EECS 4314: Hadoop Concrete Architecture Report

Mina Zaki
Alexander Aolaritei
Dillon Bondarenko
Camillo John (CJ) D'Alimonte
Jeremi Boston
Daniel Nowak

November 2, 2016

Contents

1 Abstract

This report discusses the concrete architecture of the Hadoop YARN subsystems. The addition of YARN has allowed Hadoop to become much more efficient than in its previous version. The YARN framework is an important part of the Hadoop model acting as the master of the system controlling the flow of resources and data. A lot of the information in the following report has been found by analyzing visual representation of the Hadoop source code using a simple editor in combination with a static code analysis tool to visualize the source code. Through this method the properties of YARN are presented, as well as the differences between conceptual and concrete, and the reasons why they are not the same. As a result, the findings show that the conceptual architecture can not always be directly translate into the concrete architecture.

2 Introduction

The concrete architecture of Hadoop will be covered, with a focus on YARN. Architectural styles and design patterns will be discussed relating to Hadoop and YARN. Next, we will compare the concrete architecture against the conceptual architecture via reflexion analysis. Discrepancies between the two architectures will be brought up in section 4. The following section 5 will discuss noteworthy aspects of the architecture and its subsystems using sequence diagrams (or state diagrams). Section 6 includes a brief discussion of dependencies between the YARN subsystems. Finally, section 7 will conclude with the lessons learned from this assignment. The purpose of this report is to present and explain the framework and functionalities of the YARN subsystem of Hadoop.

3 Architectural Styles

3.1 Master & Slave

One of the ever present architectural styles dominating Hadoop and its subsystems is called the Master-Slave architecture. This architecture is a modification of the main-subroutine architectural style, which supports system reliability and fault tolerance. This style is most commonly used in instances of process control, embedded systems, large-scale parallel computations and fault-tolerant systems. Not only is it found all across Hadoop, but also in YARN.

A general explanation of this architectural style would be of a single device which has unidirectional control over one or several other devices. The components of this style are the master and the slave. The master component is responsible for communication, coordination, computation and most importantly it controls the slave components. On the other hand, the slave component is responsible for a specific action it must perform for the master. We can see an example of this architecture in action in Figure 1, where a sequence diagram represents the distribution of tasks by the master among its slaves [2].

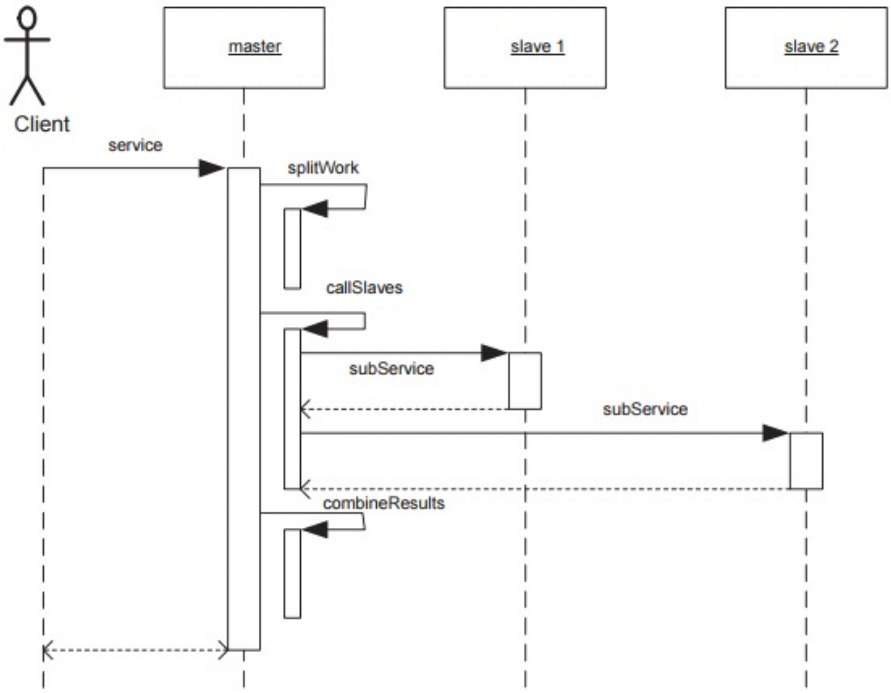

Figure 1: Sequence Diagram showing Master-Slave Interaction

A real life example of this architectural style can be seen in instances such as parallel computing. Where the master component takes a complicated task, divides it into several identical tasks which are to be completed by its slaves [2]. Such as the computation of a matrix, where each row is computed by a different slave.

3.2 Managers

Looking at YARN we can see that the master component is the Resource manager, which can be seen in Figure 2 [3]. Which is responsible for knowing where its slaves are, their resources, and most importantly it decides where these resources will be allocated [3]. The slave component is called the node manager, which can be seen in Figure 3 [3]. It is responsible for communicating with the master with regards to its heartbeat and its resource capacity. However, more importantly it offers its resources to the master, which can also be split further into containers [3]. From this we can conclude that the master-slave architecture is used within YARN.

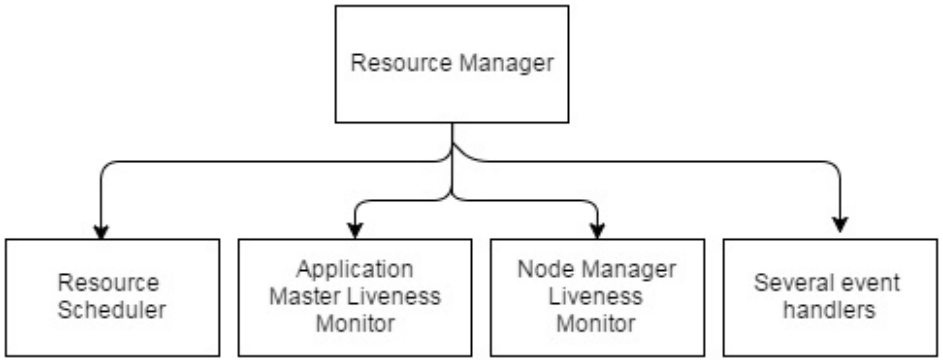

Figure 2: Components (slaves) of the Resource Manager

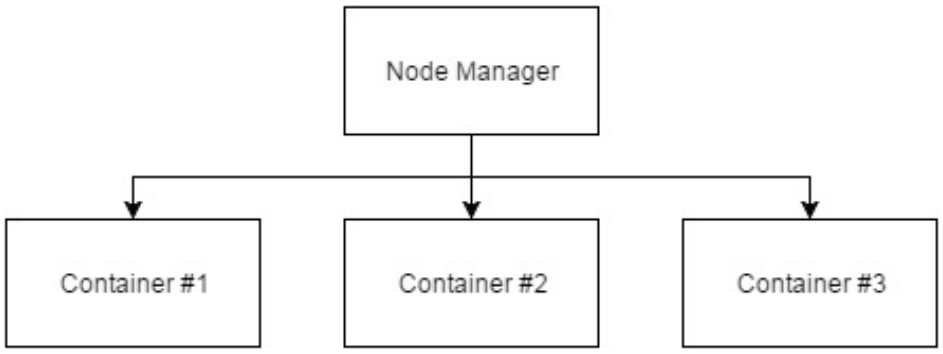

Figure 3: Components of the Node Manager

3.3 Advantages & Disadvantages

The advantages of this system are that it supports fault tolerance, which is the ability for a system to continue operating properly despite a failure in one or many of its components. It also supports parallel computation, as can be seen in our earlier matrix example. Furthermore, it is also a highly scalable architecture, due to the ease in which slaves can be increased. However there do exists some disadvantages to the architecture. For one the slaves are isolated beings, there is no shared state [2]. There is also the issue of possible latency in respects to master slave communications [2]. Finally, the architecture can only be applied to issues that are decomposable, able to split into multiple smaller issues.

4 Design Patterns

4.1 Design Concepts

The Application Submission Client submits an application to the YARN ResourceManager (RM) through the creation of a YarnClient object. The application context is set up once the YarnClient has commenced as preparation for the first container that contains the ApplicationMaster (AM) begins. Once completed, the application is submitted. The YARN ResourceManager will then launch the ApplicationMaster (as specified) on an allocated container. The ApplicationMaster communicates with the YARN cluster, and handles the application's execution. Operations on the application are performed in an asynchronous manner.

The main tasks of the ApplicationMaster during the application launch time are as follows:

- communicating with the ResourceManager to negotiate and allocate resources for future containers;
- after container allocation, communicating YARN NodeManagers (NMs) to launch application containers.

The first task is often performed asynchronously through an AMRMClientAsync object, with event handling methods specified in an explicit AMRMClientAsync.CallbackHandler type of event handler. The second task can be performed by launching a runnable object that then launches containers when there are sufficient containers allocated.

During the execution of an application, the ApplicationMaster communicates NodeManagers through the NMClientAsync object as all container events are handled by the NMClientAsync CallbackHandler. Client start, stop, status update and error reporting are handled by a typical callback handler. The ApplicationMaster will also report execution progress to the ResourceManager by handling the getProgress() method of AMRMClientAsync CallbackHandler.

4.2 Interfaces

The following are some integral interfaces to the core functionality of YARN:
Client <->ResourceManager

- By using YarnClient objects.

ApplicationMaster <->ResourceManager

• By using AMRMClientAsync objects, handling events asynchronously by AMRMClientAsync.CallbackHandler ApplicationMaster <->NodeManager

• Launching containers and communicating with NodeManagers by using NMClientAsync objects; handling container events by NMClientAsync.CallbackHandler

To note, the three main protocols for YARN applications (ApplicationClientProtocol, ApplicationMasterProtocol and ContainerManagementProtocol) are preserved as the 3 clients wrap these 3 protocols to provide a simpler programming model for YARN applications.

4.3 Object-Oriented Design Patterns (OODPS)

There is extensive use of OODPs throughout the source code of YARN. OODPs make heavy use of key object-oriented principles including, but not limited to interfaces, information hiding, polymorphism and intermediary objects. The design patterns used in YARN can be divided into two sub-categories:

• Structural Patterns – concerns the process of assembling objects and classes

• Decorator Pattern: Attaches additional responsibilities to an object dynamically. Provides a flexible alternative to sub-classing for extending functionality. Elements seen in the ApplicationMaster <->ResourceManager interface relationship

• Behavioural Patterns – concerns the interaction between classes or objects

• Iterator Pattern: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Elements seen in all the above interface relationships

• Observer Pattern: Defines a one-to-many dependency between objects; as one object changes state, all of its dependents are notified and updated automatically. Elements seen in the ApplicationMaster <->ResourceManager interface relationship

The principle of Program to Interface, not to Implementation is extensively used throughout the development of YARN. By programming to interfaces, the application will always be open to modification while not suffering from extensive change in code.

5 Reflexion Model

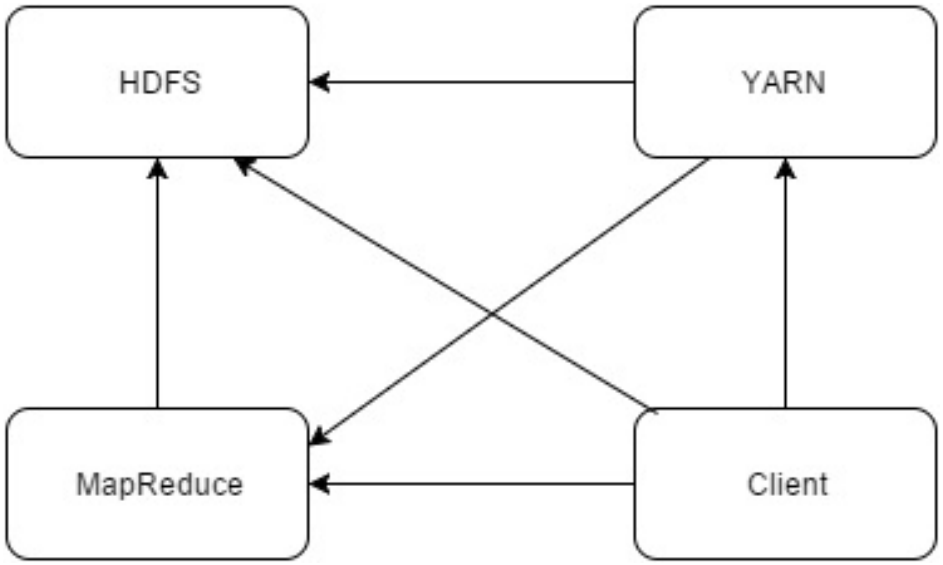

Figure 4: Conceptual Architecture of Hadoop from Assignment 1

Figure 4, presents the conceptual architecture that was uncovered from the Conceptual Architecture Assignment. Figure 5, displays the reflexion model for the concrete architecture of Hadoop. For this diagram, the dashed lines indicate divergence from the conceptual architecture of the Conceptual Architecture

Assignment. The reflexion model was created by first taking the concrete architecture uncovered using LSEdit and then replacing the some of the dependencies with the divergences of the conceptual model. In the following section, we will discuss the discrepancies between the conceptual architecture and the concrete architecture.

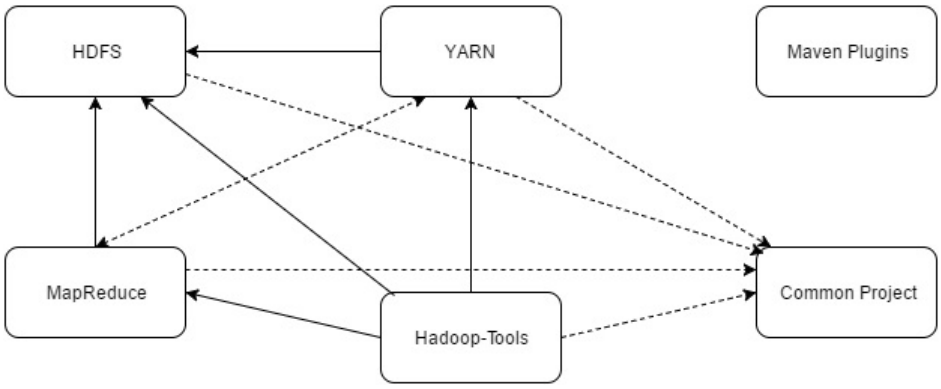

Figure 5: Reflexion Model for Concrete Architecture of Hadoop

6 Conceptual vs Concrete

This section will go in depth as to what differed from the conceptual architecture found from assignment 1 compared to the concrete architecture found from the tools provided in class.

Conceptual architecture - as its name implies - are the ideas thought up of during documentation. This is when classes and their dependencies are created abstractly without physical implementation and as such provide a blueprint to what the system will look like. These blueprints follow the System Requirement Specifications for the project as close as possible.

The concrete architecture is the physical code of the system - what dependencies are required in the system as the code asks for presently. Concrete architecture may not always stay cemented in place, as user feedback, or new features that come with updates may change the internal structure of the code to require a few new objects here and there.

6.1 Discrepancies Found

The most important factor here to note, is that there is now a two-way dependency between MapReduce and YARN. The reason for this being that MapReduce is originally handed all the data to split up and distribute, but what's it going to do with the data once that task is completed? It's given back to the YARN system for use in the classes defined there.

The Client has also been removed from the picture. In the conceptual stage of developement: the client was originally put in there as a placeholder. It represents what the client (the user) has access to in the system. It is not found in the concrete architecture since "client" from earlier is just an abstract being.

Common Project is a new item here. It contains artifacts of every class needed in the system. An artifact can be thought of as code snippets to help YARN as a whole to point calls in the right direction. Most importantly it contains initiation sequences for each of the dependant classes.

7 Noteworthy Aspects

The YARN subsystem is broken down into two major responsibilities of a JobTracker which include the Resource Manager and the Job Scheduler/Monitor. These two sections are then separated into two background processes called the ResourceManager and ApplicationMaster and are illustrated in the sequence diagram provided in Figure 6. The ResourceManager is responsible for managing all applications in the system and has a scheduler that allocates resources to other running applications. The resources are allocated

based upon constraints such as user limits and queue capacities. In comparison, the ApplicationMaster is responsible for communicating with the scheduler such that resource statuses are tracked and progress is monitored. The ApplicationMaster runs as a container where a container is defined as an entity that incorporates resource elements such as CPU, memory, and networks.

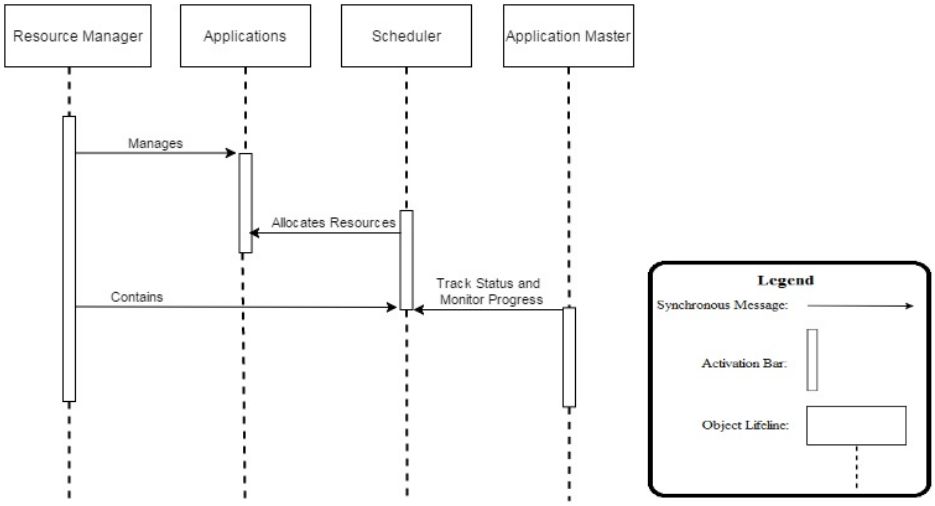

Figure 6: Sequence Diagram

7.1 Roles of ResourceManager and Application Master

The roles of the ResourceManager and ApplicationMaster can be better observed in the state diagram in Figure 7 which depicts the use case process of creating a YARN application. When a YARN application is created, a YarnClient object is instantiated and initialized using a predefined configuration. A YarnClientApplication object is then created and contains information about the clusters minimum/maximum resource capabilities which is then used to set the specification of the container that the ApplicationMaster will be launched from. Afterwards, other necessary information for the YARN application is established for the setup process such as setting up the application submission context, setting local resources for the application master, setting up the shell script so that it's available for when the application is executed, setting the environment variables, setting resource type requirements, and so on. The YARN application is then submitted using the submitApplication method along with the application context as its parameter. At this point, the ResourceManager accepts the application and begins to allocate a container with the required specifications and then afterwards will launch the ApplicationMaster on that container. The ResourceManager can then manage the state of the application by using the information contained in the YarnClient ApplicationReport. The ApplicationMaster will communicate with the ResourceManager with the purpose of allocating resources for containers and then communicate with the NodeManager's so that application containers can be launched on the allocated containers. The ApplicationMaster communicates with the ResourceManager for container allocation asynchronously by using an AMRMClientAsync object and event handling methods in the AMRMClientAsync.CallbackHandler.

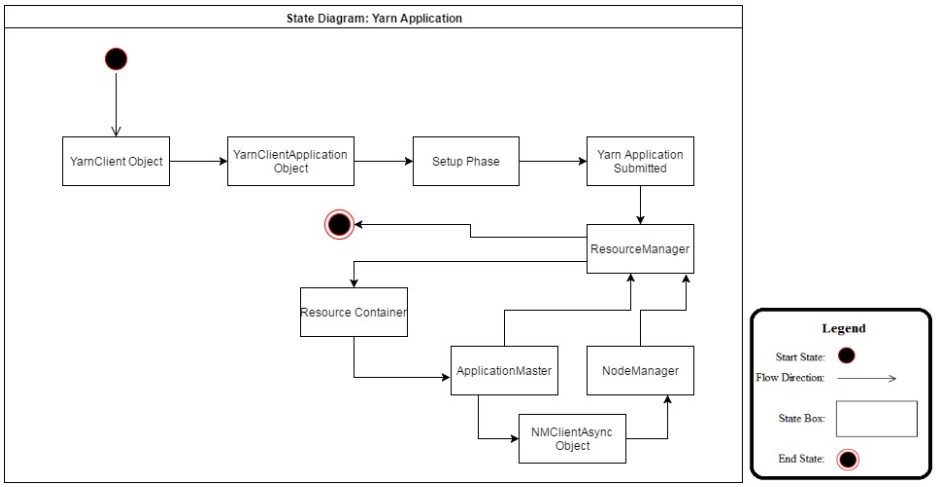

Figure 7: Creating YARN Apps

The task of having the ApplicationaMaster communicate with the NodeManager with the purpose of launching application containers is implemented by launching a runnable object that launches the allocated containers. Information about specification and environment is specified in the ContainerLaunchContext. This is required so that the ApplicationMaster can perform the container launch.

8 Concrete View

The following diagram in Figure 8 represents the concrete architecture of the YARN subsystem in Hadoop. The main components are separated into six more subsystems, the API, the Client, the Server, the Common, and lastly the Registry subsystem. The dependencies in Figure 8 show which of the subsystems interact with each other in order to perform their tasks.

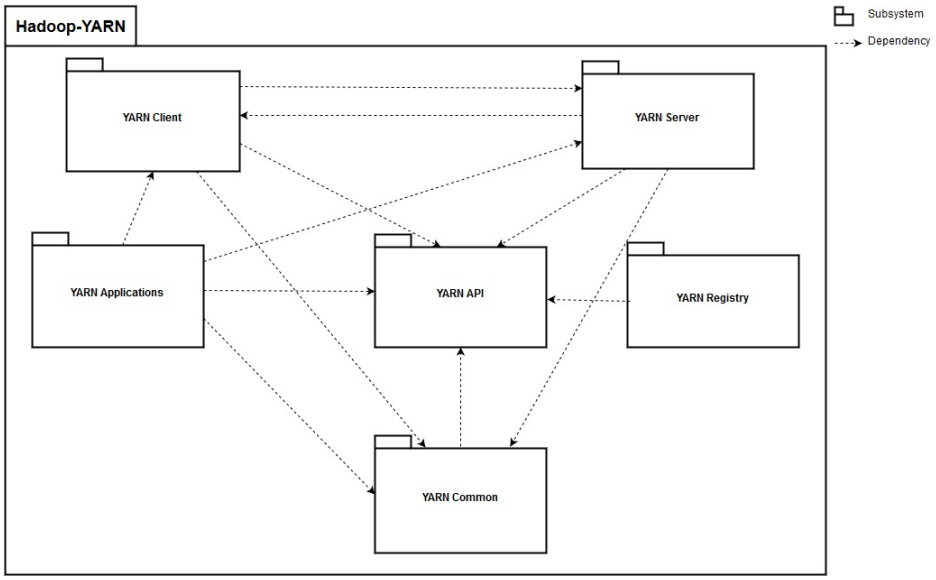

Figure 8: Creating YARN Apps

One of the most important subsystems is the YARN API, providing an interface for remaining subsystems of YARN and allowing communication between subsystems. Not only does it provide an interface but also

provides various protocols, exceptions, configurations and utilities used by all the subsystems. The next subsystem is the YARN Registry. This subsystem is used to facilitate the communication between YARN-deployed services through the use of YARN API, and to allow Hadoop services to be registered and discovered allowing services to be moved more easily [5]. Since the registry uses the API to communicate information to other subsystems it is dependent on the API subsystem. Moving on, another subsystem, the YARN Common subsystem which is what the server, client and applications subsystems are dependent on. The reason for these dependencies is the fact that it is a core library for the YARN framework which the three aforementioned subsystems make use of in their implementation. The following two subsystems YARN client and YARN Server are dependent on each other. In this situation we have the client providing information to the servers about the applications they need to run, and also creating the many containers to hold the applications running on separate servers. On the other end we have the server performing the tasks requested by the client. Finally, the remaining YARN Applications subsystem depends on the client and the server subsystems. The Applications subsystem must first wait for a YARN client to setup the application context. Once completed, the client creates an application container that contains the ApplicationMaster which is then submitted [7]. With the application container created, the application now depends on the server to run the program and complete its given tasks. Although there are many more dependencies within the subsystems, these have been selected to represent an overview of the interaction between the Hadoop-YARN subsystems.

9 Lessons Learned

It is difficult to analyze and report the concrete architecture of a system. In comparison, the concrete architecture has more dependencies than the conceptual architecture. To try and represent all dependencies of the concrete YARN architecture is very difficult under a time constraint. Furthermore, conceptual architecture can not always be directly translated into the concrete architecture.

The style of "High cohesion - Low coupling" is a great statement here concerning design. Using too many dependencies may cause the system to be vulnerable to fault when one of the dependant system malfunctions. Furthermore, it makes error handling easier without having to trace through call stacks.

10 Limitations of Findings

Since Hadoop is such a large system, it is quite easy to miss design patterns or architectural styles used. Furthermore, the conceptual architecture that was uncovered in Assignment 1 is our interpretation of what the developer's view of the system was, it can easily differ from the developer's view of the conceptual architecture.

11 Conclusion

In conclusion, the concrete architecture of Hadoop differs from its conceptual architecture. As shown in the relexion model, the concrete architecture has more dependencies than the conceptual architecture. One of the main architectural styles used in Hadoop and YARN is Master-Slave. This is a highly scalable architecture that is able to support the many nodes in the system. The two main aspects of the YARN subsystem are the Resource Manager and the Job Scheduler. In future, it would be interesting to investigate deeper into the dependencies of the YARN subsystem.

12 Bibliography

[1] "Apache Hadoop YARN." Apache Hadoop 2.7.2 –. N.p., n.d. Web. 30 Oct. 2016. [noteworth]

[2] "Architectural Styles and Patterns." Software Architecture Advanced Topics in Science and Technology in China (n.d.): 34-88. Web. 30 Oct. 2016. [master slave 1]

[3] "Hadoop Internals." Hadoop Architecture Overview ·. N.p., n.d. Web. 30 Oct. 2016. [master slave 2]

[4] Industry, By. "Introducing Apache Hadoop YARN - Hortonworks." Hortonworks Introducing Apache Hadoop YARN Comments. N.p., 03 Aug. 2012. Web. 30 Oct. 2016. [me]

[5] "Introduction and Concepts." Apache Hadoop 2.7.3 – The YARN Service Registry. N.p., n.d. Web. 30 Oct. 2016. [concrete 1]

[6] Kaiser, Gail E. SIGSOFT '95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering: Washington, District of Columbia, USA, October 10-13, 1995. New York: ACM, 1995. Web. 30 Oct. 2016. [reflexion]

[7] Objects., By Using YarnClient. "Hadoop: Writing YARN Applications." Apache Hadoop 2.7.2 –. N.p., n.d. Web. 30 Oct. 2016. [noteworthy concrete 2]