AN ABSTRACT OF THE PROJECT OF

Shujin Wu for the degree of Master of Science in Computer Science presented on February 17, 2017.

Title: ScaleIO Storage Driver: A Template Driver for CoprHD

Abstract approved: _____

Eric Walkingshaw

CoprHD is an open source software-defined storage and API platform which creates an abstraction layer over multi-vendor heterogeneous storage systems. It offers the ability to discover, pool and automate the management of the storage ecosystem with the help of storage drivers establishing connections between CoprHD and storage systems. On the demand of attracting more attentions from third-party storage companies, CoprHD community proposed a southbound driver SDK to simplify the process of developing a storage driver for CoprHD. ScaleIO storage driver, being the first one based on this southbound SDK, is implemented by us with Intel and EMC to serve the purposes to verify the southbound SDK and explore an effective way for the third-party driver development. This ScaleIO storage driver also acts as a template driver for the CoprHD community.

ScaleIO Storage Driver: A Template Driver for CoprHD

by

Shujin Wu

A PROJECT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented February 17, 2017
Commencement June 2017

Master of Science project of Shujin Wu presented on February 17, 2017.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my project will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my project to any reader upon request.

Shujin Wu, Author

ACKNOWLEDGEMENTS

This report would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Eric Walkingshaw for the continuous support of my study and research during my M.S. program, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this report. I could not imagine having a better advisor and mentor for my M.S. program.

Besides my advisor, I would like to thank the rest of my graduate committee: Dr. Rakesh Bobba and Dr. Amir Nayyeri, for their constant support, encouragement and insightful comments.

My sincere thanks also go to Shayne Huddleston, Director of IT Infrastructure Services OSU, for not only providing me an opportunity to work on the project but also encouraging and trusting me to lead the development team. I have learned a lot from this job with him.

I would like to thank the experts involved in this Project: Evgeny Roytman, Jason Davidson from EMC Corporation (Dell-EMC) and Anjaneya Chagam (Reddy) Principal Engineer at Intel for his guidance and supports towards our team. Without their passionate participation and input, this project would not have been completed. I would also like to acknowledge my team members Prathamesh Patkar, Varun Rajgopal and Taylor Cuilty for being supportive and grateful.

I thank EECS Department OSU and Nicole Thompson, for consistently providing me financial support in the form of GTA throughout my project and support facility.

Last but not the least, I would like to thank my family for helping me spiritually throughout my life. This accomplishment would not have been possible without you. Thank you.

Shujin Wu

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

There is no doubt that we are in an era of data explosion. In most current data centers, storage accounts for 40% of the budget [3]. There is an undeniable demand from data center suppliers to reduce the cost of managing and scaling the diverse storage solutions. In the storage industry, 60% of companies are committed to storage defined storage (SDS) approach [6], which brings many benefits to the industry, including scalability and simplified storage management interfaces.

EMC CoprHD is one of the leading SDS solutions in the industry. It creates an abstraction layer over multi-vendor heterogeneous storage systems to discover, pool and automate the management of the storage ecosystem [4]. However, CoprHD doesn't generically support all possible storage systems, and its original storage driver solution requires solid knowledge of CoprHD internals, which is a time-consuming learning process for third-party developers. To simplify and accelerate the driver development, EMC proposed a new CoprHD Southbound API. In this report, I describe the design and implementation of a ScaleIO storage driver based on this new southbound API. This driver mainly serves two purposes: one is to testify the design and implementation of the southbound SDK, the other is to act as an exemplary driver for third-party companies to simplify their driver development process.

This report will serve as a tutorial of how to develop a storage driver for Co-

prHD. And it's organized as follows: Chapter 1 introduces the background and motivation of this project and gives an overview of our approach. How we applied agile development process to our project is presented in Chapter 2. Requirement analysis for the ScaleIO storage driver is discussed in Chapter 3, while Chapter 4 explains the architectural design model of the ScaleIO driver and the detailed design models of my components: REST client factory, snapshot and consistency group operations. Chapter 5 introduces the development environment and the reusable modules. Chapter 6 explains the motivation and implementation principles of the automated deployment solution, and Chapter 7 discusses the tools and technologies used for the project. Finally, Chapter 8 concludes this project.

1.1 Background

1.1.1 Software-defined Storage

It's true to say that hardware has largely defined storage systems and delivered storage features for over the past 20 years. However, the drawbacks of this hardware-centric approach are also evident. First of all, there must be another system rolled in alongside once a system reaches its capacity, or customers would have to go through a painful upgrade and migration. Likewise, another set of systems and processes are also needed to support backup and disaster recovery [14]. What's worse, silos are isolated from each other, and each of these silos is optimized to run a particular workload, which complicates the data center management. Further-

more, the lack of visibility into data makes storage systems even more complicated to manage [10]. Simply put, as data volumes grow, storage management becomes not only complex and inefficient, but also expensive.

Software-defined storage (SDS) differentiates from traditional storage in how storage is managed and deployed. SDS separates the storage hardware from the software that manages the storage infrastructure, and it provides a storage service interface which allows the data owners to describe requirements on both the data and its desired service levels [2].

Generally speaking, SDS must include the following functionalities [2]:

- **Automation:** SDS should simplify the management of storage infrastructure and reduce the cost of storage infrastructure maintenance.
- **Standard Interfaces:** SDS should define a set of APIs for management, provisioning, and maintenance of storage devices and services.
- **Virtualized Data Paths:** Virtualized data paths are Block, File and Object interfaces to which applications can write data.
- **Scalability:** Software-defined storage could seamlessly add new storage devices into the storage infrastructure without maintaining another set of systems and processes.
- **Transparency:** Storage consumers should be able to monitor and manage their storage consumption against available resources and costs.

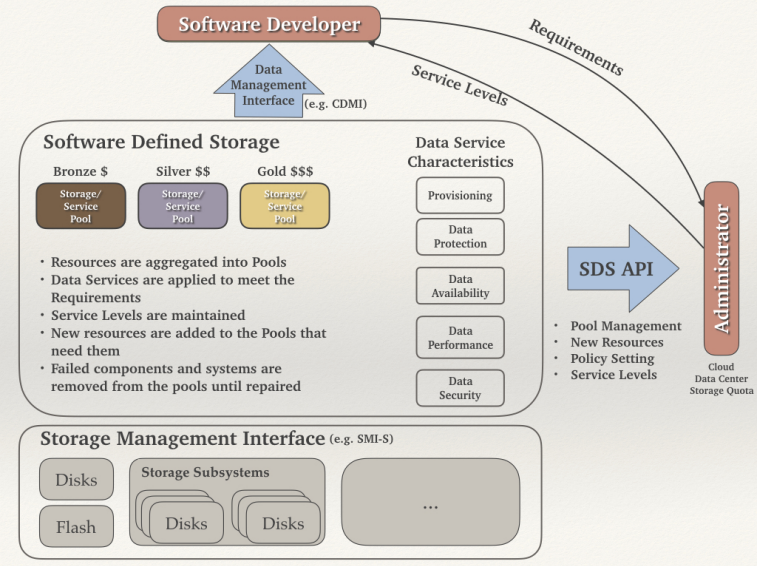

Figure 1.1: Software Defined Storage [2]

Figure 1.1 illustrates the concepts behind software-defined storage. Software developers express their requirements for the data they own via a data management interface, such as CDMI (Cloud Data Management Interface). Their desired service levels can be delivered through a combination of the SDS solution and the administrators. Currently, SDS aggregates resources into storage pools. To meet the requirements of service levels, SDS maintains a set of data service characteristics that the storage pools can apply. What's more, SDS utilizes a standard storage management interface, such as SMI-S (Storage Management Initiative Specification), to automate management of storage resources and discover their capability. Administrators are also empowered to manage pools, resources and service policies via abstract interfaces [2].

1.1.2 CoprHD and its Storage Drivers

CoprHD is a software platform for building a storage cloud. It does not provide storage on its own, but holds an inventory of all storage devices in the data center and understands their connectivity [9]. As shown in Figure 1.2, in the northbound interface, CoprHD can be integrated with traditional, cloud and Cloud Native Computing stacks. Within CoprHD, self-service provisioning is via REST APIs and service catalogs. CoprHD as an SDS solution discovers heterogeneous storage systems and classifies them into virtual storage arrays and pools with storage policies.

The capability to discover and operate on a storage system is carried out through the CoprHD storage driver which leverages the southbound API. Figure 1.3 shows the legacy architecture of the CoprHD storage driver. Currently, all of the CoprHD drivers are based on this design. This version of CoprHD storage driver is tightly coupled with CoprHD infrastructure. The driver developers have to know how to work with CoprHD database, zookeeper, task completers and workflows to develop a storage driver for CoprHD. Thus, this driver model works for in-house driver development where the developers have a solid understanding of CoprHD internals but meanwhile is hardly suitable for third party development.

1.2 Overview of Our Approach

To simplify the process of adding support for new storage systems, EMC proposed a new southbound driver SDK, through which, driver implementation does not have

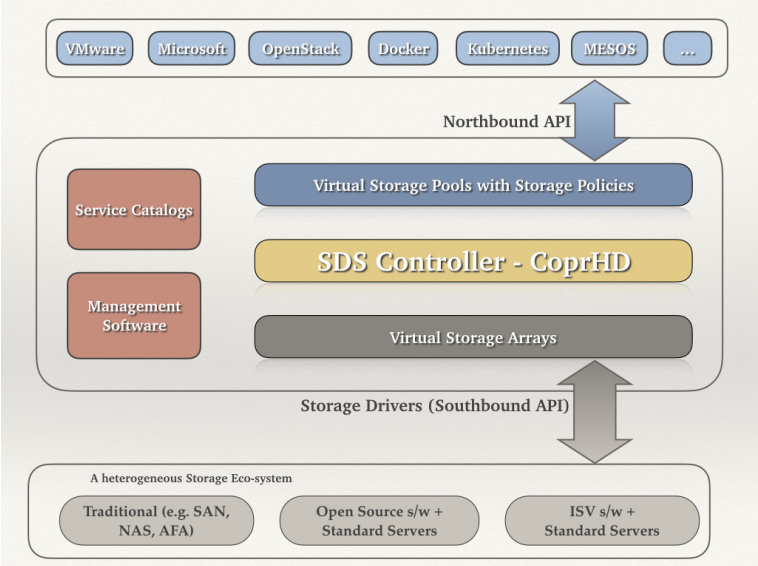

Figure 1.2: CoprHD Eco-system

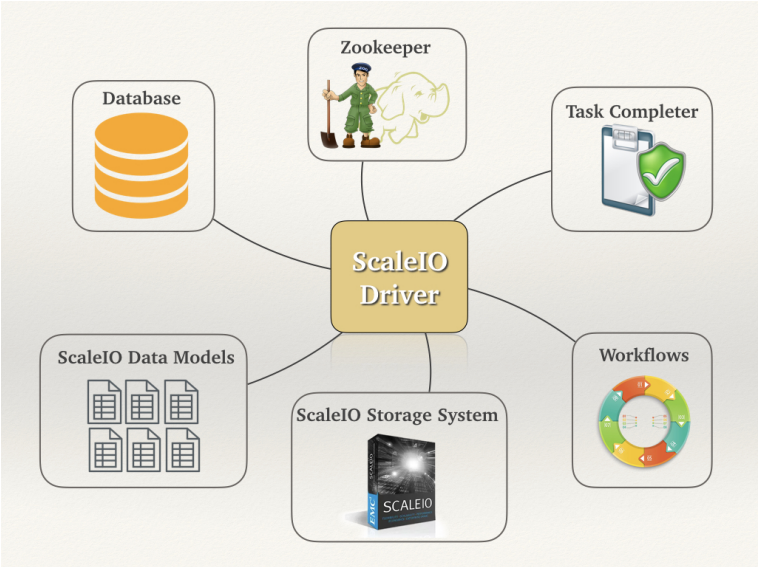

Figure 1.3: CoprHD Legacy Driver Architecture

any dependency on CoprHD infrastructure or its persistent data model classes. Our ScaleIO storage driver is based on this southbound SDK. The idea is to leverage ScaleIO REST API to implement the methods that defined in the SDK. Overall, our team follows an agile process and performs Test Driven Development. The driver is verified by unit tests and integration tests, and I also created an automated deployment solution using Ansible to ease the deployment and testing processes.

1.3 Context within the Project

CoprHD has become open source for contribution since July 2015, and it is under active development since then. By its "x-wing" release [13], CoprHD makes enhancement in many aspects, such as usability and data protection. The following sections present the contributions related to the storage driver development.

1.3.1 Other Drivers

CoprHD is actively working on integration with cloud and Cloud-Native stacks through various drivers. Here are some major ones:

- **CoprHD driver for OpenStack Integration:** Previously in the OpenStack environment, CoprHD can only be used as a storage system managed by Cinder storage controller through a Cinder driver. Now a new driver solution is proposed to position CoprHD as an alternative storage controller for OpenStack customers [5].

- **CoprHD driver for Ceph Integration:** Ceph is a modern distributed object store and file system in the storage industry. Developers are working on a Ceph driver to enable end users to handle Ceph volumes via CoprHD.

- **CorpHD driver for Flocker Integration:** Flocker is an open source CDVM (Container Data Volume Manager) for Dockerized applications. The CoprHD Flocker driver empowers CoprHD to deliver persistent storage for Docker containers via Flocker [8].

1.3.2 Cornerstone of the ScaleIO Driver

- **Southbound Driver SDK Development:** The ScaleIO storage driver is dependent on southbound driver SDK. And since southbound SDK development was ongoing simultaneously with the ScaleIO driver, any defect found in SDK will block the corresponding development of ScaleIO driver component. The southbound SDK was targeted in CoprHD 2016 Q2 Yoda release [13].

- **Southbound SDK Enhancements:** This project aimed at plugging storage drivers into CoprHD via the user interface (UI), which is the ideal solution for deployment. Our Ansible solution is an alternative solution for it.

- **CoprHD and ScaleIO Test Environment:** Curt Bruns from Intel created a Vagrant solution to set up a virtual environment for three ScaleIO VMs and one CoprHD VM [1]. Our team leveraged his solution to set up three ScaleIO

VMs as our testing environment. Besides, CoprHD community published multiple approaches to deploying CoprHD, which is the foundation of my Ansible solution.

1.4 Contributions

This ScaleIO driver is implemented by a team of four. As the team leader, I led the meetings to determine and track the progress of each development goal, assuring on-time delivery of the driver features. I also developed a *ScaleIORestClient* module and a snapshot operation module for the ScaleIO storage driver. Regarding deployment, I created an Ansible solution for CoprHD and its storage drivers. Meanwhile, I would like to acknowledge my team members for their achievements. Varun Rajgopal implemented the discovery functionality, Taylor Cuilty developed the standard volume operations, and the clone operation module was implemented by Prathamesh Patkar.

This ScaleIO driver project has served the following purposes. It revealed the defects of the southbound SDK which can be fixed before other third-party developers started developing their CoprHD drivers. And as an exemplary driver, it simplifies the development process for third-party companies.

Chapter 2: Development Process

2.1 Test Driven Development

Our development process follows Test Driven Development (TDD) and is also associated with agile methods. Namely, our development of a feature consists of several iterations, where the test cases are written before the feature is deployed. The development process usually refers to following steps: to identify the initial requirements; to write unit tests that fail before writing any functional code; to write functional code until all unit tests pass; if necessary, to refactor the code or the tests to assure all the updated requirements are met. Additionally, to ensure that the addition of a new feature or refactoring of the code doesn't break the features already delivered, the tests are run as part of a regular process for building the driver.

One of the benefits of the approach is it allows us to expose issues regarding requirements in early stage since test cases could help define the scope of this driver project. Also, we sent the test plan to the community for review before we start writing code, and the feedback confirmed that our test cases covered all the required functionalities. TTD process also reduced the number of bugs since it assures that the code, for which tests were deployed, works [16].

2.2 Team Collaboration

As I mentioned earlier, one of the missions of the ScaleIO storage driver is to evaluate the design of the southbound SDK. Our team started with the first version of the SDK, while EMC southbound SDK team is working on SDK development as well as the driver support layer. As shown in 2.1, any update on southbound SDK from the EMC team will trigger an iteration of the driver development. Likewise, any issue we found in the SDK will be reported to the SDK team, and the corresponding feature development of the driver will be blocked until they fix the issue.

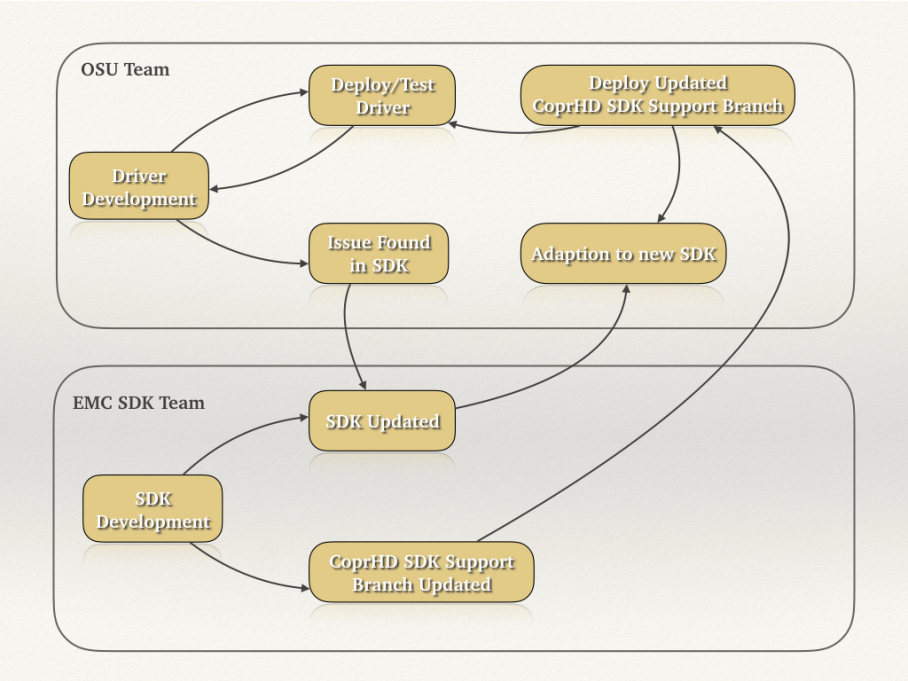

Figure 2.1: Development Cycle

Further, driver integration test involves more collaborations. Many issues can

only be exposed during this period. Any modification made in the adaption of generic CoprHD storage driver API to the specific operations of ScaleIO array will result in the redeployment of SDK, ScaleIO driver and even the CoprHD instance. This repeating process is also the primary motivation for the Ansible automated deployment solution, which I will discuss in the later chapter.

Chapter 3: Analysis and Requirements

The goal of requirement analysis is to understand what mission a ScaleIO driver attempts to accomplish and what approach is adopted to reach this aim. You will find the high-level system architecture of our approach to ScaleIO driver in section 3.1 and the components of the southbound SDK in section 3.2. Section 3.3 explains the functionality that we have implemented in the ScaleIO driver.

3.1 Top Level System Architecture

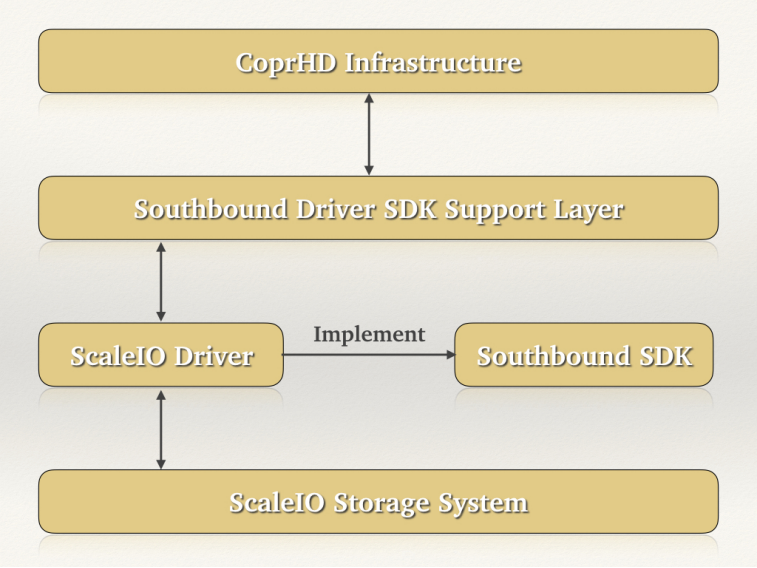

Figure 3.1: Top Level System Architecture

As shown in figure 3.1, our approach decouples the storage driver from the

CoprHD infrastructure, and all the driver calls will be invoked from the southbound SDK support layer. To implement a CoprHD driver, what we need is to implement the interfaces that are defined in the southbound SDK and utilize the ScaleIO Rest API to interact with the ScaleIO storage system to perform the required operations.

3.2 Components of Southbound SDK

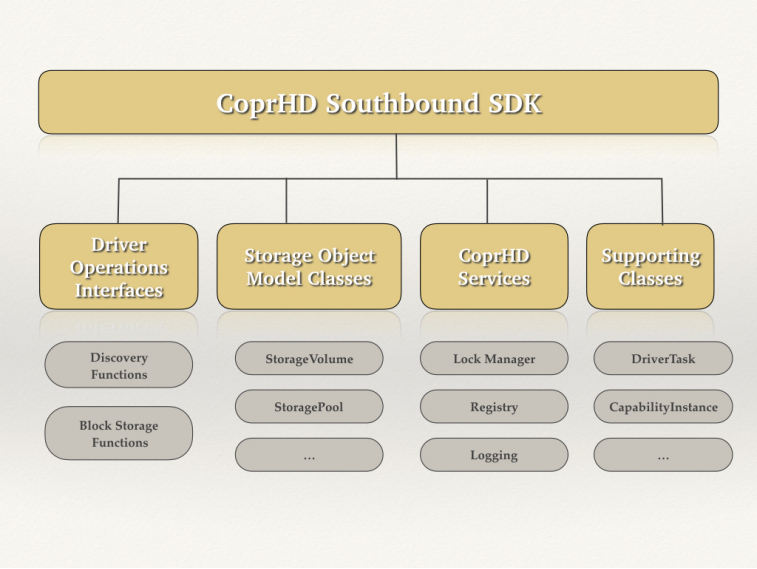

Figure 3.2: Southbound SDK Components

Our driver development is tightly dependent on the SDK. The CoprHD community releases the draft version of the southbound SDK jar through the wiki page, which as shown in figure 3.2, includes the following artifacts [15]:

- Set of interfaces which the drivers have to implement
- Set of object model classes which are utilized in interface methods

- Interfaces for CoprHD services available to storage drivers
- Supporting classes
- Public libraries used by the SDK
- Abstract base class for the storage drivers

3.3 ScaleIO Driver Functionality

The ScaleIO driver supports storage system discovery and block storage operations. Discovery of storage systems refers to the actions to register ScaleIO storage systems into CoprHD, after which CoprHD is capable of managing these registered storage systems. However, a ScaleIO storage system does not necessarily register as one storage system in CoprHD. Instead, each protection domain in ScaleIO can be viewed as a storage system. Moreover, once discovery operation is completed, the information of protection domain together with the corresponding pools and ports are exposed to users via the CoprHD UI. Afterward, CoprHD will perform discovery periodically to check the connectivity between CoprHD and its storage systems. Note that during our driver development, the CoprHD UI does not support discovery operations from the SDK driver, so discovery request made through CoprHD command line tool. Eventually, users will be able to give system credentials through CoprHD UI to register a storage system.

Block storage operations contain but are not limited to volume, clone, mirror and snapshot operations, and they are not independent. One example is volume

CoprHD Southbound SDK Snapshot Operation Interfaces	Functionality	ScaleIO Storage System
createVolumeSnapshot()	Create volume snapshots.	Supported
restoreSnapshot()	Restore volume to snapshot state.	Not Supported
deleteVolumeSnapshot()	Delete snapshots.	Supported
createConsistencyGroupSnapshot()	Create snapshot of consistency group.	Supported
deleteConsistencyGroupSnapshot()	Delete consistency group snapshot.	Supported
createConsistencyGroup()	Create block consistency group. (Without volumes)	Not Supported
deleteConsistencyGroup()	Delete block consistency group.	Not Supported

Table 3.1: Snapshot Operations Requirement Analysis

creation must be executed before any clone, mirror or snapshot operation. To get rid of the shackles of execution orders, we employ the ScaleIO command line to perform necessary block operations so that the implementation of the operations with higher priority do not block the ones with lower priority. The block operation request flow starts from the CoprHD UI and is sent to the SDK driver through southbound SDK support layer. Unlike discovery operations, a block operation might be invoked by multiple users simultaneously. Therefore, our driver must be capable of handling concurrent requests, which Chapter 4 will cover.

Although we have to implement all the interfaces that are defined in the SDK, not all of the functionality is supported in ScaleIO storage system. For example, ScaleIO storage system does not support mirror operations. Table 3.1 is an example of how we examine the functionality of ScaleIO towards the southbound SDK.

Chapter 4: Design

4.1 The Architectural Design Model for the Driver

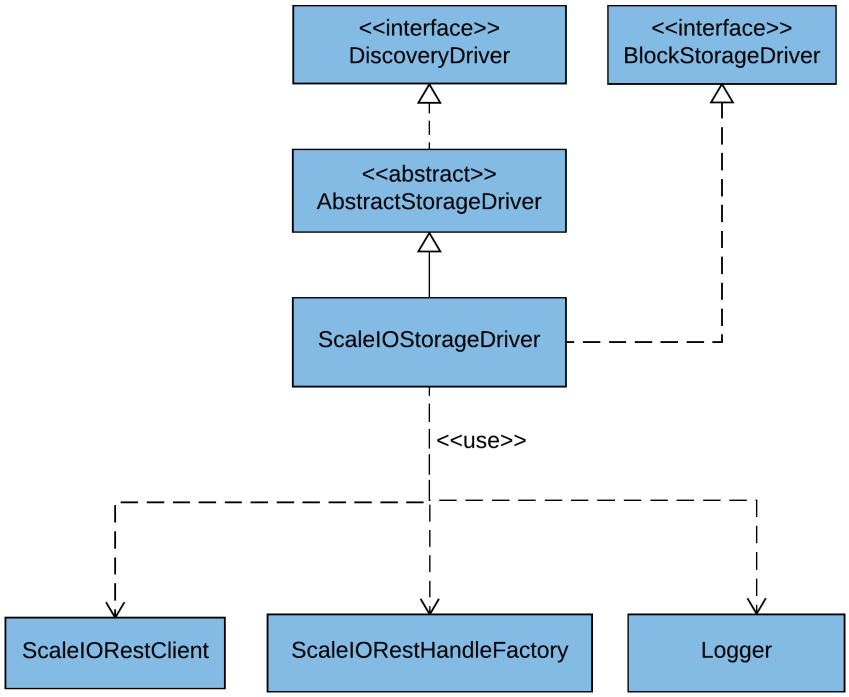

Figure 4.1: Driver Class Diagram

A CoprHD storage driver consists of two components: storage system discovery and block storage operations. As shown in figure 4.1, *ScaleIOStorageDriver* class extends *AbstractStorageDriver* class, and it has to implement all the methods defined in *DiscoveryDriver* and *BlockStorageDriver* interfaces. *AbstractStorageDriver*

class also reserves a registry service for the driver to store necessary information in CoprHD, such as system credentials. In addition, the ScaleIO storage driver relies on ScaleIO REST API to perform the storage operations, so a *ScaleIORestHandleFactory* class is created to obtain a *ScaleIORestClient* instance for each request. The ScaleIO REST component will be discussed later. Further, a public logger library is also shipped in the southbound SDK jar to log the status of each key step of a driver operation for both debugging and maintenance purposes.

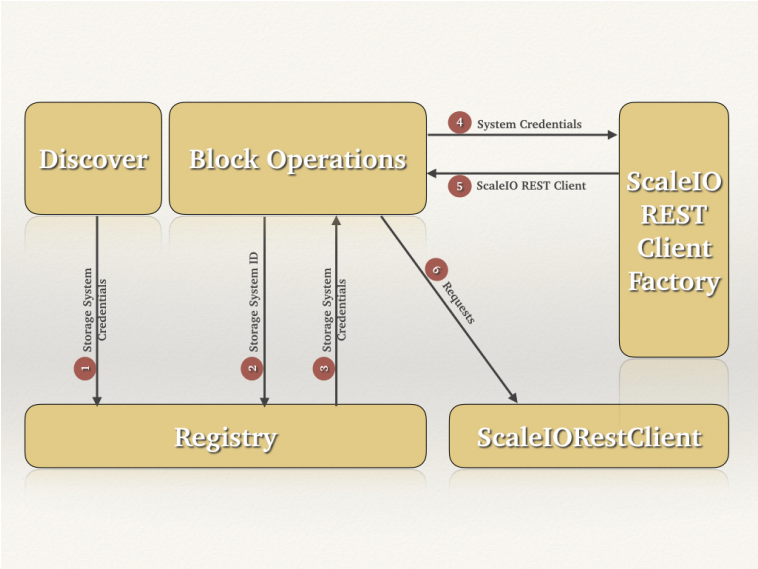

Figure 4.2: Driver Work Flow

Figure 4.2 demonstrates a workflow of ScaleIO driver:

1 Driver receives a request to discover a ScaleIO storage system. Protection domains are registered as storage systems in CoprHD, and the system credentials are stored in the registry service.

2 On receiving a request of a block operation, the driver sends another request to the registry service to retrieve the system credentials.

3 The registry service sends system credentials back.

4 The driver calls *ScaleIORestHandleFactory* to obtain a *ScaleIORestClient* instance based on system credentials received.

5 *ScaleIORestHandleFactory* returns a corresponding *ScaleIORestClient* instance.

6 The block operation requests are handled by the *ScaleIORestClient* instance.

4.2 The Detailed Design Model

4.2.1 ScaleIORestClient Module

The ScaleIORestClient component is the most important module for the ScaleIO storage driver. It turns storage requests into REST-ready and delivers them to the ScaleIO storage system. This module is also a use of the singleton design pattern. We maintain a *ConcurrentHashMap* variable to store a REST client instance for each storage system, and the singleton design pattern guarantees that only one *ConcurrentHashMap* variable is in the application runtime. Figure 4.3 illustrates the steps to obtain a *ScaleIORestClient* instance.

ScaleIORestClient module also implements the abstract factory design pattern. As shown in Figure 4.4, *RestClientFactory* is an abstract factory class that defines interfaces to create REST clients. CoprHD could have any number of de-

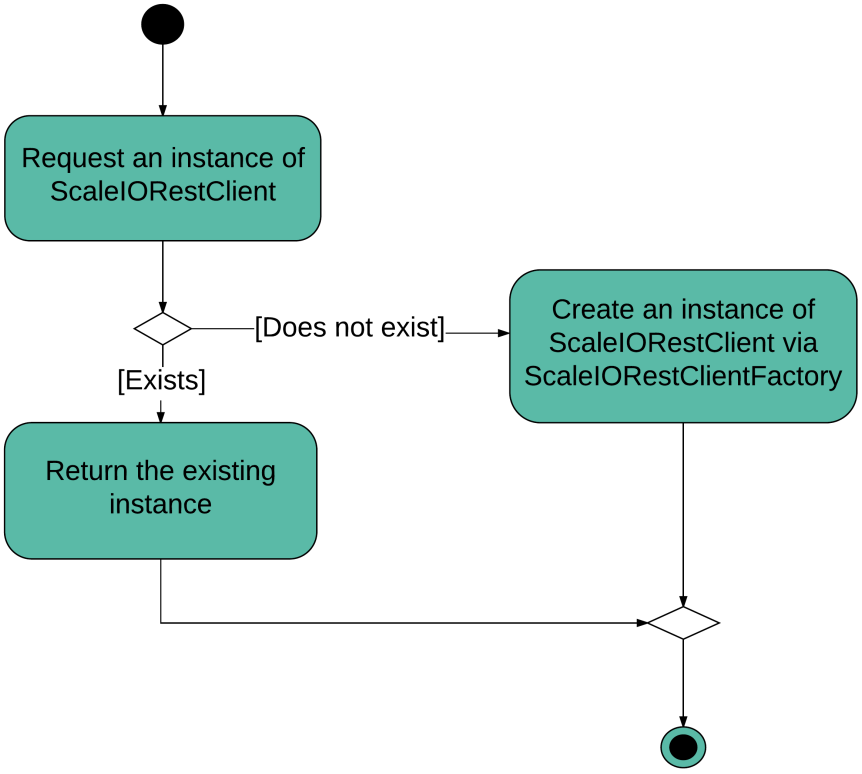

Figure 4.3: Activity Diagram: ScaleIORestClient

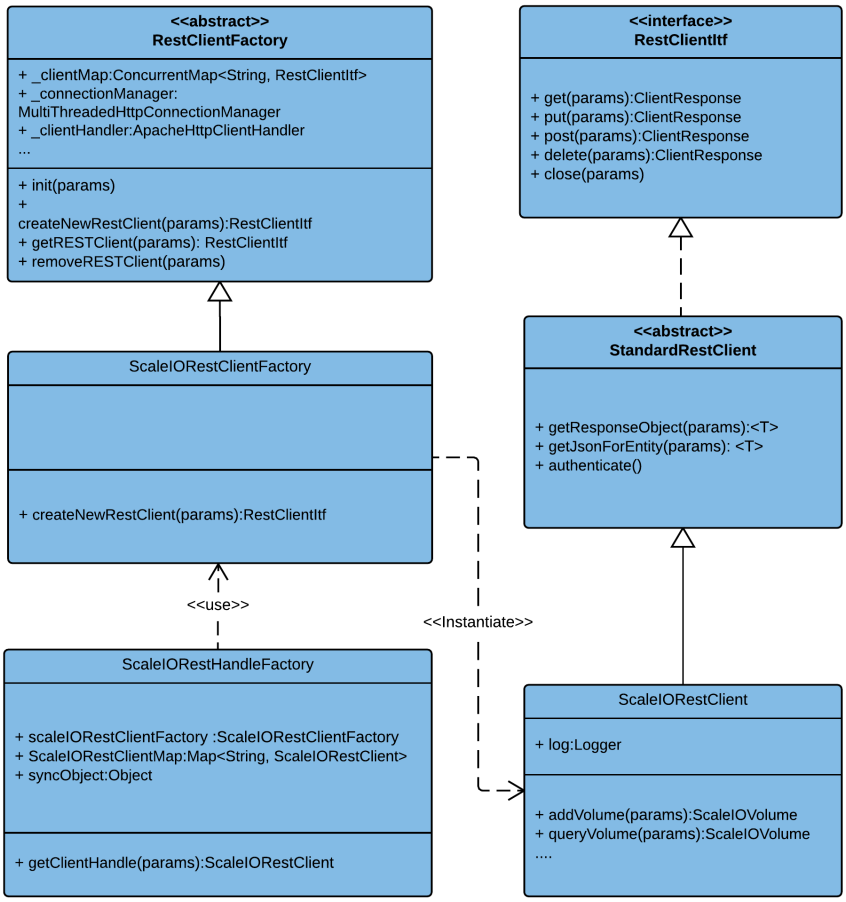

Figure 4.4: ScaleIORestClient Module Class Diagram

rived concrete versions of the *RestClientFactory* like *ScaleIORestClientFactory* and *IsilonRestClientFactory*, each with a different implementation of *createNewRestClient()* method that would create a corresponding instance like *ScaleIORestClient* or *IsilonRestClient*. Both *ScaleIORestClient* and *IsilonRestClient* are derived from an abstract class - *StandardRestClient*. CoprHD does not care which concrete REST client instance it gets from the factories, since it only uses the generic interface - *RestClientItf*.

4.2.2 Snapshot and Consistency Group Operations

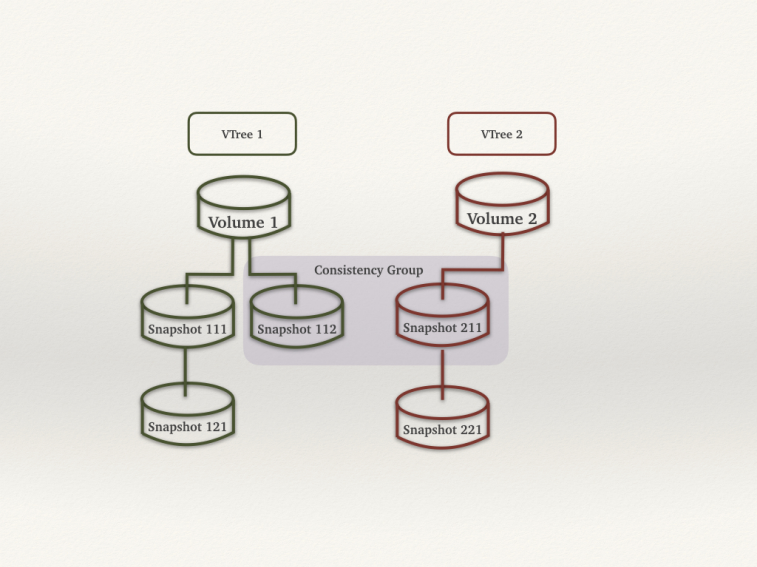

Figure 4.5: Snapshots and Consistency Groups in ScaleIO [7]

ScaleIO offers the ability to snapshot a single volume at a time, and the structure related to all the snapshots generated from one volume is referred to as a

VTree (short for Volume Tree). It's a tree spanning from the source volume as the root, and the snapshots are presented as the children of the parent volumes [7]. Note that it is allowed to take a snapshot of a snapshot volume. In figure 4.5, Snapshot 111 and Snapshot 112 are snapshots of Volume 1. Snapshot 121 is a snapshot of Snapshot 111. Together, Volume 1, Snapshot 111, Snapshot 112 and Snapshot 121 are the VTree of Volume 1. Users can remove the entire VTree or snapshots rooted at the same volume with one command.

Further, a snapshot is a new unmapped volume once it's created in the system. Namely, snapshots may be manipulated in the same manner as any other volume by the ScaleIO storage system [7]. However, unlike standard volumes, snapshots are thin-provisioned, which means that the full copy of the data is not copied over upon creation, but when it is mapped.

All snapshots taken together automatically form a consistency group in the ScaleIO storage system, and they are consistent in the sense of their creation time, which helps the situation where multiple VMs need consistent copies of multiple volumes but the source application may not able to be quiesced at the time. However, ScaleIO does not prevent you from deleting one snapshot in a consistency group. In figure 4.5, Snapshot 112 and Snapshot 211 compose a consistency group if they are taken together.

In CoprHD, before the user fires off a request to create consistency group snapshots, a consistency group which contains several volumes should exist. However, ScaleIO does not allow creating a consistency group explicitly. To qualify CoprHD for ScaleIO consistency group operations, we create a dummy empty consistency

group as shown in figure 4.6. After that, users can add volumes in the group and snapshot-ing the entire consistency group. The idea is to store the dummy consistency group identifiers to the registry service and map them to their corresponding consistency group ID once the consistency groups are formed in ScaleIO.

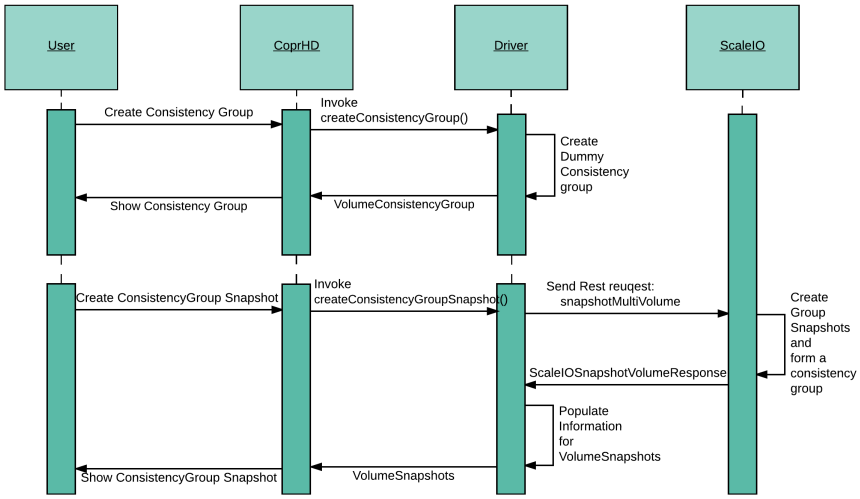

Figure 4.6: Sequence Diagram: Create consistency Group vs Create consistency Group Snapshots

Our team also agrees on a general workflow for the ScaleIO block storage operations. Figure 4.7 demonstrates the workflow for snapshot operations. At the beginning of each operation, we log the location and the starting time of the request, and then prepare and send the request via ScaleIO REST client. Once it gets the response, we will populate the information to involved entities that are defined in SDK model classes. If any exception occurs, it will be thrown and handled by the SDK support layer.

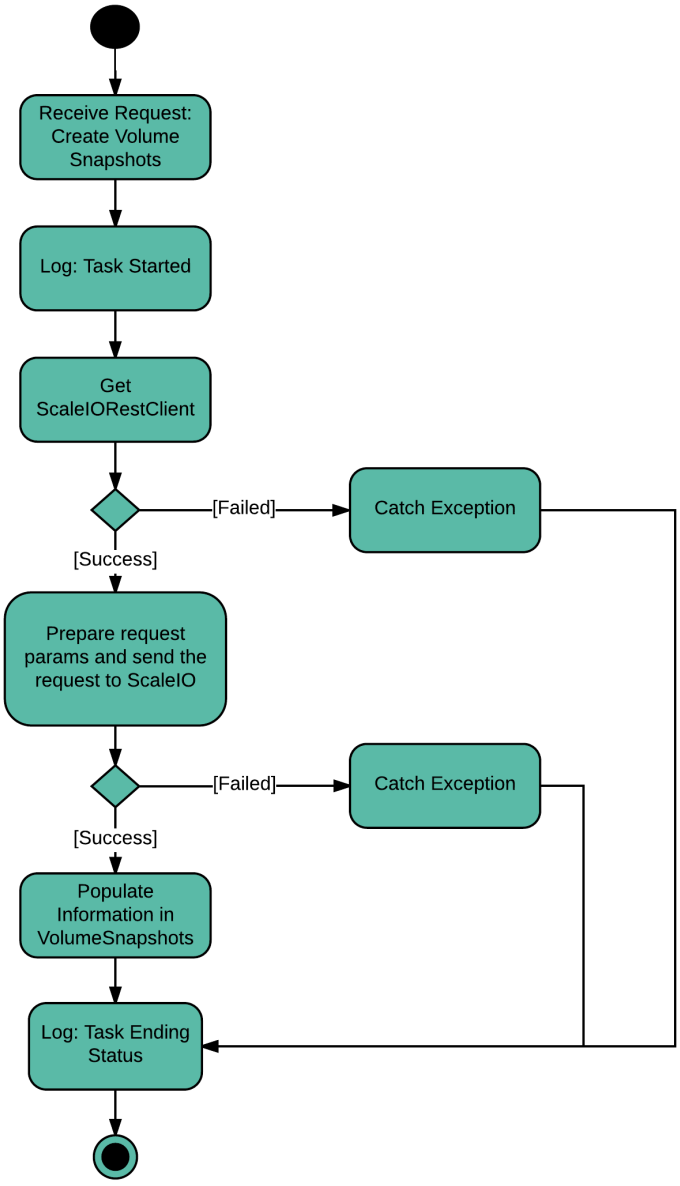

Figure 4.7: Activity Diagram: Create snapshots

Chapter 5: Implementation

5.1 Development Environment

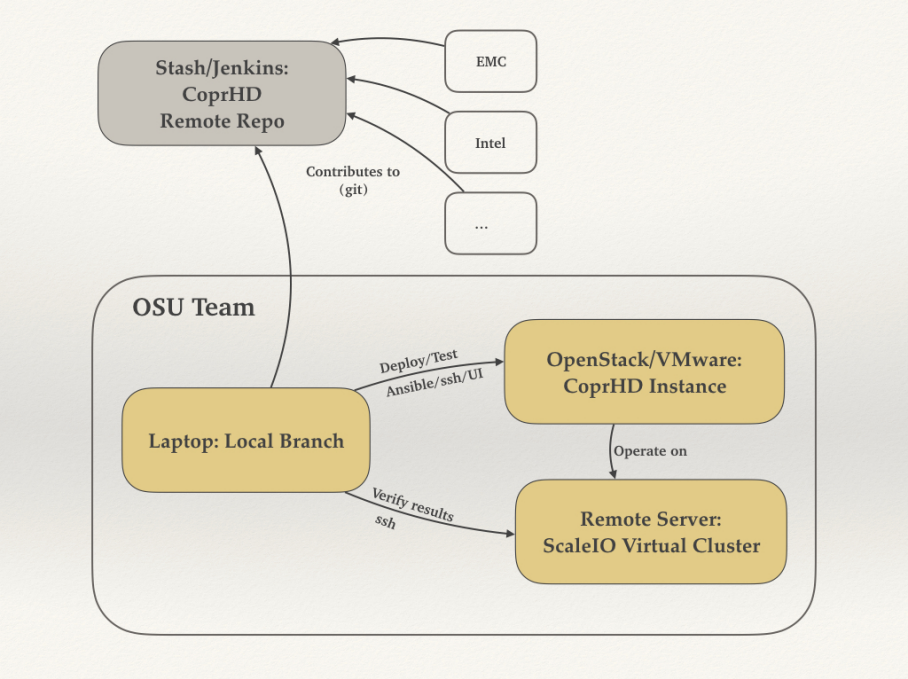

Figure 5.1: Development Environment

The development environment for the CoprHD driver implementation is composed of Integrated Development Environment (IDE) and the test environment. The IDE is usually installed and configured locally to avoid any inconvenience caused by the internet, while the test environments are placed remotely to allow access to larger storage resources. During the ScaleIO driver development, we only

performed unit tests to verify the functionality of the driver. Those unit tests directly interact with the ScaleIO test cluster and are independent of the CoprHD infrastructure. Our ScaleIO test cluster is set up in Virtualbox following Curt Bruns's Vagrant solution. However, unlike his solution to configure the cluster locally, we deploy it to a remote server to have sufficient storage to test on.

Since during the integration test, all the driver operations are invoked from CoprHD UI, a CoprHD instance is required in this period. CoprHD community posts some how-to articles to guide developers to build and run a CoprHD instance from scratch , and we adapt it to our OpenStack environment. However, this solution is problematic and requires painful occasionally debugging, which leads to the creation of my Ansible solution.

5.2 Reusable Modules

As the first team outside EMC to develop a CoprHD driver, we are a trial run for future third party developers. One of our missions in this development process is to identify the items that would be challenging or time-consuming for a third-party developer and figure out a way to speed up the overall driver development. To determine the reusable modules is one of our achievements. Here are some reusable modules:

- **Error Handling Modules:** Containing CoprHD service error models and generic exception interfaces.
- **Service Utils Modules:** Including but not limited to REST utils, such

as generic REST client interfaces, abstract classes of REST client factory, standard REST client, etc.

Chapter 6: Deployment

6.1 Issues with CoprHD Deployment

CoprHD deployment is error-prone for a variety of reasons. First of all, there are hundreds of branches in the CoprHD repository and a high number of commits are pushed to different branches every day, some of which are merged to the master branch, resulting in various levels of code changes. Our test branch is based off the master branch, so occasionally we need to update the branch with the up-to-date master branch. Although CoprHD employs Jenkins automation server, which will trigger a CoprHD build whenever a commit pushed to the repository and refused the commit if the build fails, the server does not detect the deployment issues beforehand.

Another cause of deployment failures is the version updates of the software packages on which CoprHD depends. CoprHD requires pre-installation of over one hundred packages on the hosted system, some of which are re-installed in every deployment. The scripts that the CoprHD community employed do not specify the versions of the software packages, and they are also not aware of failed installation of one package.

Ansible Playbook	Functionality
coprhd-env-setup.yml	Set up system environment for CoprHD
coprhd-deploy.yml	Deploy CoprHD
coprhd-uninstall.yml	Completely remove CoprHD from system
scaleio-driver-deployment.yml	Deploy ScaleIO driver

Table 6.1: Ansible Major Playbooks

6.2 CoprHD Deployment with Ansible

Ansible is a very powerful open source automation language aiming to deliver high productivity for a broad range of automation challenges in many respects, especially suitable for systems that involve multiple software packages and extensive custom coding. Table 6.1 lists some playbooks I wrote for the major deployment tasks. Playbook *coprHD-env-setup.yml* prepares a hosted system environment for CoprHD, while playbook *coprHD-deploy.yml* downloads CoprHD repository to the hosted system and carries out the steps to build and run CoprHD. Ansible allows specifying the versions of the packages and the commit of the adopted CoprHD branch in the playbook, and when the playbook is executed, it prints out the recap of the deployment sub-tasks and stops at where it fails. Therefore, this plot-like deployment brings exceptional benefits towards debugging, and it's also

less error-prone.

6.3 Driver Deployment

The manual deployment of the CoprHD driver follows the steps below:

1 Log in to the CoprHD instance via ssh.

2 Update multiple configuration files in different places. All these configuration files will be reset if the CoprHD instance is re-deployed.

3 Build the driver into a jar file and upload to the CoprHD instance.

4 Restart the CoprHD services.

5 Start the testing driver.

Similarly, this manual deployment is not only time-consuming but also error-prone. For example, it's relatively easy to make mistakes while modifying the configuration files. Playbook *scaleio-driver-deployment.yml* in table 6.1 automates the process to deploy the driver, with which, we can deploy the driver by one command. However, it is desired to deploy the storage driver though the CoprHD UI, and hopefully, we can do that in the near future.

Chapter 7: Tools and Technologies Used

As an open source project, CoprHD uses many software development and collaboration tools to help the CoprHD contributors across the world. Table 7.1 lists the major tools we used during our driver development. One of the advantages of these tools is that most of them are integrated into one site, where contributors can quickly switch one tool to another. The CoprHD community also maintains a website to collect CoprHD resources and latest news, which not only lowers the threshold for new contributors but also makes it easier for people to track the overall status of CoprHD. Communication tools like Hipchat ensure timely communication and assistance from CoprHD experts. The project management tool JIRA facilitates checking the status of issues or desired features through a unified platform. The automation server Jenkins relieves developers from the wearisome part of the software development process, such as continuous integration. CoprHD handles deployment via Bash script, but I believe Ansible is a better option.

CoprHD Tool	Alternatives	Functionality
Stash	Git, SVN	Version Control
JIRA	OpenProject, Google Code Hosting	Issue & Project Tracking
Hipchat	Slack	Team Group Chat
Confluence	SocialText, Wiki Spaces	Organize & Create Documents
Jenkins	Bamboo	Continuous Integration
Google Group	Quicktopic, Copperproject	Forum, Q&A
Cisco Weber	Skype	Online Meeting and Conference

Table 7.1: Software Development and Collaboration Tools

Chapter 8: Conclusions

Let us reconsider our process to develop a ScaleIO driver for CoprHD. First, we examined the methods defined in the southbound SDK to determine what operations are expected in the driver. Then we studied the ScaleIO storage system to understand which methods are supported. With the requirements in mind, we created unit tests, which were approved by the community later, for each supported method. And then comes the most important process of the driver development - driver design. We first decided to make use of the ScaleIO REST API to perform storage operations. Following that, we confirmed the high-level system design and then split it up to detailed design for each component. Regarding the snapshot module, my job includes verifying the input/output with the SDK team, understanding the ScaleIO snapshot operation and creating the code flow. I also determined the design patterns in this period. After design, implementation of the driver became straightforward. Integration test was the final process to verify the storage driver.

I have achieved the following aims in this project: a tutorial for CoprHD storage driver development; some essential components of the ScaleIO storage driver including the *ScaleIORestClient* module and the snapshot operation module; an Ansible deployment solution for CoprHD and its driver.

This work has made the following contributions. It identifies the challenges for

third-party developers during driver development and creates a test-out solution for them. It revealed SDK bugs that got fixed before other third-party developers started writing CoprHD drivers. Most importantly, this work creates a ScaleIO storage driver which will serve as a template driver for third party developers and accelerate their development process.

Bibliography

[1] Curt Bruns. Vagrant-coprhd-scaleio-devstack, May 2016.

[2] Mark Carlson, Alan Yoder, Leah Schoeb, Don Deel, Carlos Pratt, Chris Lionetti, and Doug Voigt. Software defined storage, Jan 2015.

[3] Anjaneya Chagam. Delivering a standards based SDS framework with an open stack SDS controller implementation, 2014.

[4] Anjaneya Chagam and Urayoan Irizarry. Introduction to CoprHD: An open source software defined storage controller, 2015.

[5] Anil Degwekar. Storage orchestration for OpenStack, Dec 2015.

[6] Laura DuBois. IBM spectrum storage suite: Meeting industry needs for software-defined storage, Jan 2016.

[7] EMC. *EMC ScaleIO User Guide*, Nov 2013.

[8] Hemanth Makaraju and Ben Perkins. CoprHD and Flocker integration, Aug 2016.

[9] Olaf Manczak and Bill Elliott. A short guide to the CoprHD architecture, July 2015.

[10] Ashish Nadkarni and Laura DuBois. Software-defined storage: A pervasive approach to IT transformation driven by the 3rd platform, Nov 2015.

[11] Mithuniro Oshiro and Urayoan Irizarry. How to download and build CoprHD, May 2016.

[12] Mithuniro Oshiro and Ben Perkins. How to build and run CoprHD, May 2016.

[13] Ben Perkins and Devanjan Sarkar. Vagrant-coprhd-scaleio-devstack, Jun 2016.

[14] Simon Robinson. Software-defined storage: The reality beneath the hype, Mar 2013.

[15] Evgeny Roytman. How to write storage driver for CoprHD, Sept 2015.

[16] Lus Sousa. Test driven development, an agile methodology, Jan 2017.

[17] Shujin Wu, Evgeny Roytman, Prathamesh Pramod PatKar, and Varun Rajgopal. ScaleIO storage driver based on southbound SDK, May 2016.

APPENDICES

Appendix A: Terminologies