University of Victoria
Faculty of Engineering
Coop Workterm Report

Hadoop Distributed File System Propagation Adapter for Nimbus

Department of Physics
University of Victoria
Victoria, BC

Matthew Vliet
V00644304
Computer Engineering
mvliet@uvic.ca

October 31, 2010

In partial fulfilment of the requirements of the
Bachelor of Engineering Degree

Supervisor's Approval: To be completed by Co-op Employer

I approve the release of this report to the University of Victoria for evaluation purposes only.
The report is to be considered (**select one**): ☐ NOT CONFIDENTIAL ☐ CONFIDENTIAL

Signature: _____ Position: _____ Date: _____

Name (print): _____ E-Mail: _____ Fax #: _____

If a report is deemed CONFIDENTIAL, a non-disclosure form signed by an evaluator will be faxed to the employer. The report will be destroyed following evaluation. If the report is NOT CONFIDENTIAL, it will be returned to the student following evaluation.

Contents

List of Figures

Hadoop Distributed File System Propagation Adapter for Nimbus

Matthew Vliet - mvliet@uvic.ca

October 31, 2010

Abstract

Nimbus is an open source toolkit providing Infrastructure as a Service capabilities (IaaS) to a Linux cluster. Nimbus, as with all Infrastructure as a Service software, needs a method of storing virtual machine disk images in a reliable and easily accessible manner. The current bottleneck preventing Nimbus from scaling to very large deployments is the lack of a high performance and reliable method for distributing and storing virtual machine images. The Hadoop Distributed File System (HDFS) is introduced as a possible solution to alleviate the bottleneck and allow Nimbus to scale to very large IaaS deployments. HDFS provides is fault tolerant distributed file system that can easily be adapted to function as an image storage and distribution system for Nimbus.

1 Report Specification

1.1 Audience

The intended audience of this report are the members of the High Energy Physics Research Computing group at the University of Victoria, and future co-op students within the group. Users of the Nimbus toolkit may also be interested as the work directly relates to that software.

1.2 Prerequisites

It is assumed that the readers of this document are familiar with the general concepts of virtualization, grid computing, compute clusters, networking, and file systems. A general knowledge of the Nimbus[1] toolkit will also be helpful.

1.3 Purpose

The contents of this report are intended to explain how the Hadoop Distributed File System(HDFS[2]) can be used alleviate a performance bottleneck observed in the virtual machine image distribution mechanisms currently used by the Nimbus toolkit. By adapting Nimbus to use HDFS as a storage and distribution mechanism, the performance bottleneck can be eliminated. Integration of Nimbus and HDFS is achieved by the implimentation of an adapter allowing Nimbus to store and retrieve virtual machine images from HDFS.

2 Introduction

In the field of cloud computing, one of the primary difficulties often encountered is the efficient distribution and management of large VM image files to nodes within the cloud environment. The most commonly employed solution to the issue of image management and distribution is to use make use of an image repository where all images are stored and managed. The repository acts as a central location to which cloud users can upload their virtual machine images, and from which machines in the cloud can retrieve the images needed to boot virtual machines. The Nimbus[1] toolkit, used by researchers at the High Energy Physics Research

Computing group (HEPRC) at the University of Victoria, is one of the many available toolkits that allows for the creation of Infrastructure-As-A-Service (IaaS) computing environments and employs the concept of a central repository for the storage of disk images.

Currently when Nimbus starts a virtual machine it must first get a copy of the disk image from a remote image repository, this process is referred to as propagation. The default method for the propagation of images from the image repository to a virtual machine manager(VMM) is to use SCP. This means that the all VM disk images will be propagated from a single server where the disk image resides. When there are only a handful of concurrent propagation requests to the single repository server there are no issues, the server load will be relatively low. When there are many tens or hundreds of concurrent propagation requests to the single repository server, the load on the server is too high to deal with in a timely manner. This report describes a method of using a distributed file system to reduce to the load experienced by a single server by spreading the load to the many servers that comprise a HDFS.

2.1 Hadoop Distributed File System

The Hadoop Distributed File Systems main goal is to provide a distributed fault tolerant file system. HDFS achieves its fault tolerance by splitting up files into blocks of a set size and distributing multiple replicas of those blocks across multiple storage nodes called data-nodes. The distribution of blocks of data takes into account factors such as the physical location of data nodes as well as where other related blocks of data are stored. Whenever possible no two replicas of the same block will be stored on the same data-node.

When accessing a file located on a HDFS the client talks to a single server referred to as the name-node. This name-node can be thought of as a namespace lookup as it does not store any of the data blocks, it simply contains a listing of which data-nodes contain which blocks of a file. When the name-node receives a request to access a file the name-node responds to the client not with the data, but with a list block ids and which data-node contains each block. The client then proceeds to request the individual blocks from the many data-nodes where the blocks are stored. The blocks of data are reassembled on the client side to create the requested file. Since no single server is responsible for transmitting the file to the client, the load is distributed across the many data-nodes that contain the blocks of data. This results in faster file transfer times over that of a single file server using a protocol such as SCP or HTTP. The speed increase is especially true when many clients are requesting files from HDFS as the load is distributed to all data-nodes rather then the single file server. A simplified example of a client requesting a file from HDFS is shown in Figure 1. In this example a client attempts to retrieve a single file, which has been broken into nine blocks, from an HDFS where there are one name nodes and four data nodes.

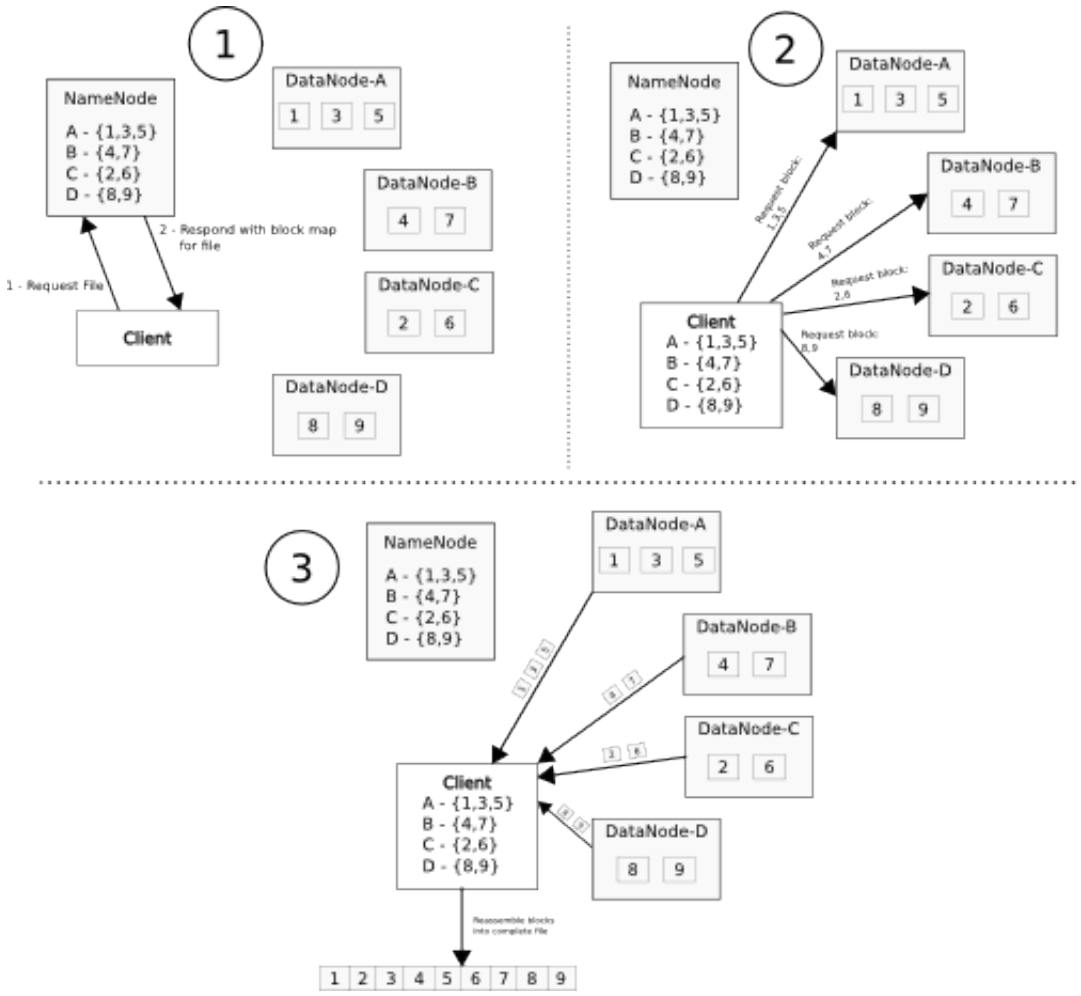

Figure 1: Example request and retrieval of a file from HDFS

To illustrate the improvement that HDFS can provide, the speeds of HDFS, SCP, and HTTP file transfers were compared by by simultaneously transmitting a 500MB file to 11 nodes of a cluster. The results from this test can be seen in Figure 2. The block replication values refers to the number of times that a unique data block is replicated across all data-nodes, a value of 10 would mean there are ten copies of the block scattered across all data-nodes. In this particular experiment, there are ten data-nodes and a single name-node in the test cluster. As can be seen in Figure 2, the fastest transfer times were achieved when using HDFS with a high block replication value.

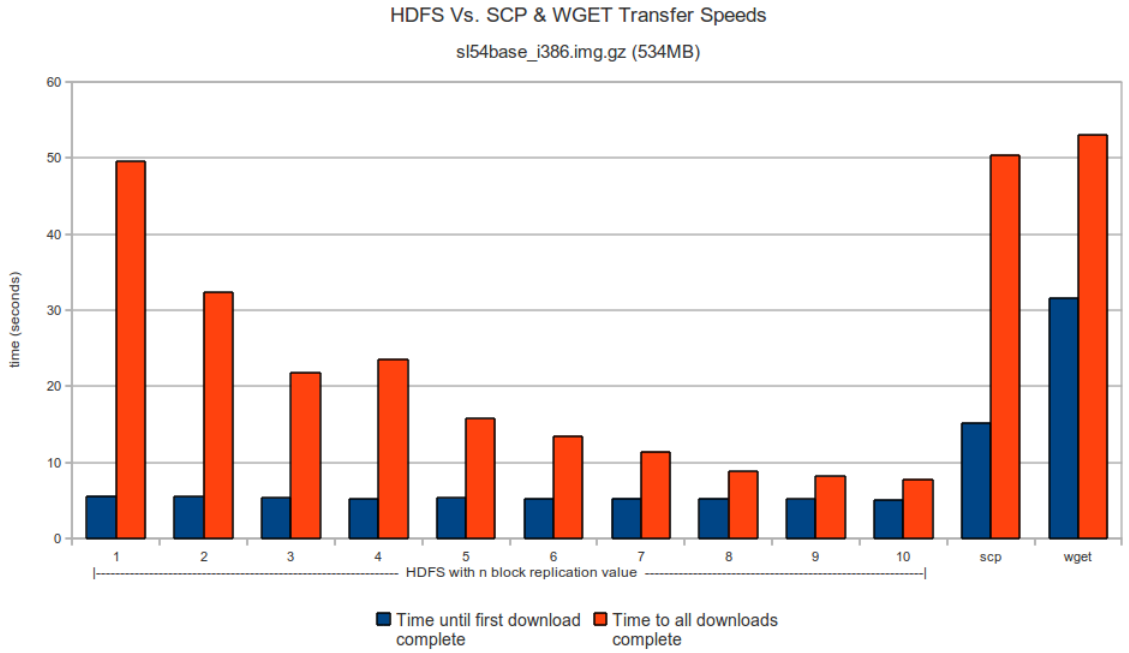

Figure 2: HDFS vs. SCP vs. HTTP file transfer speeds

3 HDFS Propagation Adapter

In order for Nimbus to communicate with HDFS a propagation adapter must be written to translate Nimbus propagation requests to requests for files from HDFS. Several methods of interfacing Nimbus with HDFS were examined, with only one method be viable for use in a propagation adapter for Nimbus. The explored methods included using the native Java hdfs library, a C language library named libhdfs[4], a language independent Thrift[5] gateway server, and finally the `hadoop` command line program supplied by a Hadoop install.

Due to workspace-control being written in Python, the method used to interface with HDFS needs to be accessible from a Python script. This immediately removes the possibility of using the native Java package easily or efficiently. The C library was initially tried, but it was found to take take significant work to write a error free wrapper allowing the library to be called Python. Another downside of the C library is that it is actually just a wrapper around the native Java library. The Thrift gateway server was also turned down as its read and write methods force UTF-8 encoding making it useless for binary encoded data. This left the `hadoop` command line program as the only option for an interface to HDFS.

The usage of the `hadoop` program is quite simple and is easy incorporated into a propagation adapter. Among its many features, the `hadoop` program provides an interface that bridges the gap between a posix system and HDFS. Among the available commands, only `copyToLocal`, `copyFromLocal`, and `test` are needed by the HDFS propagation adapter.

3.1 Propagation

When attempting to propagate an image from an HDFS repository onto a VMM, the propagation adapter first checks the propagation source to see if the image exists. This is done by calling the `hadoop` program and utilizing the `test` command as seen in Figure 3. The `-e` flag informs the program to test if the specified

path exists on the specified HDFS.

```
hadoop fs -fs hdfs://<name-node> -test -e <path_to_image>
```

Figure 3: hadoop source test command

Once the source has been confirmed to exist, the adapter then runs the `copyToLocal` command to transfer a copy of the image from the specified HDFS to a location local to the VMM. The format of this command is seen in Figure 4. Once the file transfer has completed, the image has been propagated.

```
hadoop fs -fs hdfs://<name-node> -copyToLocal <path_to_image> <local_path>
```

Figure 4: hadoop copyToLocal command

3.2 Unpropagation

Unpropagation is a feature supported by Nimbus that allows for a virtual machine image to be saved back to a repository after it has been used to execute a VM. The HDFS propagation adapter supports unpropagation with one restriction. This restriction being that the unpropagation destination must not already exist. In other words, there can not exist a file on the HDFS with the same path and name as requested destination for unpropagation. The naming restriction is imposed as a safeguard to prevent overwriting existing data on the HDFS.

Once the destination is confirmed to not exist, unpropagation to an HDFS repository is accomplished by running the `copyFromLocal` command. This command acts to transfer a file from the local system to a remote HDFS. The structure of this command can be seen in Figure 5

```
hadoop fs -fs hdfs://<name-node> -copyFromLocal <local_path> <remote_destination>
```

Figure 5: hadoop copyFromLocal command

4 Future Work

Future work for the HDFS propagation adapter will mainly consist of maintenance and bug fixes. The adapter is complete, however future work can take place on better integration of HDFS with Nimbus. In the current version of Nimbus HDFS is supported as a propagation scheme, it is not however intimately aware of the HDFS itself. In future versions of Nimbus it will be possible to give the Nimbus service more control over the management of disk images stored on the HDFS.

5 Conclusion

In order to compare the performance benefit of using HDFS verses the previous methods of image propagation a fully working propagation adapter was written to allow Nimbus to communicate with HDFS. This adapter provides the basic functionality needed for Nimbus to retrieve and store images on a HDFS. Although the provided functionality works and is all that is need, the current implementation does not a provide a tight integration with the all of the Nimbus tools. The project is considered a success in that the initial requirements of allowing Nimbus to retrieve and store images on HDFS have been met. There is a large amount of room to further improve the system as mentioned in the Future Work section.

6 Acknowledgements

I would like to thank Dr. Randall Sobie for this work term opportunity. Additional thanks to Ian Gable for guidance on this work term, the Nimbus developers, and all members of the HEPRC group.

7 Glossary

Hadoop A map-reduce based data analysis environment developed by Apache.

HDFS Hadoop Distributed File System. The default file system bundled with a Hadoop installation.

Nimbus EC2 compatible VM management software aimed at the scientific community.

Node A physical computer within a cluster.

SCP Secure Copy Protocol. A secure remote file transfer utility.

UTF-8 A form of Unicode text encoding

VM Virtual Machine

VMM VM Manager. A piece of software that manages the creation and termination of VMs.

References

[1] Nimbus toolkit - http://www.nimbusproject.org/

[2] Hadoop Distributed File System - http://hadoop.apache.org/hdfs/

[3] Hadoop - http://hadoop.apache.org/

[4] libhdfs - http://wiki.apache.org/hadoop/LibHDFS

[5] Apache Thrift - http://incubator.apache.org/thrift

Appendices

A Source Code

A.1 propagate_hdfs.py

```
from commands import getstatusoutput
import os
from time import time
from propagate_adapter import PropagationAdapter
from workspacecontrol.api.exceptions import *


class propadapter(PropagationAdapter):
    """Propagation adapter for HDFS.

    Image file must exist on HDFS to validate propagation
    Image file must not exist on HDFS to validate unpropagation
    """

    def __init__(self, params, common):
        PropagationAdapter.__init__(self, params, common)
        self.hadoop = None                  # Hadoop executable location
        self.parsed_source_url = None       # Source URL when propagating
        self.parsed_dest_url = None         # Destination URL when unpropagating

    def validate(self):
        self.c.log.debug("Validating hdfs propagation adapter")

        self.hadoop = self.p.get_conf_or_none("propagation", "hdfs")
        if not self.hadoop:
            raise InvalidConfig("no path to hadoop")

        # Expand any enviroment variables first
        self.hadoop = os.path.expandvars(self.hadoop)
        if os.path.isabs(self.hadoop):
            if not os.access(self.hadoop, os.F_OK):
                raise InvalidConfig("HDFS resolves to an absolute path, but it does not seem to exist:

            if not os.access(self.hadoop, os.X_OK):
                raise InvalidConfig("HDFS resolves to an absolute path, but it does not seem executable
        else:
            raise InvalidConfig("HDFS contains unknown enviroment varibales: '%s'" % self.hadoop)

        self.c.log.debug("HDFS configured: %s" % self.hadoop)

    def validate_propagate_source(self, imagestr):
        # Validate uri format
        if imagestr[:7] != 'hdfs://':
            raise InvalidInput("Invalid hdfs url, must be of the form hdfs://")
```

```
    # Validate file exists on filesystem
    cmd = self.__generate_hdfs_test_cmd(imagestr)
    try:
        status,output = getstatusoutput(cmd)
    except:
        errmsg = "HDFS validation - unknown error when checking that file exists."
        self.c.log.error(errmsg)
        raise UnexpectedError(errmsg)
    # 0 returned if file exists
    # 1 returned if file does not exist
    if status:
        errmsg = "HDFS validation - file does not exist on hdfs."
        self.c.log.error(errmsg)
        raise InvalidInput(errmsg)

def validate_unpropagate_target(self, imagestr):
    # Validate uri format
    if imagestr[:7] != 'hdfs://':
        raise InvalidInput("Invalid hdfs url, must be of the form hdfs://")

    # Validate file does not exists on filesystem already
    cmd = self.__generate_hdfs_test_cmd(imagestr)
    try:
        status,output = getstatusoutput(cmd)
    except:
        errmsg = "HDFS validation - unknown error when checking that directory exists."
        self.c.log.error(errmsg)
        raise UnexpectedError(errmsg)
    # 0 returned if file exists
    # 1 returned if file does not exist
    if not status:
        errmsg = "HDFS validation - File already exists at destination: %s" % imagestr
        self.c.log.error(errmsg)
        raise InvalidInput(errmsg)

def propagate(self, remote_source, local_absolute_target):
    self.c.log.info("HDFS propagation - remote source: %s" % remote_source)
    self.c.log.info("HDFS propagation - local target: %s" % local_absolute_target)

    cmd = self.__generate_hdfs_pull_cmd(remote_source, local_absolute_target)
    self.c.log.info("Running HDFS command: %s" % cmd)
    transfer_time = -time()
    try:
        status,output = getstatusoutput(cmd)
    except:
        errmsg = "HDFS propagation - unknown error.  Propagation failed"
        self.c.log.error(errmsg)
        raise UnexpectedError(errmsg)

    if status:
        errmsg = "problem running command: '%s' ::: return code" % cmd
```

```
        errmsg += ": %d ::: output:\n%s" % (status, output)
        self.c.log.error(errmsg)
        raise UnexpectedError(errmsg)
    else:
        transfer_time += time()
        self.c.log.info("Transfer complete:  %fs" % round(transfer_time))

def unpropagate(self, local_absolute_source, remote_target):
    self.c.log.info("HDFS unpropagation - local target: %s" % local_absolute_target)
    self.c.log.info("HDFS unpropagation - remote source: %s" % remote_source)

    cmd = self.__generate_hdfs_push_cmd(remote_source, local_absolute_target)
    self.c.log.info("Running HDFS command: %s" % cmd)
    transfer_time = -time()
    try:
        status,output = getstatusoutput(cmd)
    except:
        errmsg = "HDFS unpropagation - unknown error.  Unpropagation failed"
        self.c.log.error(errmsg)
        raise UnexpectedError(errmsg)

    if status:
        errmsg = "problem running command: '%s' ::: return code" % cmd
        errmsg += ": %d ::: output:\n%s" % (status, output)
        self.c.log.error(errmsg)
        raise UnexpectedError(errmsg)
    else:
        transfer_time += time()
        self.c.log.info("Unpropagation transfer complete:  %fs" % round(transfer_time))

#--------------------------------------------------------------------------
# Private helper functions

def __generate_hdfs_pull_cmd(self, remote_target, local_absolute_target):
    # Generate command in the form of:
    # /path/to/hadoop/bin/hadoop fs -fs <files system uri> -copyToLocal <src> <localdst>
    if not self.parsed_source_url:
        self.parsed_source_url = self.__parse_url(remote_target)

    ops = [self.hadoop, "fs",
           "-fs", self.parsed_source_url[0]+'://'+self.parsed_source_url[1],
           "-copyToLocal", self.parsed_source_url[2], local_absolute_target]
    cmd = " ".join(ops)
    return cmd

def __generate_hdfs_push_cmd(self, remote_target, local_absolute_target):
    # Generate command in the form of:
    # /path/to/hadoop/bin/hadoop fs -fs <files system uri> -copyFromLocal <local> <dst>
    if not self.parsed_dest_url:
        self.parsed_dest_url = self.__parse_url(remote_target)
```

```
    ops = [self.hadoop, "fs",
           "-fs", self.parsed_dest_url[0]+'://'+self.parsed_dest_url[1],
           "-copyFromLocal", local_absolute_target ,self.parsed_dest_url[2]]
    cmd = " ".join(ops)
    return cmd

def __generate_hdfs_test_cmd(self, imagestr):
    # Generate command in the form of:
    # /path/to/hadoop/bin/hadoop dfs -fs <file system uri> -test -e <path>
    if not self.parsed_source_url:
        self.parsed_source_url = self.__parse_url(imagestr)

    ops = [self.hadoop, "fs",
           "-fs", self.parsed_source_url[0]+'://'+self.parsed_source_url[1],
           "-test", "-e", self.parsed_source_url[2]]
    cmd = " ".join(ops)
    return cmd

def __parse_url(self, url):
    # Since Python2.4 urlparse library doesn't recognize the hdfs scheme,
    # we need to parse the url by hand.
    url = url.split('://')
    if len(url) != 2:
        raise InvalidInput("url not of the form <scheme>://<netloc>/<path>")
    scheme = url[0]
    netloc, path = url[1].split('/', 1)
    # Add leading / back in since it was used to partition netloc from path
    path = '/'+path
    return (scheme, netloc, path)
```