MapReduce-System (37)

Fabian Kleinrad (07), 5BHIF

March 2022

Contents

1 Introduction

In this project, the technology MapReduce is being simulated. Thereby a simple system has been developed to imitate the functionality of a MapReduce application. All of the functionality in this project is written with C++17 and compiled with the help of the meson build system[1]. The communication is based on the TCP protocol and realized using the C++ library asio[2].

Furthermore, to increase performance and usability, protocol buffers[3] are utilized. Protocol Buffers enable the serialization of data structures in an efficient manner, which simplifies working with messages sent between parties in the MapReduce system.

To make use of the advantages of a MapReduce architecture, a simple use-case consisting of counting the number of character occurrences in a plain text document has been implemented. This kind of application was chosen due to its simplicity, which enables the focus of this project to stay on MapReduce rather than a test application.

2 MapReduce

MapReduce is a programming model developed to decrease the computation time of large data sets. It was invented by Google, the reason being the need to compute various kinds of derived data. Examples would be inverted indices or representations of the graph structure of web documents. These applications all have simplicity in common. There are no complex operations needed to accomplish said tasks. Furthermore are, these kinds of processes characterized by accepting large amounts of input data and reducing it to a fraction of itself. MapReduce presents a solution to parallelization, fault-tolerance, data distribution, and load balancing. It is based on the principle of map and reduce, which is eponymous for the technology.[4]

---

[1] *The Meson Build system.* URL: https://mesonbuild.com (visited on 03/30/2022).

[2] *asio C++ Library.* URL: https://think-async.com/Asio/ (visited on 03/30/2022).

[3] *Protocol Buffers.* URL: https://developers.google.com/protocol-buffers (visited on 03/30/2022).

[4] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* URL: https://static.googleusercontent.com/media/research.google.com/de//archive/mapreduce-osdi04.pdf (visited on 04/03/2022).

2.1   Input Data

MapReduce processes unstructured or semi-structured data. The system is designed to accept large amounts of data in bulk. In the first step of the MapReduce process, this data is split into subsets to allow for data distribution. The way this data is divided depends on the implementation and the type of input data present. The result of splitting the raw data is a set of key/value pairs.[5]

2.2   Map

The map function accepts a set of key/value pairs, with the implementation being provided by the user. The result of this phase is once more a set of key/value pairs. These result pairs represent the significant information contained in the input data. These significant data pairs directly influence the result, and all unneeded information is discarded. The name map stems from assigning a quantity attribute representing the value of the resulting pairs to a quality attribute.[6][7]

2.3   Reduce

Sorted key/value pairs get passed to the reduce function, which groups and reduces the set of data points. The logic of this grouping functionality depends on the application of MapReduce. For that reason, the reduce function is also implemented by the user.[8]

---

[5]Thomas König Thomas Findling. *MapReduce - Konzept*. URL: https://dbs.uni-leipzig.de/file/seminar_0910_findling_K%C3%B6nig.pdf (visited on 04/04/2022).

[6]10.

[7]3.

[8]10.

2.4 Additional Phases

2.4.1 Shuffle

In most implementations of a MapReduce model, a shuffle phase is carried out between the map and reduce phase. The shuffle phase sorts the resulting key/value pairs from the mapping phase. This is done to group similar keys into a cluster that can then be reduced by a single worker.[9]

2.4.2 Combine

An additional combine phase can be used after mapping to reduce the network traffic. Thereby, the large number of key/value pairs resulting from the mapping phase are reduced before they are transferred over the network. However, because of this local aggregation of data, it is possible to slow down the instead through shuffling optimized process of reducing data.[10]

---

[9]10.

[10]*MapReduce Tutorial*. URL: `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.pdf` (visited on 04/04/2022).

2.5 Process
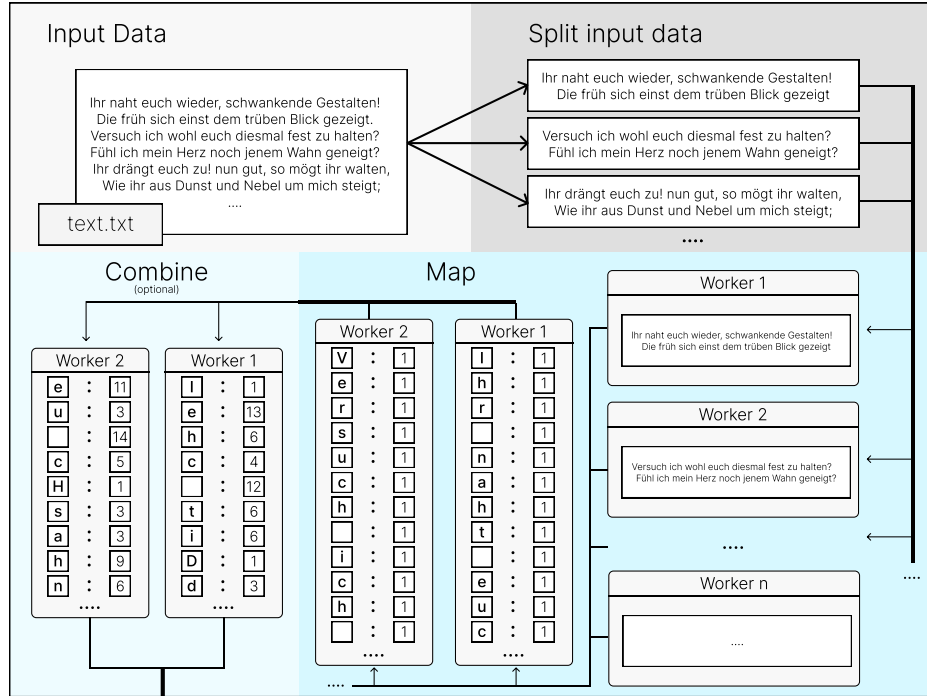
Figure 1: Procedure of a MapReduce operation. The use-case depicted in this example represents one of two possible applications provided by the system developed as part of this project. Continued in figure 2.

Upon receiving the data to be processed, it needs to be split up into chunks to be assigned to workers. Splitting vast amounts of data has to be kept as simple as possible. By reducing the complexity of dividing the input data, performance is increased because the advantage of distributed computation is better utilized.

Individual data chunks then get sent to workers, which process the mapping phase. After receiving the data, which in this case takes the shape of a string, the workers execute the user-defined mapping function for the given input. In this example, the resulting key/value pairs represent individual characters and the number of occurrences.

Due to the nature of this application, the result after mapping the input data are a vast amount of key/value pairs. To decrease network load, a combine phase is being performed. The combine phase acts as a preliminary reduce

phase for the individual workers. Depending on the number of duplicates present in the input data, the combine operation can decrease the key/value pair bundle to a fraction of the original size, further increasing performance.
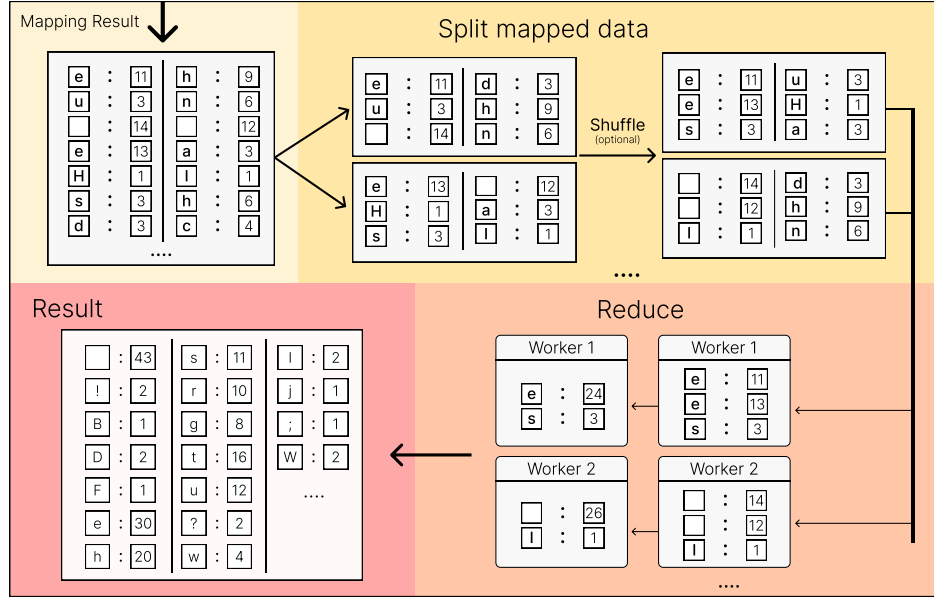
Figure 2: Procedure of a MapReduce operation. Continuation of figure 1.

Each worker sends their mapping result to the master, where the results get accumulated. Once every worker has finished, the master initiates the second phase. Similar to the mapping phase, the accumulated results must be divided up into chunks and distributed to available workers. The data chunks get shuffled to guarantee the most optimal outcome from the reduce phase. The shuffle phase ensures that pairs with the same key get grouped into the same data chunk. If no shuffling is performed, the risk of the split data consisting of only heterogeneous keys remains. This would result in the reduce phase having little to no effect on decreasing the number of key/value pairs.

Upon receiving mostly homogeneous data, the worker executes the user-defined reduce function. In this example, the value of pairs with similar keys gets aggregated. The extant key/value pairs then get sent to the master.

The result of this application being the number of character occurrences in a given input string.

7

3 Classes

The class structure in this project is oriented in the MapReduce solution Disco[11]. In contrast to the realization, disco provides, only three parties are present in this project. Disco uses a central master to act as an interface between the client and the workers, which are referred to as slaves in disco. Additionally, disco has a server role. This server manages several workers and acts as an intermediary between the worker and master. However, the role of the server has not been realized in this implementation to steer away from high complexity and enable the realization of a reliably working model in the time span of this project.

3.1 Master

The master acts as an interface connecting the clients to workers. Additionally, the master acts as the central server accepting and managing all connections. To simplify handling these tasks, the master is supported by the ClientManager and WorkerManager. These run as independent threads and allow for a delimited program structure.

Master
- workerManager: WorkerManager - clientManager: ClientManager
+ acceptConnection(): void

Figure 3: Structure of the Master class.

Figure 3 depicts the structure of the Master class. The master contains instances of ClientManager and WorkerManager, which run separately and take care of the tasks the master needs to handle. This leaves the core master class with an acceptConnection method, which asynchronously accepts asio clients, which are then delegated to either the WorkerManager or ClientManager.

---

[11]*Disco Documentation*. URL: https://disco.readthedocs.io/en/develop/index.html (visited on 04/04/2022).

3.1.1 ClientManager

The ClientManager handels MapReduce client connections. The client refers to a human user of the MapReduce system.

```
                          ClientManager
  ──────────────────────────────────────────────────────────
  + join(client:shared_ptr<ConnectionObject>): void
  + leave(client:shared_ptr<ConnectionObject>): void
  + registerJob(job_id:int, client_id:int): void
  + sendResult(job_id:int, result:map<string, int>): void
  + generateID(): int
```
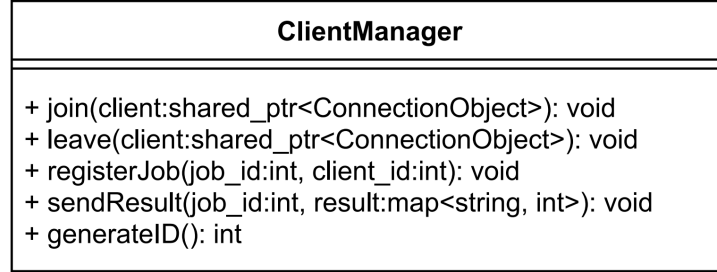
Figure 4: Structure of the ClientManager class.

For the master to be able to add connections, the ClientManager provides a *join* function. To support disconnecting clients, a *leave* function is available. These functions take an instance of a connection represented by a shared pointer on a connection object. This means the ClientManager can keep track of ongoing client-master connections and act as a communication interface between them.

Additionally the *registerJob* and *sendResult* function are depicted in figure 4. These two functions handle job management on the client-side of the master. Thereby ongoing jobs are being tracked, and upon finishing, the client gets sent the resulting data. Lastly, the class contains a *generateID* function, which generates unique ids to identify clients.

3.1.2 WorkerManager

The WorkerManager works analogously to the ClientManager, whereas the WorkerManager handles the master-worker communication. The WorkerManager also provides *join* and *leave*, which are used by the master to delegate connections.

Similar to the ClientManager, the WorkerManager provides a *assignJob* function. This function acts as the interface between the ClientManager and WorkerManger. The functionality of distributing a job received from a client is thereby realized. To allow for results to be forwarded to the WorkerManager, the functions *mapResult* and *reduceResult* exist. These methods are used by the ConnectionSession, where communication takes

```
                    WorkerManager
_____
+ join(client:shared_ptr<ConnectionObject>): void
+ leave(client:shared_ptr<ConnectionObject>): void
+ mapResult(job_id:int, worker_id:int, result:vector<pair<string, int>>): void
+ reduceResult(job_id:int, worker_id:int, result:map<string, int>): bool
+ assignJob(Job job): bool
+ reAssignTask(worker_id:int): void
+ generateID(): int
```

Figure 5: Structure of the WorkerManager class.

place.

To increase fault-tolerance, WorkerManager provides a $reAssignTask$ function, which is used when a worker fails to perform its task correctly. The $generateID$ function works the same as in the ClientManager and is used to identify workers uniquely.

3.2 Client

The Client class represents the link between the user and the MapReduce system. It implements a user interface to enable the user to communicate with the system and place job requests.

```
                         Client
_____
- client_id: int
_____
+ signOn(): void
+ signOff(): void
+ sendJob(job:Job): void
+ printResultsPlain(sorted:bool): void
+ printResultsHistogram(sorted:bool): void
```

Figure 6: Structure of the Client class.

In order to initiate the connection to the master server, the ClientManager implements a $signOn$ function. Similarly, the class also provides a $signOff$ function to terminate the connection. The function $sendJob$, which is depicted in figure 6 is used to place job request. To

enable visualization of job results, the functions *printResultsPlain* as well as *printResultsHistogram* are implemented by the ClientManager.The Client class represents the link between the user and the MapReduce system. It implements a user interface to enable the user to communicate with the system and place job requests.

3.3 Worker

The Worker is the powerhouse of the MapReduce system. It handles the computation of tasks that are assigned by the master. For this purpose, the Worker implements the functions *handleMap* and *handleReduce*, which can theoretically be replaced by any logic accepting and returning the same data types as MapReduce intends.

Worker
- worker_id: int
- handleMap(type:int, data:string, job_id:int): void - handleReduce(type:int, data:KeyValuePairs, job_id:int): void + signOn(): void + signOff(): void

Figure 7: Structure of the Worker class.

The *handleMap* and *handleReduce* functions take the type of job, data, and job id as parameters. Whereas the *handleMap* function accepts the raw data in the shape of a string, the *handleReduce* function is provided with a set of key/value pairs.
To control the connection to the master, the functions *signOn* and *signOff* are provided, which work analogously to the functions in the ClientManager.

4 Helper Classes

4.1 Pipe

To centralize the asio network connection, the pipe class is used. It provides a network interface to simplify sending and receiving messages. The pipe

uses an asio socket to transfer messages. Additionally, the class ensures the thread-safe access to the socket.

Pipe
- socket: asioSocket - is_closed: bool
+ sendMessage(message:Message): void + recieveMessageType(): MessageType + operator>>(message:Message): void

Figure 8: Structure of the Pipe class.

4.2   ConnectionObject

To represent the client and worker connections, the ConnectionObject is used. It provides structure for connections and stores vital information used for handling ongoing connections.

ConnectionObject
+ id: int + is_available: bool + last_active: timepoint
+ sendMessage(Message): void + isConnected(): bool + closeConnection(): void

Figure 9: Structure of the ConnectionObject class.

To identify a ConnectionObject, it contains an id, as depicted in figure 9. This id is not unique across all ConnectionObjects due to the distinction between worker and client connections. Additionally, the ConnectionObject stores information about the state of availability. However, this attribute is only used in worker connections.
In an effort to increase fault-tolerance and detect broken connections the

ConnectionObject stores the time, when the last message was received. It enables the server to periodically check connections and end them if no response has been received.

4.3 ConnectionSession

The ConnectionSession handles the communication between the master and workers as well as clients. The basic structure of implementing asio, multi-user communication this way, has been inspired by the asio-documentation[12].

ConnectionSession
- sendMessage(Message): void - isConnected(): bool - closeConnection(): void + start(): void
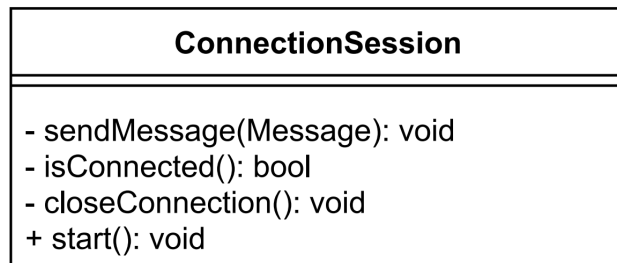
Figure 10: Structure of the ConnectionSession class.

To start the process of receiving and sending messages the ConnectionSession provides a *start* function. Additionally the ConnectionSession implements the functions *sendMessage*, *isConnected* and *closeConnection* from the ConnectionObject class.

4.4 Job

The Job class allows for uniform structure when processing jobs.

Job provides a variety of constructors to simplify using the Job class in all phases of the MapReduce process. It contains an id that is unique across all jobs to identify jobs. Furthermore, a job stores information about the type of job and current status. The different states of a job are defined in the *JobStatus* enum, depicted in figure 12.

---

[12]*chat_server.cpp*. URL: https://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio/example/cpp11/chat/chat_server.cpp (visited on 04/05/2022).

```
┌─────────────────────────────────────────────────────┐
│                        Job                          │
├─────────────────────────────────────────────────────┤
│ + id:int                                            │
│ + type:JobType                                      │
│ + data:string                                       │
│ + status:JobStatus                                  │
│ + results: vector<pair<string, int>>                │
│ + mappers: int                                      │
│ + reducers: int                                     │
├─────────────────────────────────────────────────────┤
│                                                     │
│ + Job(type:JobType, data:string)                    │
│ + Job(type:JobType, data:string, id:int)            │
│ + Job(activeJob:ActiveJobStruct, worker_id:int)     │
│ + Job(activeJob:ActiveJob)                          │
│                                                     │
└─────────────────────────────────────────────────────┘
```
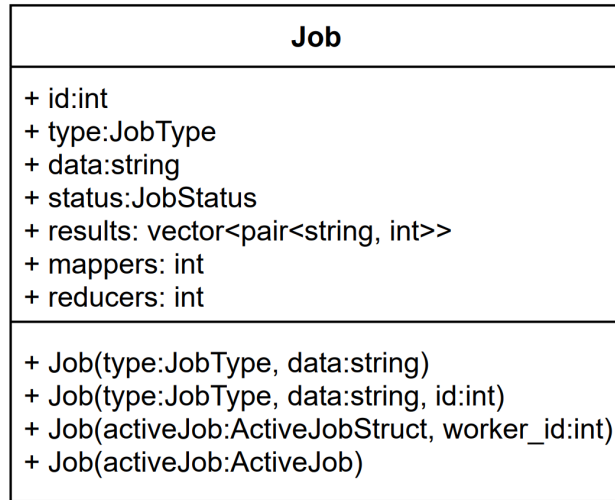
Figure 11: Structure of the Job class.

4.4.1 ActiveJob

To allow for tasks to be reassigned upon failure, the ActiveJob class provides all information. Due to the reciprocal relationship between ActiveJob and Job, it is necessary to split the ActiveJob into two classes. The class ActiveJob, depicted in figure 13 is the derived from the class ActiveJobStruct, depicted in figure 14.

The class ActiveJobStruct provides the structure of an active job. Thereby information on all phases of the MapReduce process is stored in an ActiveJob. Additionally, to the attributes contained in the Job class, it also allows storing the data each worker is currently processing.

14

```
+----------------------------+
|    <<enumeration>>         |
|      JobStatus             |
+----------------------------+
| job_new                    |
| job_queuedMap              |
| job_queuedReduce           |
| job_mapping                |
| job_mapped                 |
| job_reducing               |
| job_done                   |
+----------------------------+
```
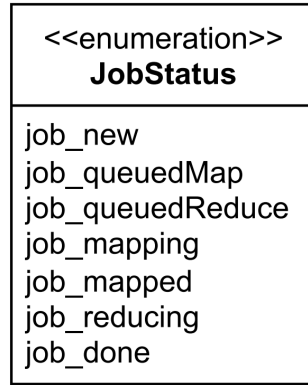
Figure 12: Enum containing job states.

```
+----------------------------+
|         ActiveJob          |
+============================+
|                            |
| + ActiveJob()              |
| + ActiveJob(job:Job)       |
+----------------------------+
```
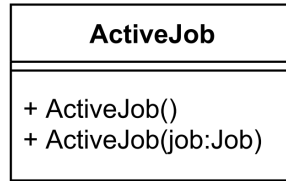
Figure 13: Structure of the ActiveJob class.

4.5   MessageQueue

Due to communication taking place between multiple parties, it is crucial to serialize the processing of messages. This concept can be realized by using a queue, in this project realized through the MessageQueue class, depicted in figure 15.

To implement the functionality of a queue, MessageQueue uses the queue data type provided by c++. Additionally, MessageQueue ensures that access to the queue happens in a thread save manner.

4.5.1   QueueItem

To store the data of an item in the queue, the QueueItem class provides the needed structure. The QueueItem stores data that can be received by the master from workers.

```
+-----------------------------------------------------------------+
|                        ActiveJobStruct                          |
+-----------------------------------------------------------------+
| + workerData:map<int, string>                                   |
| + workerReduceData: map<int, vector<pair<string, int>>>         |
| + results: vector<pair<string, int>>                            |
| + reducedData: map<string, int>                                 |
| + job_id: int                                                   |
| + is_active: bool                                               |
| + status: JobStatus                                             |
| + type: JobType                                                 |
+-----------------------------------------------------------------+
|                                                                 |
| + addWorker(worker:int, data:string): void                      |
| + addWorker(worker:int, data:vector<pair<string, int>>): void   |
| + removeWorker(worker:int)                                      |
| + addResults(results:vector<pair<string, int>>): void           |
| + addReducedData(data:map<string, int>): void                   |
| + contains(worker:int): bool                                    |
| + getWorkerData(worker:int): string                             |
+-----------------------------------------------------------------+
```

Figure 14: Structure of the ActiveJobStruct class.

```
+-------------------------------------+
|            MessageQueue             |
+-------------------------------------+
| - queue: queue<QueueItem>           |
+-------------------------------------+
| + push(item:QueueItem): void        |
| + pop(): QueueItem                  |
| + isEmpty(): bool                   |
+-------------------------------------+
```

Figure 15: Structure of the MessageQueue class.

4.6 MessageGenerator

To simplify working with protobuf messages, the MessageGenerator provides various functions that return the respective message.

QueueItem
+ type: MessageType + jobType: JobType + job_id: int + dataRaw: char* + dataReduce: vector<pair<string, int>> + dataResult: map<string, int>
+ QueueItem(type: MessageType)

Figure 16: Structure of the QueueItem class.

MessageGenerator
+ SignOn(id:int, type:ConnectionType): SignOn + SignOff(id:int, type:ConnectionType): SignOff + Confirm(id:int, type:ConnectionType): Confirm + Authentication(type:ConnectionType): Authentication + Assignment(id:int, type:ConnectionType): Assignment + JobRequest(type:JobType, data:string, mappers:int, reducers:int): JobRequest + TaskMap(type:JobType, data:string, job_id:int): TaksMap + Ping(): Ping + ResultMap(result:vector<pair<string, int>>, job_id:int): ResultMap + TaskReduce(type:JobType, data:vector<pair<string, int>>, job_id:int): TaskReduce + ResultReduce(result:map<string, int>, job_id:int): ResultReduce + JobResult(job_id:int, result:map<string, int>): JobResult

Figure 17: Structure of the MessageGenerator class.

5 Class-diagram

6 Network communication

6.1 protobuf

Protobuf is used to send structured data over an asio TCP connection. In this project, the version proto3 is being used. Protocol Buffers allow for defining the structure of a message. This happens in a '.proto' file, which is then compiled with the help of protoc. Protobuf supports a variety of different programming languages. In the case of C++, protobuf generates header files.
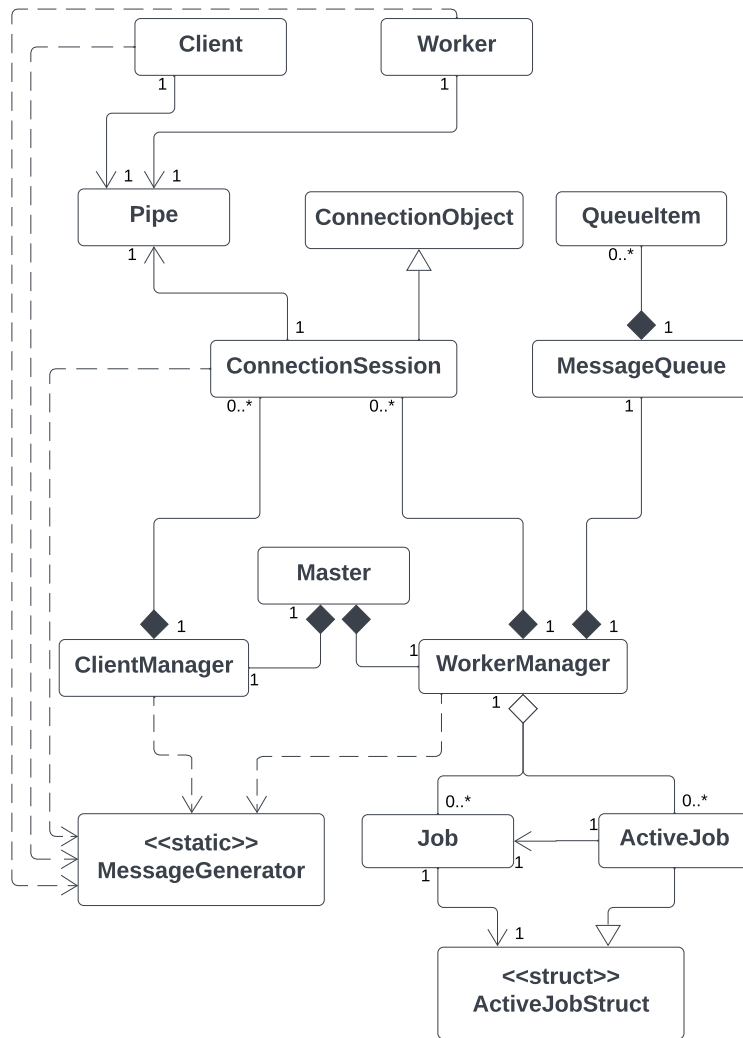
Figure 18: UML class diagram representing the structure of this project.

These contain the code necessary to instantiate predefined message objects and serialize or parse them.

6.2 Messages

6.2.1 Authentication

An Authentication message is used to identify the type of connection. In this project, there are two possible types, client or worker. Upon successfully connecting to the master, an Authentication with the respective connection type is sent.

6.2.2 Assignment

The Assignment message is used to assign ids to workers and clients. An Assignment is sent in the ConnectionSession after a new connection gets accepted by the master. It contains the id that the connection partner gets assigned and the type of connection, either client or worker.

6.2.3 SignOff

A SignOff message is either sent from a worker or client upon encountering an error that results in the program's termination. This is performed to let the master know when a client or worker is no longer available. The SignOff message can also be sent from master to client or worker to force them to stop. This behavior can be observed when the master program terminates. A SignOff contains the id of client or worker that is about to disconnect and the respective connection type.

6.2.4 Confirm

A Confirm is used the ensure that important messages have been delivered. An example of its application is to confirm that an Assignment has been delivered. This is especially important because all later communication is based on ids. If a client or worker has not been assigned an id, it would not be possible for future messages to be accepted by the master due to the lack of identification of the sender.

6.2.5   TaskMap

The TaskMap message gets sent from master to worker to assign a task to the respective worker along with the needed information to perform it. The message contains the job id, which uniquely identifies the job. This represents essential information to match the result sent by the worker to the original job. Furthermore, the type of job and data is being transferred.

6.2.6   TaskReduce

The TaskReduce message works analogously to the TaskMap message. However, instead of the data consisting of a string, that being the case with TaskMap, in TaskReduce, the data gets transferred in the shape of key/value pairs.

6.2.7   ResultMap

ResultMap is a message that is sent by the worker upon finishing the mapping task. It contains the mapping result in the form of key/value pairs. Furthermore, it contains the job id to enable the master to match tasks to jobs.

6.2.8   ResultReduce

ResultReduce gets sent by the worker upon finishing the reduce task. The result is transferred in the shape of key/value pairs. Additionally, it contains the id of the job.

6.2.9   JobRequest

A JobRequest is sent from client to master. It contains information on the job to be performed and the data to be processed. Furthermore, the client can specify the number of mapping workers and reducing worker it desires.

6.2.10   JobResult

The JobResult is the reply to the client from the master after finishing the job. It contains the result of the MapReduce operation in the form of key/value pairs.

6.2.11 Ping

The Ping message is used for checking connections. After the master detects a worker being inactive for an extended period of time it sends pings in one-second intervals until a reply is received or five pings have been sent. If a worker receives a Ping, it replies with a Ping message.
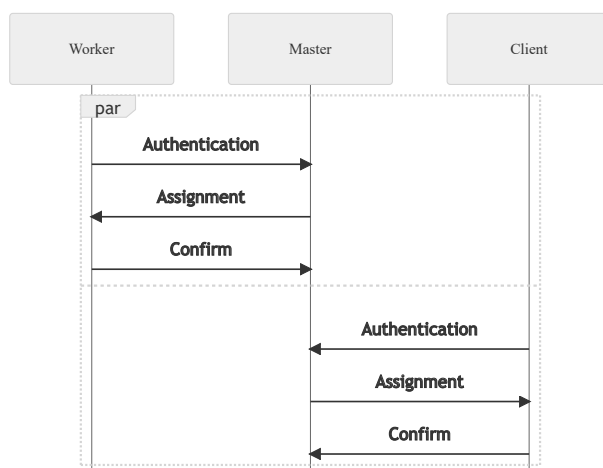
6.3 Sequence Diagram

Figure 19: Sequence diagram depicting the network traffic of the MapReduce system. Continued in figure 20

7 Parallelization and fault-tolerance

The characteristic of a MapReduce system is its ability of parallel and fault-tolerant processing. This is also, to an extent, realized in this project. Upon starting a MapReduce master it accepts a variable amount of workers and clients. If a client places a job request while no workers are available, then the job gets queued by the WorkerManager and assigned to workers when available. This allows multiple clients to place jobs simultaneously without discarding jobs unable to process at the current time.

The premise of MapReduce being distributed computation of one whole operation is vulnerable to failing at the weakest link in the chain. To
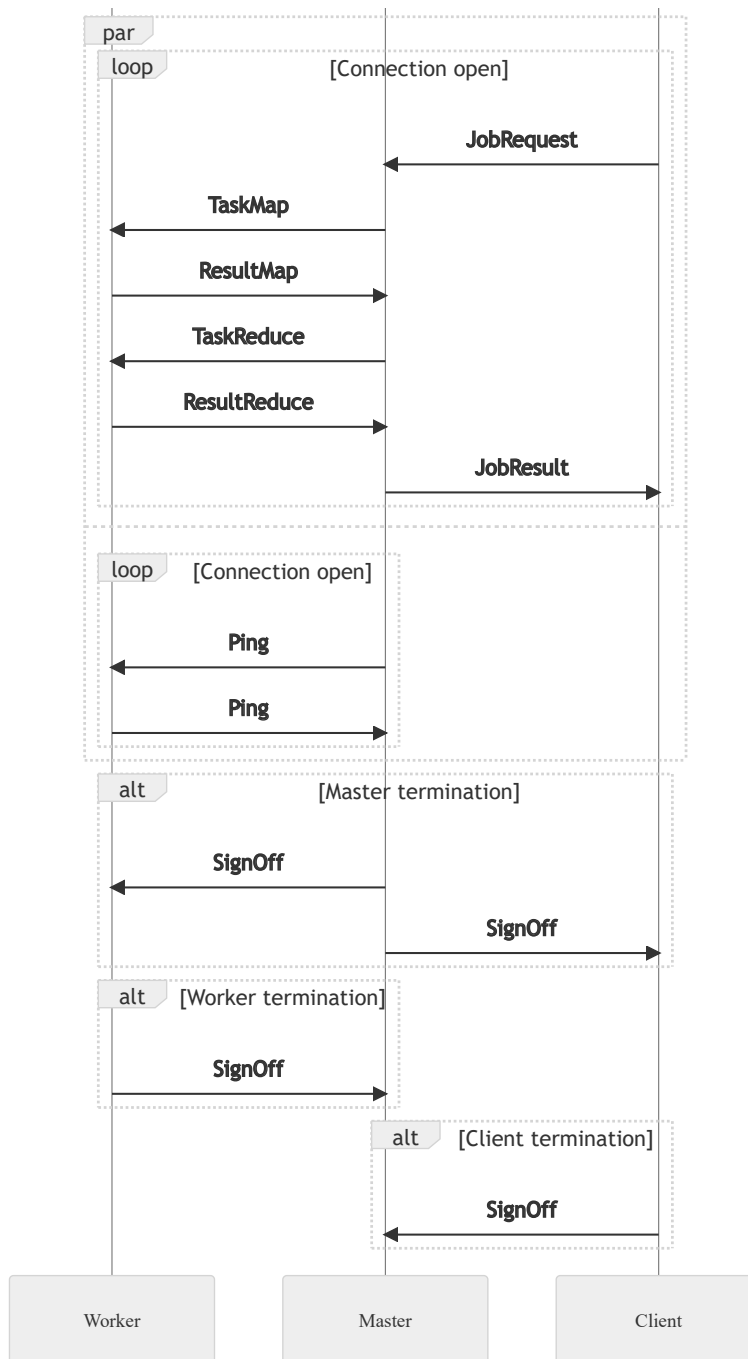
par

loop   [Connection open]

JobRequest

TaskMap

ResultMap

TaskReduce

ResultReduce

JobResult

loop   [Connection open]

Ping

Ping

alt   [Master termination]

SignOff

SignOff

alt   [Worker termination]

SignOff

alt   [Client termination]

SignOff

Worker   Master   Client

Figure 20: Sequence diagram

22

prevent this from happening, the master can reassign failed tasks. These failed tasks behave the same as jobs. If no workers are currently available, they get queued. In the case of multiple tasks failing from the same job, they get merged into one job and distributed to workers once available.

There are different reasons why a worker fails his task. The easiest for the master to detect is if a SignOff message is received from a worker that is currently assigned a task. If the worker is unable to communicate over the network, the master will timeout the connection after 15 seconds since the last response and reassign the task.

8   Usage

To use MapReduce, clone the repository Kleinrad/kleinrad_project[13]. After all dependencies listed in section 9 are satisfied, compile the project via meson. To ensure a successful build use meson 0.59.1+. Once compiled the three executables have been generated: *mr_worker*, *mr_master*, and *mr_client*.

Start the *mr_master*, which then will wait for incoming connections. After that, it is possible to start as many *mr_worker*s and *mr_client*s as needed. To place a request, use the command line interface provided by the client.

---

[13]$kleinrad_project$. URL: `https://github.com/Kleinrad/kleinrad_project` (visited on 04/05/2022).

8.1 Command Line

The Client supports a variety of command-line arguments and commands.

8.1.1 Arguments

ip - specify ip-address of MapReduce master

port - specify MapReduce port

8.1.2 Commands

Furthermore, the client provides a command-line interface to place job requests and display job results. The available commands consist of:

help - print available commands, along with parameters

quit - end client session

send - send a job request

print - print last job results

The *send* and *print* commands accept parameters to specify the request further.

send $< jobType > \ [-f] \ < data >$

The job type can either be 0 or 1. These represent different reduce operations. Job type 0 counts the number of character occurrences, whereas job type 1 counts the number of word occurrences. The optional parameter $-f$ can be used to provide a txt file as the input data. If the parameter $-f$ is non-existent, the *data* can be a string of variable length.
To display the results returned to the client, the *print* command can be used.

print $[-s] \ < printType >$

The print command has the optional parameter $-s$. Thru the use of this parameter, the output will be sorted by the value of the results. Furthermore, the print command requires a *printType* to be specified. This can either be 0 or 1. Print type 0 returns a list containing the results of the last job, along with the percentage share per occurrence relative to the whole input text. Print type 1 prints the results as a Histogram.

24

8.1.3 Configuration

The configuration is performed via json. To use json in c++ the nlohmann
json library[14] is used. With this json file, the port of the master and the ip
and port a worker connects to are specified.

9 Libraries

Purpose	Technology
Build Tool	Meson
Command line interface	CLI11
Configuration files	json
Data serialization	protobuf
Logging	spdlog
Network Communication	asio
Programming Language	C++ 17

Table 1: In this table used external technologies are listed.

---

[14]*JSON for Modern C++.* URL: https://github.com/nlohmann/json (visited on
04/05/2022).

10   Project Structure

```
/
├── LICENSE
├── meson_options.txt
├── meson.build
├── README.md
├── CHANGELOG.org
├── config.json
├── include
│   ├── client.h
│   ├── clientmanager.h
│   ├── connectionobject.hpp
│   ├── connectionsession.h
│   ├── job.hpp
│   ├── master.h
│   ├── messageQueue.hpp
│   ├── pipe.hpp
│   ├── protoutils.hpp
│   ├── worker.h
│   └── workermanager.h
├── src
│   ├── Message.proto
│   ├── client.hpp
│   ├── clientmanager.hpp
│   ├── connectionsession.cpp
│   ├── master.cpp
│   ├── worker.cpp
│   └── workermanager.cpp
├── doc
│   ├── doc.pdf
│   ├── doc.tex
│   └── references.bib
└── build
```

References

[1] *asio C++ Library*. URL: https://think-async.com/Asio/ (visited on 03/30/2022).

[2] *chat_server.cpp*. URL: https://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio/example/cpp11/chat/chat_server.cpp (visited on 04/05/2022).

[3] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. URL: https://static.googleusercontent.com/media/research.google.com/de//archive/mapreduce-osdi04.pdf (visited on 04/03/2022).

[4] *Disco Documentation*. URL: https://disco.readthedocs.io/en/develop/index.html (visited on 04/04/2022).

[5] *JSON for Modern C++*. URL: https://github.com/nlohmann/json (visited on 04/05/2022).

[6] *kleinrad_project*. URL: https://github.com/Kleinrad/kleinrad_project (visited on 04/05/2022).

[7] *MapReduce Tutorial*. URL: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.pdf (visited on 04/04/2022).

[8] *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers (visited on 03/30/2022).

[9] *The Meson Build system*. URL: https://mesonbuild.com (visited on 03/30/2022).

[10] Thomas König Thomas Findling. *MapReduce - Konzept*. URL: https://dbs.uni-leipzig.de/file/seminar_0910_findling_K%C3%B6nig.pdf (visited on 04/04/2022).