Distributed Flow Control and Intelligent Data Transfer in High Performance Computing Networks

Mehdi Sadeghi

Supervisors:
Prof. Dr. Katharina Mehner-Heindl
Dr. Adham Hashibon

MASTER THESIS

On
Master Degree Program

Communication and Media Engineering

in
University of Applied Sciences Offenburg

March 2015

Declaration of Authorship

I declare in lieu of an oath that the Master Thesis submitted has been produced by me without illegal help from other persons. I state that all passages which have been taken out of publications of all means or un-published material either whole or in part, in words or ideas, have been marked as quotations in the relevant passage. I also confirm that the quotes included show the extent of the original quotes and are marked as such. I know that a false declaration will have legal consequences.

March 31, 2015

Mehdi Sadeghi

Copyright

Abstract

This document contains my master thesis report, including problem definition, requirements, problem analysis, review of current state of the art, proposed solution, designed prototype, discussions and conclusion.

During this work we propose a collaborative solution to run different types of operations in a broker-less network without relying on a central orchestrator.

Based on our requirements, we define and analyze a number of scenarios. Then we design a solution to address those scenarios using a distributed workflow management approach. We explain how we break a complicated operation into simpler parts and how we manage it in a non-blocking and distributed way. Then we show how we asynchronously launch them on the network and how we collect and aggregate results.

Later on we introduce an implemented prototype that demonstrates the proposed design.

To Rozita

Acknowledgement

Foremost, I would like to express my gratitude to my advisor Dr. Adham Hashibon for his continuous support, creativity, encouragement and enthusiasm during the course of this thesis.

Besides my advisor, I would like to sincerely thank my supervisor Prof. Dr. Katharina Mehner-Heindl for her insightful comments, scientific support, and encouragement. Her support has always helped me to continue with more self-confidence.

Preface

This thesis is a research and development effort to accomplish data intensive operations in a distributed manner with a collective but decentralized approach toward workflow management while minimizing data transfer during such operations.

During this thesis I have been responsible to study the problem of my client, define it clearly and assist them in finding a proper solution. Either with finding an already existing approach and integrating it into their development process or with proposing a new approach to address their needs.

My activities include analyzing the problem, collecting requirements, studying state of the art software frameworks and related products, analyzing them against requirements and proposing a solution.

In case of proposing a new approach, it was desired to develop a prototype to demonstrate it. Therefore an application has been developed in order to experiment ideas discussed in this work. This open source application is available online[1] for review. The source files of the current document are also available online[2].

If there are any comments and improvements regarding this document, the author appreciates an email to **sadeghi@mehdix.org**.

[1] https://github.com/mehdisadeghi/konsensus
[2] https://github.com/mehdisadeghi/cme-thesis

Contents

List of Figures

Listings

Chapter 1

Introduction

We are accomplishing this work in order to facilitate creating data-intensive and computationally intensive scientific[1] applications. In this chapter we introduce the problem domain and typical workflow and environment which is in practice today. This work is done in a european material science research institute, therefore throughout this thesis we have their needs in mind[2]. We begin with introducing the problem context.

1.1 Problem Context

European scientific communities launch many experiments everyday, resulting in huge amounts of data. Specifically in molecular dynamics and material science fields. There are many different simulation software which are being used to accomplish multiscale modeling [3] tasks.

These tasks often involve running multiple simulation programs over pre-existing datasets or datasets which are produced by other simulation software to achieve desired results. These datasets often resemble models of physical systems, i.e. information about atoms, molecules, etc. This information would be stored in numeric structures and arrays, however there are other data types such as images, which have their own use cases (depending on the field). In material science community it is about particles and their attributes such as coordinates in a given space and velocity.

Nowadays this is a common practice in many fields of science, where typical non-expert users have to write scripts in order to run their intended applications, log-in to clusters, find all required data sets, move them to a folder accessible by their script, launch and monitor status of the submitted jobs and finally collect output files.

Often there would be more than one data set involved in an operation. Those data sets would be spreaded among multiple machines. In some cases (such as ours) there would be datasets being produced on multiple destinations and they would be

[1] So-called e-Science

[2] See *Material Informatics* http://en.wikipedia.org/wiki/Materials_informatics

[3] For more information see Computational Multiscale Modeling of Fluids and Solids Steinhauser, Martin 2008

requested on other machines for some operations. Then users would have to deal with even more complexity or they would not be able to run such operations at all or would probably have a poor performance if they do.

Every community uses different simulations but they share similar process. In this non-managed approach non-experts have to deal with the complexities of high performance computing systems.

The aforementioned workflow is not flexible and is not managed at all. While simpler and smaller experiments could be handled this way, larger and more complicated experiments require different solutions. Such experiments are the source of many high performance computing (HPC) problems, specially workflow management and data transfer.

Our goal is to find a solution that let us to mange this situation, take the complexities a way from users, create a distributed platform and minimize the transferred data. Moreover we want to deliver these promises regardless of the workstation that user works with to be able to run a network of computers which collaborate together to deliver the task. This will in turn help us to have more parallelism and avoid single point of failure.

1.2 Objectives

There are two main objectives in these thesis as the title suggests:

- distributing the workflow of an application, i.e. the state
- minimizing the amount of transferred data during an operation

First of all we want to focus on collaboration in a distributed application. This is how multiple computers will manage to finish an operation collectively in a distributed environment. We want to find a way to keep the state of the running operation distributed amount all the participants.

Next we want to avoid unnecessary data transfer during an operation as much as possible. We prefer to have the operation be transferred rather than data. However this is not possible all the time, hence minimizing and smart transfer are mentioned.

Both of these objectives are tailored toward the context that this work is done. This means that even thought there are existing workflow management tools and data transfer solutions but those tools do not meet our requirements. This will be discussed in more detail in chapter 2: Requirements.

1.3 Terms and Definitions

We will use a number of terms through this report. Here are the meaning for each.

Node Refers to one computer in the network.

Dataset Consumed and produced data by scientific applications .e.g. NumPy arrays or HDF5 datasets. Data or Data sets also refer to this one.

Application The prototype which has been developed to show case the proposed solution, see 6.

Instance An instance of the application running on a node.

Peer One instance of the network application which is in collaboration with other local or remote instances.

Operation Some functions, carrying logic of our application, which users want to run on datasets.

Task Same as the operation with more emphasize on the output rather than the functionality.

Service Remote procedures provided by the application which could be called remotely.

System The combination of nodes, datasets, application, instances, operations and services as a whole.

User A scientist, researcher or student who uses the system.

1.4 Typical Environment

While working in an institute, often there are many computers which users connect to them remotely. In a typical scientific and research environment users have their own windows or Linux machine and meanwhile they can SSH to other Linux machines on the same network with their user credentials. In such environments it is common to have computer clusters which users access using SSH. Normally there is a job scheduling software such as Sun Grid Engine (SGE) installed on the clusters and users have to submit their jobs using the tools provided by corresponding cluster software. Moreover such job schedulers enforce some policies to job submissions.

There are more common characteristics about these environments which we name a few:

- Users do not have administrator rights and root access to the machines
- Using network shared storage is very common
- Institutes often use LDAP[1] and users can login to any machine with their credentials

[1] https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

While running multiple scientific programs, they often need to exchange data back and forth in order to accomplish one operation. Some operations such as comparison require multiple datasets at the same time which also called All-Pairs problem. [7]. Such operations involve more computation and are more complicated to address. Because those required datasets might not be available on the same machine, hence should be transferred. Having these said, it is cheaper to transfer the operation rather than the data whenever possible.

In another words the programs that we need to run over existing datasets are distributed among multiple computers, clusters or HPC sites. The nature of our experiments, makes it necessary to launch multiple such programs in one run to achieve the desired outcome. This is one reason that we look for distributed solutions which not only have to distribute state of the program among these machines, but have to provide smarter ways to move data between these machines during operation executions.

1.5 Document Overview

This document is organized into 8 chapters. The current chapter is for the opening and introducing the context.

In chapter two we explain the requirements. We try to introduce our fields of interest. These requirements will form our orientation during the rest of this work.

In chapter three some related works will be discussed. We will assess each of them against our requirements and interests.

In chapter four different aspects of the problem will be discussed. We will analyse multiple scenarios. This chapter would prepare the basics for the proposal.

In chapter five we will explain the proposed solution. We will suggest an approach that could solve the discussed scenarios.

In chapter six we talk about our designed prototype and its implementation. We will talk about the chosen technologies and their advantages.

In chapter seven we discuss about the applicability of the proposed solution, some issues and future work.

And the last chpater is for conclusion.

Chapter 2

Requirements

Every group has its own needs. That is the reason that we have many different solutions in the market. We focus on requirements of material science community in this work, even though the results could be applied to similar fields as well.

One of the efforts in material science community is an ongoing european project called SimPhonY. It is focused on delivering a product which will run multiple distributed third-party simulation engines under the cover of one application. This application has to keep state of ongoing simulations and provide them with necessary datasets while running[1]. Our requirements which will be introduced further in this chapter all originate from this project and our results more likely to be used in the same project.

In this chapter we explain the requirements according to our context.

2.1 Assumptions

We have a certain problem which we want to focus on rather than reintroducing solutions that already exist. Here we go through assumptions which are necessary to understand the next chapters.

2.1.1 Data Characteristics

In our context data is mostly numerical and explains characteristics of physical particles such as atoms and molecules. This data is being used in multiscale simulations. One important aspect of our datasets is that they are not critical and we can reproduce them. We are more concerned about preserving the running state of a distributed application rather than preserving datasets, hence is the focus on distributed workflow.

2.1.2 Data Transfer

We assume a data transfer approach is already in place. This could be any file system which supports network storage. Rather than going into details of how data could

[1] http://www.simphony-project.eu/about-simphony/

be transferred more efficiently, we will focus on finding which data to be transferred and from which computer to which destination.

2.1.3 Independent Work

Our work is independent from any simulation or any context related tools. We do not rely on any specific software or very specific datasets. When we refer to domain related topics, it is to help us to better understand the practical use cases of this thesis.

2.2 Requirements

Before bringing in the requirements we take a moment to better depict the environment that these requirements come from.

It is common that users work with scientific data with help from SciPy project[1]. This project alone is a reason why we are biased toward Python, as later we will discuss. SciPy project includes NumPy which is an N-dimentional array package which often is used by users and programs to deal with datasets.

There is a well known portable scientific data file format called HDF5[2]. This format is de facto standard for storing large datasets in many scientific fields which need to store numerical data. Later on we will see that we have used HDF5 as storage in our prototype, even though we are not dependant on it. NumPy and HDF5 play well together, i.e. there are many tools and libraries to store and retrieve NumPy data structurs within a HDF5 file. HDF5 does not need a runtime, having HDF5 library is enough to access and manipulate it.

It is also good to mention that our intended users normally work in places where there is a cluster and they access it using their own workstations via local area network (LAN). It is also very common that they have minimum access rights to their workstations, since administrators do not want to install programs that might increase the risk of problems and the cost of maintenance.

And one more point about operations that we talk about them a lot in this text. Users can run simulations on their own laptop and workstations (which is very common), but when it comes to more time consuming tasks, they need to work with clusters. Usually they login to remote machine or they use some local programs that will submit their scripts to remote resources. However this is not a versatile approach and we need to change this.

We are looking for a solution that puts the user first. We want to be able to give a single user, the opportunity to run complicated operations right from her workstation, while preserving the workflow out of her machine. To help them easily install our application, and to use it with minimum learning curve. We want to let the user to focus on their main tasks rather than distracting them with data and workflow management and put no burden on them.

Having said this short introduction, we continue with requirements.

[1] "A Python-based ecosystem of open-source software for mathematics, science, and engineering" www.scipy.org

[2] Hierarchical Data Format http://www.hdfgroup.org/HDF5/

2.2.1 Distributed State

We need a distributed solution which eliminates the need to have central orchestration. We want to decrease the dependency between running programs therefore we want to have some sort of distributed application which distributes the operations to all other computers. The main requirement here is distributing the state of application between all or a number of nodes, in such a way that we could bootstrap a new application to a certain state. To make it more clear we need to look at the network of peers as one whole which has knowledge of currently running operations along with available datasets and preserves this state among all the nodes.

2.2.2 Distributed Data

We want to be able to store the result of an operation on different computers, i.e. distribute it on the network. This comes from the nature of our workflows. Normally we have huge datasets which represent certain models i.e. particles of a fluid or gas inside a container and we want to run multiple operations on those datasets. These are normally available on different computers of an institute, and if there are multiple institutes cooperating on a topic then we have to fetch data from remote computers. Therefore we need to consider distributed data management.

2.2.3 Data Endpoint Abstraction

We need to abstract the absolute path of required data from users. To run every operation we provide certain data files as input. This is part of the manual step of running an operation and it makes it fragile. Currently users have to take care of storing input files in correct folders before running their scripts, or they have to copy the input files from the shared network file system to the appropriate runtime folder. This is something that we want to disappear. We want the system to manage the input data and its absolute location.

2.2.4 Server Agnostic

We want to have the same user experience regardless of the machine that we connect to. It is very common that a user moves from one workstation to another one or connects to different machines using SSH. We want to be able to provide the requested information to the user, no matter to which machine she connects.

We could have any number of active computers in our network which are running an instance of our program and we want to let the users to talk to each of them and be able to launch same operations and get the same result.

It also means that if one users initiates an operation in first computer and then goes to the next one and asks for the status of the operation, she should be able to do that as if she is working with the first computer.

2.2.5 Runtime Control

While an operation is running we want to have full control over every step. In traditional approaches using job scheduling systems this is not possible. Except basic control such as stop, resume or similar operations, users can not control the runtime behavior of their program. In contrast we want to control all aspects of an operation.

2.2.6 Easy Deployment

We look for a solution that is easy to deploy onto new machines. A fully automated installation process is required. Unnecessary dependencies should be avoided. Users in science field are not professionals in the field of computer and therefore installing complex software requires system administrators to come in. Such installations costs money because user herself is not able to finish it without help from others. Moreover it makes it non-feasible for single users or small groups to try the software.

2.2.7 User Space Solution

We need a user space solution. It means that it should be possible to deploy, install and run the software without having full control over the machine which is supposed to run it. Any software which needs administrator or root access rights is not of our concern. Therefore the software itself along all of its dependencies should be installed and should be able to work correctly in user space, under a normal Windows or Linux user account with no special rights except the default ones.

2.2.8 OS Agnostic

We want to be able to run the solution on both Linux machines and Windows machines. It is common that users have two machines, one for office tasks with Windows OS and the other for running simulations with Linux. However Linux is the favorite machine to run the software but it could be possible to run it on Windows as well.

2.2.9 Light Weight

We look for a light weight solution which could be run on both laptops and stronger machines, with minimum dependencies and launch time.

Chapter 3

Related Work

In this chapter we will go through a number of existing solutions. First we discuss some well known HPC products which offer a complete set of features to launch and operate an HPC site. These are mostly large scale software which include every aspect of data intensive computing. Then we introduce data distribution solutions which are designed to manage big data. Afterwards we introduce some state distribution solutions which could be utilized in a possible solution to spread object states among a number of programs. At the end we introduce some existing efforts in scientific workflow management field. At each part we introduce a the parameters which are interesting for us and then we analyze the given product against them.

It is also important to mention that we have only considered mostly free and open source projects. Projects which need royalty fees or limit the use cases and their code is not available publicly have not been considered. Rationale for this decision is to achieve a sustainable solution for small groups with limited budgets. Therefore it is crucial to avoid any notable costs in relation with using products and implementing non-free and non-open standards or protocols. In case of closed source programs it would not be possible to extend them and in case of protocols it is obviously a bad choice because it will impose future risks on us. Therefore we decide to avoid such products, standards or protocols all together.

3.1 Grid Computing Solutions

There are various products in high performance computing field, such as grid middlewares, distributed storage systems, data storage management systems, workflow managers, operating systems for massively parallel super computers and so on. Most of these products are targeted toward super computers and large scale operations. However we require more light weight solution toward small, distributed groups. Therefore in this section we do not want to go through all the existing products in HPC field, which is out of scope, instead we discuss a number of them which are more likely to be used in European scientific communities, specially in the working environment that I am accomplishing my work.

It is importance to notice that we asses these products against requirements of small and agile scientific groups which often only have access to limited resources.

Moreover we look for an application level solutions and not a sole product. Here are the extracted parameters according to the discussed requirements:

- Deployment complexity
- Data provision methods
- State preservation
- Centralization
- Required user rights
- Runtime control
- Applicability

Here go through a number of projects which are widely being used.

3.1.1 UNICORE

UNICORE is one of the main providers of European Middleware Inititative(EMI) [12]. It is an open source software under BSD license[1]. It follows a client/server architecture and is implemented in Java programming language. In consists of three main layers, user, server and target system layer. Jobs will be executed from the client machine and the resulting output files will be downloaded to the same machine as well. It provides job execution and monitoring on grids.[29]

It has a graphical client based on Eclipse IDE. There is also a command line interface available. GUI provides workflow design and execution means and allows users to design complex workflows and combine multiple applications while selecting the desired amount of RAM and number of CPUs on target resources. It introduces a concept called GridBeans that allows users to extend the GUI to take advantage of software available on the grids and visualizing output data.

The service layer of UNICORE is composed of Gateway and number of other components. Gateway, as its name says, is the entry point to a UNICORE site. It is like the middle-man between inside and outside world in UNICORE. Client makes job specification in Job Service Description Language (JSDL)[2]. This layer exposes resources via Web Services Resource Framework (WSRF) compliant web services[3] for file transfer, job submission and management. There are also a wide range of file transfer protocols supported for site-to-site and client-to-site[4].

It has been used in super-computing domain to allow exposing and managing of available computing resources.[28]

[1] https://www.unicore.eu/unicore/
[2] http://en.wikipedia.org/wiki/Job_Submission_Description_Language
[3] http://en.wikipedia.org/wiki/Web_Services_Resource_Framework
[4] https://www.unicore.eu/unicore/architecture/service-layer.php

Deployment complexity

Deployment requires Java Runtime Environment in place, which if not available would require further assistance from IT administrators to be installed. It is also intended to be installed on a cluster site not normal workstations. To access web features a browser plugin should be installed[1]. The server is composed of multiple components that user has to download and install each of them separately.

Required user rights

To install the server components admin rights on Windows or root access on Unix-like operating systems are necessary. Even though running the client does not need any special permissions.

Data provision methods

Output files are produced on remote resource (service layer) and can be downloaded to client machine on demand. Input files should be transferred to the UNICORE storage using the client. There are also command line tools such as *uftp* to transfer files to remote storages[2]. UNICORE relies on standard file system as storage mechanism. However there are efforts to extend it to support Hadoop Distributed File System as a storage mean.[3]

State preservation

The UNICORE service layer contains the state of the application and running jobs. Upon connecting to a UNICORE site users can utilize the GUI to explore the site and access the running jobs for control and monitoring purposes. The details of possible operations are covered in UNICORE user manuals.

Centralization

UNICORE follows a standard client-server architecture therefore it is centralized. It sits on top of a cluster and will let clients to connect to it via defined services. There is no means of inter-site and inter-unicore information exchange. UNICORE is a grid middleware and infrastructure and therefore it is not intended for inter-site state distribution.

Runtime control

It is possible to use UNICORE's web services from third party applications, however this does not give control over the run-time internals of the application, instead it is merely a way to write extensions for the program.

[1] https://www.unicore.eu/documentation/manuals/unicore/files/client_intro.pdf

[2] https://www.unicore.eu/documentation/manuals/unicore/files/uftpclient/uftpclient-manual.pdf

Applicability

There are a number of considerations that prevent us from taking UNICORE as a solution. However it is a well established and mature product on its own.

UNICORE is supposed to be a site manager software. A group have to install it on a cluster and then users will be able to access the resources. Only a well-informed and skilled group can install and maintain such a product, which contradicts with our initial requirement that is aimed toward small groups. Moreover we look for a user space solution which is not the case about UNICORE.

Then next point is distribution of data and state. Even though the product which is installed on a site will preserve state of jobs and will alow accessing remote file systems, still it is not a good fit for our case. We need to have automatic inter-site interoperability, both for data distribution and state of the system, i.e. running jobs, which is not fulfilled by this product.

Runtime control over a job execution is another limitation that we face with this solution. We need to be able to deliberately interfere in every step of execution of a job and define arbitrary policies for them. To fulfill this, we more need a framework rather than an application. Applications such as UNICORE represent computing backends and job schedulers rather than a platfrom to build new solutions on top of them.

3.1.2 Globus Toolkit

Globus Toolkit (GT) is the widely developed application for resource management and grid computing today.[10] It lets people share resources, such as computational power, databases, etc online while preserving the provider's local autonomy. It consists of various modules and allows further services to use GT libraries to build new services.[14]

Unlike UNICORE which is heavily dependent on its Eclipse based client, GT does not provide an interactive user interface. However it offers its services in form of multiple command line tools that makes them suitable to be used in scripting[1].

Deployment complexity

GT has various components that the user might not need all of them and should install whatever she is interested in. Services only install on Unix-like operating systems. Providing a Unix-like environment such as *cygwin* one can install it under Windows as well. It is also possible to install it directly from pre-compiled binaries or install from source, since GT is free software and is available mainly under Globus Toolkit Public License (GTPL)[2].

[1] http://toolkit.globus.org/toolkit/docs/4.0/data/key/
[2] http://toolkit.globus.org/toolkit/docs/6.0/licenses/

Data provision methods

GT implements GridFTP protocol[1] as defined by Open Grid Forum (OGF). Users have to use command line tool as well as some GUIs provided by GT to move files between local and remote machines before and after executing jobs.

State preservation

GT has a component called Grid Resource Allocation and Management (GRAM) which provides job submission, management and monitoring. GRAM is not a Local Resource Manager (LRM)[2], instead it utilizes them to execute jobs on remote sites. GT supports WSRF specification, therefore it has statefull web services, however there is no notion of inter-side workflow and state preservation.

Centralization

GT could be installed on multiple machines to allow users to run GT services on them. These machines will communicate based on X.509 security standard. Having multiple machines, Globus Gatekeeper will dispatch services on them allowing GRAM to submit jobs onto whatever LRM available.

Required user rights

User should have admin rights to install and configure Globus Toolkit along with its components.

Runtime control

Even though GT allows very flexible scripting and lets to develop services using its libraries, there is no notion of full runtime control rather than predefined routines such as MPI[3].

Applicability

Like UNICORE, GT is a product rather than a framework to build new services on top of it. However it is more flexible in terms of developing new services on top of it. It could be utilized as another backend to our solution which makes it possible to take advantage of a wide range of HPC resources in our solution. But itself alone is not sufficient for our mission.

[1] https://www.ogf.org/documents/GFD.20.pdf

[2] Local job managers control a resource directly and let the jobs to be executed on them, e.g. Sun Grid Engine (SGE), Condor, etc.

[3] http://toolkit.globus.org/alliance/publications/papers/gempi.pdf

3.2 Distributing Data

3.2.1 Distributed File Systems

One way to achieve fault tolerant and reliable data storage and access is to use distributed file systems (DFS). In this case the data will be replicated over a network of storage servers with different magnitudes based on the underlying file system. We will discuss a number of free and open source solutions.

Hadoop Distributed File System (HDFS)

Here we discuss HDFS and not Hadoop itself, but here is a short description of what Hadoop is: "Apache Hadoop is a set of algorithms (an open-source software framework written in Java) for distributed storage and distributed processing of very large data sets (Big Data) on computer clusters built from commodity hardware[1]." Thus it targets the same goals that we have but in very large scale with a different approach. "The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware." The primary objective of HDFS is to store data reliably even in the presence of failures.[26, tp. 3]

"Hadoop1 provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce [9] paradigm."[25]

"HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes."[25]

Deployment Complexity HDFS requires Java 1.5.x, ssh, sshd and rsync to be installed. Three basic modes are available: local, pseudo-distributed and fully distributed mode. XML configuration, installation of local and pseudo distributed modes are almost straightforward, but for fully distributed note extra steps are required according to quick start guide.[2]. It is also portable. "HDFS has been designed to be easily portable from one platform to another."[25]

Fault Tolerance According to Hadoop "Hardware failure is the norm rather than the exception." There is also heartbeat technique in place. "Each DataNode sends a Heartbeat message to the NameNode periodically." "The DataNodes in HDFS do not rely on data protection mechanisms such as RAID to make the data durable. Instead, like GFS, the file content is replicated on multiple DataNodes for reliability."[25]

Accessibility HDFS provides three main access methods:

1. FS Shell "includes various shell-like commands that directly interact with the Hadoop Distributed File System (HDFS) as well as other file systems that Hadoop supports"

[1] http://en.wikipedia.org/wiki/Apache_Hadoop
[2] http://hadoop.apache.org/docs/stable/

2. Administrative Commands a set of commands to control HDFS from command line interfaces (CLI)

3. WebHDFS to interact with HDFS via HTTP, i.e. REST[1] API

Applicability Hadoop is targeted toward big data and thousands of nodes and it requires MapReduce[9] programming technique. Hadoop is a promising tool if we deal with multi-terabyte datasets. But it has its own requirements, one need to adapt the programming techniques along with running an Hadoop site to benefit from its advantages. Since we are after a distributed workflow framework (according to our requirements), HDFS could serve as a backend but Hadoop itself can not replace our system, at least not untill we deal with smaller datasets (large in Hadoop terms is beyond terabytes of data).

For our current state I do not find Hadoop a suitable solution. It will put unnecessary burden on us, however it is amoung top solutions for data centers and very large sites. But still I could think of applying MapReduce itself as an algorithm in some of our scenarios.

3.3 Distributing State

In this section we go through a number of existing methods to distributed an object or in another terms to distribute the state.

3.3.1 Distributed Hash Tables (DHT)

Distributed Hash Tables (DHT), best known for their application in building torrent tracking software, are distributed key/value storages. DHTs could let us to have a key/value store and distributed it in a decentralized way among a network of peers.

Kademlia

Kademlia is a p2p DHT algorithm introduced in 2002. We first tried to use it as a distributed key/value store but it is not suitable for our case and changes do not propagate only to a few neighbors [16].

In our case to keep track of the available data on the network of collaborating peers, we tried a DHT implementation (I was not aware of the problem in the beginning).

Our tests showed that even though DHT is fault-tolerant and reliable for file distribution, it is not adequate for our realtime requirement to find our required data. In one test we ran two peers, one on an Internet host and another one on local host. Here are the client and server codes:

Listing 3.1: A twisted application to run Kademlia DHT

```
1 from twisted.application import service, internet
2 from twisted.python.log import ILogObserver
```

[1] Representational state transfer

```
3
4  import sys, os
5  sys.path.append(os.path.dirname(__file__))
6  from kademlia.network import Server
7  from kademlia import log
8
9  application = service.Application("kademlia")
10 application.setComponent(ILogObserver,
11     log.FileLogObserver(sys.stdout, log.INFO).emit)
12
13 if os.path.isfile('cache.pickle'):
14     kserver = Server.loadState('cache.pickle')
15 else:
16     kserver = Server()
17     kserver.bootstrap([("178.62.215.131", 8468)])
18 kserver.saveStateRegularly('cache.pickle', 10)
19
20 server = internet.UDPServer(8468, kserver.protocol)
21 server.setServiceParent(application)
22
23
24 # Exposing Kademlia get/set API
25 from txzmq import ZmqEndpoint, ZmqFactory, ZmqREPConnection,
26  ZmqREQConnection
27
28 zf = ZmqFactory()
29 e = ZmqEndpoint("bind", "tcp://127.0.0.1:40001")
30
31 s = ZmqREPConnection(zf, e)
32
33 def getDone(result, msgId, s):
34     print "Key result:", result
35     s.reply(msgId, str(result))
36
37 def doGetSet(msgId, *args):
38     print("Inside doPrint")
39     print msgId, args
40
41     if args[0] == "set:":
42         kserver.set(args[1], args[2])
43         s.reply(msgId, 'OK')
44     elif args[0] == "get:":
45         print args[1]
46         kserver.get(args[1]).addCallback(getDone, msgId, s)
47     else:
48         s.reply(msgId, "Err")
49
50 s.gotMessage = doGetSet
```

In the above example we have used *twisted networking library*[1] and one Python implementation[2] of *Kademlia* DHT algorithm[16]. This will start a peer-to-peer network and will try to bootstrap it with another peer on the give IP address. Thereafter it will open another endpoint to expose a simple *get/set* method for the rest of application for communicating with the network.

The next part is a few lines of code to communicate with this network:

Listing 3.2: Multipart messages with ZeroMQ REQ/REP pattern

```
#
# Request-reply client in Python
# Connects REQ socket to tcp://localhost:5559
#
import zmq

# Prepare our context and sockets
context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:40001")

# Set request
socket.send(b"set:", zmq.SNDMORE)
socket.send(b"the key", zmq.SNDMORE)
socket.send(b"the value")
print socket.recv()

# Get request
socket.send(b"get:", zmq.SNDMORE)
socket.send(b"the key")
print socket.recv()

# Invalid get
socket.send(b"get:", zmq.SNDMORE)
socket.send(b"not existing")
print socket.recv()
```

This simple client will try to connect to the previously opened port and send get/set messages.

Configuring this p2p network is a little tricky. The network should work correctly even if nodes enter and leave the network. During our tests in development environment we observed some problems with initializing the network, but while the network was initialized leaving and entering the network had no effect on the results.

Having the number of nodes increased up to 3 the reliability shows up again. When we set a value for a key in one node we can not guarantee that getting the value for that key on other nodes will return the updated one. With a number of tests I can confirm that two nodes which are bootstrapped with the same third node

[1] Twisted Matrix Project https://twistedmatrix.com/

[2] A DHT in Python Twisted https://github.com/bmuller/kademlia

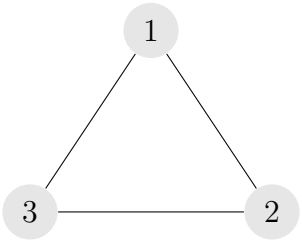

Figure 3.1: A network of three peers

does not provide the accurate result every time and it is not clear for me why this happens. See figure 3.1 on page 18.

After running more tests, we figured out that the possible source of the above mentioned problems was the confusion in using *binary* and *string* in Python, so it was an error in our side.

Firewall Problems In a test having one process running on a server in Internet and outside of the local network and having two different processes running on one laptop but on different ports it is observed that the changes (sets) in the Internet does not replicate to the local processes but the changes from local processes are being replicated to the other process.

Conclusion Having a network between local and Internet processes in the above mentioned method is not reliable. Repeating the tests with only local processes which are bootstrapping to one of them and running the setter/getter methods showed that even in this scenario it is not reliable and one can not guarantee that the desired value will be returned.

3.4 Other Works

There are a number of other notable works that we did not discuss here but they are worth mentioning. Moreover this enlarges our vision of state of the art and let us to know more similar efforts which we might take advantage of them in future.

3.4.1 The Raft Consensus Algorithm

"Raft[19] is a consensus algorithm that is designed to be easy to understand. " "Consensus is a fundamental problem in fault-tolerant distributed systems. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value, that decision is final." [1]

[1] http://raftconsensus.github.io/

3.4.2 RADICAL Ensemble MD Toolkit

As described on their website[1]: "The Ensemble MD Toolkit is a Python library for developing and executing large-scale ensemble-based Molecular Dynamics (MD) simulations and workflows. It is being developed by the RADICAL Research Group at Rutgers University. Ensemble MD Toolkit is released under the MIT License."

3.4.3 Concoord

"ConCoord is a novel coordination service that provides replication and synchronization support for large-scale distributed systems. ConCoord employs an object-oriented approach, in which the system creates and maintains live replicas for Python objects written by the user. ConCoord converts these Python objects into Paxos Replicated State Machines (RSM) and enables clients to do method invocations on them transparently as if they are local objects."[2]

ConCoord has been used as underlying technology to create OpenReplica[3], a coordination service for distributed applications.[1]

3.4.4 COSMOS

"A Python library for workflow management that allows formal description of pipelines and partition- ing of jobs. In addition, it includes a user interface for tracking the progress of jobs, abstraction of the queuing system and fine-grained control over the workflow."[15]

3.4.5 Weaver

"Weaver[5] is a high-level framework that enables the integration of distributed computing abstractions into scientific and data processing workflows using the Python programming language. This takes advantage of users' familiarity with Python, minimizes barriers to adoption, and allows for integration with a rich ecosystem of existing software."[4]

3.4.6 iRod

From iRod's documentation[5]:

iRODS is open-source, data management software that lets users:

- access, manage, and share data across any type or number of storage systems located anywhere, while maintaining redundancy and security, and
- exercise precise control over their data with extensible rules that ensure the data is archived, described, and replicated in accordance with their needs and schedule

[1] http://radical-cybertools.github.io/ensemble-md/index.html
[2] https://pypi.python.org/pypi/concoord
[3] http://openreplica.org/
[4] http://cs.uwec.edu/~buipj/research/software/weaver.html
[5] http://irods.org/wp-content/uploads/2012/04/iRODS-Overview-November-2014.pdf

3.4.7 Ceph

"Ceph is a distributed object store and file system designed to provide excellent performance, reliability and scalability[1]."

[1] http://ceph.com

Chapter 4

Problem Analysis

In this chapter we try to analyze our problem in depth and find out different aspects of it. We discuss a number of important elements such as possible operation types and we formulate the way we apply operations on datasets.

There are a number of possible use cases in our problem domain. To demonstrate these cases we assume we have a number of nodes and datasets, and we need one or more datasets to do certain operations. In this chapter we explain possible combinations of operations, nodes and datasets.

4.1 Operations

In a final solution there would be many services, some will carry administrative tasks such as getting a list of currently running tasks, or a list of available datasets. These services do not change state of the system. It means that even though they could affect the performance of the running machine, i.e. with querying the database, they will not make any permanent change into data stores or the running instance. We are not interested in these services.

There are a number of other services who carry the business logic of our application. Calling of these services will probably change the state of the running instance and might store persistence data or create new datasets. Moreover they are often data intensive and will trigger some workflows to begin. We are interested in these services and we call them operations. They could be any scientific operation, however we do not discuss the detail of them. Instead we are interested on categorizing them based on their characteristics, such as type and number of required datasets.

4.1.1 Types

We divide data intensive operations into two main groups, the linear operations and non-linear ones. This is simply comes from the nature of the operation, if it should be applied to input datasets in parallel or serial. We describe this with a simple algebraic notion.

Linear Operation

Being linear means that the operation could be broken into smaller operation and then run in parallel. Having two datasets we can apply the operation on each of them separately and the aggregate the results. Here is the algebraic notation of a linear operation which acts on two datasets:

$$f(a+b) = f(a) + f(b)$$

Being linear or non-linear only matters when we have to operate on more that one dataset, or we want to break the input into many parts.

Another subtle point here is that for a linear operation we can simply run the operation on the machine that contains the required dataset, avoiding any dataset transfer among different locations in case the requested dataset is not available locally on the machine which has received the command to start the operation.

Non-linear Operation

In contrast to linear there is non-linear operation. This type of operations require all the inputs to be processed at once. It is not possible to apply the same operation on each of the input datasets and aggregate the result at the end. This means that these operations could not be run in parallel. Here is the algebraic notion of such operation:

$$f(a+b) \neq f(a) + f(b)$$

A sample of non-linear operations is comparison, when it is required to compare all elements on two different datasets. Such a use case happens in many areas of science and engineering. These operations are also called All-Pairs. [7]

In non-linear case the complexity of running operations on multiple remote datasets dramatically increases. When the required datasets are available on the same machine which starts the operation, there is no problem. However when the datasets are not available on the same machine or even not on a single remote machine, we have to make at least one data transfer. In this case at least one dataset should be moved to the location of the other datasets to make it possible to run the non-linear operation on one single machine which contains both of required datasets.

4.1.2 Data files

As operations need input and produce output datasets, we have to see how the input data will be provided and where the output data will be stored. Every input or output needs an explicit address, an endpoint, either local or remote. Typically users will copy files around and will move them using file transfer tools to a working directory and then will launch desired script using job schedulers. However they do not pass large data files around and will mention their location inside their script files, which normally points to some shared ftp storage. The scheduler program in turn will run the script at some point of time in future and then will look for data files inside working directory. Finally any output file will be created in the same

directory or a location explicitly defined in the script. For each operation we need one or more input datasets which might be available on the same node that wants to run the operation initially or could reside on other nodes. We describe some characteristics of these data files shortly.

Input

One need to pay attention that we do not pass complete and real datasets as input parameters, instead we use identifiers to find the required dataset. Currently for most users these identifiers are nothing than the name of the data files inside the working directory or the explicit path of a dataset on a network machine. It is assumed that the system has knowledge of available datasets and can find them providing an identifier, in this case file name. Another point about data files is that they are normally not mission critical and could be reproduced, hence is the emphasizes on the state. The last point to mention is that input data is not managed by the system.

Output

Operations create output datasets which normally are small in size, therefore we ignore the transfer cost of operation results in our work. These data files will be normally stored in working directories and are of less importance to us. In typical environments users check output directory for their result and again they use conventional file transfer tools to have that data locally or provide it to some visualization tools to be visualized. However all these steps are manual and no control and value added services could be built on top of them. Users are not able to track their activities and there is no history left about them (except probably some server logs), no reports could be made and no administrative decisions could be made about usages, user activities and so on. This all means that the output data files are not managed.

Intermediate Output

With some of current tools, users can pipe output of one simulation program to another program. For example if we have dataset a as input this call would be $f(g(a))$. We will address this further in our design as a *Mixed Operation*. With a fastforward we would say that such operations will be handled asynchronously during a number of parent-child operations, just like function calls in a programming language or mathematics.

4.2 Dataset Identification

When we ask for an operation and we want to store the result somewhere on the network we have to think about an identifier for them. We need a consistent way of naming datasets. If we ask users to provide result dataset names it will break soon, because we would have duplicated names. The naming should be managed by system, as well as data management and transfer itself. However, we have to

provide a user friendly way for naming, one idea is to assign tags to datasets. If a user search for these tags, any dataset which has that tag will appear in the query results. Another approach is to store a database of datasets and operations. Having such a database lets the system to make a relation between datasets, operations and possible other desired factors.

Even though we want to avoid duplicates in our network, it does not mean we do not want redundancy and replication for our data, but it means that we rely on other solutions, such as distributed file systems with built-in replication, to do this task.

4.2.1 Data Manipulation

Normally we do not manipulate existing datasets. Each operation results in a number of new datasets. However if we opt for a storage mechanism such as Hierarchical Data Format (HDF) we might want to store and retrieve datasets from a single data file presumably in HDF5 format. But it depends on our further design and it does not change the fact that each operation produces new datasets and we have to store it.

4.3 Scenarios

According to the operation type and number of input datasets, a number of combinations are possible. In this section we introduce them as scenarios. We begin with a simple one and we gradually add details to it and make new scenarios. The following text describes the scenario and it contains expectations of my supervisor about the internals of the system, therefore it goes a little into design of the system. However in the next chapter we will explain our final approach, therefore any design related material in this chapter represent only ideas and expectations.

For the following scenarios we assume these general statements to be true:

1. The user has neither a prior knowledge where the datasets are stored
2. Nor of how many servers are present on the network

4.3.1 Scenario 1 - Linear operation with one input set

In this scenario we have a linear operation, e.g. Op^A on $Node^A$ which requires one single dataset such as $Dataset^1$ which is available on one of the other peers.

We have a distributed network of collaborating servers, where in this case, we consider two computers. Each server has its own storage and maintains a number of datasets on it. These servers collaborate together to accomplish issued commands. User in this case wants to perform one operation on a dataset that resides only on one of the servers.

The user connects to one of the servers, which we call a client for now. This server is assumed to be part of the network, though it may not have any local data stored on it. The user interactively (or non-interactively) issues a command on a set of datasets providing some kind of identification. This command is broadcasted by

the client to all servers in the network. All servers receive this command and check whether they have the data locally. The server which has the data performs the operation and the others ignore it. The result of the operation in this case, remains on the same server which the original dataset was on.

- Note: further we will see in our design that the notion of *client-server* does not describe our design, since it would be more *peer-to-peer* and we would have temporary *result collectors* only.

We assume the user has already queried the available data in the entire network by issuing something like "list datasets" which outputs dataset names and ids.

The following table shows two servers, each has one dataset. The user is connected to S2.

Server ID	*Dataset ID*	*Client*
S1	DS1	No
S2	DS2	Yes

Let us assume the data sets are 10^6 random numbers and the operation is to transform the real random numbers to a set of [0 or 1] depending on whether the number is even or odd. This operation is assumed to be a user defined method that operates on the data set and represents user's intended logic[1].

- Note: A dataset can be defined as an object that has an id, and a one dimensional array (Python list).

The user issues the command like this from a command line interface:

- real2bin(DS1) will result in Broadcast(real2bin(DS1))
- Note: it is assumed that all functions are already defined on all servers

The client broadcasts this function to all servers. Each server will check if the dataset with this id exists, if so will run the command.

This means that each server, especially the client, has to know about all data sets existing in all servers. It does not need to have the actual data, but needs to know about it. So that when the user issues the above command, she would not get a non-existing dataset error from the client, just because the data is not there.

Such operations will be done in two simple steps. First would be publishing the recieved operation to all of the participating peers. This step is basically transfer the operation to another peer, but in a distributed manner. Afterwards, when each peer has received the operation request, the peer which contains the desired dataset would run the operation in one step, hence this operation requires one dataset.

In chapter 5 we explain in detail our suggested solution.

[1]This scenario is partially from a raw requirement text by my advisor.

4.3.2 Scenario 2 - Linear operation with two input sets

This is similar to scenario one, except that the operation requires two datasets to operate. In this scenario we need at least three peers involved. We assume the first peer has no data of our interest therefore it should cooperate with others to accomplish the request. Our operation in this case requires two different input datasets which are not available on the first peer and we should access them on other peers.

We assume the data distribution is like the following table:

Server ID	*Dataset ID*	*Client*
S0	—	Yes
S1	DS1	No
S2	DS2	No

In this case user will issue a command to S^0 which involves datasets DS^1 and DS^2 which none of them are available on S^0. In this case we need to transfer the operation to other nodes. Since this operation is linear we want to break it so it could run in parallel on S^1 and S^2 on resepective datasets. And then, when the intermediate outputs are ready, they would reside on S^1 and S^2. Then we would need to transfer one of them to the other node to calculate the final result. We have to transfer the smaller dataset to have less transfer cost. Finally when both intermediate data availabe on one of S^1 or S^2 we run the final operation there.

The simple algebraic notion for these steps would be like these:

$$\Sigma = f(DS^1 + DS^2)^{S0} = f(f(DS^1)^{S1} + f(DS^1)^{S2}) \tag{4.1a}$$

Then we launch them on other nodes

$$DS^{S1} = f(DS^1)^{S1} DS^{S2} = f(DS^1)^{S2} \tag{4.1b}$$

We move the smaller intermediate dataset to the other node.

$$DS^S 2 \longrightarrow S^1 \tag{4.1c}$$

The transfer scenario would be different in case of mixed operations or non-linear operations.

4.3.3 Scenario 3 - Linear operation with 2+ input sets

This scenario is slightly different than scenario two only about the number of input datasets. All the assumptions and requirements remain the same. Except that we would have one more more extra machines containing the rest of datasets. However we consider the worst case here, where every machine contains only one of the required datasets and the initial machine has none of them. In reality there would be cases than all the data would be available on the same machine or a number of datasets would exist in one remote machine. The worst data distribution for this case would be like this:

Server ID	*Dataset ID*	*Client*
S0	—	Yes
S1	DS1	No
S2	DS2	No
S3	DS3	No

Theoretically there could be more datasets but it is unlikely and often it does not exceed two inputs.

4.3.4 Scenario 4 - non-Linear operation with one input set

With the similar assumptions as before, we have only a different type of operation. With one dataset there is no difference between this scenario and first scenario, where the operation is linear. In case this dataset could be broken into smaller parts we should consider the operation type to prevent any unexpected results.

4.3.5 Scenario 5 - non-Linear operation with two input sets

This case is very complicated. We can not solve it like the previous ones, and we need to make extra decisions. In this kind of operation we would need both datasets at the same time in one machine in order to produce any results. Therefore it is not possible to distribute this operation on multiple machines like we can do for a linear operation with multiple inputs.

4.3.6 Scenario 6 - non-Linear operation with 2+ input sets

This is an extension to the previous scenario. If we could find a solution for scenario 5, we would extend it to cover this scenario as well. There is no fundamental difference between this scenario and the last one. Again we consider the same data distribution as described for scenario 3.

4.3.7 Scenario 7 - mixed operations

We previously discussed a number of scenarios to run atomic operations on one ore more datasets. The solution to above mentioned scenarios will be discussed in detail in the next chapter. However these are the simple cases and do not cover all possible operations that we need. Here we introduce operations which are composed of another operation types which we call them *sub-operation*. Here is the main assumption before mixing operations:

- **Any sub-operation will produce in a new dataset**

This is necessary in order to simplify the problem and allow us to make some reasonable results for a limited set of cases and let aside other possibilities for further work. This would be enough for us to demonstrate our main problems and also to build our basic proposal on top of that.

Here is the algebraic notion of such a problem, f and g are linear functions:

$$\begin{aligned}\Sigma &= f(a + g(b + c)) \\ &= f(a) + g(b + c) \\ &= f(a) + g(b) + g(c)\end{aligned}$$

In the above text we assumed linear operations, however we would have mix of linear and non-linear operations. We will discuss this further in next chapter, while proposing a solution to solve such operations.

4.4 Decision Making

The main decision that we need to make at every scenario is whether we should transfer the required data or we need to delegate the operation to an instance on a node which already has the data. To make a decision we need to answer a number of questions. First we need to know the location of the data:

1. What is the operation type?
2. Are required datasets available locally?

These points derive directly from our two main requirements about distributing workflow information and eliminating data location. But how these questions serve those purposes?

First of all we need to recognize the type of the operation that we are going to launch. This operation could resemble any of the discussed scenarios in this chapter. This will make it clear for us if we can directly jump into distributing the workflow and launching the task or we need to take further steps into breaking the operation into smaller units. We will discuss this in detail in next chapter, while describing our design.

The next import question regarding operations is data availability. If a local machine has received an operation and the required dataset is available on local machine then there is no need for any transfer, then we would need only to distribute the operation information among collaborating peers.

Answers to the above questions will help us to decide whether on which machine we should run the operation. Running an operation remotely means that we will not transfer back the requested data from another machine, instead we will launch the operation on the machine containing the data. This is different from conventional approach of transferring methods or executable to a remote resource and executing it there. In our case we assume that we have the same service API available on all the participating machine. Therefore we only need to decide on which machine we have to forward the request. In case of forwarding or delegating a request to other machines we would need to preserve regarding workflow information on all the participating machines. This will be discussed in next chapter as part of our distributed workflow management design.

Chapter 5

Proposed Solution

In this chapter we explain how we can solve the scenarios which we explained in previous chapter. We introduced seven different scenarios, three of them for linear operations and three of them for non-linear ones, and one mixed operation type. We begin with the most basic scenario, solving a data intensive operation which requires one dataset. Then we will extend it to accept more datasets and we will solve them. We will cover only solving linear operations and non-linear operations will be left for further work.

5.1 Big Picture

To make it easier to consume all this text we will make an effort to show a big picture of our design beforehand. Here it is.

There would be multiple computers, our application installed on each of them. This machines do not depend on each other to operate. Each of them runs the same program instance as others. Each one has the same set of services that other instances and users can call. Each instance has a number of datasets. These peers will exchange knowledge of existing datasets with each other. Each instance contains atleast two distributed internal structures, we call them *distributed stores*, these are similar to distributed key/value stores. One store for operations, and another one for datasets. When an instance receives a service call for an operation, it will store some basic information and identification about it inside its operation store. This store automatically will inform other peers about these changes, so others will have the same knowledge afterwards.

Upon an incoming request, an instance would be able to accomplish the task alone, only if it has access to datasets locally. Otherwise it will do nothing and will simply ignore it. But since the operation store will distribute this information seamlessly, any other peer has the opportunity to launch the operation if they have the desired data, if it is not the case they will only update their internal store and do nothing. But if they had the data, they would run the desired operation and will update the state of the operation to a meaningful one, such as *processing*, and will distribute it accordingly to inform others about the new state of this operation.

In case of a mixed operation the receiver will break it into smaller operations and will self-launch them accordingly and will register a meta operation and will assign

the aforementioned smaller operations as sub/child operations. We will also register this meta operation into the internal operation storage will then automatically distribute it like any other operation. This way we behave the same way with simple or complicated operations and we use the same interface to interact with them.

In case of operations which have a number of child operations there would be a need to aggregate the result of sub operations. This would be done in a seamless way as well. We will introduce a *collector* peer which will randomly take control and collect the results.

When an operation is done, the result dataset will be given to the internal distributed dataset store of the responsible peer - the peer which is running it - and it will be stored in a backend storage but the unique identifier of the dataset will be distributed and other peers will get the knowledge of a new dataset and the container peer respectively. To query results users have to use the operation id that they have received upon the initial service call.

This way we can apply a collaboration technique to a number of autonomous peers. This design allows us to have peers which are able to work independently, but meanwhile are member of a larger network of peers and participate in accomplishing larger tasks.

We will cover these in more detail during this chapter.

5.2 Basic Design

The basic idea that we will follow in this chapter, relies on breaking the operations into smaller units which we can solve them in one step, such as only one operation or service call. In our design the simple operations are the building blocks for mixed ones. We build them on top of the atomic units, which we know how to solve them. This idea has the advantage of allowing us to reuse our work and decrease the complexity of implementing more complicated operations. However we would have increased complexity in messaging parts.

We assume that we have the information about the datasets available on all machines i.e. in form of a distributed table with entries containing the node address and dataset id. Based on this information the application can decide if it has the required data or not. We will explain this in detail later in this and next chapter.

Based on this algorithm the application implicitly delegates operations to the other nodes (instances of the same program), where the data is available. It would be a non-blocking service call, just like signaling others about an incoming request. Along with any operation change, the distributed workflow manager will synchronize the information among the peers. Any change in datasets in any collaborative node will also be synchronized with peers and will be added to a distributed list.

Here is a short definition of operations from design point of view:

- Simple Operation Any operation that has only one input dataset. We solve these.
- Complex Operations Operations with more that one input dataset. We turn them into simple operations and we solve them.

- Mixed Operation Any operation that has another operation as input. We turn them into simple and complex operations and then we solve them.

5.2.1 Break and Conquer - Recursive Call

This makes the high level algorithm that we using in this work. We break operations into smallest possible operations and we implement them. Then we build other operations using these small units. For example we solve scenario number one, as discussed in chapter 4 to run one single operation on a single dataset. In order to run it successfully we need to find the corresponding dataset and in case it is not available locally we have to launch the operation on the node which has it. Then when we come to scenario number two, i.e. applying the same operation on two datasets we break it into two smaller units, and one *meta operation*.

After having two smaller operations and one meta operation, we launch the smaller operations implicitly. Actually we break scenario number two into two instances of scenario one and one meta operation to observe the overall process. The implementation of this process will be presented in the prototype.

5.2.2 Non-blocking Calls

We base our design on non-blocking service calls. This means all the calls in our system would be asynchronous. When a user calls a service, she would instantly receive an id, instead of being blocked for the real result. The rationale here is because of possible long-execution times in our use cases. One needs to think of a service call in our design as *request submission*. Not only the interaction between user and peers is asynchronous but the inter-peer service calls follow the same path. Any service call regarding running an operation would be non-blocking and will result in an unique identifier.

5.2.3 Dataset identification

When a user or peer wants to submit a request for an operation, they would not provide a real dataset as input. They would instead provide the unique id of a dataset existing on our network of collaborative peers. The our system, i.e. the peer who has received the request, will look at its internal dataset store to see if it has the dataset locally or not. This will happen in the pre-processing step. In any case a signal will be dispatched to other peers about the new operation. The nature of these signals are also non-blocking.

5.2.4 Distributed operation

To realize the above mentioned method, we need to distribute any single operation. To achieve this we assign one unique id to every incoming operation. This will happen before doing any real work on the request. In our prototype we have implemented this with *decorators* in Python programming language. From this point of time, the operation will be known and tracked with this id. One can imagine this as a ticket which allows monitoring every change made to an operation. Apart from

id we will store name of the operation and its input datasets. This will allow us to relaunch this operation in case we need to. The store that keeps this information is distributed among all participating nodes. Any further information such as results will be attached to this store during the process. There are concerns running a distributed store that needs further attention and we try to cover them in further discussions and further work.

5.2.5 Distributed store

We will use this term many times in the next sections, therefore we have to explain it. Basically we talk about a simple key/value storage. Currently the storage mechanism is not important for us, it could be memory or anything else. This stores are like dictionaries, the keys would be the unique identifiers. Either id of an operation or id of a dataset. Then we will store further information about that object as the value for that key. We will take advantage of very simple structure to make it easy to be exchanged among peers.

From one side, these stores would be simple repositories to read/write key/value pairs. This simplifies dependent parts. From another point of view these stores are distributed objects, but not really. The keep sending signals about any change in their internals. These signals will be caught and handled by another component respectively. The other component will then signal other peers about certain changes that has been happened in this store. Other peers then will catch this message and will unpack the message and will update their own stores.

This way with minimum coupling we would have a distributed storage which its distributed nature is hidden from the objects which need to use it.

5.2.6 Messaging

To maintain a network solution we utilize messaging to have peer-to-peer communication. We define a number of different message types known to the system. Upon arrival of each message type the peer will take corresponding actions. There are a few message types that are important during this chapter and we introduce them here.

Delegation call

Used when a peer asks other to run an operation instead of it. Other peers will run the requested operation if they have the desired data and will update the distributed stores as a consequence.

Operation news

This happens when a peer wants to inform other peers about a change in its operation store. Other peers will only update their store respectively. This message will not cause others to run an operation, but this will cause the collective peer to take further actions if required.

5.2.7 Meta operations

Nature of simple operations is simple. An operation either will be handled locally or a dispatched signal will be handled by a peer who has the requested data. The status and result dataset id will be stored inside the operation store under the operation's unique id.

As soon as it comes to next scenarios this simplicity vanishes. Then we have to deal with multiple operations under one initial request from users. We have decided to repeat the same structure for operations with one or more datasets. Therefore in pre-process we check the number of inputs and we create the same number of smaller operations. This is only for linear operations, as we will not cover non-linear operations. Nevertheless, an input could be an operation call in case of mixed operations. In this case again we go back to our main patter, *break and conquer*. We create a meta operation and we store it. We create sub operations for **level one** operations and we will store its id in our meta operation as child operation. The same story will happen subsequently for any sub operation as well. Meaning that those might also create their own sub-operations and so on.

To better understand this structure think of how method calls work in programming languages. A programmer can call functions and pass other function calls as input arguments. The runtime will begin to execute each function call and will use a stack to restore the last execution point to continue from there after finishing the nested call. We apply the same mechanism but in a distributed and non-blocking way.

5.2.8 Collectors

In last part we discussed meta-operations. But who will be responsible to aggregate the result of these small operations spread all over the network? Here we introduce a new role for a peer, *collector*. During the pre-process phase, where we create sub-operations we randomly assign a flag for one of sub-operations to indicate that it should collect the results. That peer - which is yet unknown - apart from running its part of the meta operation, will take an extra responsibility. That is, it would check frequently to see if the sub-operations are done. As soon as the sub-operations are done, it will launch the target function, bypassing the pre-process phase.

In this part the meta operation will be labeled complete and the resulting dataset will be stored. If the target function - which carries the desired business calculation - needs any number of datasets, they will be copied to local machine to complete the process. Since we expect the resulting datasets have small sizes this would not cause a problem. Moreover in case of linear operations we can optimize the selection of this machine much easier than non-linear operations where we need to copy a dataset to another machine. In such cases we would select the machine which contains the larger dataset.

5.3 Realizing Scenarios

5.3.1 Scenario 1 - Linear operation with one input set

We introduced this scenario in last chapter and this was the assumed data and service distribution:

Server ID	*Dataset ID*	*Client*
S1	DS1	No
S2	DS2	Yes

In this case S^2 receives a request to run an operation such as Op^A on DS^1. However DS^1 is available on the other node. Upon getting such a request we will take a number of steps on the receiver (machine which received the command) and the container (the one with desired data) machine and on neutral machines which neither have received the request nor have the desired dataset.

On the receiver machine:

1. If operation requires one dataset continue, otherwise go to next scenario.
2. Store the operation in distributed operation store and get the id
3. Return operation id to user
4. If data store has the dataset locally do these
   (a) Run desired operation on the dataset
   (b) Store the result dataset in dataset store # Currently we choose a random peer to store results there
   (c) Add the result dataset id to the operation # Will be distributed
   (d) Update the operation state # Will be distributed

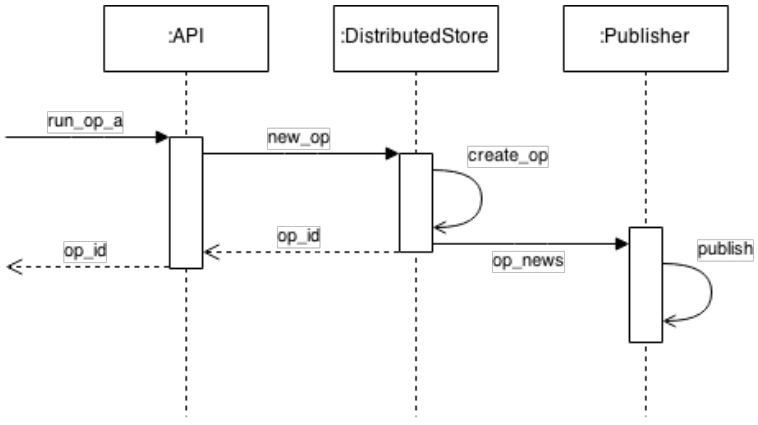

Figure 5.1: Steps we take upon receiving a new operation when the desired dataset is not available locally

From the above steps two types of signals will be published to other peers. Each of them will trigger certain actions. First we show the flow will happen upon receiving an operation update message:

1. An operation update signal is received.
2. Update our internal operation store silently (without distributing this change).

Next one is a delegate message:

1. A delegate signal is received.
2. If we have desired dataset do these, otherwise do nothing:
   (a) Run desired operation on the dataset
   (b) Store the result dataset in dataset store # Currently we choose a random peer to store results there
   (c) Add the result dataset id to the operation # Will be distributed
   (d) Update the operation state # Will be distributed

5.3.2 Scenario 2 - Linear operation with two input sets

First lets have another look at our data distribution and peers as described in analysis chapter:

Server ID	*Dataset ID*	*Client*
S0	—	Yes
S1	DS1	No
S2	DS2	No

The S0, in this case, is the peer who receives the command and initiates the request. Each of the other two peers, S1 and S2, has one of the required datasets, but not both of them.

In order to realize this scenario we apply **divide and conquer** and **produce-consume-collect** methods as described earlier in this chapter. Then we create one meta operation and two subs. Sub operations will operation exactly as described in previous section and their results will be reflected in our distributed stores.

As an example we assume that in this case we have two arrays, each consisting of 10^6 random numbers. We have to first transform these datasets into a set of [0 or 1] based on the number being even or odd (use case 1) and then we make a third dataset which contains the sum of every two corresponding numbers in range of [0 to 2]. (in our prototype we have implemented this)

- Note: in this case each pear is able to run the requested linear operation on one or more datasets.

The notation of above mentioned approach will be like this:

$$f(a + b) = f(a) + f(b)$$

In order to run this operation in a collective way, we need to think of the type of service calls in our system, whether they are blocking or non-blocking. Since often the operations in HPC environments are time consuming and long-running, we consider the non-blocking approach. The operation will be **submitted** to the

collaborative network and the system decides where and how to store result of an operation (a dataset). This allows us to design our system in a decentralized way, where each peer will inform others (neighbors) about a request using **publish-subscribe** pattern, where the peer will publish a request and the peer which contains the dataset will react to the published request and will run the operation. All the other peers who do not have the requirements (the dataset for now) will ignore it. As described for first scenario, they will store the details of running operations.

For this operation the following steps will take place on the receiving peer: On the receiver machine:

1. If operation requires two dataset continue, otherwise go to next scenario.
2. Store the operation along with input dataset names in the distributed operation store and get the id
3. Create sub commands for each dataset and update the operation store
4. Setup parent operation id for sub-operations
5. Randomly choose one of sub-operations as collector peer (with setting an extra parameter)
6. *Self-launch* the sub operations - This will resemble scenario 1
7. Return operation id to user

For other peers there is only one change while updating an operation:

1. An operation update signal is received.
2. Update our internal operation store silently (without distributing this change).
3. If we are assigned as collector of this operation do these:
   (a) Find the parent of this operation
   (b) Check state of its sub-operations
   (c) If sub-operations are done do these:
      i. Download the result of sub-operations to this machine # Results are small in size
      ii. Run the parent operation on them
      iii. Store the result and update the operation store

- With the use of operation ids we eliminate the need to get a result dataset name from user but we still can accept **tags** from users.
- We assume every operation involving more than one dataset is made of other operations which are already defined in the system.

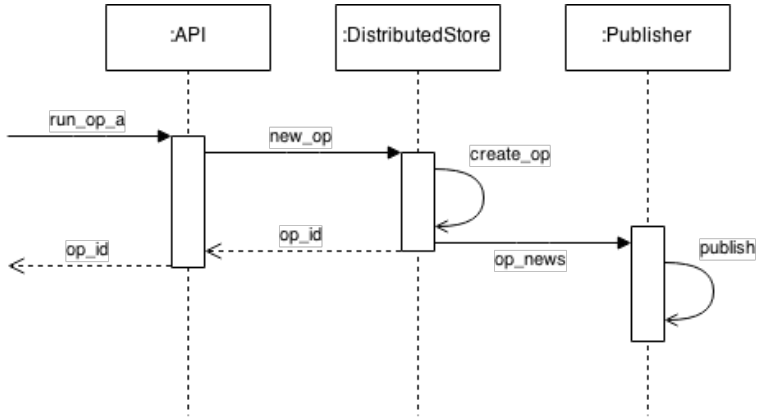

Figure 5.2: When a peer is notifed about an operation when it has the desired dataset

5.3.3 Realizing Queries

Having an operation id at hand a user can query the result of operations. Since each peer has enough information about the running operations it can return list of operations and list of datasets. So user can get informed about current operations and existing datasets and query for operation results or further details.

When user provides an id we return the operation details, such as sub-operation ids, state of each operation, duration, parent operation id (in case this operation is a child), operation name (the requested service), result dataset id, submit time and etc.

Providing a dataset's id we can return the dataset or *download* and store it in a local file to be observed by user. We discuss more on possible extensions in further works section.

Chapter 6

Prototype

6.1 Architecture

There are a number of possible ways to design a system to run distributed data intensive operations. Here are three well known approaches:

Conventional Approach there is no distributed application in this case. Applications run on desktop machines and they access data on on network stores such as NFS mounted or other distributed file systems. This is pretty much the same thing that many users of data intensive scientific applications do today.

Centralized Approach this approach is we have a central orchestrator which users connect to it directly or using a client to submit their operations. This is similar to the traditional client/server architecture. In many HPC distributed applications such as UNICORE, this would be the software which is installed on the cluster and could have multiple other machines -as resources- under its control. The emphasis in this model is providing a managed access to distributed computing resources.

Decentralized Approach in this approach we eliminate the orchestrator peer and the network of application instances should collaborate in a decentralized fashion to keep track of data and control flow for each task. This is the paradigm that we will follow in this chapter as our proposed design.

6.1.1 Collaborative design

To give a better understanding of our solution one should think of it as a distributed collaborative application. Even though one instance of our application has same basic functionalities as multiple peers together but it has been designed for collaboration and a single instance will only be functional if all the requested data are available locally. Nevertheless this makes it possible to use application in standalone mode with no peers which might come beneficial to some users with only local data.

We have picked a decentralized design, where peers will share the knowledge of running operations and existing datasets with one another. The next stark point

is that they will collaborate to accomplish one simple or complicated operation. In terms of delegating a simple operation from a node which does not have the required data - but has received the command to run such operation - to a node which contains the data.

6.1.2 Hexagonal Architecture

In a traditional approach toward application design we would have three tiers, i.e. client layer (GUI), business layer (logic) and database layer. These tiers correspond to a one dimensional application architecture, where there are only two *sides* assumed to exist around application logic, client and database. However this is not the case when we have a multi-dimensional architecture, where there are multiples input/output channels around our business logic. In the latter case we have to use a so called hexagonal architecture. [8] In such an architecture applications receive signals from multiple communication means at the same time. This signals will trigger the appropriate internal business logic, therefore they can't be layered in one dimension.

Here is a high level view of the components of the system:

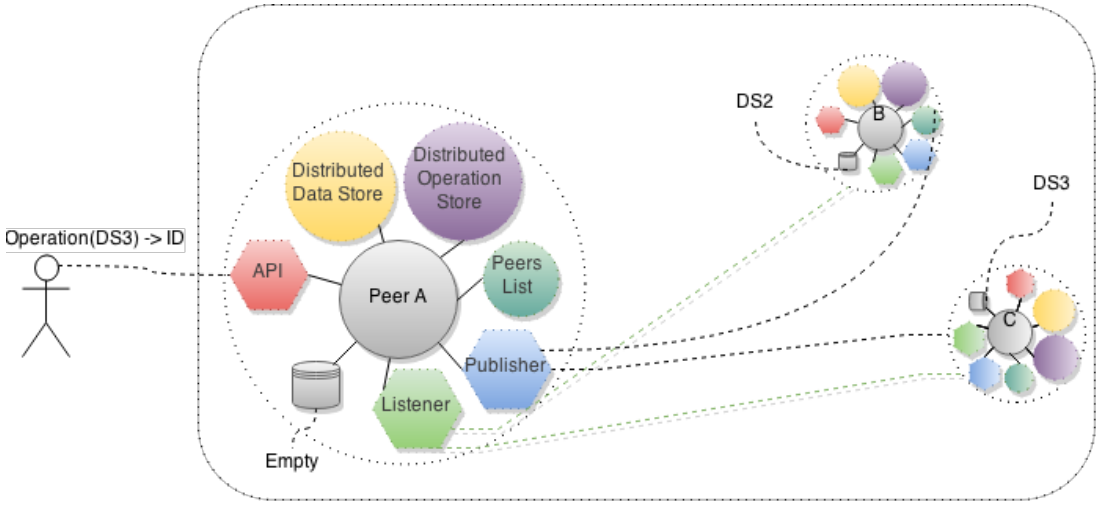

Figure 6.1: High level network view of the system

6.1.3 Actors

There are two types of actors in our problem domain.

User A user who launches, control and monitor an operation. Typically they are employees of scientific institutes or universities. The goal of these users is to utilize the program to launch some kind of simulation and get back the result.

Instance Every instance can launch and observe an operation on other instances on other nodes. If we launch a single instance network, then there would be

no other instance to talk to, therefore any recursive service call will happen on the same instance and not on any other one. When there are more than one instance on one network, on instance has the ability to call services/operations on other instances, basically using the same channel that one normal user does. This will make our application to act as a user of itself. We will utilize this when we introduce recursive service calls inside our application, hence the term *collaboration*.

6.1.4 Messaging

When it comes to choosing messaging technology, the field is dominated with centralized solutions. There are multiple solutions which bring messaging into applications, e.g. RabbitMQ, ActiveMQ, Celery and so on[1]. Most of them are centralized, even though they allow distributed paradigms, their still need central servers.

Because of the nature of our application, peers might join and leave the network. Therefore we need an transport layer which is designed having this in mind. We have selected ZeroMQ as our transport layer because it is trivial to build effective distributed messaging systems on top of that. It is an open source project and has a very active community with hundreds of contributors. It is written in C programming language and has bindings for multiple programming languages including Python. It is also multi-platform and is available on different architectures and operating systems.

Supported Patterns

ZeroMQ supports different messaging paradigms, among many others:

1. PUB/SUB which is publish/subscribe
2. REQ/REP which is similar to traditional client/server

We only use publish subscribe during the course of this work.

Publish/Subscribe

Since we need to inform other peers about certain topics, we need a mechanism to inform them all together. Any peer which is interested in other peers will *subscribe* to their *news channel*. The publish-subscribe pattern is the best way to realize this. ZeroMQ allows us to subscribe to any number of *publisher* channels. Here are some important aspects of ZeroMQ publish-subscribe sockets:

- An application can subscribe to non-existing or non-running publishers.
- A publisher will maintain separate queues and will keep messages for any subscribed party.
- A publisher will simply drop the messages if there are no subscribers.
- A publisher could have any number of subscribers.

[1] A compelling list could be found at http://queues.io/

- A subscriber could subscribe to any number of publishers.

The above mentioned features gives let use to design distributed. We could subscribe to peers regardless of their current state. They might fail and come back again but ZeroMQ will try recover and reconnect when the other peer is available. In the other hand, the subscribers will not lose their messages if they go offline and come back online again.

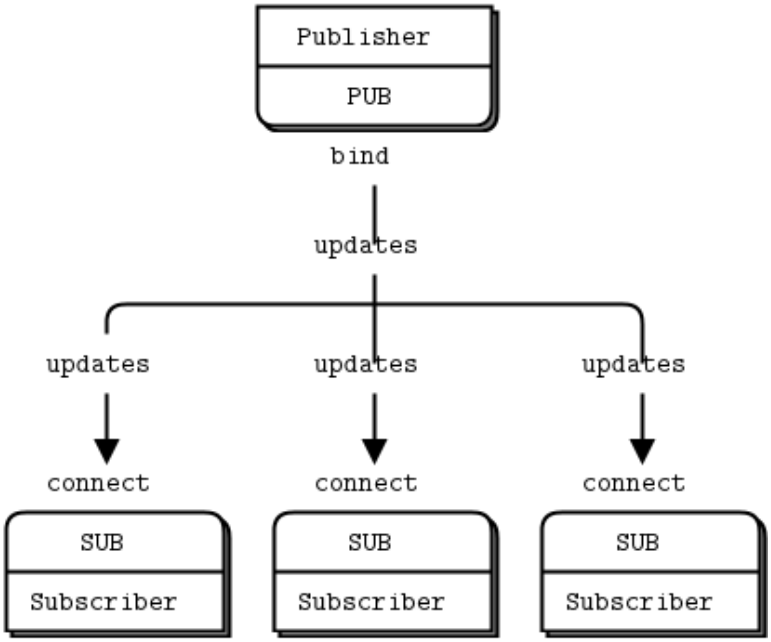

Figure 6.2: Publish-subscribe pattern from ZMQ guide. [18]

Message types

We have decided to have a number of messages which will be exchanged between peers. Each peer will listen to all the other peers via subscribing to their news channel. We assign a numeric id to each of these messages to identify them. Here is a short overview of these topics (they are called topic in ZMQ):

Delegation informs other about a request to be handled

Operation update informs others about a change in operation distributed store

Pull request ask others to download a temporary dataset from the provided endpoint

Topic Handlers

For each of these topics there are certain topic handlers designed to maximize flexibility of application and make it easy to introduce new topics and change topic handlers without touching or with minimum change to existing code.

Topic matching

Topic matching or message filtering is a technique as described in Advanced Message Queuing Protocol (AMQP) v1.0 specification[1]. For each peer we have one publisher channel and one subscriber channel. The publisher is responsible to publish any given message to all the subscribers. If we could assign certain message types in transport layer it makes it easier to handle them upon arrival. ZeroMQ provides a very basic matching algorithm for this purpose. It prefixes a message with a number then a blank space.

This is how a message will be published:

Listing 6.1: Message publisher code

```
1  import zmq
2  import msgpack
3  context = zmq.Context()
4  socket = context.socket(zmq.PUB)
5  # A sample local ip address
6  socket.bind("tcp://192.168.1.1:4000")
7  message = {'id': '247506b8-09e9-40d8-968d-6f739ba802d3'}
8  packed = msgpack.packb(message)
9  # An agreed constant for operation news
10 topic = 10013
11 socket.send('%s %s' % (topic, packed))
```

Messagepack is a serializer that we use in our transport layer. "MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller. Small integers are encoded into a single byte, and typical short strings require only one extra byte in addition to the strings themselves." as described by its developers[2].

This is how a subscriber will receive a message:

Listing 6.2: Subscriber receives and unpacks a message

```
1  import zmq
2  import msgpack
3  context = zmq.Context()
4  socket = context.socket(zmq.SUB)
5  # connecting to the publisher endpoint
6  socket.connect("tcp://192.168.1.1:4000")
7  while True:
8      msg = socket.recv()
```

[1] https://www.amqp.org/
[2] http://msgpack.org/

```
9       topic, delimiter, packed = msg.partition(' ')
10      topic = int(topic)
11      message_dict = msgpack.unpackb(packed)
12      # call appropriate handler here ..
```

6.1.5 Coupling

We have followed event driven pattern in our design. The application components are loosely coupled and communicate using signals.

To do so we have used a Python thread-safe signaling library called *blinker*[1].

The main cases that we use signaling is when a component wants to publish a message. In that case it would send a signal with corresponding topic and message. The publisher component will grab that signal, and will issue the real message to the peers.

In the other hand, when a new message arrives we will use dynamically registered handler objects. Than handler will run its logic code and will notify further components by issuing corresponding signals.

Here is a basic overview of publishing process. First in sender component:

Listing 6.3: Publishing an internal signal with blinker

```
1 from blinker import signal
2 publish_signal = signal('publish') # signals are named
3 publish_signal.send(sender, {'topic': 10013})
```

The publisher component will be notified in case of a new publish message. This is the code that initialized the publisher upon application start:

Listing 6.4: Initializing the publisher upon application start

```
1 def _run_publisher(self):
2     context = zmq.Context()
3     socket = context.socket(zmq.PUB)
4     socket.bind("tcp://%s:%s" % (self.ip, self.config['PUB_PORT']))
5     def publish_handler(sender, topic=None, **kwargs):
6     packed = msgpack.packb(kwargs)
7     socket.send('%s %s' % (topic, packed))
8     publish = signal('publish')
9     publish.connect(publish_handler, weak=False)
```

As seen in above code snippet, the publisher assigns a function to be run when a new signal arrives. The *weak=False* is to prevent *publish_handler* to be garbage collected when it goes out of scope with having a non-weak reference to it.

6.1.6 State

In current design we have no state machine but a number of objects have states. The most important ones are:

[1] "Fast, simple object-to-object and broadcast signaling" - https://pypi.python.org/pypi/blinker

- Operation store
- Dataset store
- Publisher

Operation store and dataset store are almost explained so far. The publisher is part of the application that has queues for each of subscribers. In case of a failure or restart, the messages inside these queues will be discarded.

6.2 Technology

We have created a Python application using Gevent[1], zeromq[2] and zerorpc[3] to be able to service multiple requests in a non-blocking way.

ZeroRPC is a remote procedure call framework which is built on top of ZeroMQ. It is capable of exposing a given object's methods as API and made them accessible using an endpoint address.

"gevent is a coroutine-based[4] Python networking library that uses greenlet[5] to provide a high-level synchronous API on top of the libev event loop."[6]

6.2.1 Programming Language

We selected Python as the main programming language to implement this project. There are a number of justifications to do so. Here are the main ones:

Multi-Platform Python is a multi-platform language. It runs on different operating systems seamlessly, hence easier deployment.

High Availability Python along with its rich standard library is available by default on almost all Linux machines. This is a great advantage for use, because we do not need to take further steps to install a runtime in highly conservative institutes.

Familiarity Python is already being used as main scripting languages in many scientific environments. This would be an advantage for us in further steps when users want to contribute to the project or maintain it.

Based on C Python is well known to be very close to C programming languages. Even though Python is slow in arithmetic operations it is possible to write speed critical parts in C and execute it directly within Python code. However in our current solution we do not have arithmetic operations.

[1] http://www.gevent.org/
[2] http://zeromq.org/
[3] http://zerorpc.dotcloud.com/
[4] PEP 380 - https://www.python.org/dev/peps/pep-0380/
[5] Lightweight in-process concurrent programming - https://pypi.python.org/pypi/greenlet
[6] See first footnote

Faster Development Since Python does not need special tools to build and deploy its scripts it is much cheaper and faster to start, build and test programs.

Aspect Oriented Support With Python it is very easy to wrap methods and apply pre-process and post-process conditions to them. We have used this aspect of the language to create operation ids, delegate service calls, distribute messages and etc before and after service calls.

6.2.2 Dependency Management

Using *pip*[1] it is very trivial to manage and install multiple dependencies of a project. It is capable of installing dependencies from remote git repositories or from the Python Package Index (PyPI)[2]. Moreover *pip* itself is a Python package. It gives us huge benefits with abstracting away the complexity of dependency management. It can bundle a package with compiled dependencies, install from Python wheel[3], uninstall, upgrade and query available PyPI packages.[4]

6.2.3 Virtual Environment

As described in the previous section, Python is the chosen programming language for this project. Along with Python, comes *virtualenv* package[5]. This is a great way of installing project dependencies into a single directory (which serves as the virtual root file system) and avoid touching operating system managed files and directories which normal users do not have access to them. While working inside a *virtaulenv*, all the changes is written to a single directory and all binary files, downloaded Python packages goes into that directory. Therefore this is the best way to deploy a Python project in user space.

6.2.4 ZeroMQ

The main library that powers our prototype is called ØMQ or ZeroMQ. ZeroMQ is an asynchronous messaging library written in C with bindings for many languages including Python. This library helps us to easily scale and use different programming paradigms such as publish-subscribe, request-replay and push-pull.

6.3 Key Compoenents

We have tried to make a modular architecture with cohesive components which are loosely coupled using dynamic method registeration. This is possible thanks to Python programming model, where functions and methods are first class objects and

[1] https://pypi.python.org/pypi/pip
[2] https://pypi.python.org/pypi
[3] A built-package format for Python
[4] https://pip.pypa.io/en/latest/user_guide.html#create-an-installation-bundle-with-compiled-depe
[5] virtualenv is a tool to create isolated Python environments

we can simply pass them around. We have applied this to topic handler registeration and internal signal handlers which we register dynamically.

6.3.1 Distributed Storages

Our aim is to distribute the information about available datasets and operations at each node. To achieve this we let our application to launch a number of communicators and publish information about it is data. Other nodes in our network have to subscribes on other nodes, ZeroMQ allows us to subscribe to multiple publishers, therefore each node can subscribe to other nodes. Nodes frequently get **news** from other nodes, for example availability of certain datasets on a node, then it can use publish-subscribe to get extra information on that particular subject.

Operation Store

Operation store is a dictionary[1] like object that hides that distributes its content to other peers. The interface provides getter and setter methods to explicitely add a new dataset to the store or get an existing one. Internally it will broadcast a specific signal along with basic information about the operation. It could be a newly added operation or an update to parts of the operation attributes such as a new dataset assigned to it or about the operation entering into a new state.

Dataset Store

Currently we have only implemented a *Random Dataset Store* which is similar to operation store it terms of hiding the storage and broadcast from the rest of the application. Meanwhile it has a few key differences. First of all it does not have an update method. Beside querying the datastore, it only allows the application to create a new dataset and will assign a unique identifier to it. There is currently no notion of updating a dataset. It will publish information about the newly added dataset, and it will use a temporary storage to buffer the dataset and will select a random peer and will use the networking layer to transfer the dataset to the randomly selected peer.

6.3.2 Decorators

We have used Python decorators extensively to apply pre and post processing during an operation call.

6.3.3 Application

We have capsulated the application itself as a top level class. A developer then would create an instance of this object and then can run it, access the application configurations, certain public objects, register topic handlers, ask for API endpoints, or application ports and so on.

[1] In Python dictionaries are simple key/value objects used extensively by the language itself

6.3.4 API

We have made a separate API Python module[1] to introduce our public API. We expose all the functions defined in this module as public services accessible from outside via application API port.

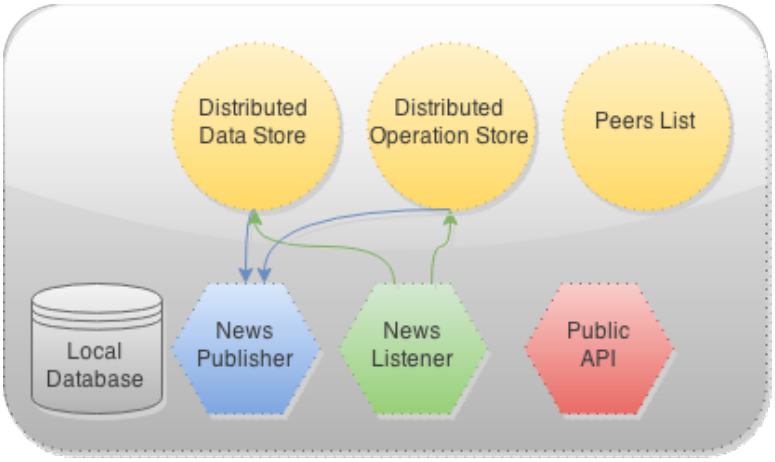

Figure 6.3: Another view of system compoenents

6.4 Layers

6.4.1 Network

This layer is responsible to run the publisher channel. Here are the responsibilities of this layer:

- Exposing application public interface
- Running publisher channel
- Listening to internal signals to publish corresponding messages
- Subscribing to peers
- Getting incoming messages from subscribers
- Find and call appropriate handler upon arrival of new messages

API

Since this is a network program we need to use a form of Remote Procedure Call (RPC) to communicate between nodes. We used a library based on zeromq called *zerorpc*. Using this library we now expose a set of APIs as services and let the nodes talk to each other based on this API. There are multiple solutions for exposing services which we do not discuss here.

[1]Modules in Python are simply files with .py extension which contain code

Publisher

Publisher is a ZeroMQ socket object which is binded to one port on the running machine. Having endpoint of this port, other peers can subscribe to it and listen to whatever it publishes.

Listener

Listener is another type of ZeroMQ socket. Having connected to a publisher's endpoint address, we can read any incoming messages, similar to regular sockets. But it has this feature of connecting to multiple endpoints.

6.4.2 Pre-processing

In Python we can simply wrap any function or method or even classes, inside another one. This is realized using decorators in Python. Basic usage of decorator in Python is like this:

Listing 6.5: Pre-processing with decorators in Python

```
1  def delegate(func):
2      @functools.wraps(func)
3      def new_func(*args, **kwargs):
4          # do preprocess
5          # call func, if desired
6          # do post process
7      return new_func
8
9  # In our code:
10
11 @delegate
12 def legacy_function(an_arg, an_option=somevalue, a_flag=False):
13     # some life-changing processing here...
```

As you can see, we wrapped a function only with putting *@delegate* on top of it. Now when *legacy_function* is called, the code inside the *new_func* block will be executed first. There we would have access to the original function and all of its parameters. We can then apply any required processing. We can even decide to call the real function all not. All of these without changing the caller code and makes our application very dynamic. This way it is really trivial to write pre-processing and post-processing for any function. We have implemented our pre and post processing using decorators.

API Calls

Since we have used zerrpc to expose our services, we have two ways of calling them. First is using the command line tool provided along with it which has the same name. The second is using zerorpc Python package which lets us to programmatically call our exposed services.

Here is how we call a service from command line:

Listing 6.6: Quering an API endpoint for available commands

```
\$ zerorpc tcp://127.0.0.1:9998
connecting to "tcp://127.0.0.1:9998"
list             Return a list of datasets or peers
echo             Echo
use_case_2       Invokes operation for use case 1 described in the report
             Accepts 'peers', 'datasets' and 'operations as option.
use_case_1       Invokes operation for use case 1 described in the report
             Accepts 'peers', 'datasets' and 'operations as option.
```

This is the the normal output when we provide the endpoint of one of our instances. We get a list of available (exposed) commands along with their documents.

Here we call a service:

Listing 6.7: Running an operation on a dataset

```
\$ zerorpc tcp://127.0.0.1:9998 use_case_1 ds1
```

This would be how we get the content of operation store after calling aforementioned command:

Listing 6.8: Running an operation from command line

```
\$ zerorpc tcp://127.0.0.1:9998 list operations
connecting to "tcp://127.0.0.1:9998"
{'1096486d-ed28-40b6-9254-9b2006e8557d':
  {'args': ['ds1'],
  'command': 'use_case_1',
  'duration': 5.635747909545898,
  'result_dataset_id': '3fb9a21c-aee3-4a6a-98b0-ad0890d75686',
  'state': 'done',
  'submit_moment': 1427122073.528059}}
```

To programmatically calling a service we would use zerorpc package:

Listing 6.9: Connecting to an API endpoint in ZeroRPC

```
import zerorpc
c = zerorpc.Client()
c.connect("tcp://192.168.1.1:4000")
answer = c.echo('hi')
```

6.5 Initialization

First of all each application instance establish its own zeromq publisher socket. Then it subscribes itself to all other nodes which are listed in config file. It will also bring up the API channel. There would be more steps such as registering handlers for various topics and preparing internal signal handlers.

6.5.1 Local Database

For testing purposes we used a temporary HDF5 file as backend storage.

6.5.2 Stores

During initialization step we initialize our dataset store with available datasets in our HDF5 file.

6.5.3 Network

All of network initialization will be done after reading the local store.

6.6 Deployment

The software is easily installable in a Python Virtual Environment. All the requirements could be installed within the virtual environment.

6.7 Test Results

To be able to assess the performance of each given solution to the mentioned scenarios we made a demo application called **Konsensus** which its code is available on Github. [**konsensus**]

6.7.1 Integration Tests

Writing integration tests for a distributed application is not as straightforward as writing unit-tests for a normal application. Our demo application acts as a server and client at the same time. Moreover we want to launch multiple network peers running on one or more machines. Testing scenarios on this network is not possible with normal mocking approaches, because we need to test the behavior of our solution in a network of collaborating peers which are not external, rather the core services of the application.

To overcome testing issues we have to launch the desired number of peers separately and then run our tests over them. To make this operation faster we changed the application to make it possible to launch any number of instances on one machine and we automated this process using a number of scripts.

Mixing Signals in Greenlets

We use Python Greenlets instead of threads. This means that our demo application runs on only one thread. This causes a problem when launching multiple apps all together with one script and inside one thread, that causes the signals for events spread among all greenlets and make trouble. To avoid this we have to run each server in a separate processes. Running them inside threads won't help as well because the blinker Python library is thread-safe so it moves signals between threads as well as Greenlets.

Chapter 7

Discussion and Outlook

7.1 Data Transfer

In current design we store files on a random temp folder for each peer. But we can take advantage of existing Distributed File Systems (DFS). We can then eliminate the complexity of data transfer among peers. DFS has not been considered in our current design but it is an essential part of distributed applications.

Currently we store result datasets randomly between peers as there is no network accessible storage. We also delegate operations to the peers which contain the data, but the data could be availabe via a shared storage. This will make the whole effort meaningless, or it could be better to say that we focus on situations that machines containg the data do not have access to shared storages. Again another possibility comes to mind, we could make a shared cloud or use a cloud storage provider to share data amoung our peers.

These assumptions somehow contradict with our initial requirement, where we having large datasets spread on multiple machines (and not on a shared storage accessible by all of them). We can also consider the price of accessing such data on shared storages and compare it with having faster local storages and transferring only necessary parts for each operation. Even though these data could be moved to a cloud but this is not the case for us.

The only improvement that I can think of here, is changing the storage strategy and use a cloud storage to keep the results (one advantage of having a flexible design is right here, where we can change our storage strategy with a change in configuration file). This will let us to avoid the expense of transfering datasets back and forth between our peers and we would enjoy the simplicity of having one *meta-disk* to work with. We don't rely on DFS in our desgin, we make the decision on which node we have to run the operation and when it comes to data transfer part we can use a *universal disk* concept to deliver the remaining data.

7.1.1 Large Dataset Transfer

To transfer large arrays over the network there are a number of considerations. Should the array be stored locally before transfer? What if the array is so big that it does not fit into the machines memory? And how the array should be transferred?

Currently we assume the result dataset to fit into the memory, therefore there is only the question of how to transfer them over the network. To prevent unnecessary copies, we consider streams to send them to other peers. In the demo application this is done with streaming sockets. The other peer will be notified and then it will fetch the desired dataset.

We need to develop a mechanism to consider dataset size for transfer. User defined files are normally small and we can safely transfer them but system datasets are large and for any transfer some sort of control should happen.

7.2 Possible Issues

7.2.1 High Load

7.2.2 Orphan Operations

7.2.3 Complexity Growth

7.3 Future Work

During this work we have focused on the aspects of the problem which were important in the context domain and we left aside many other small and big problems without considering them during this project. The main reason was that we wanted to work on problems which were new and genuine because for other aspects there are already many well-defined solutions available, so we did not spend our time for them. Moreover one should consider that this project is not solely an implementation but is a research on finding ways to embed distributed solutions into other projects.

In the following sections we talk shortly about the topics which we have not covered but this work can be extended to include them as well.

7.3.1 Non-linear Operations

The main part which have not been covered yet is non-linear operations.

7.3.2 Network Discovery

Currently the peers are configured in the beginning and there is no dynamic peer recognition. This might be done in a number of ways such as sending broadcasts or using third party projects such as Zyre [1]

At this stage there is no network discovery, because it is not our main problem. It can be done later as an improvement.

[1] https://github.com/zeromq/zyre

7.3.3 Bootstrapping

With having address of only one peer we would be able to configure and a new peer and join the network. There should be a mechanism among peers to identify joining and leaving peers. But our context is different than a peer-to-peer applications which peers join and leave frequently. In our case most of peers run a long time and bootstrapping is more a way to get the state of currently running workflows and let others know about the new peer.

7.3.4 Data Popularity

There are algorithms developed to calculate data popularity over time and then replicate them over peers for easier access. If we want to move toward any type of data replication we would need to use this algorithms.

7.3.5 Security

There is no user management and secure communication in our initial requirements however this would be required if we want to manage user rights or introduce limitations or simply to keep a history of activities for each user. Moreover to secure inter-peer communications we might use X.509 certificates. Further more since we've used ZeroMQ as underlying transport channel we can use its more advanced security features such as Elliptic curve cryptography[2] based on Curve25519[4] to add perfect forward secrecy, arbitrary authentication backends and so on.

7.3.6 Fault Tolerance

In current work there is no failure recovery mechanism, since it was not part of the requirements. In case of a failure or exception in any collaborating peer not only the failed instance should be able to recover itself into a correct state, moreover the other peers should maintain a valid state for on-going distributed workflows and keep their internal state up-to-date.

Like other topics in this section this one is not of our interest too. The point is there are existing solutions for these problems and we want to let our application to be able to demonstrate the main problem which would be deciding about data transfer routes and distributing the information about currently running operations.

7.3.7 Web Monitoring

Before starting my thesis we have developed a job submission and monitoring web application in order to get to know job scheduling backends and the workflow and user requirements. We called this tool Sqmpy and it is also open source and available on Github[1]. We can use Sqmpy project as a monitoring tool for konsensus network. Providing one peer address it can query the rest of peers and connect or subscribe to their news channel. Having this we can always see which nodes are offline and

[1] A Simple Queue Manager https://github.com/mehdisadeghi/sqmpy

which ones are online. This also gives us a platform to extend monitoring and control features to the web. Currently we have made the required software platform to achieve this. In the Sqmpy project we can simply maintain realtime connections to the browsers and since our web framework is written in Python, with minimum cost we can integrate it with konsensus which is written with Python as well.

Chapter 8

Conclusion

Even though there are many solutions designed for HPC problems, still there are requirements for smaller groups which are not satisfied, such as:

- Making scientific applications user friendly
- Providing *smarter* solutions which get out of users way, i.e. hiding the systems complexity from ordinary users
- The system manages data endpoints, not users
- Less deployment and maintenance cost
- More flexibility to control application at runtime

During this work we addressed some of these needs:

- The problem was defined and requirements where defined
- We went through the state of the art
- A solution approach was proposed
- A prototype was developed
  - Based on open technologies
  - Runs in user space
  - Open source and freely available on Github

Our approach is very flexible to be extended and it is easy to build new services on top of the existing framework which provides the distributed operation and storage mechanisms to applications.

References

[1] Deniz Altinbuken and Emin Gun Sirer. "Commodifying replicated state machines with openreplica". In: (2012).

[2] *An encryption and authentication library for ZeroMQ applications*. 2015. URL: http://curvezmq.org/.

[3] Wasim Bari, AhmedShiraz Memon, and Bernd Schuller. "Enhancing UNICORE Storage Management Using Hadoop Distributed File System". English. In: *Euro-Par 2009 – Parallel Processing Workshops*. Ed. by Hai-Xiang Lin et al. Vol. 6043. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 345–352. ISBN: 978-3-642-14121-8. DOI: 10.1007/978-3-642-14122-5_39. URL: http://dx.doi.org/10.1007/978-3-642-14122-5_39.

[4] D. J. Bernstein. *A state-of-the-art Diffie-Hellman function*. 2015. URL: http://cr.yp.to/ecdh.html.

[5] Peter Bui, Li Yu, and Douglas Thain. *Weaver: Integrating Distributed Computing Abstractions into Scientific Workflows using Python*. 2010. URL: http://ccl.cse.nd.edu/research/pubs/weaver-clade2010.pdf.

[6] Weiwei Chen and E. Deelman. "WorkflowSim: A toolkit for simulating scientific workflows in distributed environments". In: *E-Science (e-Science), 2012 IEEE 8th International Conference on*. Oct. 2012, pp. 1–8. DOI: 10.1109/eScience.2012.6404430.

[7] Jared Bulosan Christopher Moretti et al. "All-Pairs: An Abstraction for Data-Intensive Cloud Computing". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* 11 (2008), pp. 352–358.

[8] Alistair Cockburn. *Hexagonal Architecture*. 2015. URL: http://alistair.cockburn.us/Hexagonal+architecture.

[9] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Proc. Sixth Symposium on Operating System Design and Implementation, 2004.

[10] Thomas Eickermann et al. "Steering UNICORE applications with VISIT". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 363.1833 (2005), pp. 1855–1865. ISSN: 1364-503X. DOI: 10.1098/rsta.2005.1615.

[11] *EMI 3 Monte Bianco Products*. 2015. URL: http://www.eu-emi.eu/products.

[12] *European Middleware Initiative*. 2015. URL: http://www.eu-emi.eu.

[13] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Advanced computing. Computer systems design. Morgan Kaufmann Publishers, 1999. ISBN: 9781558604759. URL: http://books.google.de/books?id=ONRQAAAAMAAJ.

[14] Ian Foster. "Globus Toolkit Version 4: Software for Service-Oriented Systems". English. In: *Network and Parallel Computing*. Ed. by Hai Jin, Daniel Reed, and Wenbin Jiang. Vol. 3779. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 2–13. ISBN: 978-3-540-29810-6. DOI: 10.1007/11577188_2. URL: http://dx.doi.org/10.1007/11577188_2.

[15] Erik Gafni et al. "COSMOS: Python library for massively parallel workflows". In: *Bioinformatics* (2014). DOI: 10.1093/bioinformatics/btu385. eprint: http://bioinformatics.oxfordjournals.org/content/early/2014/07/24/bioinformatics.btu385.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/early/2014/07/24/bioinformatics.btu385.abstract.

[16] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". English. In: *Peer-to-Peer Systems*. Ed. by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Vol. 2429. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-44179-3. DOI: 10.1007/3-540-45748-8_5. URL: http://dx.doi.org/10.1007/3-540-45748-8_5.

[17] C. Moretti et al. "All-pairs: An abstraction for data-intensive cloud computing". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. Apr. 2008, pp. 1–11. DOI: 10.1109/IPDPS.2008.4536311.

[18] *ØMQ - The Guide*. 2015. URL: http://zguide.zeromq.org/page:all.

[19] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *Proc. USENIX Annual Technical Conference*. 2014, pp. 305–320.

[20] S. Pandey et al. "A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments". In: *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. Apr. 2010, pp. 400–407. DOI: 10.1109/AINA.2010.31.

[21] K. Plankensteiner, R. Prodan, and T. Fahringer. "A New Fault Tolerance Heuristic for Scientific Workflows in Highly Distributed Environments Based on Resubmission Impact". In: *e-Science, 2009. e-Science '09. Fifth IEEE International Conference on*. Dec. 2009, pp. 313–320. DOI: 10.1109/e-Science.2009.51.

[22] K. Ranganathan and I. Foster. "Decoupling computation and data scheduling in distributed data-intensive applications". In: *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*. 2002, pp. 352–358. DOI: 10.1109/HPDC.2002.1029935.

[23] K. Ranganathan and I. Foster. “Decoupling computation and data scheduling in distributed data-intensive applications”. In: *Proc. 2002 High Performance Distributed Computing IEEE International Symposium* 11 (2002), pp. 352–358.

[24] S. Shumilov et al. “Distributed Scientific Workflow Management for Data-Intensive Applications”. In: *Future Trends of Distributed Computing Systems, 2008. FTDCS ’08. 12th IEEE International Workshop on*. Oct. 2008, pp. 65–73. DOI: 10.1109/FTDCS.2008.39.

[25] *The Hadoop Distributed File System*. URL: http://www.aosabook.org/en/hdfs.html.

[26] *The Hadoop Distributed File System: Architecture and Design*. 2007. URL: http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf.

[27] *Twisted Matrix Project*. 2015. URL: https://twistedmatrix.com/.

[28] *UNICORE*. 2015. URL: https://www.unicore.eu/unicore/architecture.php.

[29] *UNICORE*. 2015. URL: http://en.wikipedia.org/wiki/UNICORE#UNICORE_in_Research_.26_Production.

[30] Jianwu Wang et al. “A High-Level Distributed Execution Framework for Scientific Workflows”. In: *eScience, 2008. eScience ’08. IEEE Fourth International Conference on*. Dec. 2008, pp. 634–639. DOI: 10.1109/eScience.2008.166.

[31] Qishi Wu et al. “Automation and management of scientific workflows in distributed network environments”. In: *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. Apr. 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470720.