MapReduce: MR Model Abstraction for Future Security Study

Citation for published version (APA):

Published in:
the 7th International Conference on Security of Information and Networks (SIN’14)

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester’s Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Open Access

Download date:08. Nov. 2019
MapReduce: MR Model Abstraction for Future Security Study

Ibrahim Lahmer, Ning Zhang
School of Computer Science
The University of Manchester
Manchester, M13 9PL, UK
ibrahim.lahmer@postgrad.manchester.ac.uk
ning.zhang@manchester.ac.uk

ABSTRACT
MapReduce is a new parallel programming paradigm proposed to process large amount of data in a distributed setting. Since its introduction, there have been efforts to improve the architecture of this model making it more efficient, secure and scalable. In parallel with these developments, there are also efforts to implement and deploy MapReduce, and one of its most popular open source implementation is Hadoop. Some more recent practical MapReduce implementations have made architectural changes to the original MapReduce model, e.g., those by Facebook and IBM. These architectural changes may have implications to the design of solutions to secure MapReduce. To inform these changes and to serve any future design of such solutions, this paper attempts to build a generic MapReduce computation model capturing the main features and properties of the most recent MapReduce implementations.

Keywords
MR: MapReduce, DFS: Distributed File System, YARN: Yet Another Resource Negotiator Master node, Resource Manager, JobTracker, TaskTrackers, NameNode, DataNodes.

1. INTRODUCTION
MapReduce, as a parallel and distributed programming framework, is becoming more and more widely used [17]. Hadoop, an implementation of the MapReduce framework, has been adopted by many companies, including the major IT players in the world, such as Facebook, eBay, IBM and Yahoo [2] [16]. Recently, Facebook engineers team [5] have claimed that they developed a new MapReduce topology called Corona. Also, Yahoo developers [10] and IBM developers [11] have also announced that they have developed and used a new MapReduce implementation, YARN. These new implementations of the MapReduce use architectures that are different from the one used by Hadoop, the previous implementation. For example, in Corona and YARN, the functions played by a single master node in Hadoop is split into two sets, respectively, played by two master nodes, a Resource Manager and a JobTracker. In other words, Corona and YARN use two masters to perform the functions which are performed by the single master in Hadoop. While, these architectural changes in the new MapReduce implementations bring a number of benefits such as improving performances and reducing the chance of creating bottlenecks in the system [5][11]. They also have implications on other aspects of the system such as security provisioning. For example, these changes will alter the way different system components interact, and how and from which entities various requests and responses are generated. These will lead to changes in the risk landscape in the system, and therefore how security should be provided. However, existing security solution designs for MapReduce have not taken into account these architectural changes in the new MapReduce implementations. For example, the encryption service has been designed by Jason and Subatra (2013). They examined their proposed security service for the old MapReduce framework version [1]. The scheme encrypts data which is uploaded by the client and stored in DFS. Furthermore, Somu, N. and et al. (2014) proposed another security service [12]. They have designed an authentication service by which the client can be authenticated to the MapReduce framework. Their design schemes show that they perform well in term of security and performance in the old MapReduce version. Their designs assume that, there is only one master node (JobTracker) in which the client authenticate to and submit his job. This master node also retrieves the input data splits which are uploaded to the DFS, and allocates the TaskTrackers (Worker nodes). However, in the current MapReduce framework implementation, it is no longer the case, there are two master nodes, one master node, Resource manager, is responsible for job submission, providing job IDs and the paths of job files to clients, then the client uploads his input data files into the DFS. The other master node, JobTracker, retrieves input splits which are uploaded by the client to the DFS, and allocate the TaskTrackers which are assigned by the Resource Manager. This raise concerns in term of security as the way of MR components interact and accessing the data stored in the DFS. An example of these concerns, in the new MR framework implementation the client authenticates to the Resource manager rather than JobTracker and his input splits are retrieved by the other master node. This means that the JobTracker also needs proper secret keys to have access to the DFS and retrieves the input splits when the...
clients job submission is completed. Also, in the new MR implementation, the worker daemons have to authenticate to the Resource Manager to obtain the admission to the MR framework rather than authenticate to the JobTracker.

The authors' main research task is to investigate and design security solutions for MapReduce applications, as designs of some security services for MapReduce should be architectural dependent, and as the existing MapReduce security solutions are largely homogeneous and only secure entrance of a job execution [13] [9]. For this, the solution should be designed based on a generic MapReduce model in literature. However, first, there is not a generic MapReduce model in literature, which captures the recent changes in the MapReduce architecture. Second, through literature research in MapReduce architectures [2][4], and exiting MapReduce applications, such as used by [5][10] [11], we have identified that there is a mismatch between the original MapReduce architecture and what has been deployed in recent real-life applications. So this paper as our first step work, tries to bridge this gap by presenting an abstract MapReduce model capturing the features and properties of the most recent deployments of MapReduce. We investigate the existing MapReduce architecture model and synthesizing from these real-life deployments and in future steps, build and present a generic security solution requirements and design for such model.

In detail, the remaining part of this paper is structured as follows. Section 2 introduces MapReduce, its architecture and architectural components and explains, at a generic level, how it is typically deployed in a Cloud environment. Section 3 analyses, in detail, a simple example of job execution process using MapReduce, and based on this analysis, the section presents an abstract model of MapReduce execution using MapReduce. It starts with an example job execution process using MapReduce, and based on this example, it constructs an abstract model for MapReduce highlights the procedures and interactions among its architectural components during a job execution process. Finally, Section 4 concludes the paper and outlines out future work.

2. MAPREDUCE ARCHITECTURE

This section introduces the MapReduce framework, highlighting its physical and logical components and explains how the framework may be deployed in practice.

2.1 An Overview

MapReduce is a new programming model that supports parallel processing and distributed resource sharing. It uses a set of distributed nodes (called servers) that work collaboratively to process a large amount of data or to execute a specified job. A job execution in MapReduce is carried out in two distinctive phases: Map and Reduce. In the Map phase, a set of nodes, called Mappers, are used to process and convert one set of data (input) into another set of data (intermediate results). This set of input data is broken into key-value pairs called tuples. In the Reduce phase, number of nodes, called Reducers combines and process the intermediate data result to produce smaller set of tuples called job result. As indicated in the framework name, in MapReduce, Map tasks are always executed before the Reduce tasks [17]. Figure 1 shows the generic sequence of the process in MapReduce applications.

2.2 MapReduce Components

![Figure 1: A typical MapReduce task sequence [3]](image)

A MapReduce system consists of a number of components which are summarized in Figure 2. As shown in the figure, these components can largely be classified into two main groups (called clusters). One is the Distributed File System (DFS or Distributed Storage) that is used to store date files during job executions. The DFS consists of one name node and multiple data nodes. The other is the Processing Framework (PF) cluster that is used to carry out job executions (or distributor computations). PF consists of two master nodes and a number of worker nodes. In addition to these physical components, there are also logical components (i.e. software components) and data components. In the following, we discuss the functionalities of these components [4] [5] [15] [16].

2.2.1 Physical Components

Physical components are actual physical nodes used to host software components, clients and servers. Depending on the roles played by the servers, they may be master workers or slave workers. The corresponding nodes hosting these components are therefore referred to as user machines, master worker nodes (or master nodes, in short) and slave worker nodes (or slave nodes, in short).

1. **User Machines**: A user machine runs a client application, which largely performs two major tasks: i) submit a job to the MapReduce master node, and ii) specify the configuration, trigger and monitor the Job execution [8].

2. **Masters nodes**: In Figure 2, master nodes are of the following types.

   (a) **Central Cluster Manager** (also called Resource Manager): This node is responsible for managing the job schedules and the cluster resources by tracking the nodes in the cluster and the amount of free resources on these nodes. There is one such node per cluster.

   (b) **JobTracker** (also called Master Node or MRApplication Master): It runs a program, called master daemon that is responsible for managing (monitoring and tracking the progress of) each individual job executions. It also negotiates with the Resource Manager regarding the use of resources (containers i.e. Slaves). Usually, there is one or more JobTrackers per cluster. In older MapReduce framework version, there is only one Master Node that plays the roles of both Resource Manager and Job Trackers.

   (c) **NameNode** (also called Catalogue Server or Metadata Data Node): This node manages and maintains a set of DataNodes, i.e. a file system and metadata for all the directories and files stored in DataNodes. It is usually part of DFS cluster and there is one per cluster.
3. **Slave nodes**: they are of two types, TaskTrackers and DataNodes.

   (a) **TaskTracker**: A TaskTracker node may host a single Mapper, a single Reducer or both. Usually, there is more than one TaskTracker node per cluster. As shown in Figure 2, each TaskTracker runs the worker daemon to execute multiple tasks.

   (b) **DataNode**: This node is also part of the DFS and, sometimes, named as a Storage Server. It is the node where the actual data files are stored in units of data blocks (each being 64MB or 128MB), and shared. Each data block has its own block ID, and, for fault-tolerance and performance considerations, it is replicated in a number of DataNodes. A DataNode periodically sends a report to the NameNode listing the blocks it stores, as shown in Figure 2. There are typically multiple data nodes in a cluster. In some cases the DataNode may be the same node as the TaskTracker node for performance consideration [15] [17].

4. **Job history daemon** is also executed on the JobTracker node. Its function is to store and archive the job history for later interrogation by the user if desired.

5. **Intermediate Buffered Data** is the output of the Map phase. Assuming that the number of reducers used in a job execution is R, then the intermediate data is a set of R regions called Partitions (i.e. for each reducer assigned to a job execution, there is a data partition output from the Map phase to process).

6. **Shuffle Handler** is a service designed to sort and group the intermediate key-values pairs according to the key (an example in the next section explains what this key is and how the key-value is used to present the data). It is run on the TaskTracker Node.

7. **Output Data** is the final result produced by the Reducers.

8. **Job Configuration** is a set of variables and parameters specific to a certain Job and specified by the User.

Figure 2 depicts both physical and logical components of the MapReduce Framework as organized in two Cluster categories.

### 2.3 Implementing and Hosing MapReduce Components in Cloud

MapReduce components are typically hosted in a virtual environment by a MapReduce provider. Virtualization allows multiple independent Operating System components be run on a single physical node. It allows the sharing of physical resources by multiple users [8]. The MapReduce framework is hosted and implemented in such environment due to the advantages provided by such technology. For instance, Server Consolidation, increase up time (e.g. fault tolerance, live migration, high availability, storage migration) and isolate applications tasks within one physical server [7]. The MapReduce application components run on the cloud system are hosted on the top of Virtual Machines (VMs). Figure 3 shows how different components of a MapReduce application such as master node (Master Daemon), worker nodes (Worker Daemons), DataNodes and NameNode are hosted. They can be run on the top of the same physical server. They are isolated using virtualization host machines and virtual switches. These different components share physical resources (e.g. CPUs and Rams). At the network level, they are isolated by using virtual switches and VLANs mechanisms. Communications among different set of MapReduce components hosted in different physical servers are carried out through external network fabric. The Hypervisor is the core of the virtualization technique. As a middleware, the Hypervisor is responsible for creating, managing and executing VM operating system instances and sharing the physical resources. There are number of Cloud
service delivery models [6], but the following two are used to host a MapReduce application. These two are different in term of services provided by the cloud providers as following:

1. **Infrastructure-as-a-Service (IaaS):** In this delivery model the customer allocates the virtual resources as needed. The IaaS provider delivers the required storage and networking devices, in form of wrapped service. They also provide a basic security needs, including physical security and system security such as firewall, Intrusion Prevention System (IPS) and Intrusion Detection System (IDS). A MapReduce application can be implemented using this service delivery model in which the application can be managed directly by the MR Client through the master nodes.

2. **Platform-as-a-Service (PaaS):** This service delivery model offers the facilities such as Software Development Kit (SDK) (e.g. Python and Java) which can be used by customers to write their own programs to develop their own applications. With this service deliver model, customers using these facilities do not have to manage the required hardware or software components such as VMs, virtual switches. CPUs and RAMs. A MapReduce application can also be implemented using this Cloud service delivery model.

![Figure 3: Hosting MapReduce components in the Cloud](image)

### 3. A GENERIC MAPREDUCE COMPUTATION MODEL

The components of the MapReduce model execute and process the input data based on a specific execution data workflow. This workflow involves a number of interactions between model components. This section, starting with a practical use case scenario, attempts to build an abstract model of MapReduce computation capturing the data flows as well as the interactions among MapReduce components during a job execution. This abstract model can serve as a basis for our future design of a security solution for MapReduce.

### 3.1 MapReduce Job Execution: an Example

To describe the data execution workflow in brief, a practical simplified example is illustrated. Suppose that there is a big number of records. These records contain temperatures of UK cities which are daily bases registered for one year.

<table>
<thead>
<tr>
<th>No.</th>
<th>Temp.</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>York</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>Durham</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>Leeds</td>
</tr>
<tr>
<td>4</td>
<td>23</td>
<td>London</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>Manch.</td>
</tr>
</tbody>
</table>

They are stored in number of database files as shown in the following tables, these files are stored in number of data blocks in the DFS. In this example, the Job is submitted by the client to the Master. The duty of the Job is to find out the maximum temperature of each city (each city has more than one record per file).

![Figure 4: A Number of files store temperatures of UK cities](image)

To simplify this example for explanation purpose, let consider number of files n = 10, each file stored in a one data block, and each input split size is the same as the data block size. Figure 5 shows the MapReduce components which are involved in processing the input data for this task, as a layout. First, the Client (User) determines the Job configuration parameters, input data format and the location of the input data. In our example, the input files are divided into input splits (10 input splits), and there is an input split per mapper to process. They are assigned to the Mappers by the Master based on job configuration files and parameters involved [14]. Each Mapper reads and parsing its input split into key-value pairs (in our example, the key is the city name and the value is its temperature). The Mapper starts the Map execution task (in this example, the map phase is to find out the highest temperature for each city in each input block i.e. file). The output result of the Mappers is the intermediate buffered data which is written locally in a number of data partitions. They are equal to the number of Reducers (in our example two data partitions and two Reducers). After the Mappers completed their task, each partition is assigned to a Reducer. The Reducers start their execution task which is finding the maximum temperature per city but before that the Reducers themselves have to perform the shuffle process. In the shuffle process the intermediate data (within the same partition) is sorted and grouped based on the key. (For example, York city (i.e. key) has a couple of temperatures values 18, 11, 23, 9 as an output of different Mappers, as listed in the intermediate result shuffle table in figure 5). The final output result of each reduce task, which is a list of cities and the maximum temperature associated with each city are written into separate files on the DFS system as shown in Figure 5.

### 3.2 Interactions of MapReduce Components

The following points describe a typical execution flow of job being executed by the MapReduce framework from the mo-
ment when the job is submitted by the client to the moment when it is successfully completed. This data execution workflow is further illustrated in Figure 6.

1. First, the user (on the Client node) runs the MapReduce program (Job Client) after it has been authenticated to the master node using his credentials. The used credentials are usually a Username and Password (Figure 6 (1)(2)). The master node might be either the Resource Manager in the new version of MapReduce framework or it is the JobTracker itself in previous version of the framework. This initial authentication is done by using Kerberos. The Kerberos protocol involves KDC Key Distributer Centre server in which the client and the server are registered. The KDC contains AS Authentication Service and TGS Ticket Granting Service. The client uses his credentials to make authentication request to the AS in order to get the TGT Ticket Granting Ticket. The Authentication Server then issues a TGT to the client. The Client then send it to the TGS requesting the service ticket. Once the client has the service ticket, he uses it to access the designated resource (i.e. service).

2. The Client requests the job ID (i.e. new application ID) and the Path to write the Job definition files (Figure 6 (3)).

3. Once the Client acquires the Job ID (Figure 6 (4)), the Client contacts the Namenode to start writing into the DFS (Figure 6 (5)), the client authenticate to the NameNode using the service ticket (contains the address of the remote service/resource as well as the secret) issued by TGT server from the first interaction between the client and AS.

4. The client is redirected to the DataNode and start copying the Job resources (Configuration file, input splits which are computed by the Job Client) to the specified directories (Figure 6 (6)).

5. Once the copying is completed the Client informs the Master Node that the Job is ready to be launched, this is called client job submission (Figure 6 (7)).

6. The Resource Manager allocates the JobTracker, and the master daemon (Application Master) is launched in this JobTracker (Figure 6 (8)(9)). This daemon creates an object and instant to present the Job and keeps track to the progress of the Jobs Tasks.

7. In order for the JobTracker to create a list of tasks to be run, it retrieves the input splits (computed by the client) from the DFS (Figure 6 (10)), whereas one map task object assigned for each input split. JobTracker also determines the reduce tasks objects.

8. The JobTracker requests TaskTrackers from the Resource Manager for all Map and Reduce tasks (Figure 6 (11)). In this request, as a performance prospective (i.e. consideration), JobTracker sends the data
location of each Map task, trying to obtain a Mapper node close to its split input.

9. Then once the tasks (either Map or Reduce Tasks) have been assigned to TaskTrackers, the mapper and reducer child daemons are started launching by the worker daemon (Figure 6 (13) (14)). However, before the map tasks started, the worker daemon retrieves the Job resources from the DFS (DataNodes) and copied them locally (Figure 6 (12)). The map task also, parses the input data into key-value pairs as explained in the previous example.

10. After Map tasks have been completed by the Mappers, the Reducers start to read the output data of the map task (Figure 6 (15)). Each reducer is assigned to one data partition as explained in the previous example. The reducer shuffles the map phase output data (i.e. intermediate data) in each data partition. This partition was specified before by the Master using user configuration file. These intermediate data at the specified partition, might be located in different nodes as they are an output of different mappers child daemons. So the reducer child daemon connects to designated Worker Nodes (daemons), where the partial data of the indexed partition stored locally, to gain access to the desired data (Figure 6 (15)).

11. Once the reduce task is executed and completed, the worker child daemon writes its output result into a file at the DFS system (Figure 6 (16)).

12. Finally the Reducer (Work daemon) informs the Master Node (JobTracker) that the task has been completed successfully or not (Figure 6 (17)). Once the entire reducers involved in the submitted job have completed the tasks assigned to them, they notify the Master Node (JobTracker). Then the JobTracker updates the client with the Job status (Figure 6 (18)).

4. CONCLUSION

In this paper, we have examined a real-life application of the MapReduce Model. Based on this examination, we have built an abstract MapReduce usecase model that captures the main characteristics of an MapReduce execution at a generic level, the functionality of each of its components and the interactions among the components in executing a computational job. This abstract model which addresses the previous MapReduce changes will be used in our future work on designing a cost effective security solution for MapReduce, because to study any security services for such model, it is necessary to understand its anatomy and including a recent MapReduce model.

5. REFERENCES


