

# MapReduce

A very primitive (and flawed) implementation of MapReduce

Alexander Lampalzer

March 10, 2019

# Contents

<b>1</b>	<b>License</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
2.1	Technologies . . . . .	3
<b>3</b>	<b>Map Reduce Algorithm</b>	<b>3</b>
3.1	Stage 1: Splitting . . . . .	4
3.2	Stage 2: Mapping . . . . .	4
3.3	Stage 3: Shuffling . . . . .	4
3.4	Stage 4: Reduce . . . . .	4
<b>4</b>	<b>Software Architecture</b>	<b>4</b>
4.1	Communication / Lifecycle . . . . .	4
4.2	Components . . . . .	6
4.2.1	Node . . . . .	6
4.2.2	Master . . . . .	6
4.2.3	CLI . . . . .	6
<b>5</b>	<b>Usage guide</b>	<b>8</b>
5.1	For End-users . . . . .	8
5.1.1	Python API . . . . .	8
5.1.2	CLI . . . . .	8
5.2	For System Administrators . . . . .	8

# 1 License

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

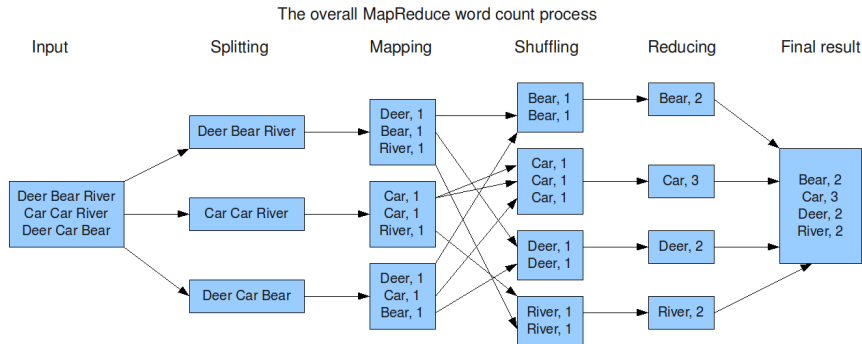


Figure 1: This diagram shows the different phases of the MapReduce algorithm, as explained in the following section.

## 2 Introduction

The goal of this project is to create fully implement the MapReduce algorithm in cpp. It is not meant for any sort of production environment, but rather to educate the author in writing c++.

### 2.1 Technologies

Purpose	Technology
Build Tool	Meson
Command line interface	CLI11
Configuration files	json
Data serialization	Protobuf
Logging	spdlog
Network Communication	asio
Network Communication	gRPC
Programming Languages	C++ 14, Python
String Manipulation	fmt

Table 1: This table lists all the technologies used for this project.

## 3 Map Reduce Algorithm

MapReduce is an algorithm for distributed, parallel processing of data. It accomplishes this, by dividing up the input data into chunks and processing these chunks in several stages. The following graphic depicts these stages and shows what input and output are needed / provided by each stage.

### 3.1 Stage 1: Splitting

The main goal of this stage, is to split up the input data into chunks, hence the name "Splitting". This can be accomplished in several ways. A common approach (and the approach used in this project) is to split at the end of each line. This is rather simple, however results in an increased amount of chunks, that have to be processed. Another approach, which is used by the popular big data processing software "Hadoop" is to split the data into block of 64MB.

In this implementation, the splitting phase is realised on the client's computer, or to be more precise, in the command line interface (CLI). Data is first ingested by this cli, then splitted and finally uploaded in chunks to the master. This is primarily due to simplicity reasons.

### 3.2 Stage 2: Mapping

In it's core, this phase is quite simple: It takes input data, processes it using whatever algorithm the user intends to use and returns a list of key-value pairs - for example [(Foo, 2), (Bar, 1)].

This implementation realises the map phase, by distributing so called "jobs" to each node. A job contains a python script and the data, that needs processing. Each node writes the code to this and executes it using python3. Obviously, this is a security risk, because running unveried code on some random machine in some data center is inherently flawed, however for reasons of simplicity in the implementation this approach was choosen.

### 3.3 Stage 3: Shuffling

The purpose of this stage is to organise the key-value pairs generated by mapping stage and grouping them by key. Each group is then independently sent to a node. In reality this stage is very complicated, because the cost of transferring data is high and in an optimal case, this should be minimized.

### 3.4 Stage 4: Reduce

This phase ingests the groups of data generated and applies some form of user-specified algorithm to it - in the end, the result is exactly one key-value pair for each group. These pairs are then accumulated and stored on the disk, for the user to retrieve and utilize.

## 4 Software Architecture

### 4.1 Communication / Lifecycle

This section will explain the communication diagram chronologically, starting at the first user input to the cli.

The first step for a user is to create a new job. This is achived, by writing a "job.json" file, as described in Section 3.1 and provide the input and this configuration file to the CLI. The CLI then goes ahead and communicates to the master, that it wishes to upload data. For the sake of minimizing impact on other user, the approach has been chosen to then open a new, sperate port

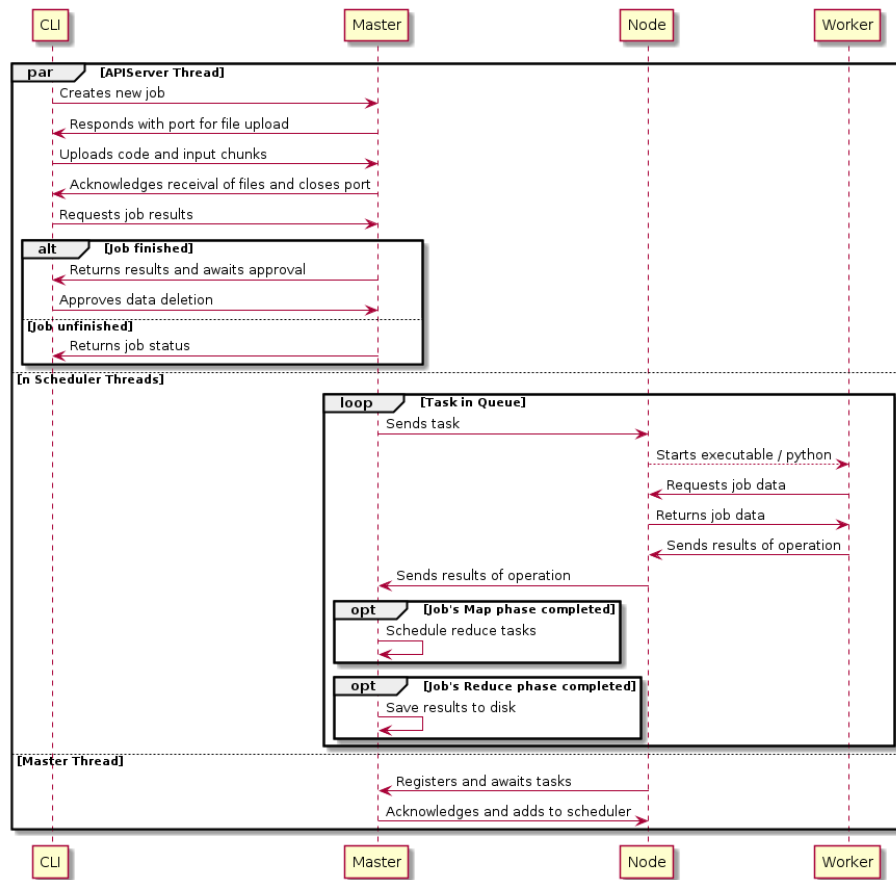


Figure 2: This diagram depicts the messages sent between the different components of this implementation. Furthermore, it is separated into threads, which are running simultaneously on the Master.

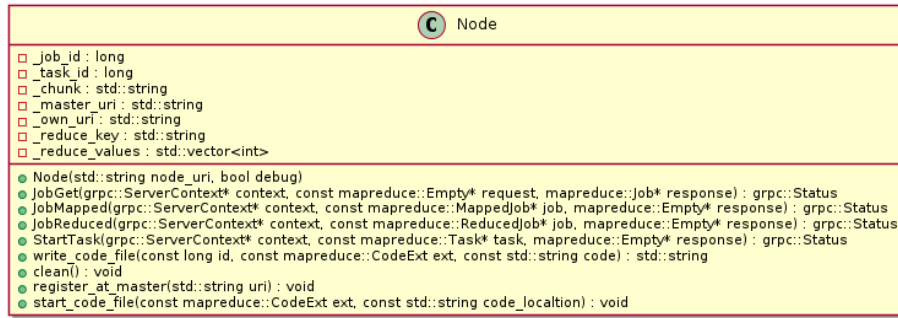


Figure 3: This class diagram shows the relationships, methods and variables of each class of a MapReduce node

on the master to receive data on. After data upload has finished, this data is ingested by the master and eventually scheduled for mapping on a Node.

Once a new node is created on the network, it registers with the master and thus is ready to be included in task scheduling.

Each time a new task is sent to the node, the code of that particular task is written to a temporary space on the disk and executed, a new Worker has been created. This worker then goes ahead and retrieves the job and its associated data from the node, processes it and sends it back to the master, via the node.

Eventually all map task are completed, at this point the master shuffles, sorts and groups the data, to then schedule new reduce tasks. The results of this phase are stored to the disk, to be later retrieved by the user.

## 4.2 Components

### 4.2.1 Node

In comparison to the master, a node is relatively simple. A node is only able to process exactly one task at a time, however that can be multiple nodes for each computer. Optimally, the number of nodes is equal to the number of CPU cores.

### 4.2.2 Master

In the diagram, the master is the central component. The APIServer is responsible of ingesting data from the CLIs, processing it and then sending it off to the master. The FIFOScheduler takes care of scheduling map and reduce jobs on all nodes - in this case FIFO stands for "First in first out", which means that in essence the tasks are put into a queue.

### 4.2.3 CLI

Because of size and complexity, the CLI has not been structured in an object-oriented manner. It is relatively straight-forward with nearly no logic involved.

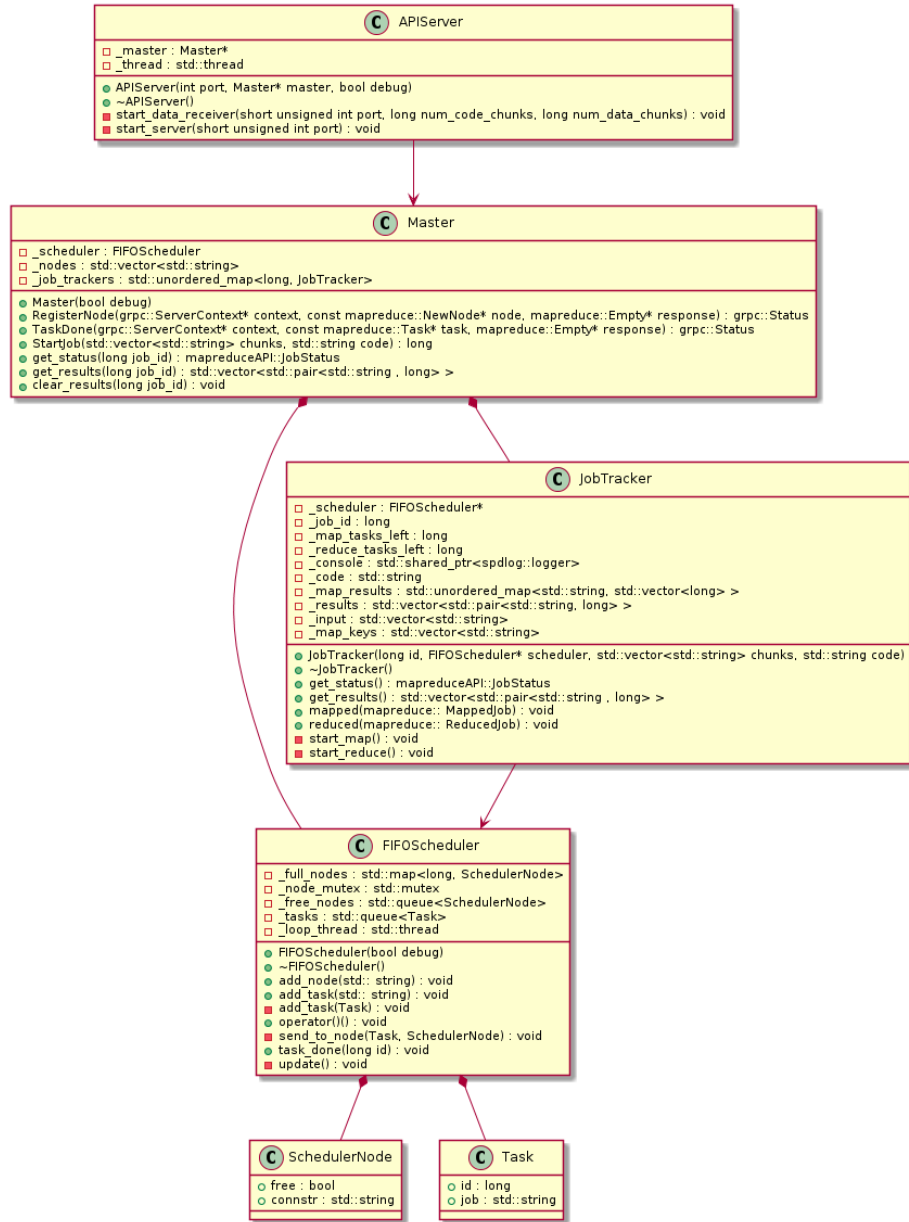


Figure 4: This class diagram shows the relationships, methods and variables of each class of the MapReduce Master



## 5 Usage guide

### 5.1 For End-users

#### 5.1.1 Python API

The python API for writing MapReduce algorithms should be mostly self-explanatory, however here is one example to count word occurrences.

```
1 from MapReduce import Worker
2
3 def map_func(chunk):
4     results = []
5     for word in str(chunk).split(' '):
6         results.append([word, 1])
7
8     return results
9
10 def reduce_func(key, values):
11     return sum(values)
12
13 Worker(map_func, reduce_func)
```

#### 5.1.2 CLI

For interacting with the MapReduce system, a clean and easy command line interface is provided. To start a job, first of all, a job configuration file aka. job.json has to be created. It has to follow the following schema:

```
1 {
2     "input": "shakespeare.txt",
3     "split": "line",
4     "code": "impl.py",
5     "ip": "127.0.0.1",
6     "port": "3000"
7 }
```

Next, the cli needs to do its magic. Just use `./cli start --config job.json` to start a new job. Wait a little while, get a coffee and call `./cli status --id <job_id>` to retrieve either the results, or the current status of the job. That's it!

### 5.2 For System Administrators

Start one master by using `./master`- there are command line parameters provided to change the Master and API ports. Start nodes by using `./node`, master IP and Port can be specified on the command line.