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 analysed 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 comlicated 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 on our network 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 theoratical 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 analysing the problem, collecting requirements, studying state of the art software frameworks and related products, analysing them against the defined requirments, 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 Terminology

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 credenticals. 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 schedueleres enforce some policies to job submissions.

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

- Users do not have admin 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 comparision require multiple datasets at the same time which also called All-Pairs problem. [6]. 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.

1.4 Overview

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

Introducation The openning chapter.

Requirements The problems that we are supposed to answer.

Literature Intoducing 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.

Conclusion Summary and future work.

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

Chapter 2

Requirements

2.1 Identical Instances

We assume that on each machine of the network the same instance of our program is running which is capable of all of our operations including A, B and C. The only consideration is the availability of Datasets, they are not available on all machines.

Assuming that operations A and B will run on the machines which contain the required data, a number of questions arise here:

1. On which machine operations C should run? A, B or C?
2. On How to transfer the required data to that machine in an optimized way?

2.2 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. For this reason we discuss regarding our needs.

2.2.1 Collaborating Network

We assume there is a network of computers which are available to run the tasks, each node is running an instance of the application. We will propose our collaboration and data transfer algorithm between them later.

2.2.2 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.

2.2.3 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.

2.2.4 Workflow

In contrast to data we are interested in workflow. We want to find a reliable approach to access and update state of our workflow on any arbitrary node which is part of our collaborative network.

Chapter 3

Related Work

In this chapter we will go through a number of existing solutions. First we introduce the parameters which are important for us and we are interested in them.

3.1 Parameters

According to our requirements there are a number of parameters which are important to use. We asses existing products against our set or requirements and we leave out other factors. Here is a list of main factors:

- Centralization
- Data abstraction
- Runtime control
- Deployment complexity
- Possiblity to run in user space
- Integrating into other programs

3.2 Grid Computing Solutions

First of all we go through a number of projects which are widely being used.

3.2.1 UNICORE

UNICORE is one of the prducts of European Middleware Inititative(EMI) [10].

3.2.2 Globus Toolkit

3.2.3 Hadoop

3.3 Distributing Data

3.3.1 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." [22, tp. 3]

"Hadoop1 provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce [8] paradigm."[21]

"HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes."[21]

Deployment Complexity src:http://hadoop.apache.org/docs/r0.18.3/quickstart.html 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).

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." [21]

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

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

3.4 Distributing State

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

3.4.1 Distributed Hash Tables (DHT)

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

Kademlia

Kademlia is a p2p DHT algorithm introduced in 2002. We first tried to use it as a distributed key/value store but it is not suitable for our case and changes do not propagate only to a few neighbours [12].

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:

```
1  from twisted.application import service, internet
2  from twisted.python.log import ILogObserver
3
4  import sys, os
5  sys.path.append(os.path.dirname(__file__))
6  from kademlia.network import Server
7  from kademlia import log
8
9  application = service.Application("kademlia")
10 application.setComponent(ILogObserver,
11     log.FileLogObserver(sys.stdout, log.INFO).emit)
12
13 if os.path.isfile('cache.pickle'):
14     kserver = Server.loadState('cache.pickle')
15 else:
16     kserver = Server()
17     kserver.bootstrap([("178.62.215.131", 8468)])
```

```
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)

def getDone(result, msgId, s):
    print "Key result:", result
    s.reply(msgId, str(result))

def doGetSet(msgId, *args):
    print("Inside doPrint")
    print msgId, args

    if args[0] == "set:":
        kserver.set(args[1], args[2])
        s.reply(msgId, 'OK')
    elif args[0] == "get:":
        print args[1]
        kserver.get(args[1]).addCallback(getDone, msgId, s)
    else:
        s.reply(msgId, "Err")

s.gotMessage = doGetSet
```

In the above example we have used *twisted* networking library[23] and one python implementation[1] of *Kademlia* DHT algorithm[12]. 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:

```
#
# Request-reply client in Python
# Connects REQ socket to tcp://localhost:5559
# Sends "Hello" to server, expects "World" back
#
import zmq

# Prepare our context and sockets
context = zmq.Context()
```

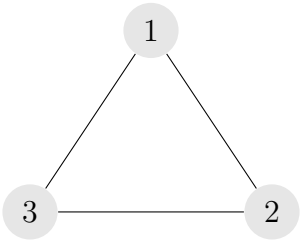

Figure 3.1: A network of three peers

```
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
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 11.

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

Firewall Problems In a test having one process running on a server in Internet and outside of the local network and having two different processes running on

one laptop but on different ports it is observed that the changes (sets) in the Internet does not replicate to the local processes but the changes from local processes are being replicated to the other process.

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

3.4.2 Concoord

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

3.5 Distributed Workflows

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

3.5.1 COSMOS

[11]

3.5.2 Weaver

[4]

Chapter 4

Problem Analysis

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 respectively, but they are not necessarily on the same nodes. In the following paragraphs we explain possible combinations of operations, nodes and datasets.

4.1 Operations

4.1.1 Types

In every scenario we want to run an operation which could be linear or non-linear.

Linear Operation

Being linear means that the operation could be broken into its components and then run in parallel or series. Here is algebic notation of a linear operation which acts on two datasets:

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

Being linear or non-linear only matters when we have to operate on more that one dataset.

Non-linear Operation

In contrast to linear there is non-linear operation. This means that this kind of operation has dependant parts and those parts could not run in parallel:

$$Operation(A + B) \neq Operation(A) + Operation(B)$$

4.1.2 Input/Output

For each operation we need one or more datasets which may be available on the same node that wants to run the operation initially or could reside on other nodes.

Input

Input files are normally not mission critical and could be reproduced.

Output

Operations create output datasets which normally are small in size, threfore we ignore the transfer cost of operation results in our work.

Locating

Every input or output needs an explicit address, an endpoint, either local or remote.

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 the result name. We need a consistent way of naming datasets. If we ask users to provide resulting dataset names it will break soon, we need to let user to somehow give some **tags** but not the real names. We have to let the user know about the result name but also let her to look for datasets by providing some tags.

The simplest problems that will happen if we store datasets with similar names is redundant work in the network. Peers will start to process and override the same dataset.

4.2.1 Data Manipulation

We will neet to let users to manipulate currently existing datasets, but very fast it comes to mind that not every dataset should be writable, we will need to categorize and identify our datasets based on some criteria. These problems are not part of my thesis but We mention it as part of problem analysis.

4.3 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. Is the data available locally?
2. If not, is the data available on another node? – Here only the physical location of data matters not the instance controlling it.

4.4 Scenarios

We begin with a simple scenario and we gradually add details to it and build new scenarios.

4.4.1 Scenario 1 (UC1)

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.

Assumptions

There are two main assumptions here:

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

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 broadcast 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. Let us assume 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 here to be a user defined method that operates on the data set.

- 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 data set 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, he/she does not get a "data structure not existent" 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.

4.4.2 Scenario 2 (UC2)

This is similar to scenario one, except that the operation requires two datasets to be done. We have a network of peers collaborating to finish some linear and non-linear tasks. 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 datasets which are not available on the first peer and we should access them on other peers.

Assumptions

The main points the same:

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

We assume the data distribution is like the following table:

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

Possible 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 straigh 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 5

Proposed Design

5.1 Possible Approaches

During this work we consider three different approaches toward preparing required data for operations.

Conventional Approach in this approach we put the required data on a network file system and all application instances will access it there utilizing NFS mounted file system or other distributed file systems.

Centralized Approach in this approach we have a central instance which will orchestrate operation delegation and operation output forwarding to other peers.

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.

5.2 Basic Idea

We assume that we have the information about the Datasets available on all of the 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.

Based on this algorithm the initial application delegates operations to the other nodes (instances of the same program), where the data is available. Our distributed workflow manager will synchronize the information on these running operation and will label the output data and will add it to the distributed data table.

After finishing operations A and B we will run operation C in either of these nodes, because the required data is partially available on these nodes. Then we have to transfer the rest of the data to one of these nodes to run the operation C which needs both parts simultaneously.

5.2.1 Break and Conquer

5.2.2 Recursive Call

5.2.3 Collectors

5.2.4 Asynchronous Calls

5.2.5 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.

Chapter 6

Prototype

6.1 Architecture

In a traditional approach toward application layering 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. [7] 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.

6.1.1 Overview

6.1.2 Actors

There are two types of actors in our problem domain.

User A user who launches, control and monitor an operation.

Instance Every instance can launch and observe an operation on other instances on other nodes.

6.1.3 Messaging

Publish/Subscribe

Filtering

6.1.4 Coupling

6.1.5 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 ØMQ

The main library that powers our prototype is called ØMQ or ZeroMQ [26]. 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.

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

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 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. [18]

6.7.1 Integration Tests

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

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

Mixing Signals in Greenlets

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

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 [27].

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[26] as underlying transport channel we can use its more advanced security features such as Elliptic curve cryptography[2] based on Curve25519[3] 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 [19]. 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] D. J. Bernstein. *A state-of-the-art Diffie-Hellman function*. 2015. URL: http://cr.yp.to/ecdh.html.

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

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

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

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

[24] 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.

[25] 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.

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

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