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 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.

February 28, 2015

Mehdi Sadeghi

Abstract

This document contains my master thesis report including the problem definition, requirements, problem analysis, an overview of state of the art, proposed solution, designed prototype, discussions and conclusion. During this work we have proposed a solution to collaboratively run various types of operations in a network without any broker or orchestrator. We have defined and analyzed a number of scenarios according to our requirements and we have implemented the solution to address those scenarios using a distributed workflow management approach. We explain how we break a complicated operation into simple parts and how we will calculate the final result without any central broker. We will show how we asynchronously launch operations on the network and how we store and collect results of collaborating peers.

Contents

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 and to minimize data transfer during such operations.

This work has not been an implementation task nor a purely theoretical work. It means that I was not supposed to create an application or develop a software from ground up (even though eventually I did), instead I have been responsible to study about and define the problem of my client and assist them either with finding a suitable solution and helping them to integrate it into their development process or propose a new approach to address their needs. My activities include but not limited to analyzing the problem, collecting requirements, studying state of the art software frameworks and related products, analyzing them against the defined requirements, proposing a solution and developing a prototype.

During this thesis an open source prototype application has been developed which is available online[1]. The source files of the current document are also available online[2]. If there are any comments and improvements regarding this document, I appreciate an email to **sadeghi@mehdix.org**.

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

Chapter 1

Introduction

1.1 Objectives

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

- distributing the workflow of 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 does not meet our requirements. This will be discussed in more detail in chapter 2.

1.2 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 We mean both consumed and produced data of scientific applications .e.g. NumPy arrays or HDF5 datasets.

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.

1.3 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 softwares which are being used to accomplish multi-scale modeling tasks.

These tasks often involve running multiple simulation programs over the existing datasets or the data which is produced by other simulation softwares. It's common to run multiple programs on existing datasets during one operation to produce the desired results. The order to run simulation softwares is normally defined within scripts written by users.

Moreover users have to provide the required data manually and copy all required files to a working directory to submit their job, and they might have to login to different machines to prepare files, submit the script, monitor the status of the job and finally collect the output files. This type of work routine is a common form of workflow management in above mentioned communities.

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.

1.3.1 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

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 amoung these machines, but have to provide smarter ways to move data between these machines during operation executions.

1.4 Document Overview

This document is organized into 8 chapters. Here is a short overview of them:

Introduction The opening chapter.

Requirements The problems that we are supposed to answer.

Literature Introducing the related work.

Analysis Different aspects of the problem will be discussed.

Proposal We will explain the proposed solution.

Prototype Implementation of the proposal.

Discussions About the applicability of the proposed solution.

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

Conclusion Summary and future work.

Chapter 2

Requirements

Every group has its own needs. That is the reason that we have many different solutions in the market. In this chapter we explain the requirements according to our context.

2.1 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 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 a one whole whick has knowledge of currently running operations along with available datasets and preserves this state among all the nodes.

2.3 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.

2.4 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.

2.5 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.

2.6 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.

2.7 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.

2.8 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.

2.9 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.

2.10 Assumptions

During this work we have a number of assumptions. We have a certain problem which we want to focus on rather than reintroducing solutions that already exist. Here are the assumptions that we have considered during this work:

2.10.1 Data Characteristics

We need to discuss more about the data. In our scientific context data is mostly numerical and explains characteristics of physical particles such as atoms and molecules. These data is being used to simulate collections of particles called models. Although our work is not dependent on these, they help us to understand the definition of the data that we often refer to in this report. One important aspect of the data that we are interested in is that it is not critical and we can reproduce it.

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

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 desinged 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 analyse 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 protocls. 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 accomlishing 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 visulaizing 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 layor 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 supercomputing domain to allow exposing and mangaing 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 intented to be installed on a cluster site not normal workstations. To access web features a browser plugin should be installed[5]. 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 transfered to the UNICORE storage using the client. There are also command line tools such as *uftp* to transfer files to remote storages[6]. 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.

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

[6] 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 stablished 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 futher sevices to use GT libraries to build new services.[14]

Unlike UNICORE which is heavily dependant 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[7].

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)[8].

[7] http://toolkit.globus.org/toolkit/docs/4.0/data/key/
[8] http://toolkit.globus.org/toolkit/docs/6.0/licenses/

Data provision methods

GT implements GridFTP protocl[9] 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 machiens 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)[10], 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[11].

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 devloping 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.

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

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

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

3.1.3 Apache Hadoop

3.2 Distributing Data

Before introducing approaches toward data distribution we define our parameters:

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

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)

"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 It needs Java 1.5.x ssh and sshd and rsync. Three basic modes are available: Local, Pseudo-Distributed and Fully Distributed mode. XML configuration, installation of Local and Pseudo Distributed modes are almost straight forward, for fully distributed note extra steps are required (official doc link is dead)[12].

Fault Tolerance "Hardware failure is the norm rather than the exception." "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]

"HDFS has been designed to be easily portable from one platform to another."

[12] http://hadoop.apache.org/docs/r0.18.3/quickstart.html

Accessibility

1. FS Shell
2. DFSAdmin
3. Browser

Applicability There is a good document here: http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf Hints: HADOOP is for big data and the programming should be different (map/reduce) and it does not look suitable for our use cases and requirements. The burden would be so high that we will waste a lot of resources. I have to put these in scientific words with more logic and references to sizes that we need and more numbers.

Users have to program their applications using Java and Hadoop to take advantage of distributed computing features in Hadoop MapReduce and HDFS. Cites? Hadoop website? https://infosys.uni-saarland.de/publications/BigDataTutorial.pdf

iRod

Ceph

3.3 Distributing State

As previous sections first we introduct some parameters:

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

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 The Raft Consensus Algorithm

[**ongaro2014search**]

3.3.2 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 neighbours [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:

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

import sys, os
sys.path.append(os.path.dirname(__file__))
from kademlia.network import Server
from kademlia import log

application = service.Application("kademlia")
application.setComponent(ILogObserver,
    log.FileLogObserver(sys.stdout, log.INFO).emit)

if os.path.isfile('cache.pickle'):
    kserver = Server.loadState('cache.pickle')
else:
    kserver = Server()
    kserver.bootstrap([("178.62.215.131", 8468)])
kserver.saveStateRegularly('cache.pickle', 10)

server = internet.UDPServer(8468, kserver.protocol)
server.setServiceParent(application)


# Exposing Kademlia get/set API
from txzmq import ZmqEndpoint, ZmqFactory, ZmqREPConnection,
 ZmqREQConnection

zf = ZmqFactory()
e = ZmqEndpoint("bind", "tcp://127.0.0.1:40001")

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[27] and one python implementation[1] of *Kademlia* DHT algorithm[16]. This will start a p2p 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:

```
1  #
2  # Request-reply client in Python
3  # Connects REQ socket to tcp://localhost:5559
4  # Sends "Hello" to server, expects "World" back
5  #
6  import zmq
7
8  # Prepare our context and sockets
9  context = zmq.Context()
10 socket = context.socket(zmq.REQ)
11 socket.connect("tcp://localhost:40001")
12
13 # Set request
14 socket.send(b"set:", zmq.SNDMORE)
15 socket.send(b"the key", zmq.SNDMORE)
16 socket.send(b"the value")
17 print socket.recv()
18
19 # Get request
20 socket.send(b"get:", zmq.SNDMORE)
21 socket.send(b"the key")
22 print socket.recv()
23
```

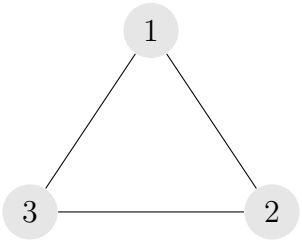

Figure 3.1: A network of three peers

```
24 # Invalid get
25 socket.send(b"get:", zmq.SNDMORE)
26 socket.send(b"not existing")
27 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 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.

3.3.3 Concoord

Describe why it is not suitable for us. It allows single object sharing.

3.4 Distributed Workflows

For our last section of this chapter we also need to define some parameteres in the beginning:

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

In this section we introduce a number of existing scientific workflow systems.

3.4.1 COSMOS

[15]

3.4.2 Weaver

[5]

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 influces 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 insterested 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 devide 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 amoung 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 scheulders. 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 explicitely 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 desribe 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 prividing an identifier, in this case filename. Another point about data files is that they are normally not mission critical and could be reproduced, hence is the emphasizis 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 get 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 (of course except the 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.

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, therfore 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 genearl 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 Scneario 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. This server is assumed to be part of the network, though it may not have any local data stores on it. The user issues, interactively (or non-interactively) a command on a set of data 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: it is assumed that at any instance of time, only one server acts as a client.

Moreover 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 intented logic.

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

The user issues the command like this from a python shell:

- real2bin(DS1) will result in -¿ Broadcast(real2bin(DS1))
- Note: it is assumed that all functions are already defined on all servers, since they execute the same environment.

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 command above, she does not get a non-existing dataset error from the client, just because the data is not there. Hence we need some interface, or some wrapper function that checks the argument for the data type, or to create some proxy interface from all data to all nodes.

In proposal chapter5 we explain in detail our suggested solution.

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

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

Theoratically 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 complecated. 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 previos 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 scneario 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 possiblities 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 discuess 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, whilc describing our design.

The next import question regarding operations is data availibility. If a local machine has received an operation and the required dataset is available on locall machine then there is no need for any transfer, then we would need only to distribute the operation information amoung 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 executables 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 discuess 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 sove 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 desgin 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 eachother. 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 laucn 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 afore-mentioned 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 identifer 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 autonomus peers. This design allows us to have peers which are able to work independantly, 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 Idea

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 operationor 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 complexitly of implementing more complicated operatins. 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 amoung the peers. Any change in datasets in any collaborative node will also be synchronized with peers and will be added to a distributed list.

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 **??** 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 scenrio 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 implementaion of this process will be presented in the prototype.

5.2.2 Non-blocking Calls

We base our desing on non-blocking service calls. This means all the calls in our system would be asynchronous. When a user calls a servie, 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 amoung all participating nodes. Any further infromation 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 futher work.

5.2.5 Distributed stores

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 identifers. Either id of an operation or id of a dataset. Then we will store futher 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 dependant 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 catched and handled buy 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 updted their own storages.

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 Collectors

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.

5.2.7 Unique IDs

5.3 Operation Types

5.3.1 Simple Operation

5.3.2 Mixed Operation

5.4 Using Prior Art

5.4.1 Data Transfer

We can take advantage of existing Distributed File Systems (DFS) to make the data available for operations. We can then eliminate the complexity of data transfer between these two nodes and delegate it to existing distributed file systems. The main point is we don't rely on DFS for all of our decision making part but we explicitly make the decision which operations to run on specific nodes and then for the data transfer part we can use a meta or universal disk concept to deliver the remaining data.

5.5 Proposed Approach

In order to calculate the result we might take a number of approachers, we start with a combination of **divide and conquer** and **produce-consume-collect** methods.

The S0, in this case, is the peer who receives the command and initiates the request. The two other peers, S1 and S2 respectively, have the required datasets. The initiator will find the corresponding datasets and will dispatch commands to run each part on each peer and then will collect the resulting datasets. This will be a blocking operation, we will wait until the other peers finish their parts and return the result to us. If the output is a number it will be returned to the user, if it is a dataset it will be stored based on defined storage mechanism, currently we use random storage. The peer will break the operation into smaller operations each one calculating result for one of datasets, this **sub-operations** will be executed like **scenario 1** and the result will be collected by initiator peer.

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].

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

The notation of above mentioned approach will be like this:

$$Operation(A + B) = Operation(A) + Operation(B)$$

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. In this way the user will provide a dataset name for storing the result. The operation will be **submitted** to the collaborative network. Later on user is able to query for the result using the key that she had provided at the time of submission. This allows us to design our system in a more decentralized way, where each peer can inform others (neighbors) about a request in a **publish-subscribe** manner, where the peer will publish a request and finish the operation. Later on the peer who has the dataset will **react** to the published request and will take further actions, all the other peers who do not have the requirements (the dataset for now) will ignore it, however they can store the details of running operations for next steps, when we will come to more complex workflow.

To show more detailed version of this operation we demonstrate the steps for it:

1. User issues the command to S0, providing DS1, DS2 ~~and a unique name for the result~~
2. System will check whether the operation is linear
3. Then it will break the command into sub-commands, each for one of datasets
4. System will generate unique ids for each sub-command

5. System will then submit the sub-commands along with dataset name and the unique id for the result dataset to **itself**, which will cause a situation like scenario 1

6. System will next have to collect the results in a non-blocking manner which we will discuss shortly.

- 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.

There is an important issue here, we create sub-operations for each operation and we run them in a non-blocking manner, this will cause it almost impossible to return the result of operation to the user in one run. One might think that we can block and query until the result of sub-operations are ready, but this is something that we want to avoid. Therefore to solve this issue in a distributed manner, we introduce an operation id for each user request. We inform all the peers via sending messages (signals) about the new operation and it is id and sub-commands. Each peer will update this operation internally based on further received messages. We also return the operation id to the user instead of any results. Then user will query for the result of operation, providing the operation id. We change the above steps like this:

1. User issues the command to S0, providing DS1, DS2 and a unique name for the result

2. **System will generate a unique id for the operation and will store it along with the parameters**

3. System will check whether the operation is linear

4. Then it will break the command into sub-commands, each for one of datasets

5. System will generate unique ids for each sub-command

6. **System will notify other peers about the incoming operation with related parameters**

7. System will then submit the sub-commands along with dataset name and the unique id for the result dataset to **itself**, which will cause a situation like scenario 1

8. System will next have to collect the results in a non-blocking manner which we will discuss shortly.

9. System will return the operation id to the user

In the other hand the other peers which are the same basically, will react to the new operation signal:

1. Receive operation update message
2. Make a local lookup if the operation should be added or updated
3. Add or update the operation in the local storage

Having the operation id and local updating storage for operations we now need to find a way to collect the results. First of all we need to decide which peer will collect the results. We take the most straight forward for now, the initiator peer, which has the knowledge of existing datasets in the network along with their sizes, will pick the peer which contains the largest dataset as the collector peer. We explicitly decide about the collector node in the beginning either by size or randomly amount the data container peers.

It is worthy to mention that the collector peer will then store the result based on the configured storage mechanism which is random storage for now, not necessarily storing on the same node.

Now we have enough information in each peer to collect, process and store the results. The peers (including the collector) will react to operation methods like this:

1. Receive operation update message
2. Make a local lookup if the operation should be added or updated
3. Add or update the operation in the local storage
4. Am I the collector? If yes do the followings:
    - check if the sub-operations are done
    - If the sub-operations are done, collect their results
    - Process the results
    - Based on the storage mechanism store the result
    - Update the operation with the result dataset id
    - Change status of operation to "done" (we need a proper state-machine here)
    - Inform other peers about the update

Now if user makes a query giving the operation id this would be the result:

1. Check operation storage
2. If the operation is marked done, return the dataset id
3. If it is not done, return the status.

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 storages such as NFS mounted or other distributed file systems. This is pretty much the same thing that many users of data intensive scentific 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 operatinos. This is similar to the traditional client/servier 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 emphasise 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 desined for collaboration and a single instance will only be functionl 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 operatinos and existing datasets with one another. The next stark point is that they will collaborate to accomplish one simple or complicated operatin. In

terms of delegating a simple operatino 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 dimentional 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-dimentional 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 dimention.

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 nomral 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

Publish/Subscribe

Filtering

6.1.5 Coupling

6.1.6 State

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.

6.2.1 Programming Language

We select 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 arithmatic 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 arithmatic operations.

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.

[1] http://www.gevent.org/
[2] http://zeromq.org/
[3] http://zerorpc.dotcloud.com/

6.2.2 Dependency Management

Using *pip*[4] 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)[5]. 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[6], uninstall, upgrade and query available PyPI packages.[7]

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[8]. 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 ØMQ

The main library that powers our prototype is called ØMQ or ZeroMQ [32]. 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 Compoenents

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.

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

Operation Store

Dataset Store

6.3.2 Message Handlers

6.3.3 Decorators

6.3.4 Application

6.4 Layers

6.4.1 Network

API

Since this is going to be a network program we need to use a form of Remote Procedure Call (RPC) to communicate between nodes. Rather than implementing ourselves we used a library based on zeromq called *zerorpc*. Using this library we now expose a set of APIs 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.

Publisher

Listener

6.4.2 Pre-processing

Incoming Messages

API Calls

6.4.3 Backend

Logic

Stores

Database

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.

6.5.1 Local Database

6.5.2 Stores

6.5.3 Network

6.6 Control Flows

6.6.1 API Call Flow

Possible Reactions

6.6.2 Incoming Message Flow

Possible Reactions

6.7 Queries

6.8 Deployment

The software is easily installable in a Python Virtual Environment.

6.9 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. [22]

6.9.1 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.

Scnenario 2 Errors

While testing scenario 2 we observed a common error. We this scenario with three different peers as the following table:

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

We also used **Random Dataset Storage** mechanism, simply to store resulting dataset of one operation on one of the network peers. The problem is when two peers decide to store the result of one operation on each other a blocking condition happens. Our approach was opening a temporary port and inform the other peer to fetch the data. Meanwhile this is exactly happening on the other peer, therefore both block.

Solution To solve the blocking peers problem we used the already running main application API instead of opening temporary PULL/PUSH zeromq sockets. This change is working fine and the peers exchanging datasets with no problem.

Chapter 7

Discussions

7.1 Possible Issues

7.1.1 High Load

7.1.2 Orphan Operations

7.1.3 Complexity Growth

7.1.4 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 datasets fit into 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.

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.

8.1 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.

8.1.1 Non-linear Operations

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

8.1.2 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 [33].

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

8.1.3 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.

8.1.4 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.

8.1.5 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[32] 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.

8.1.6 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.

8.1.7 Web Monitoring

Before starting my thesis I 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 [23]. 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 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.

References

[1] *A DHT in Python Twisted*. 2015. URL: https://github.com/bmuller/kademlia.

[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] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Mehdi Sadeghi. *Konsensus, a distributed flow control and data transfer backend*. 2015. URL: https://github.com/mehdisadeghi/konsensus.

[23] Mehdi Sadeghi. *Sqmpy, A simple queue manager to submit and monitor jobs on computing resources*. 2015. URL: https://github.com/mehdisadeghi/sqmpy.

[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.

[32] *ZeroMQ*. 2015. URL: http://zeromq.com/.

[33] *Zyre*. 2015. URL: https://github.com/zeromq/zyre.