runtime structures as COMPSs is a Java native program.

![Figure 4.7: The $<PSCOId>$ class.](image)

The $<PSCOId>$ has by construction the same hashcode than the associated persisted SCO. Hashcodes are used by COMPSs to index its internal structures, most of them hashmaps.

The $<PSCOId>$ also keeps the identifier of the corresponding PO. The $<back-ends>$ list keeps the hostname or ip of the hosts where the PO is stored or has a replica.

When an Object or a SCO not persisted yet is a task parameter, they are sent serialized in advance to the host where the task is going to be executed later. The bigger the object is in main memory the bigger the serialization file to be transferred.
Chapter 4. Integration

But when a persisted SCO is a task parameter, only the serialization of the corresponding \textit{<PSCOId>} object is sent in advance to the task execution host.

Once the \textit{<PSCOId>} is deserialized in the remote host, the identifier is used to obtain a new persisted SCO that allows the access to the PO with such identifier in the underlying storage platform, from that remote host and from the task to be executed on it. This is done by the COMPSs worker calling the \textit{<SCO getByID(String id)>} method of the Storage Interface.

The serialization file of a \textit{<PSCOId>} is smaller than the serialization of the persisted SCO and faster to be transferred. Or in other words, \textit{<PSCOId>} objects are managed by COMPSs like any other Object but its serialization is smaller and faster to transfer.

4.2.2 API Integration

As explained before, the \textit{<Loader>} modifies the main code of the application replacing each call to a method declared in the Task-Selection Interface by a call to the \textit{<executeTask>} API method in order to execute the method as a remote task.

The \textit{<executeTask>} method belongs to the API package. The API package is the entry point to the COMPSs runtime, for example, the Python and C/C++ bindings call the API package directly to start/stop the runtime or to execute a task.

Internally, the \textit{<executeTask>} method calls the \textit{<processParameters>} method. The \textit{<processParameters>} processes the task parameters. Task parameters come in chunks of 3 elements: value, type and direction.

\begin{verbatim}
data.txt FILE_T IN
5 INT_T IN
Object@8jdadf OBJECT_T IN_OUT
\end{verbatim}

If a task parameter has type \textit{<SCO T>}, that is the task parameter value is a SCO not persisted, COMPSs manages the SCO like any other Object. From this point the COMPSs runtime can be seen as a black-box, considering that any SCO not persisted will be managed as regular Objects inside the runtime.

If a task parameter has type \textit{<PSCO T>}, that is the task parameter value is a persisted SCO, COMPSs manages the persisted SCO in a different way. In the \textit{<processParameters>} method a new \textit{<PSCOId>} object is created and substitutes the persisted SCO inside the runtime. A \textit{<PSCOId>} requires less memory, a smaller serialization on disk and can be transferred faster to the worker nodes.
4.2.3 Engine Integration: Task Scheduler

Eventually, all tasks pass through the \texttt{<TaskScheduler>}. If a task parameter has type \texttt{<PSCO\_T>} the task parameter value will be a \texttt{<PSCOId>} object at that point.

The default scheduler policy is based on data-locality. The \texttt{<getLocations>} method of the Storage Interface is used to query the underlying storage platform for the PO locations, using the identifier provided in the \texttt{<PSCOId>} object.

The \texttt{<getLocations>} method response, a not-empty list of hostnames or ips, is set in the \texttt{<backends>} property of the \texttt{<PSCOId>} object.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{compss.png}
\caption{Execution flow diagram of a COMPSs application.}
\end{figure}

4.2.4 Adaptors Integration: Worker

Once a task arrives to the worker assigned by the Task Scheduler, If a task parameter has type \texttt{<PSCO\_T>}, the locations in the \texttt{<backends>} property of the \texttt{<PSCOId>} object, the task parameter value, are examined as follow:

- If the task parameter has direction IN and the worker hostname or ip is one of the locations, nothing needs to be done. The Persistent Object (PO) is already in the local storage backend.
- If the task parameter has direction IN but the worker hostname or ip is not one of the locations, a new replica needs to be done in the worker
host. Therefore, the worker calls the <newReplica()> method of the Storage Interface to make a replica of the PO in the local storage backend.

- If the task parameter has direction IN_OUT or OUT, that is, the PO is going to be modified by the task, the worker calls the <newVersion()> method of the Storage Interface to make a modifiable version of the PO in the local storage backend.

Once all the task parameters are in the worker host, the task is ready to be executed. There are two task execution modes, compss or external:

- The <compss> execution mode means that the task will be executed by the COMPSs worker. This is the default mode for any parallel application execution. The COMPSs worker will call by Java reflection the method defined in the task.

- The <external> execution mode means that the task will be executed out of COMPSs by the underlying storage platform. In this mode, the COMPSs worker calls the <executeTask()> method of the Storage Interface that will execute the method defined in the task in the local storage backend.

The task execution mode is a runtime configuration option defined as:

```
--task_execution=<compss|external>
```

Task execution under COMPSs or external.
Default: compss

4.3 Integration with PyCOMPSs

The Python binding uses the Java Native Interface (JNI) library to invoke directly the methods exposed by the COMPSs API. That is, invokes directly the <executeTask> method when it needs to execute a task on a remote resource.

The term "directly" used before means that no instrumentation is done like happens in COMPSs. In PyCOMPSs, the code associated to the <@task> decorator is responsible to invoke the <executeTask> method with the right arguments. At execution time, this code is interpreted like the rest of the Python application.

Main differences with a Java application, comes from the fact that Python objects cannot be managed inside the COMPSs runtime that is a Java native program.
If a task parameter is a Python object, it is serialized in a file and the filename of the serialization is passed as task parameter value instead. Therefore Python objects are managed as files in the COMPSs runtime.

When the task parameter is a Python SCO, instead of being serialized the SCO is persisted in the storage platform and its storage identifier is passed as task parameter value.

The `<executeTask>` method will then call the `<processParameters>` method to process each task parameter. When the parameter has type `<PSCO_T>` but the parameter value is not a `<PSCOId>` object as expected but a string, the COMPSs runtime understands that such string is the identifier of a Persistent Object and creates a new `<PSCOId>` object as done in the Java case.

From this point, `<PSCOId>` objects are managed as usual by the COMPSs runtime until the Python worker is called to execute the task.

The Python worker can’t manage `<PSCOId>` objects because they are Java objects created by the COMPSs runtime, so only the string with the identifier for the Persistent Object is passed to the Python worker as task parameter value.

The Python worker receives then the identifiers of Python Persisted Objects as strings. The Python worker use each identifier to obtain a Self-Contained Object for each Persistent Object.

Self-Contained Objects will be used inside the task to access the data stored in the Persistent Object needed to complete the task.
Chapter 5

Infrastructure

5.0.1 Development Infrastructure: BSCGrid

During the development process BSCGrid has been the infrastructure used to test the integration of COMPSs with dataClay storage platform. BSCGrid is a cluster at BSC-CNS shared by different projects.

Three hosts at BSC-CNS Grid were available to test the integration. Host \textless bscgrid20\textgreater was selected to run the COMPSs master. Hosts \textless bscgrid05\textgreater and \textless bscgrid06\textgreater run a COMPSs worker each. These hosts are not exclusive, there can be processes from other projects running on them.

![Development infrastructure diagram](image)

Figure 5.1: Development infrastructure. Hosts used from BSCGrid cluster.
dataClay is a distributed storage platform for Persistent Objects. Persistent Objects can be stored on any dataClay backend and there is one backend running on each worker node. dataClay also starts in one node a process called <Logic-Module> that manages the metadata.

BSC-CNSgrid20 details:
- Processors: 2 x AMD Opteron 6140 8-Core
- Memory: 32 GB of RAM
- Disk: 1 x 2TB 6G SAS 7.2K

BSC-CNSgrid05, BSC-CNSgrid06 details:
- Processors: 1 x Intel Xeon 8-Core E5-2440
- Memory: 16 GB of RAM
- Disk: 1 x 4TB 6G SAS Seagate ST4000NM0023

5.0.2 Development Infrastructure: Minerva
Minerva is the cluster used to test PyCOMPSs and the Hecuba storage platform. Nodes of Minerva have the following characteristics each:
- Processors: 16 Cores
- Memory: 24 GB of RAM
- Disk: 4 TB

Minerva is a cluster shared by different projects and nodes are not exclusive. On each worker node an Hecuba backend is running. A Hecuba backend is an Apache Cassandra database backend.

5.0.3 Production Infrastructure: Marenostrum III
MareNostrum is a supercomputer hold by the Barcelona Supercomputing Center, the most powerful in Spain and one of seven supercomputers in the Spanish Supercomputing Network.

MareNostrum III is based on Intel SandyBridge processors, iDataPlex Compute Racks, a Linux Operating System and an Infiniband interconnection.

The current Peak Performance is 1.1 Petaflops. The total number of processors is 48,896 Intel SandyBridge-EP E5–2670 cores at 2.6 GHz (3,056 compute
nodes) with 103.5 TB of main memory.

See below a summary of the system:

- 37 iDataPlex compute racks.

Each one composed of:

- 84 IBM dx360 M4 compute nodes
- 4 Mellanox 36-port Managed FDR10 IB Switches
- 2 BNT RackSwitch G8052F (Management Network)
- 2 BNT RackSwitch G8052F (GPFS Network)
- 4 Power Distribution Units

All IBM dx360 M4 node contain:

- 2x E5–2670 SandyBridge-EP 2.6GHz cache 20MB 8-core
- 500GB 7200 rpm SATA II local HDD

Nodes are differentiated by RAM as follows:

- 128 nodes contain 8x 16G DDR3–1600 DIMMs (8GB/core) Total: 128GB/node
- 128 nodes contain 16x 4G DDR3–1600 DIMMs (4GB/core) Total: 64GB/node
- 2880 nodes contain 8x 4G DDR3–1600 DIMMs (2GB/core) Total: 32GB/node

42 heterogeneous nodes contain:

- 8x 8G DDR3–1600 DIMMs (4GB/core) Total: 64GB/node
- 2x Xeon Phi 5110P accelerators
- 1.9 PB of GPFS disk storage

Interconnection Networks

- Infiniband Mellanox FDR10: High bandwidth network used by parallel applications communications (MPI)
- Gigabit Ethernet: 10GbitEthernet network used by the GPFS Filesystem.

Operating System: Linux - SuSe Distribution 11 SP3

Marenostrum has been the infrastructure used to run the experiments and get the results presented on this document.
6.1 COMPSs dataClay Application

The compilation of an application that runs with COMPSs and dataClay needs to be done in three steps:

6.1.1 Step 1: Application Class Registry

First, those classes (*.java files) in the application that the programmer wants them to be Self-Contained Objects need to be compiled with the Java compiler. Later the compiled classes (*.class files) have to be registered in dataClay.

To register classes in dataClay a tool called <Registrator> have been developed. <Registrator> is a Java program that connects with the <LogicModule> of dataClay and registers the class passed as argument and its subclasses recursively.

The <Ant> script developed to do this automated looks like follows:

```bash
echo "CLEAN & BUILD MOLECULE CLASSES"

cd ${HOME}/apps/DataModels/Molecule
ant clean lib

cd

echo "REGISTER CLASSES"
java -cp ${HOME}/dataClay/dataClayclient.jar:
${HOME}/apps/DataModelProviders/Registrator/lib/registrador.jar
severo.registrator.Registrator severo.moleculeArray.Molecule
${HOME}/apps/DataModels/Molecule/lib/molecule.jar

The <severo.moleculeArray.Molecule> canonical class name and the path to the jar file with the classes, <~/apps/DataModels/Molecule/lib/molecule.jar>, are passed to the <severo.registrator.Registrator> main class as arguments.

The output of the <Registrator> tool looks like:
REGISTER CLASSES

[dataClay] Connecting to BSC-CNSgrid05.BSC-CNS.es:7001 ...
[dataClay] No global.properties file found at /home/cdiaz/cfgfiles/global.properties.
Using default values.
Connected!
[dataClay] Using client library version 4327
[NETTY] CL channel [id: 0x3a44ec15] registered

Creating an interface for class: severo.moleculeArray.Molecule

Registering contract for user : Registrator
Registering contract for user : Enricher
Registering contract for user : Producer
Registering contract for user : Consumer
[dataClay] Finishing connections ...
[NETTY] CL channel [id: 0x3a44ec15] unregistered
Done!

The `<severo.moleculeArray.Molecule>` application class has been registered for the user `<Producer>` among others.

6.1.2 Step 2: SCO Class Generation

Once an application class has been registered, dataClay can generate the corresponding SCO class.

To obtain SCO classes a tool called `<GetStubs>` have been developed. `<GetStubs>` is a Java program that connects with the `<LogicModule>` of dataClay and gets the SCO classes (stubs) registered for a certain user.

The `<Ant>` script developed to do this automated looks like follows:

```bash
echo "GET STUBS"
export dataClayCLIENTCONFIG=${HOME}/apps/Tools/GetStubs/cfgfiles/client.properties
${HOME}/apps/DataClients/Producer/stubs
```

The output of the `<GetStubs>` tool looks like:

GET STUBS
Chapter 6. Compilation

[DataClay] Connecting to BSC-CNSgrid05.BSC-CNS.es:7001 ...
[DataClay] No global.properties file found at
/home/cdiaz/cfgfiles/global.properties.
Using default values.
Connected!
[DataClay] Using client library version 4327
[Netty] CL channel [id: 0xd54bcc84] registered
[DataClay] Finishing connections ...
[Netty] CL channel [id: 0xd54bcc84] unregistered
Done!

All the SCO classes (stubs) the user <Producer> has permission are copied
to the folder <~/apps/DataClients/Producer/stubs>

6.1.3 Step 3: Final Application Compilation

Once the SCO classes have been obtained the whole application can be compiled.

The <Ant> script developed to do this automated looks like follows:

```bash
# echo "CLEAN & BUILD PRODUCER APP"
cd ${HOME}/apps/DataClients/Producer
ant clean lib
cd

echo "ALL DONE!"
```

The output of the script looks like:

Buildfile: /home/cdiaz/apps/DataClients/Producer/build.xml

clean:
 [delete] Deleting directory /home/cdiaz/apps/DataClients/Producer/build
 [delete] Deleting directory /home/cdiaz/apps/DataClients/Producer/lib

init:
 [mkdir] Created dir: /home/cdiaz/apps/DataClients/Producer/build

compile:
 [javac] Compiling 2 source files to
 /home/cdiaz/apps/DataClients/Producer/build
 [copy] Copying 4 files to /home/cdiaz/apps/DataClients/Producer/build
 [copy] Copying 3 files to /home/cdiaz/apps/DataClients/Producer/build

lib:
 [mkdir] Created dir: /home/cdiaz/apps/DataClients/Producer/lib
Chapter 6. Compilation

[jar] Building jar: /home/cdiaz/apps/DataClients/Producer/lib/producer.jar

BUILD SUCCESSFUL

The jar file <producer.jar> contains the final application compiled with the SCO classes.

6.2 PyCOMPSs & Hecuba Application

As Python is an interpreted language there is no need to compile the source files of a Python application to run it with PyCOMPSs & Hecuba.

There is also no registration process needed for Hecuba. Tables for Persisted Objects in Cassandra are created by Hecuba automatically from the description provided in the Self Contained Object. The following code shows a Python Self Contained Object called "MoleculeDictionary" this class inherits from the "StorageObj" class provided by the Storage API implementation done in Hecuba and therefore its instances will be SCOs.

The class is called "MoleculeDictionary" because it will be mapped to a Python dictionary by Hecuba and this dictionary will be persisted on a table in Cassandra as both, dictionary and table, are <key, value> structures. The annotation <@ClassField> is used to indicate which is the name and type of the corresponding column in the table that will be created in Cassandra, and whether the field is the key column (#y) or not (#n) in the table.

```python
from pyapi.storageobj import StorageObj
class MoleculeDictionary(StorageObj):
    @ClassField id str #y
    @ClassField atoms str #n
    @ClassField center str #n
```

To abstract the programmer from the use of dictionaries a class field of type "MoleculeDictionary" could be added to the original application class, "Molecule". Once the dictionary field is marked as persistent all the data stored on it will be persisted in Cassandra. The name for the table to be created in Cassandra is passed as argument in the <makePersistent> call.

The <get> and <put> class methods could be implemented in order to retrieve or store a "Molecule" object by its identifier (its key value) from or into the persistent dictionary.
from app.MoleculeDictionary import MoleculeDictionary
class Molecule:
    dictionary = MoleculeDictionary()
    dictionary.make_persistent('MoleculeDictionary')

@staticmethod
def get(id):
    ...
...

@staticmethod
def put(m):
    ...
...

def __init__(self, id):
    ...
...

In the main application "Molecule" objects could been used as follows and the programmer will be unaware of the use of dictionaries:

from app.Molecule import Molecule

if __name__ == '__main__':
    n = int(sys.argv[1])
    print '[Producer] - Running with ', n, ' molecule(s)

#Create & Persist
for i in range(0,n):
    m = Molecule(i)
    Molecule.put(m)

# Barrier
for i in range(0, n):
    m = Molecule.get(i)
    print m.id

Notice that only when the <put> class method is called for a "Molecule" object, <m>, such object is persisted in Cassandra.
7.1 dataClay Deployment Components

dataClay has two deployment components:

7.1.1 dataClay Logic Module

The Logic Module (LM) is a TCP service that keeps a central repository of: user account information, registered data models, object metadata and both model contracts and data contracts.

LM is the responsible of user authentication both for session handling, during the initialization of user-level applications, and for management operations including the registration of classes, enrichments, contracts, etc.

LM is also in charge of on-demand stub generation for client applications, and deployment of execution classes to Data Service backends propagating any possible enrichment. In short, LM manages all the dataClay operations out of the execution critical path.

7.1.2 dataClay Data Service

The Data Service (DS) is a distributed TCP service that handles all object operations including persistence management (store, load, and delete objects) and method execution (processing execution requests on loaded objects).

To this end, DS backends (DS nodes) are deployed provided with a PostgreSQL database where the byte-array representation of the objects is stored, Netty libraries to communicate with each other, and the runtime environment for each supported language (currently Java and Python).

Furthermore, every DS backend handles two cachés: session caché and metadata caché; in order to prevent the LM to become a bottleneck. The former
enables the DS backends to validate current sessions, so that once an application is initialized and authenticated with LM it can be executed communicating directly to DS backends. The latter gives DS backends awareness of objects datasets and locations, thus enabling them both to check permission rights on data access, and to know which external DS backends to communicate when processing requests requiring objects not stored in local storage.

7.1.3 PostgreSQL

Each dataClay Data Service (DS) uses a PostgreSQL backend or instance to store data.

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department. POSTGRES pioneered many concepts that only became available in some commercial database systems much later.

PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features: complex queries, foreign keys, triggers, updatable views, transactional integrity, multiversion concurrency control.

For more information about PostgreSQL visit its website:

http://www.postgresql.org/

7.2 COMPSs Deployment Components

COMPSs has two deployment components:

7.2.1 COMPSs Master

The COMPSs Master (CM) is a Java program that orchestrates the parallel execution of the user application.

The CM parses and instruments the main class of the user application following the Task Selection Interface (TSI). The CM executes the main method of the instrumented application. When a method declared in the TSI is found, it is executed as a remote task. Data dependencies between task are detected and a Task Dependency Graph (TDG) is internally used to resolve them. Task
free of dependencies are scheduled to remote execution on the available resources.

The CM monitors the remote execution of each task. A fault tolerance mechanism is in charge of resubmitting or reschedule a task if the resource goes down temporally or permanent. If all tasks are executed successfully the application will end and the CM synchronize data and cleans the execution. If a task fails the application execution ends with an error and log information can be used to debug the application.

### 7.2.2 COMPSs Worker

A COMPSs Worker (CW) is a Java program that executes tasks on a remote resource. A unique CW is started on each resource and it remains waiting for tasks to be executed.

If data needed for a task is not present at the local resource the CW ask for the data to be transferred on the resource. When data is ready, the Java method associated to the task is invoked by Java Reflection or executed by an external component, like a storage platform (dataClay or Hecuba).

The CW manages a data cache for future tasks that uses previous data. If either a transfer or a method execution fails, it is notified to the CM and log information can be used to debug the task execution. The CW keeps alive till the CM sends a shutdown command to it.

### 7.3 COMPSs & dataClay deployment in MN3

COMPSs and dataClay have been installed both in Marenostrum III (MN3). Details of COMPSs installation can be found in the COMPSs Installation Manual.

The Load Sharing Facility (LSF) is the utility used at Marenostrum III for batch processing support, so all jobs must run through it. To submit a COMPSs & dataClay Job in Marenostrum a LSF script is provided.

A COMPSs & dataClay Job needs 4 nodes at least to run properly. The number of nodes is specified in the submission script.

The steps done in a COMPSs & dataClay Job are:

1. The LSF Job starts with a set of assigned nodes.
2. Start dataClay, the Logic Module (LM) runs alone on one node, on the other nodes, except one, a Data Service (DS) backend is started.
3. Run the user application with COMPSs. The COMPSs master runs alone in one node (the one exclude in point 2). On the other nodes with a DS backend a COMPSs Worker is started, then both processes run concurrently on each node. Once master and workers are ready the application execution starts.

4. The application finish eventually and COMPSs Master and Workers stop automatically.

5. Stop dataClay. Logic Module and Data Services are stopped.

6. The LSF Job ends.

Different processes are started on nodes. To have them running concurrently the method recommended by the Marenosrum Support Team was the LSF \texttt{<blaunch>} command.

\texttt{<blaunch>} launches parallel tasks on a set of hosts

\texttt{blaunch [-n]}
\begin{verbatim}
[-u host_file | -z host_name ... | host_name]
[-use-login-shell | -no-shell ]
command [argument ... ]
\end{verbatim}

\texttt{<blaunch>} only works under LSF. It can only be used to launch tasks on remote hosts that are part of a job allocation. It cannot be used as a standalone command.

7.4 Hecuba Deployment Components

Apache Cassandra database is the deployment component of Hecuba. On each worker node a Cassandra backend or instance is started following a ring topology.

Apache Cassandra is a massively scalable open source NoSQL database. Cassandra is perfect for managing large amounts of structured, semi-structured, and unstructured data across multiple data centers and the cloud. Cassandra delivers continuous availability, linear scalability, and operational simplicity across many commodity servers with no single point of failure, along with a powerful dynamic data model designed for maximum flexibility and fast response times.

Cassandra built-for-scale architecture means that it is capable of handling petabytes of information and thousands of concurrent users/operations per second.
Cassandra is a partitioned row store database: Cassandra’s architecture allows any authorized user to connect to any node in any data center and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL.

Automatic data distribution: Cassandra provides automatic data distribution across all nodes that participate in a ring or database cluster. There is nothing programmatic that a developer or administrator needs to do or code to distribute data across a cluster because data is transparently partitioned across all nodes in a cluster.

Built-in and customizable replication: Cassandra also provides built-in and customizable replication, which stores redundant copies of data across nodes that participate in a Cassandra ring. This means that if any node in a cluster goes down, one or more copies of that node’s data is available on other machines in the cluster.

Cassandra supplies linear scalability: Cassandra supplies linear scalability, meaning that capacity may be easily added simply by adding new nodes online. For example, if 2 nodes can handle 100,000 transactions per second, 4 nodes will support 200,000 transactions/sec and 8 nodes will tackle 400,000 transactions/sec.

For more information about Cassandra please visit its website:

http://cassandra.apache.org/
8.1 Benchmark

Two versions of a synthetic benchmark have been used to validate the integration of COMPSs programming model and runtime with the dataClay storage platform.

The benchmark produces objects, each object simulates a molecule. There is no pretended relation with a real molecule further than a better understanding of the benchmark.

Molecules are persisted in files on one version and in dataClay on another. Those molecules could be used later by a consumer application. The benchmark presented here is called then $<Producer>$.

Both versions are a COMPSs application, the one that persists molecules in files is called $<ProducerFile>$, and the one that persists molecules in dataClay is called $<ProducerDataClay>$.

Both versions have the same phases:

- Create: creates $<n>$ empty molecules.
- Initialize: initializes each molecule with $<m>$ atoms.
- Compute: computes the center of mass of each molecule.
- Persist: persists the molecule in a file or in dataClay depending on the benchmark version.
- Results: simulates an access to each molecule from the master node to get its data. It can be seen as a phase where results are collected.

In both versions, the phases $<create>$ and $<results>$ are executed sequentially in the master node while the phases $<initialize>$, $<compute>$, and $<persist>$ are
defined as tasks and executed in parallel in the worker nodes.

Between both versions there is a difference in the order that such phases are executed, this is illustrated in Figure 8.1. There is a change on when take place the \texttt{<persist>} phase.

![Figure 8.1: Benchmark phases of the File and dataClay versions.](image)

In the File version, the molecule is initialized and its center of mass is computed first and after that the molecule is written to a file. In the dataClay version, the molecule is persisted in dataClay first and later initialized and the center of mass computed.

### 8.2 Previous Considerations

For a better understanding of the benchmark structure and behaviour some questions are answered in this section.

#### 8.2.1 When persist a SCO with dataClay?

To get benefit of dataClay is important to persist SCOs before doing any operation on them. If we initialize a SCO before make it persistent, more data have to be transferred into dataClay once its \texttt{<makePersistent>} method is called.

To call the \texttt{<makePersistent>} method first on a SCO makes that all posterior modifications are done directly in dataClay which is a faster option.

In a similar way, to make that the computation phase take place in dataClay, that is, doing the computation where the data is and therefore doing things faster, the SCO must be persistent first.
8.2.2 Are both benchmark versions equivalent?

Both versions generate the same number of tasks. They also generate the same Task Dependency Graph (TDG) as illustrated in Figure 8.2.

![Task Dependency Graph (TDG) of the File and dataClay benchmark versions with 16 objects and 16 SCOs respectively.](image)

Figure 8.2: Task Dependency Graph (TDG) of the File and dataClay benchmark versions with 16 objects and 16 SCOs respectively.

Figure 8.2 shows the TDG of an execution of the \texttt{<ProducerFile>} and the \texttt{<ProducerDataClay>} benchmark versions, with 16 molecules.

The central chain corresponds to the \texttt{<results>} phase, that is, each spot represents a data access done in the master node to a molecule. The other side-chains of 3 spots each represent the tasks that are executed in the worker nodes.

Side-chains are executed in parallel, but tasks in one of this chains are executed in programming order because they have a data dependency between them on the molecule object that they are processing.

From the point of view of the number of tasks and dependencies, both benchmark versions are equivalent.
8.2.3 What happen in the Compute phase?

There is no significant difference in performance between both benchmark versions in the compute phase but the time of doing versions of objects with COMPSs or with dataClay.

The \texttt{\<init\>} and \texttt{\<computeCenterOfMass\>} tasks modify the object they depend on, therefore a new version of the object is created on the host the task is going to be executed. Once the version object is created the only difference is that for dataClay exists the overhead of calling the \texttt{\<executeTask\>} method of the Storage Interface, for dataClay to execute the task method in the local storage backend.

8.2.4 What happen in the Results phase?

The code for the \texttt{\<results\>} phase in both benchmark versions is exactly the same:

```java
// Results
for (Molecule molecule : molecules) {
    System.out.println("[Producer] " + molecule.getName() + " with center mass " + molecule.getCenter()[3]);
}
```

An access to each molecule is done sequentially in the master node. In this case, it is to get the name and the mass computed for each molecule. The purpose of this access is to model the fact that at the end of the parallel execution the application may only need to access from the main routine executed in the master node to a small piece of data stored remotely in a worker node.

This access to a molecule is solved different on each benchmark version. In the File version, COMPSs has to transfer back to the master the full object from the worker node where it is stored in cache. The bigger the object is the longer the transfer to be done.

But with dataClay, there is no need to transfer the full object. In this case, the molecule is a persisted SCO, that is, a wrapper to a persistent object stored in a backend. The wrapper allows to query the storage platform to get only an attribute of the persistent object, its name and the mass in this example. So, to access to a small piece of data like a single attribute there is no need to transfer the full object to the master node. Later in the experiments results it is shown that this feature is what makes dataClay worth to be used together with COMPSs.
8.3 Experimental Analysis

The experimental analysis have been done running the `<Producer>` benchmark, previously described, with COMPSs & dataClay on Marenostrum III. PyCOMPSs & Hecuba runs also in Marenostrum III but this alternative is not mature enough. Therefore the following sections presents only the evaluation of COMPSs & dataClay.

8.3.1 Memory Issues

There are two kind of memory problems that can happen on nodes when running the experiments:

- Out-of-Memory Exception: It happens when the Java Virtual Machine is configured with few memory and the Garbage Collector is not longer able to free enough memory. At this point the execution ends with an out-of-memory exception thrown by the JVM and a dump file is generated (hs_err_pid<pid>.log file).

- Process Killed: It happens when a process requires more memory but there is no more memory free on the node. At this point, the Linux kernel sends a signal to kill such process. The `<cgroups>` (abbreviated from control groups) Linux kernel feature is what limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes and kills them when limits are passed.

The figure below shows a failed experiment, where in one worker node (bottom right), a process try to pass the node memory limit and the operative system of the node kills the process.
The solution for the first problem, the Out-of-Memory exception, is to increase the maximum amount of memory the JVM can use, this is done using the `-Xmx` JVM option. But to avoid the second problem, the process to be killed, the sum of memory used by all the processes in the node never should exceed the memory limit the node has.

The nodes in this experiments have 32GB of memory each. The JVMs used by COMPSs & dataClay respectively, were configured to have a maximum of 20GB. This doesn’t mean that the JVM will use 20GB to execute the application, it could use much less, but on the other hand there is no guarantee that still the sum of the memory used by all the processes in the node doesn’t pass the node memory limit, it is something that only can be seen at execution or post-execution time. For example, the figure below shows that with this configuration the previous experiment ends successfully with around 26GB of memory used on each worker node.
Figure 8.4: Memory profile of a Job that fits the node memory limit.

In the experiments done, with 2 worker nodes, the minimum and maximum memory set on each JVM was \(-Xms 2g\) and \(-Xmx 20g\) respectively. With this values experiments with 500 objects doesn’t pass the worker node memory limit. For more objects and same number or worker nodes, maybe another memory configuration will be needed in order to avoid the memory problems explained here.

### 8.3.2 Object Size

*Which is the object size that makes dataClay worth to be used together with COMPSs?*

In the experiments done, the Molecule class will be the only class registered as SCO in dataClay. Changing the object size can be done just changing the number of atoms of each molecule.
Figure 8.5: Molecule class.

If each molecule is read from a file on disk, where the molecule has been codified in a simple way, for example in a human readable format, just writing its values in sequential order. As the number of atoms come from $10^2$ to $10^6$, the size of such file on disk is shown in Table 8.1.

<table>
<thead>
<tr>
<th>Class Name - Allocated Objects</th>
<th>Bytes Allocated</th>
<th>Objects Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>sun.misc.FloatingDecimalBSCToBinaryBuffer</td>
<td>576 B</td>
<td>24</td>
</tr>
<tr>
<td>org.apache.xml.stream.XMLInputSource</td>
<td>1,024 B</td>
<td>32</td>
</tr>
<tr>
<td>float[][] getDummyAtoms</td>
<td>3,344 B</td>
<td>125</td>
</tr>
<tr>
<td>float[] getDummyCenter</td>
<td>16,144 B</td>
<td>658</td>
</tr>
<tr>
<td>void computeCenterOfMass</td>
<td>2,016 B</td>
<td>1</td>
</tr>
<tr>
<td>float[][][]</td>
<td>20,016 B</td>
<td>1</td>
</tr>
<tr>
<td>float[][]</td>
<td>120,144 B</td>
<td>10,035</td>
</tr>
<tr>
<td>float[]</td>
<td>4,816 B</td>
<td>1</td>
</tr>
<tr>
<td>float</td>
<td>16,656 B</td>
<td>1</td>
</tr>
<tr>
<td>double</td>
<td>16,656 B</td>
<td>1</td>
</tr>
<tr>
<td>Table 8.1: Size of file on disk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To know how much memory needs each molecule, a memory profiling has been done using the VisualVM profiling tool [https://visualvm.java.net/]. Figure 8.6 shows the profile for 1 molecule and a number of atoms from $10^2$ to $10^6$.

Table 8.1 summarizes the file size and amount of memory needed for 1 molecule and a number of atoms from $10^2$ to $10^6$. The size in memory almost double the size on disk, this could be because to represent a number as float in memory more bytes are needed than to write the same number with chars on disk.

<table>
<thead>
<tr>
<th># Atoms in a Molecule</th>
<th>File Size</th>
<th>Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 10^2$</td>
<td>1,8K</td>
<td>3.3K</td>
</tr>
<tr>
<td>$5 \times 10^2$</td>
<td>8,9K</td>
<td>16.1K</td>
</tr>
<tr>
<td>$1 \times 10^4$</td>
<td>18K</td>
<td>32.1K</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>88K</td>
<td>160K</td>
</tr>
<tr>
<td>$1 \times 10^4$</td>
<td>176K</td>
<td>320K</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>879K</td>
<td>1.6M</td>
</tr>
<tr>
<td>$1 \times 10^5$</td>
<td>1,8M</td>
<td>3.2M</td>
</tr>
<tr>
<td>$5 \times 10^5$</td>
<td>8,6M</td>
<td>16.0M</td>
</tr>
<tr>
<td>$1 \times 10^6$</td>
<td>18M</td>
<td>32.0M</td>
</tr>
</tbody>
</table>

Table 8.1: Number of atoms, File and Memory size.

The experiments were done with 2 worker nodes. 500 is the number of molecules that runs without memory problems on 2 workers, and allows to increase the number of atoms till $1 \times 10^6$.

More than 500 molecules implies more objects and tasks and more memory will be needed by the COMPSs runtime and the Storage Platform leading to memory problems on nodes.
Chapter 8. Analysis

The execution time by object size is plotted on Figure 8.7.

![Figure 8.7: COMPSs alone / with dataClay execution time vs. object size.](image)

These are the execution times plotted on Figure 8.7:

<table>
<thead>
<tr>
<th></th>
<th>alone</th>
<th>with dataClay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[alone]</td>
<td>7.69</td>
<td>7.69</td>
</tr>
<tr>
<td></td>
<td>7.65</td>
<td>7.65</td>
</tr>
<tr>
<td></td>
<td>19.94</td>
<td>19.94</td>
</tr>
<tr>
<td></td>
<td>37.52</td>
<td>37.52</td>
</tr>
<tr>
<td></td>
<td>182.37</td>
<td>182.37</td>
</tr>
<tr>
<td></td>
<td>368.96</td>
<td>368.96</td>
</tr>
<tr>
<td></td>
<td>1805.26</td>
<td>1805.26</td>
</tr>
<tr>
<td></td>
<td>3682.27</td>
<td>3682.27</td>
</tr>
<tr>
<td>[with dataClay]</td>
<td>7.64</td>
<td>7.22</td>
</tr>
<tr>
<td></td>
<td>7.30</td>
<td>7.30</td>
</tr>
<tr>
<td></td>
<td>7.91</td>
<td>7.91</td>
</tr>
<tr>
<td></td>
<td>9.92</td>
<td>9.92</td>
</tr>
<tr>
<td></td>
<td>23.33</td>
<td>23.33</td>
</tr>
<tr>
<td></td>
<td>47.48</td>
<td>47.48</td>
</tr>
<tr>
<td></td>
<td>196.35</td>
<td>196.35</td>
</tr>
<tr>
<td></td>
<td>556.51</td>
<td>556.51</td>
</tr>
</tbody>
</table>
Figure 8.8 shows in logarithmic scale that for objects with a size in memory equal or greater than 0.16 MB (160 KB), COMPSs with dataClay give better execution times than COMPSs running alone.

Figure 8.8: COMPSs alone / with dataClay execution time (logarithmic scale) vs. object size.
Chapter 8. Analysis

Figure 8.9 shows the Paraver trace for the benchmark File version with 500 objects of 16MB each. The execution runs on 1 master and 2 worker nodes. In the trace, the first thread is the master and there are 4 threads more for each worker. The light-blue color in a line means that the thread is idle, therefore, the trace shows that workers are computing, that is executing tasks, at the beginning but once the parallel computation ends they stay idle meanwhile the master is getting objects from them that are going to be accessed in the application.

Objects are accessed in the Results phase of the benchmark to get the result of the Compute phase. In the File version this causes a synchronization of the object that means that it has to be transferred from the worker node the updated object is, to the master node. The trace shows that the Result phase is a bottleneck and takes a long time to complete.

![Figure 8.9: Trace for benchmark File version with 2 workers.](image)

Figure 8.10 shows the equivalent execution for the benchmark dataClay version. It can be seen that as objects are accessed through dataClay in the Results phase, there is no need to do a synchronization (transfer) of the object and there is no bottleneck this time, only the fields of the object needed are requested, not the entire object. At first sight both versions takes the same time when workers are computing in parallel.

Therefore, object synchronizations are the cause of the time difference between both benchmark versions. It is seen here that the use of dataClay allows to access the data avoiding big data movements.
8.3.3 Scalability

Scalability is the capability to increase resources to yield a linear (ideally) increase in service capacity.

Scaling up is the commonly used term for achieving scalability using better, faster, and more expensive hardware. Scaling up includes adding more memory, adding more or faster processors, or simply migrating the application to a more powerful, single machine. Typically, this method allows for an increase in capacity without requiring changes to source code. Administratively, things remain the same since there is still only one machine to manage.

Scaling out leverages the economics of using commodity PC hardware to distribute the processing load across more than one server. Although scaling out is achieved using many machines, the collection essentially functions as a single machine. By dedicating several machines to a common task, application fault tolerance is increased. Of course, from the administrator’s perspective, scaling out also presents a greater management challenge due to the increased number of machines.

Only the parallel part of an application can be scaled-out, as the workload is spread among the resources, therefore scaling out the benchmark File version will not give relevant improvements as the sequential part, the Result phase done by the master, dominates the execution.

Figure 8.11: Trace for benchmark File version with 2 workers. The Results phase is done sequentially by the master and dominates the execution.

Figure 8.12: Trace for benchmark File version with 16 workers. The execution time does not scale-out.
Figure 8.12 shows a trace for the benchmark File version that has been run with 16 workers. Although the parallel computation scale-out in the worker threads, there is no relevant improvement in the execution time as the sequential part in the master thread dominates the execution as mentioned before.

The benchmark dataClay version has been scaled out in Marenostro III from 2 to 64 worker nodes.

Although for 2 worker nodes and an object size of 0.16MB (160KB), COMPSs with dataClay starts giving better performance than COMPSs alone. When more worker nodes are added there is no improvement in the execution time for a fixed workload (input). This is because, in the benchmark used, task duration is direct proportional to the object size. The smaller the objects the shorter the tasks are. Adding more worker nodes will only yield to have idle workers because the 2 initial ones can handle all the workload.

Therefore, to scale out with a fixed workload and no idle worker nodes, bigger objects with a size of 16MB each were used in the experiments done. With bigger objects the tasks are longer and the workload is greater, that is, the 2 initial worker can’t handle easily the workload and when more worker nodes are added the execution time improves.

![Benchmark Scale-out](image)

Figure 8.13: Benchmark Scale-out.
These are the execution times plotted in Figure 8.13:

\[
\text{with dataClay} \quad 188.482 \quad 97.870 \quad 52.862 \quad 32.238 \quad 24.881 \quad 33.521
\]

Experiments show that the benchmark dataClay version doesn't scale out linearly as shown in Figure 8.13. The reasons why the benchmark doesn't scale as expected are explained later.

### 8.3.4 Speedup

*Speedup is the effect of applying an increasing number of resources to a fixed amount of work to achieve a proportional reduction in execution times*, defined as:

\[
\text{Speedup} = \frac{\text{Small system elapsed time}}{\text{Large system elapsed time}}
\]

Speedup is linear if the speedup is \(N\). That is, the small system elapsed time is \(N\) times larger than the large system elapsed time.

For example if a system with 2 worker nodes does the job in "n" seconds and a larger system that doubles (\(N=2\)) the number of worker nodes, that is 4 worker nodes, does the job in "n/2" seconds, then the speedup is "n/(n/2)=2", which is a linear speedup because such speedup is achieved due to a 2 times powerful system.

The above is only true, if adding more worker nodes does not cause an overhead in the system. For example, adding more worker nodes could make the master node to take more time when scheduling a task to a resource, as more resources are available but their status and data locality are checked for each of them.

In our case, the benchmark speedup is sub-linear. The reasons to have a sub-linear speedup are explained later.
8.3.5 Locality

In computer science, the principle of locality, is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed. There are two basic types: temporal and spatial locality.

Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality refers to the use of data elements within relatively close storage locations.

Locality is merely one type of predictable behavior that occurs in computer systems. Systems that exhibit strong locality are great candidates for performance optimization through the use of techniques such as the cache, prefetch or prediction.

The <Producer> benchmark get benefit from temporal locality. That is, tasks with a dependency on the same object are sent to the same worker node in order to reuse the object stored in the node. When such tasks go to the same worker node the performance is the best possible. But when one of those tasks is sent...
to a different worker node there is a penalization due to the need of transfer the object to a different node.

In the dataClay benchmark version, 3 types of tasks are sent to the worker nodes in programming order: `<makePersistent>`, `<init>` and `<computeCenterOfMass>`. As the `<init>` and the `<computeCenterOfMass>` modify the object a new version of the object must be done previously to these tasks. The cost of doing a new version is greater if the object is not already in the node because data needs to be transferred. The situation when data is not present in the node where a task is going to be executed is called here a "data locality miss".

In the case of the `<init>` tasks the cost of a data locality miss is smaller because the object is empty, but in the case of `<computeCenterOfMass>` the cost of a data locality miss is the highest one because the object has been initialized before and a big object will be transferred to the node where the task is going to be executed.

Table 8.2 shows the number of transfers due to a data locality miss with high cost, for the same workload and different number of worker nodes. As such transfers happen between workers, the relation between number of transfers per worker is shown also in the table. Later we will see that these transfers with a high cost happen at the end of the execution, when few tasks remain pending, and for the shorter executions, those with 16, 32 or 64 workers, this fact will have a bigger impact in the execution time.

<table>
<thead>
<tr>
<th>#Workers</th>
<th>#High-cost transfers</th>
<th>#Transfers/#Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>2.9</td>
</tr>
<tr>
<td>16</td>
<td>30</td>
<td>1.9</td>
</tr>
<tr>
<td>32</td>
<td>41</td>
<td>1.3</td>
</tr>
<tr>
<td>64</td>
<td>110</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Table 8.2: Number of transfers due to a data locality miss with high cost.

8.3.6 Sub-linear Speedup

Two main factors are the reason to have a sub-linear speedup, the management overhead that increasing the number of workers introduce in the master and data locality misses. This factors have a greater impact the shorter the executions are.
8.3.6.1 Management Overhead

The `<TaskScheduler>` component has a window of N tasks free of dependencies and ready to be executed on a set of resources. When a resource is free the N task are explored to select the task with highest score, that is, the task with more data locality for that resource. Each time that a resource ask for a task the `<TaskScheduler>` will be busy processing the request with a time-cost factor of O(N).

If more resources are added, there could be more resources free at a time, let’s say M. There will be M resources pending for a task and the `<TaskScheduler>` will have a time-cost factor of O(MxN) to serve the last request.

At first sight, with more resources more tasks could be processing in parallel. But also with more resources, the service time of the `<TaskScheduler>` component could increase and those resources could be idle instead waiting for the next task. This is shown in Figure 8.16 where a blue space between two tasks means that the resource is idle just waiting for the next task while the `<TaskScheduler>` serves the other resources.
Another kind of management overhead is the time needed to initialize the workers and the storage backends. Although they are done in parallel the initialization needs a time and when the execution times are shorter this time becomes important. This is shown in Figure 8.26 with a trace view.

### 8.3.6.2 Locality misses

Each task in the scheduler window has data locality affinity for a resource. In our case it is the resource that stores the greater number of task parameters.

In figure 8.17 rectangles are tasks to be executed. Tasks are in a queue. The scheduler window has size 2 and moves from left to right over the queue. Each resource has assigned a different color. The circle in the right is the current request for a task with the color of the resource it comes from. The color of a task gives the data locality affinity for a resource. Each row in the picture represent a step in time.

If the requests arrive in round-robin from the resources the data locality matches 100%. There is always a task in the window that has data locality affinity for the current resource.

But in a real scenario, the order the requests arrives could not follow a round-robin pattern, a certain resource could be faster or there could be communication problems. In figure 8.18, two requests from the same resource are received con-
secutively, in this case thanks to the scheduler window there is always a task with data locality affinity for the current resource and data locality misses are avoided.

But if figure 8.19 two request from the same resource are received consecutively and the scheduler window is not enough to avoid a data locality miss for the last request.

Figure 8.17: Requests arrive in a round-robin way.

Figure 8.18: Requests arrive out-of-order without locality miss.

Figure 8.19: Requests arrive out-of-order with locality miss.
Two considerations can be done:

1. As the number of resources grows the probability that the requests arrive out-of-order to the `<TaskScheduler>` grows and more data locality misses could take place.

2. There is another phenomena, that take place when the number of tasks pending of execution is smaller than the window size. It could happen that in that case there is no task with data locality affinity for the current request in the window and a task from the window is assigned to the resource with a data locality miss. As the number of tasks in the window decrease this situation become worse and more data locality misses take place.

Whenever data locality misses take place, during the execution (1) or at the end (2) the impact of them in the execution time is greater for shorter executions.

### 8.3.7 Traces

Execution traces have been generated when the dataClay benchmark version was scaled-out. Traces are plotted in the following figures:

Figure 8.20: 500 objects of 16MB, 1 master thread and 2 workers with 4 threads.

Figure 8.21: 500 objects of 16MB, 1 master thread and 4 workers with 4 threads.
Figure 8.22: 500 objects of 16MB, 1 master thread and 8 workers with 4 threads.

Figure 8.23: 500 objects of 16MB, 1 master thread and 16 workers with 4 threads.

Figure 8.24: 500 objects of 16MB, 1 master thread and 32 workers with 4 threads.

Figure 8.25: 500 objects of 16MB, 1 master thread and 64 workers with 4 threads.
As more workers are added the execution time scale-out until the overheads of adding more resources become relevant:

1. The "Start" and "Stop" times of the COMPSs runtime and the Storage Platform become relevant for the shorter executions.

2. The scheduler spend more time when there are more resources to serve and this service time become critical when the execution time is shorter.

3. There are more data movements due to data locality misses when more workers are added and the shorter the execution time the greater the impact of the data transfer time.

Figure 8.26: 500 objects of 16MB, 1 master thread and 64 workers with 4 threads.
Figure 8.26 is a zoom-in of the trace corresponding to the execution with more workers, 64. The execution parts different from computing have been marked as follow:

1. Start
2. Schedule
3. Transfer
4. Synchronize
5. Stop

The synchronization part correspond to the synchronization of the $<\text{PSCOId}>$ objects that are smaller than the original objects synchronized in the benchmark File version. The synchronization time is smaller for the benchmark dataClay version than for the benchmark File version but also when executions are shorter this time also becomes relevant. As seen here, the parts where no computation is done become critical as they dominate the execution time.
Chapter 9

Conclusions

Advances in data storage technology with Non-Volatile RAMs (NVRAMs) or Storage Class Memories (SCMs) devices, with low latencies, high bandwidth and a byte-addressable interface, have opened the path for a new generation of Object Storage Platforms that take advantage of this high-performance and byte-addressable storage devices.

dataClay has been presented in this master thesis as one of this new emerging object storage platforms. dataClay is one of the projects of the Storage System group at the Barcelona Supercomputing Center (BSC-CNS). dataClay stores the objects and their relationships as they are in volatile memory. dataClay stores data, code and behaviour policies of the object. This differs from other approaches that offer objects as the persistent abstraction that only store the data.

Hecuba is another example of Object Storage Platform presented in this work. Hecuba is a project of the Autonomic Systems and e-Business Platforms at BSC-CNS. Hecuba explodes the space locality where related objects are stored in blocks that are replicated in the storage platform backends. A block of related objects can be therefore efficiently accessed and iterated in the same host.

To allow the scientific community to get benefit from these new object storage platforms in their high-performance applications, programming models and middleware need to be adapted. In this work, the COMPSs programming model for distributed applications, the main research line of the Workflows and Distributed Computing group at BSC-CNS, and its corresponding middleware, the COMPSs runtime, has been integrated with both dataClay and Hecuba object storage platforms.

The integration starts understanding two important concepts that are key in this new storage paradigm:

- The Persistent Object (PO) abstraction, presented here for Non-Volatile RAM (NVRAM) devices as the equivalent to the file abstraction for Hard Disk
The Self-Contained Object (SCO) class, presented here as the equivalent to the File class available in many object oriented programming languages. As the File class allows to work with files stored in HDDs, the SCO class allows to work with Persistent Objects in NVRAM devices.

After the analysis of the features and the architecture of programming model, middleware and object storage platforms, an application interface has been proposed, the Storage API. The Storage API defines the classes and methods that allow both, the application and middleware, to work with Self-Contained Objects.

The use of the Persistent Identifiers in such integration, has been key not only to save memory in the middleware structures, as Self-Contained Objects are managed through its identifiers, but also to work with Self-Contained Objects that come from other languages different from the one the middleware has been written. In this sense, to allow the interoperability with other languages different from Java, the binding-common library has been adapted to work with Self-Contained Objects from languages like C/C++ and Python through its Persistent Identifiers.

A distributed computing application that works with a big amount of data usually requires during the computation the synchronization of data, with the host the user is connected to, or the collection of the results at the end of the execution. This is a common behaviour in scientific applications and a benchmark that models such behaviour has been used in the experimental analysis.

As seen in this work, if these applications run just with a distributed middleware like COMPSs data movements will take place due to synchronizations or accesses from the main application thread. These data movements become critical as most of the execution time will be spent on them. dataClay offers an implementation of the Self-Contained Objects, that allows the access to the data of a Persistent Object from the application without big data movements, as only the required field is transferred.

Experiments done in the supercomputer Marenostrum III hold by the Barcelona Supercomputing Center, using two versions of the same benchmark, one that stores objects in files and another that stores them in dataClay, show that:

For objects with a size equal or greater than 0.16MB (160KB), the use of COMPSs & dataClay give better execution times than COMPSs running alone.

That is, there is a point where the object size implies long synchronizations in the file version of the benchmark that start to dominate the execution time.
Chapter 9. Conclusions

Things become worse if synchronizations are done in the sequential part of the application as adding more resources will not give any relevant improvement, this have been shown when scaling-out the file version of the benchmark where only the time of the parallel part got reduced.

On the other side, experiments show that the dataClay version of the benchmark scales out better but not as could be expected. There are some aspects that arise as the executions become shorter:

- The "Start" and "Stop" times of COMPSs & DataClay become relevant.
- The COMPSs runtime, especially the Scheduler component, spend more time when there are more resources to serve and this service time become critical.
- With more resources, there are more data movements due to data locality misses and the shorter the executions the greater the impact of the transfer times.
- The synchronization times are smaller for the dataClay version of the benchmark but also when executions are shorter they become relevant.

But despite the points mentioned above, that could be improved, the benefits in the execution time that a framework like COMPSs & DataClay could provide, make the marriage of programing models and these new generation of object storage platforms a path worth to explore to face the exascale and big-data challenges of the 2020 horizon.
Chapter 10

Future Work

10.1 Other Benchmarks

The `<Producer>` benchmark is a synthetic benchmark used for the validation and analysis of the integration of COMPSs with dataClay. But in order to compare this integration with other alternatives more standard benchmarks should been used.

In this sense other benchmarks like the `<K-means>`, `<Wordcount>`, `<Sort-by-key>` or `<Matrix-multiplication>` are been adapted to run with COMPSs and dataClay. Figure 10.1 shows a trace of the execution of the `<K-means>` benchmark with 2 workers and the following parameters:

```
KMeans with random generated points
-----------------------------------
Running with the following parameters:
- Clusters: 100
- Iterations: 10
- Points: 8000000
- Dimensions: 50
- Fragments: 256
```

Figure 10.1: Example of a `<K-means>` benchmark trace with 2 workers.

In this case the points will be grouped into k=100 clusters. The process is iterative and can finish by number of iterations or by convergence. Each point
has $d=50$ dimensions. To make the process parallel the amount of points have been divided into fragments. Points on each fragment are grouped into the $k$ clusters and partial results are merge on each iteration.

Here, the Fragment class was selected to inherit from the Self-Contained Object class and its objects (class instances) to be stored as Persistent Objects.

## 10.2 COMPSs & Hecuba Analysis

The analysis of the integration of COMPSs with object storage platforms has been focused on dataClay as its installation in Marenostrum supercomputer was done and validated earlier than Hecuba.

At the moment of writing this document Hecuba was on its first steps in Marenostrum. Some small executions have been already done but still some debug and adjustments are needed. Figure 10.2 show a trace of an execution of the Wordcount benchmark with 1900 millions of words, that is, 10GB of data aprox.

Figure 10.2: Trace of a Wordcount benchmark execution with COMPSs and Hecuba.

Once the installation of Hecuba had been validated by the BSC groups, an analysis like the one done with dataClay should be done with Hecuba.

## 10.3 The Scheduler

It has been seen that, in a middleware for distributed computing like COMPSs, the scheduler is a component that has a relevant impact on the application performance.

Not only things like the increment in the service time when more resources are added have to be considered but also the capacity of the scheduler to match
Chapter 10. Future Work

the data locality when a task is assigned to a resource. If the scheduler doesn’t match the data locality, more time will be spend with data movements between resources.

The policy the scheduler follows must take into account the information about data locality that the underlying object storage platform provides. But not only should consider the number of objects that are stored in a resource but also the size of such objects. It is a better decision to move some objects that have a small size than just one object but very big.

In this sense the Storage API should be extended in order to provide a method that allows the Scheduler to ask for the size of a Persistent Object. This method should then been implemented by the object storage platform.

10.4 Consumer

If Persistent Objects shared by two distributed applications are modified in parallel by one, <Producer>, and read in parallel later by another, <Consumer>, the <consolidateVersion> of the corresponding Self-Contained Objects in the first one must be invoked to make the second able to see the changes in the Persistent Objects. The <Consumer> application could be, for example, an application to visualise the results of the <Producer> one in real time.

Some experiments in the development infrastructure have been done to test the scenario proposed here, even with several <Consumer> applications. Similar experiments and a posterior analysis could be done in Marenostrum supercomputer.
11.1 Benchmark Source Code

11.1.1 Molecule.java

package severo.moleculeArray;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.io.Serializable;

public class Molecule implements Serializable {
    private String name;
    private float[][] atoms;
    private float[] center;

    public Molecule() {
    }

    public Molecule(int i) {
        String name = "Molecule" + i;
        setName(name);
    }

    public void init(int n) {
        this.center = getDummyCenter();
        this.atoms = getDummyAtoms(n);
    }
}
public String getName() {
    return name;
}

public void setName(String name) {
    if (name == null) {
        throw new IllegalArgumentException("[Molecule]: ERROR, argument cannot be null");
    }
    this.name = name;
}

// Computes the center of mass of the molecule based on its atoms.
public void computeCenterOfMass() {
    float sumX = 0;
    float sumY = 0;
    float sumZ = 0;
    float totalMass = 0;

    System.out.println("[Molecule]: Computing center of mass of molecule " + this.getName());
    float[] atom;
    int n = this.atoms.length;
    System.out.println("[Molecule]: Number of atoms is " + n);
    for (int i = 0; i < n; ++i) {
        atom = this.atoms[i];
        sumX += atom[3] * atom[0];
        sumY += atom[3] * atom[1];
        sumZ += atom[3] * atom[2];
        totalMass += atom[3];
    }

    if (this.center == null) {
        System.err.println("[Molecule]: Center is null");
    } else {
        this.center[0] = sumX / totalMass;
        this.center[1] = sumY / totalMass;
        this.center[2] = sumZ / totalMass;
        this.center[3] = totalMass;

        System.out.println("[Molecule]: X : " + this.center[0]);
        System.out.println("[Molecule]: Y : " + this.center[1]);
        System.out.println("[Molecule]: Z : " + this.center[2]);
    }
}
System.out.println("[Molecule]: Mass : " + this.center[3]);
}
return;
}

// Dummy method to generate atoms for a molecule
private float[][] getDummyAtoms(int nAtoms) {
    float[][] atoms = new float[nAtoms][];
    for (int i = 0; i < nAtoms; i++) {
        atoms[i] = new float[] { 1, 1, 1, 100 }; // Dummy atom coordinates and mass
    }
    return atoms;
}

private float[] getDummyCenter() {
    float[] center = new float[] { 0, 0, 0, 0 }; // Dummy center coordinates and mass
    return center;
}

public void writeToFile(String filename) {
    try {
        File file = new File(filename);
        FileOutputStream fop = new FileOutputStream(file, true);
        OutputStreamWriter osw = new OutputStreamWriter(fop);
        BufferedWriter bwriter = new BufferedWriter(osw);
        // Molecule (name)
        bwriter.write(this.getName());
        // Atoms (point coordinates and mass)
        int n = this.atoms.length;
        for (int i = 0; i < n; i++) {
            bwriter.write(" ");
            bwriter.write(String.valueOf(this.atoms[i][0]));
            bwriter.write(" ");
            bwriter.write(String.valueOf(this.atoms[i][1]));
            bwriter.write(" ");
            bwriter.write(String.valueOf(this.atoms[i][2]));
            bwriter.write(" ");
            bwriter.write(String.valueOf(this.atoms[i][3]));
        }
        // Center (point coordinates and mass)
        bwriter.write(" ");
        bwriter.write(String.valueOf(this.center[0]));
        bwriter.write(" ");
        bwriter.write(String.valueOf(this.center[1]));
        bwriter.write(" ");
        bwriter.write(String.valueOf(this.center[2]));
        bwriter.write(" ");
        bwriter.write(String.valueOf(this.center[3]));
    }
}
bwriter.write(String.valueOf(this.center[1]));
bwriter.write(" ");
bwriter.write(String.valueOf(this.center[2]));
bwriter.write(" ");
bwriter.write(String.valueOf(this.center[3]));
bwriter.newLine();
bwriter.close();
bsw.close();
fsop.close();
System.out.println("[Molecule] " + name + " written to file");
} catch (IOException e) {
    System.out.println("[Molecule] " + e.getMessage());
}

private float[] getCenter() {
    return center;
}
11.1.2 ProducerFile.java

```java
package severo.producer;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import severo.moleculeArray.Molecule;

public class ProducerFile {

    // MASTER MAIN
    public static void main(String[] args) throws Exception {

        // Read arguments
        int nMols = Integer.parseInt(args[0]);
        int nAtoms = Integer.parseInt(args[1]);

        String filename = "molecules_" + nMols + "_" + nAtoms + ".dat";
        System.out.println("[Producer] Running with " + nMols
                + " molecules with " + nAtoms + " atoms...");

        // Create
        System.out.println("[Producer] Create molecules (sequential)...");
        ArrayList<Molecule> molecules = new ArrayList<Molecule>();
        for (int i = 1; i <= nMols; i++) {
            Molecule molecule = new Molecule(i);
            molecules.add(molecule);
        }

        // Initialize
        System.out.println("[Producer] Initialize molecules (parallel)...");
        for (Molecule molecule : molecules) {
            molecule.init(nAtoms);
        }

        // Compute
        System.out.println("[Producer] Compute molecules (parallel)...");
        for (Molecule molecule : molecules) {
            molecule.computeCenterOfMass();
        }

        // Write
        System.out.println("[Producer] Write molecules to a file (parallel)...");
    }
}
```
for (Molecule molecule : molecules) {
    molecule.writeToFile(filename);
}

// Results
for (Molecule molecule : molecules) {
    System.out.println("[Producer] " + molecule.getName() + " with center mass " + molecule.getCenter()[3]);
}

molecules.clear();
System.out.println("[Producer] End of execution.");
11.1.3 ProducerFileItf.java

```java
package severo.producer;

import integratedtoolkit.types.annotations.Method;
import integratedtoolkit.types.annotations.Parameter;
import integratedtoolkit.types.annotations.Parameter.Direction;
import integratedtoolkit.types.annotations.Parameter.Type;

public interface ProducerFileItf {

    @Method(declaringClass = "severo.moleculeArray.Molecule")
    void computeCenterOfMass();

    @Method(declaringClass = "severo.moleculeArray.Molecule")
    void init(@Parameter int n);

    @Method(declaringClass = "severo.moleculeArray.Molecule")
    public void writeToFile(
        @Parameter(type = Type.FILE, direction = Direction.OUT) String filename
    );

}
```
11.1.4 ProducerDataClay.java

```java
package severo.producer;

import java.util.ArrayList;
import severo.moleculeArray.Molecule;
import storage.StorageItf;

public class ProducerDataClay {
    // MASTER MAIN
    public static void main(String[] args) throws Exception {
        // Read arguments
        int nMols = Integer.parseInt(args[0]);
        int nAtoms = Integer.parseInt(args[1]);
        System.out.println("[Producer] Running with " + nMols
                         + " molecules with " + nAtoms + " atoms...");

        // Create
        System.out.println("[Producer] Create molecules (sequential)...";
        ArrayList<Molecule> molecules = new ArrayList<Molecule>();
        for (int i = 1; i <= nMols; i++) {
            Molecule molecule = new Molecule(i);
            molecules.add(molecule);
        }

        // Persist
        System.out.println("[Producer] Persist molecules (parallel)...";
        int i = 1;
        for (Molecule molecule : molecules) {
            molecule.makePersistent("Molecule" + i);
            i = i++;
        }

        // Initialize
        System.out.println("[Producer] Initialize molecules (parallel)...";
        for (Molecule molecule : molecules) {
            molecule.init(nAtoms);
        }

        // Compute
        System.out.println("[Producer] Compute molecules (parallel)...";
        for (Molecule molecule : molecules) {
```
molecule.computeCenterOfMass();
}

// Results
System.out.println("[Producer] Barrier (sequential)...");
for (Molecule molecule : molecules) {
    System.out.println("[Producer] " + molecule.getName() + " with center mass " + molecule.getCenter()[3]);
}
molecules.clear();
System.out.println("[Producer] End of execution.");
11.1.5 ProducerDataClayItf.java

```java
package severo.producer;

import integratedtoolkit.types.annotations.Method;
import integratedtoolkit.types.annotations.Parameter;
import integratedtoolkit.types.annotations.Parameter.Type;
import severo.moleculeArray.Molecule;

public interface ProducerDataClayItf {
    @Method(declaringClass = "severo.moleculeArray.Molecule")
    void computeCenterOfMass();

    @Method(declaringClass = "severo.moleculeArray.Molecule")
    void init(
            @Parameter int n
    );

    @Method(declaringClass = "severo.moleculeArray.Molecule")
    void makePersistent(
            @Parameter(type = Type.STRING) String alias
    );
}
```

11.2 Storage API

11.2.1 SCO Interface

```java
package storage;

public interface SCOItf {

    /**
     * An object is not persistent anymore after this method is invoked.
     * @throws StorageException Throws a storage exception if the process fails.<br>
     * The exception message should provide a useful description of the error.
     */
    void deletePersistent() throws StorageException;

    /**
     * Returns the identifier of a persistent object in the underlying storage system.
     * @return The identifier of the persistent object or null if the object is
     */
}
```
**not persistent.**
* @throws StorageException Throws a storage exception if the process fails.<br>
* The exception message should provide a useful description of the error.
*/
String getID() throws StorageException;

/**
* Converts an object into a persistent object.
* <p>
* Persistent objects are managed by the underlying storage system.
* @param name Object name. Required for some storage systems.
* @throws StorageException Throws a storage exception if the process fails.<br>
* The exception message should provide a useful description of the error.
*/
void makePersistent(String name) throws StorageException;

11.2.2 Storage Interface

package storage;

import java.util.List;

public interface StorageItf {

/**
* Merges the changes made in a versioned object into the original one.
* <p>
* Both objects must be in readonly state. Identifiers are never merged.
* @param versionedObjectID The identifier of the versioned object.
* @throws StorageException Throws a storage exception if the cosolidation
* process fails.<br>
* For example:<br>
* - If there is no object with ID versionedObjectID<br>
*/
void consolidateVersion (String versionedObjectID) throws StorageException;

/**
* Deletes the referenced object.
* @param objectID The identifier of the object to delete.
* @throws StorageException Throws a storage exception if the process fails.<br>
* For example:<br>
* - Connection problems, etc.<br>
* The exception message should provide a useful description of the error.
*/
void delete (String objectID) throws StorageException;

/**
 * Returns the object referenced.
 * @param objectID The identifier of the object.
 * @return The object itself.
 * @throws StorageException Throws a storage exception if the process fails.
 * For example:
 * - If there is no object with ID objectID
 * - Others (connection problems, etc.)
 * The exception message should provide a useful description of the error.
 */
Object getByID (String objectID) throws StorageException;

/**
 * Returns a list with the location (hostname or IP address) of the referenced object and also the location of each of
 * its replicas if any.
 * @param objectID The identifier of the object.
 * @return A list with the locations.
 * At least the location of the original object that is referenced must be
 * in the list.
 * @throws StorageException Throws a storage exception if the process fails.
 * For example:
 * - If there is no object with ID objectID
 * - Others (connection problems, etc.)
 * The exception message should provide a useful description of the error.
 */
List<String> getLocations (String objectID) throws StorageException;

/**
 * Creates the environment needed for the storage layer.
 * <p>
 * The environment variables must be kept in the interface implementation, and they are never exposed to the users of this interface. For example,
 * the session identifier, versioning information, etc.
 * @param cfgfile Path to the configuration file.
 * @throws StorageException Throws a storage exception if the initialization process fails.
 * For example:
 * - If the format of the configuration file is not correct or incomplete
 * - Others (connection problems, etc.)
 * The exception message should provide a useful description of the error
 */
/*

void init (String cfgfile) throws StorageException;

/**
 * Creates a replica of the referenced object in the specified host.
 * <p>
 * A replica is a copy of the original object and has the same identifier.<br>
 * A replica only can be done if the original object is in readenly state.<br>
 * The replica object will be in readonly state as the original one.<br>
 * @param objectID The identifier of the object to be replicated.<br>
 * @param hostname Name of the host where the replica object must be placed.<br>
 * It can be either a hostname or an IP address.<br>
 * @throws StorageException Throws a storage exception if the replication
 * process fails.<br>
 * For example:<br>
 * - If there is no object with ID objectID<br>
 * - If there is no backend in host hostname<br>
 * - If the replica could not be created in the specified host<br>
 * - Others (connection problems, etc.)<br>
 * The exception message should provide a useful description of the error.
 */
void newReplica (String objectID, String hostname) throws StorageException;

/**
 * Creates a new version of the referenced object in the specified host.
 * <p>
 * The versioned object is a copy of the original object but with
 * a new identifier.<br>
 * To make a new version the original object must be in readonly state.<br>
 * The versioned object will not be in readonly state and can be modified.<br>
 * @param objectID The identifier of the object to be versioned.<br>
 * @param hostname Name of the host where the versioned object must be placed.<br>
 * It can be either e hostname or an IP address.<br>
 * @return The identifier of the versioned object.<br>
 * @throws StorageException a storage exception if the versioning
 * process fails.<br>
 * For example:<br>
 * - If there is no object with ID objectID<br>
 * - If there is no backend in host hostname<br>
 * - If the new version could not be created in the specified host<br>
 * - Others (connection problems, etc.)<br>
 * The exception message should provide a useful description of the error.
 */
String newVersion (String objectID, String hostname) throws StorageException;
/**
 * Executes a method on a specific host asynchronously.
 * @param objectIDstr ID of the target object.
 * @param operationSignature Signature of the method to be executed.
 * @param params Parameters for the method.
 * @param destHost Destination host where the method has to be executed.
 * @return The id of the executed request that will receive the callback handler 
 * with the corresponding response.
 * @throws StorageException
 */
String executeTask(final String objectIDstr, final String operationSignature,
final Object[] params, final String destHost, final CallbackHandler callback)
throws StorageException;

/**
 * Processes the callback event produced by a task execution.
 * @param CallbackEvent the event to be processed
 * @return The task result. If it is a SCO, you can retrieve its ID by 
 * calling getID method.
 * @throws StorageException
 */
Object getResult(final CallbackEvent callbackEvent) throws StorageException;

11.2.3 Storage Exception

package storage;

public class StorageException extends Exception {

}

11.2.4 Callback Handler

package storage;

public abstract class CallbackHandler {

/**
 * Method that will process the callback event.
 */
* @param e event received that must be processed.  
*/
protected abstract void eventListener(final CallbackEvent e);
}

11.2.5 Callback Event

package storage;

public final class CallbackEvent {

    public enum EventType {
        FAIL,
        SUCCESS;
    }

    /* Request ID */
    private String requestID;

    /* Type of event */
    private EventType type;

    /* Message for error passing, etc. */
    private String message;

    /* Constructor */
    public CallbackEvent(){}
}


References


