Department of Computer Science

Master's Thesis

The Holumbus Framework

Distributed computing with MapReduce in Haskell

Submitted on:
February 26th, 2009

Submitted by:
Stefan Schmidt
Dipl. Ing. (FH)
Preesterkoppel 2
25560 Schenefeld, Germany
Phone: +49 (0) 48 92 / 7 32
E-Mail: stefanschmidt@web.de

Supervised by:
Prof. Dr. Uwe Schmidt
Fachhochschule Wedel
Feldstraße 143
22880 Wedel, Germany
Phone: +49 (0) 41 03 / 80 48 45
E-Mail: si@fh-wedel.de

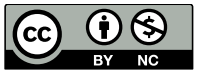
The Holumbus Framework

Distributed computing with MapReduce in Haskell

Master's Thesis by Stefan Schmidt

Abstract

Although current computers are very fast, the processing of hundreds of gigabytes of data still may take several hours or even days on a single processor system. When using multiple computers in parallel for the computation, the costs for design and implementation of a distributed system are very high. The MapReduce concept - developed at Google Inc. - encapsulates the efforts of parallelism in a single library. It provides a simple but powerful framework to build distributed applications without having deeper knowledge of parallel programming. This thesis presents a MapReduce framework provided with an efficient communication architecture and a distributed data storage solution written in Haskell, a purely functional programming language.



Layout done with the help of Timo B. Hübel's template, LaTeX 2ε, KOMA-Script and BibTeX.

Contents

List of Figures

List of Listings

1 Introduction

1.1 Motivation

In the last twenty years, the main focus of the world economy has shifted from the production and trading of industrial goods to the creation and transfer of information. This process was accompanied by fundamental changes in the economical and social systems and led to the term "information age" [LV03].

The essential condition for this development was the ability to store and transfer information electronically, more precisely with the help of computers. The development of the World Wide Web in 1993 has sped up this process. Today nearly every newspaper, book and scientific paper is published electronically via the Internet. In the last five years, even non-textual media like audio and video documents have become widely accessible.

The existence of such a huge information source has led to a new problem, the filtering of unimportant information from the important one. This is commonly known as "information overload" [EM02].

In computer science, the field of information retrieval [BYRN99] tries to compensate these negative effects. New tools are developed for filtering and processing the data individually, to get only those pieces which are really important for the user. Most of these approaches accomplish this by processing all available documents.

Modern computers provide high processing powers and Moore's Law [Moo65] can still be applied to the progress in development, but the manipulation of multiple gigabytes of data is still not feasible on a single PC. The computation has to be split between several machines.

A distribution of work among a network of computers is more complicated than developing an application which runs on a single computer. The programmer has to find a way to divide the work - a task that might not be straightforward. Afterwards he has to implement different programs and coordinate their communication. The risk for the occurrence of failures and the time for their correction will increase significantly in such a system. Even if all programs contain no functional errors, it is not assured that they can interact properly.

In spite of all this work, there is only a small advantage for further data manipulation tasks. In general it is very likely that many parts of the system will have to be rewritten when the main algorithm changes.

All these problems lead to the idea of encapsulating the mechanisms for splitting the work between several computers in a library. With such a framework, this problem has to be solved only once and can be used for different data processing tasks. When building a system for manipulating huge amounts of data, the programmer only has to provide his own application code and does not have to mind the pitfalls of a distributed system. A very powerful but simple approach for implementing such a framework is provided by the MapReduce concept [DG04].

1.2 Scope

The major aim of this work is to develop and implement a distributed MapReduce library using the purely functional programming language Haskell [Has]. It has to provide the application programmer with a simple but flexible interface for building his own distributed MapReduce systems. This software is part of the Holumbus framework which is developed at the University of Applied Sciences in Wedel, Germany. Because of this the MapReduce library is named "Holumbus-MapReduce".

At the moment only few projects deal with the implementation of distributed systems in Haskell, a deeper research in this topic is necessary. These investigations led to the implementation of two further libraries, "Holumbus-Distribution" and "Holumbus-Storage".

Holumbus-Distribution provides the programmer with modules to build a flexible and powerful communication system in Haskell. Holumbus-Storage adds the functionality of a distributed data storage solution to the MapReduce software.

The practical use case for all three libraries is the implementation of a distributed webcrawler and indexer for the “Holumbus-Searchengine” library.

1.3 Related Work

Although the concept of MapReduce was invented over fifty years ago in the process of the research in functional programming languages like LISP, its adaptation to a distributed environment took place no more than five years ago. The first reported system was build at Google Inc. [DG04] for the parallel construction of the large indices used by the Google searchengine.

Because the Hayoo! searchengine [Hay] written by Timo B. Hübel and Sebastian Schlatt at the University of Applied Sciences in Wedel needed an improvement of its existing index generation process, the distributed MapReduce approach was chosen.

Although there is a close relation to the Holumbus-Searchengine the codebase of the MapReduce framework is independent.

1.4 Outline

Before starting with the main topic of this thesis, a general introduction to concurrency, distributed systems and MapReduce is given in chapter 2. In the following chapter 3 the existing parts of the Holumbus framework are introduced and the main features of the required systems are analyzed. In chapter 4 the actual implementation of the software is described. At the beginning a brief illustration about the general project structure is given and the main modules are named. After this follows a detailed description for every elementary component. Chapter 5 closes this work by comparing the actual implementation with the previously defined requirements. If some requirements are not met, a brief outlook for future work is given.

The first part of the appendix describes some interesting example programs implemented during the development process while the second part gives a general impression how to use this software for the creation of customized applications.

2 Fundamentals

This chapter introduces the basic concepts which are necessary for understanding this thesis. The first section gives a general overview about concurrency and parallelism. The second part discusses the ideas of distributed systems. A specialized kind of distributed system, a MapReduce system, is introduced in the last section.

2.1 Concurrency and Parallelism

2.1.1 Definition

In the common opinion, a program consists of a sequence of operations which are executed one after another. This was true in the earlier days of computer programming. But today, this model has been expanded.

Most modern programs and operating systems are able to do more than one single operation at the same time or at least create the impression of parallel program execution. Programs with a graphical user interface (GUI) are a good example. While doing some computations in the background, a GUI-program should still be able to react to user input. If the program does not respond, the user might think it is crashed and terminate the computation altough it is still running.

The major requirement to do multiple actions at the same time, is the ability to run operations concurrently. Concurrency in this context means that the code of the

programs can be split into independent segments. The created parts ideally share no common resources and therefore the order of their computation does not affect their result. They can be processed one after another or alternately.

Parallelism is a special case of concurrent execution. It implies the possibility of running independent parts simultaneously at the same time. This is only possible, if the computer system consists of more than one processing unit. This is true for multiprocessor or multicore systems. On a single processor system, parallelism can not be reached, but is simulated for the user by switching between multiple actions in a short amount of time. In some parts of the literature, parallelism is referred as "real concurrency" [Sno92].

2.1.2 Processes and Threads

There are two levels of concurrency, the concurrent processing of separate programs and the processing of different parts of a single program.

The first scenario is handled by the operating system. When a program is started, the system creates a process for it. Among the program code and data, a process contains information about the used ressources. In general, processes do not share their ressources, for example each process has its own address space and cannot interfere with the memory of other programs.

To execute different parts of one process concurrently, it is separated into threads. Unlike processes, the threads of one program share their ressources and memory. The decision which parts of a program are executed in separate threads is taken by the programmer. There are two different types of threads which determine, who is responsible for their management.

Kernel threads are scheduled by the operating system, whereas user threads are maintained by the thread library of the application program. Kernel threads can take benefit of the operating system, especially its ability of having full control over the physical processor. One of their drawbacks is the increased scheduling overhead used by the operating system. In general the creation of user threads consumes less system ressources.

2.1.3 Communication

Previous subsections said that concurrent processes and threads are independent of each other. Regarding their computational purpose this is true, but the output of one process can be used as input for another process. Such a scenario is widely known as the producer-consumer-problem.

Even if two processes are totally independent regarding their computation, they have to share the ressources of the underlying computer system. For a proper use, most of the system ressources can only be accessed by one process at a time, for example the control over the system's standard output channel. The access to those ressources has to be controlled.

These two examples show that there is a strong need for two threads to communicate with each other. Communication can be synchronous or asynchronous. Synchronous communication requires that two processes have to wait until both are ready to exchange data. In an asynchronous environment, the sending process does not have to wait for the receiver. The data is stored in a buffer until the receiving process reads it. This type of communication does not need a synchronization, but the sender does not know when its message will be actually received or processed.

There are several techniques for realizing inter-process communication. Two of them are shared memory and message passing. Although the first one is mainly used in combination with synchronous communication and the second method with asynchronous, both of them do not predefine possible type of data exchange.

Shared Memory

As the name might imply, using shared memory as a communication method means that the communication partners share a distinct amount of memory. Both can read and write to the memory and use it to exchange data.

Since two threads of the same process always run in the same address space, there is no need for extra effort to implement shared memory. A global data object is suitable for this purpose. But when using different processes, it is necessary that the operating system provides an interface for installing and accessing shared memory between them.

Reading and writing the shared memory needs to be coordinated between the communication partners. It is very likely that the sequence of reading and writing operations on the shared data is significant, for example a data object can only be read from the memory, if it was written before, otherwise the result of the reading operation cannot be predicted. The violation of these rules is known as race condition [TW06].

The coordination between the processes has to ensure that only one process at a time can access the shared memory. A set of instructions, which can only be executed by a well-defined number of processes at the same time is a critical section.

The access to such a section can be controlled by a semaphore [Dij02]. In most implementations this is a shared variable holding an integer value. This number indicates how many processes are currently executing the critical section. Everytime a

process wants to enter the critical section, it checks the variable. If the value is smaller than the maximum number of allowed threads, the variable is increased and the thread enters the section. Otherwise it waits until the value of the variable becomes smaller than the threshold again.

When a process leaves the section, it decreases the variable to give other processes the opportunity to access the shared memory. This concept assumes that the functions for entering and leaving the critical section are atomic. That means a process cannot be interrupted while executing these operations.

Message Passing

Message passing is a more abstract communication concept than shared memory. In some use cases its underlying implementation might rely on shared data objects and access control via semaphores. But in most cases the programming interface hides the underlying communication mechanisms to the application programmer.

As the name of this method implies, two processes communicate with each other by exchanging messages. Every process, which should be able to receive messages, has a mailbox for storing incoming messages. The receiving thread frequently checks the mailbox for new messages, reads them and reacts to their content. After a message is read, it is deleted from the mailbox. The mailbox itself generally consists of a FIFO-buffer for storing new data packages. To prevent buffer overflows, the mailbox usually has a fixed size. When a process wants to transmit information to another process, it puts a copy of the message in the mailbox of the receiver. If the receiver's mailbox is full or not available, there are different possibilities on how to react. Either the sending function waits until it can deliver the message or it returns with an error and lets the application programmer decide how to handle this event.

The abstract functionality of this communication method makes it applicable to more use cases than shared memory. Like shared memory, it can be used between processes and threads which are running on the same physical computer. But it is expansible to data exchange over a network connection. This type of process communication is one of the key aspects of distributed systems.

2.2 Distributed Systems

2.2.1 Definition

Andrew S. Tanenbaum has defined a distributed system as a set of independent processing units which behave to the user as a single system [TS07]. Other scientists

extend this concept with the need of a transport system to link the system nodes.

Although this definition is not very specific, it covers all important use cases of distributed systems in computer science. A set of computers which are connected by a network and share some parts of their work can be considered as a distributed system like a single computer which consists of multiple processors. Today, distributed systems can be found in many applications.

2.2.2 System Features

The general idea for building a distributed system is that sharing the work between multiple processors is faster than doing it on a single processor. A distributed system is capable of parallel execution, not just sequential execution like on a single processor system.

The performance and usability of a distributed system is described by four indicators [CDK01]:

- Openness
- Transparency
- Scalability
- Fault tolerance

The criteria of openness judges the ability of the distributed system to interact with other systems or the user himself. A good way to fulfil this requirement is to open the interfaces for outside communication, so that external components or users can access the system easily. One consequence of openness might be that the system is able to be extended by new components, even without the need to shut it down first.

As said in the definition by Tanenbaum [TS07], transparency of a distributed system means, that the system components should behave like a single processor for the user. This includes for example the hiding of the system structure, its components and the internal state.

In general there are two dimensions which are used to measure the ability of a distributed system to scale: the number of the processors and the amount of work.

In theory when increasing the number of the processing units, the system gains more computational power. Although in practice, adding new components is accompanied by an increase of the administrative payload. In some systems there might be a limit for the total number of components. Above this point, additional components do not

increase the system's overall processing power significantly, because the increase of management overhead compensates the gain of computational power.

It is also important to test the system's capability to cope with a high workload. This may increase the network communication or even overload some system components. The system should be able to handle this and continue its work.

Fault tolerance is implied by the first three criteria as it is the basic problem of the implementation of distributed systems. Not only one of the system components may be faulty, even the communication between them may be interrupted. This increases the probability of failures, because of this some parts of the system may not work correctly. In fact for many systems it is very unlikely to work without failures [DG04].

Talking about fault tolerance always leads to talking about communication. The occurrence of failures within a component might be handled by common programming and testing techniques, but errors within the communication layers are not easily detected and corrected.

2.2.3 Communication Architecture

Every system component has to fulfil one or more tasks, called a role. The distribution of roles in a system is called the architecture of the system. It determines the kind of data exchange between the nodes. The sum of all communication paths in the network is known as communication architecture.

In general there are two types for a system architecture, the client-server and the peer-to-peer model [Ham05]. In some systems a mixture of both is implemented.

Client-Server

The client-server model consists of two types of nodes in the network, a large number of clients and a small number of servers. The clients and the servers communicate via a request and response scheme. A client sends a request to a server to process a specific task. The server executes the task and sends a response message including the result back to the client. In general the clients are not able to communicate directly with each other.

The clients add no extra functionality to the system. Increasing the number of clients results in a higher workload for the server and therefore a lower performance of the system.

Only the server owns system ressources which can be used for computations. If there is a failure on the server, the whole system is unable to continue working. Without

introducing redundancy like additional machines, the server is a single point of failure in the model.

Despite these disadvantages, the client-server model is the most common communication model. It is easy to implement and enforces a clear separation of tasks in a network.

Peer-To-Peer

In a peer-to-peer environment all the network nodes are equal. Every one of them is able to offer the same services and to communicate with other nodes. Every node can join and leave the network at any time.

The general idea in a peer-to-peer network is that there is no central server. Because of this it is not easy to detect and address special nodes in the network, especially in networks with a large number of peers and a high fluctuation.

Therefore many practical applications use a directory server to make the addressing of a particular node easier. It is important to prevent the directory server to become a single-point-of-failure.

The peers are equal and therefore they have to take care of the computing ressources on their own. Every peer has its own ressources. Adding new peers to the network automatically adds computational power of the whole network. Failures of a single node can be compensated by other peers and the system is still able to work correctly.

Peer-To-Peer networks are widely used in filesharing applications like Bittorrent [Bit].

2.3 MapReduce

Although parallelism in a distributed system is always possible, this does not mean that is can speed up all single computations. In a client-server architecture for example the work on the server might still be executed in sequence and therefore does not benefit from the parallelism between client and server. In general the reduction of processing time in such an environment can be achieved by giving the server more ressources than the clients. For computations on very large data sets it might be more feasible to split the work among multiple servers. So the algorithm performed by the server should be executeable in parallel.

For this the sequential algorithm has to be transformed into a parallel algorithm. But finding such a parallel algorithm is not always an easy task. In fact it is still a

topic for research if it is possible to find a parallel counterpart for every sequential algorithm [BM96].

Parallelism can be reached for many problems by splitting the input data and run the sequential algorithm on multiple processors at the same time. After each processor finished its work all computation results need to be merged together to get an overall result. With this method there is no need to modify the original algorithm to be applicable to parallel computation.

In most cases the creation of distributed systems is more complex than building a single application. Therefore the code for splitting the work, controlling the progress and merging the output is placed inside a framework. The application programmer only adds his own algorithm to manipulate the data and does not have to care about parallel programming issues.

It is obvious that the framework needs a simple but powerful interface to link the application code. One of the approaches for building such a library leads to the MapReduce concept which is used in commercial computer systems today.

2.3.1 Basic Concept

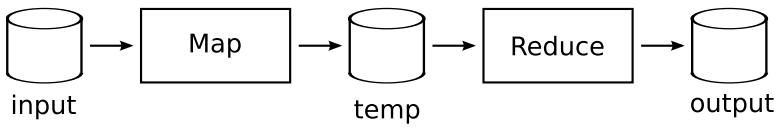


Figure 2.1: *General sequence of MapReduce*

In a MapReduce framework, the computing of the overall result takes place in two distinct phases, the map and the reduce phase. Both of them are processed in sequence, but within these stages parallel computing is possible. This general MapReduce sequence is illustrated in figure 2.1.

The names and the general functionality of these two steps are borrowed from functional programming languages like LISP and Haskell. Map and reduce are standard functions in those languages and are used for list processing. Their general use - without any parallel execution - is explained as follows.

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Listing 2.1: *Definition of a map function in Haskell notation*

The map function as shown in listing 2.1 takes a list of values and a transformation function as input. The transformation function is applied to every single list element and a new list with the results of the transformations is returned.

```
reduce :: (a -> b -> a) -> a -> [b] -> a
reduce _ z []     = z
reduce f z (x:xs) = reduce f (f z x) xs
```

Listing 2.2: *Definition of a reduce function in Haskell notation*

A list and a reduce transformation function are the inputs for the reduce function. As displayed in listing 2.2, the reduce function combines all elements of the input list to a single result value. In this example, the reduce transformation is applied to each element of the input list from left to right, but the other direction can be implemented as well.

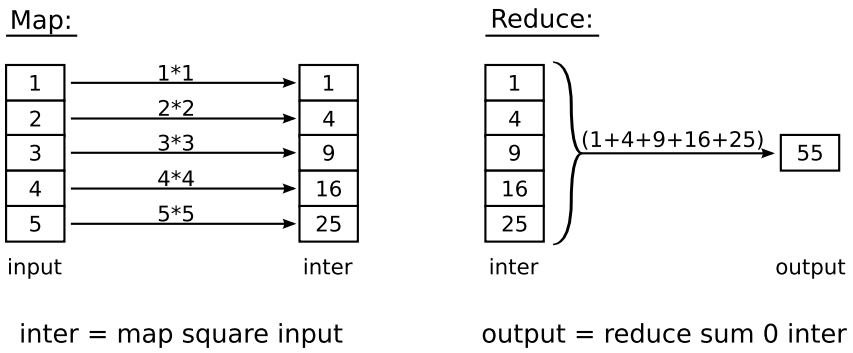


Figure 2.2: *Example for the use of map and reduce*

A basic example for the internal processing of map and reduce is shown in figure 2.2. First, the map function is called with the list [1,2,3,4,5] and the square function. The intermediate result is a list which contains only square numbers. The reduce functions takes this list and calculates the overall sum of all elements.

2.3.2 Use of Parallelism

The basic map and reduce functions in the previous section do not use parallelism. The idea is to distribute the computation among several computers. A machine which processes the map phase is called a "mapper". The output of all mappers is the input for the reduce phase which is computed by the "reducers". The MapReduce framework takes care of providing the mappers and reducers with input data and manages the data exchange between the two phases. To be able to do this, the two basic functions need to be extended to be processed in parallel.

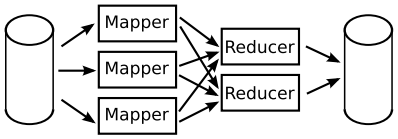


Figure 2.3: *Dataflow with multiple mappers and reducers*

Transforming the map function to parallel execution is straightforward. Since the computations of the result elements in the output list are independent from each other, these can be done in parallel. First, the input list has to be split up and distributed to multiple computers in the system. Instead of one single output list, the map phase produces a set of lists. Each single element contains a valid result.

In contrast to the map function, the basic reduce function from listing 2.2 is not suitable for parallelism. Because every list element is significant for the overall result, splitting the input would result in getting an incorrect overall result. Therefore, some modifications to the basic concept have to be made. The limitation that the reduce phase only produces one single result is not applicable in a parallel environment. It is acceptable that every reducer produces it's own result. The result of the reduce phase is now a set of values and it is free to the user to combine these single results or not.

The output values of the MapReduce system will be more expressive if every result value is calculated from a predefined set of input values. At best the input values for each reducer are distinct from the inputs of all other reducers. With this precondition, the results of the reduce phase are distinct and therefore independent from each other. To achieve this, the data has to be grouped and sorted between the map and reduce phase by the framework. This is only possible if an ordering relation is defined upon the data. To fulfil this requirement, the MapReduce concept is expanded to work only with key-value-pairs as data types. Since the grouping is done on the basis of the keys, only these data types need an ordering relation. The value data types are independent from this limitation.

Figure 2.4 illustrates an exemplary MapReduce cycle with the sorting of the data between the map and reduce phase. The system consists of three mappers and two reducers. For a better clarity, only the keys are shown in this image and not the complete key-value-pairs. In this example, the data type of the keys in the map phase are integer numbers, but before the reduce step, their data type changes to characters. After the map function, the pairs are grouped by their keys and for each of these groups, the reducers produce an output value.

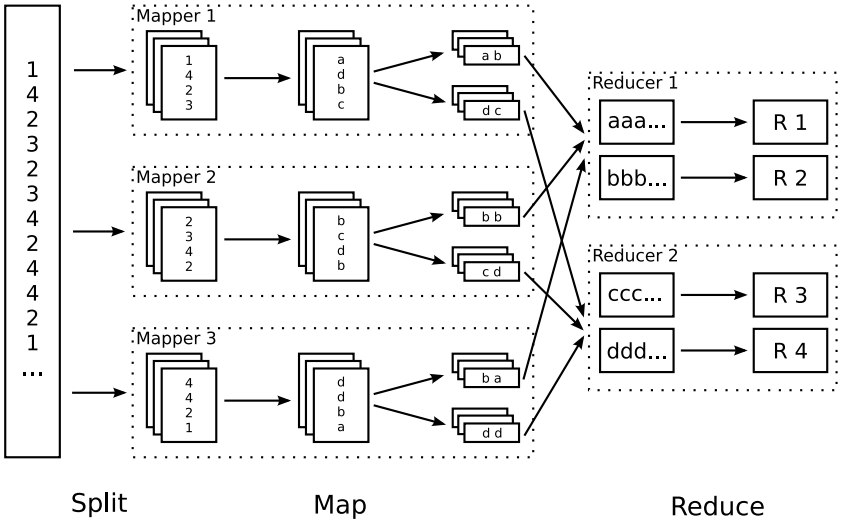


Figure 2.4: *Distribution of the data in a MapReduce system*

2.3.3 Related Projects

Although the idea of implementing map and reduce functions in functional programming languages is not new, the transfer of this concept into a distributed framework is still under research. Current studies examine possible advantages of using MapReduce in a multi-core environment [RRP+07; CKL+06] or in combination with specialized hardware [CAK+07; HFL+08]

The number of commercial applications is still very small, but there are three mentionable projects: Google MapReduce and Apache Hadoop and Disco.

Google MapReduce

Google MapReduce was developed in 2004 at Google Inc. [DG04]. It was designed to be able to run processes on the large input data of the company's search engine. Although the system itself is proprietary and only little detail knowledge is published, it is regarded as the prototype for a distributed MapReduce system.

The core software is written in C and it runs on thousands of machines in the company's datacenters around the world.

Apache Hadoop

The Apache Hadoop system is a distributed MapReduce project of the Apache Foundation [Had]. As it is published under the Apache Licence, its sourcecode is available for

download. One aim of the project is the ability to be platform independent, therefore it is implemented in Java.

It is used in many commercial applications. In 2008, Yahoo! started to build parts of searchengine's index with the help of Hadoop [Yah]. It is also used in the Amazon product searchengine A9.com [A9c] and the Amazon Elastic Compute Cloud [Ama].

Disco

The distributed MapReduce system Disco [Dis] was developed in 2008 by the Nokia Research Center in Palo Alto. Its sourcecode can be downloaded and is published under the revised BSD license.

In the Disco framework the map and reduce functions need to be specified in Python [Pyt], but the core of the system is written in the functional programming language Erlang [Erla]. Therefore this software might be a good example for the development of a similar framework in Haskell.

Currently there are no reports that Disco is used in bigger commercial applications, but like Hadoop it can easily be used in Amazon Elastic Compute Cloud [Ama], too.

3 Analysis

3.1 Environment

3.1.1 Holumbus-Searchengine

The Holumbus-Searchengine library [Hol] was developed by Timo Hübel [Hüb08] and Sebastian Schlatt [Sch08] as part of their Master's Thesis. The aim of their work was to develop and implement a framework for building specialized search engines. The user should be able to adopt this library to his own needs and generate indices for his own document sets. These indices can be used for a fast search in the input set.

One program which uses this framework is the Hayoo! searchengine [Hay]. It generates an index for the online documentation of the most important Haskell libraries and provides a web-based userinterface. Together with the existing Haskell API searchengine Hoogle [Hoo], Hayoo! is one of the main sources for the Haskell community to search the online documentation.

3.1.2 Limitations

Holumbus-Searchengine provides a MapReduce framework which is used in the document crawler and indexer. The major limitation of this MapReduce implementation is a missing distribution mechanism. The MapReduce programs can only be executed

on one single computer and cannot be shared between multiple machines over the network. Therefore one of the major features of MapReduce - the use of parallel computing - cannot be used.

For a relatively small document set like the online documentation for the Hackage database with approximately 15000 pages and 80 MB of data, the crawler and indexer process takes more than 20 hours on a modern computer. Since the documentation is frequently updated, it is desired to update the searchindex weekly, but with such a long processing time this is not feasible.

3.1.3 Enhancements

There are two possible solutions to speed up the index creation. First of all, at this stage of development the Holumbus-Searchengine is not capable of updating an existing index structure. If there are only few changes to the underlying data set, updating the index will be much faster than doing a complete rebuild [LZW04].

The second solution is to improve the MapReduce system. With a distributed MapReduce implementation, the processing time could be decreased just by adding hardware to the system. But the existing MapReduce implementation of the Holumbus-Searchengine library cannot be distributed among multiple computers. So the potential of this approach for manipulating huge amounts of data has hardly been exhausted.

The second solution seems to be the most promising. When increasing the size of the processed document set, at some point it will become unfeasible to build and update a searchindex on a single computer [BP98]. Even for document sets with only multiple gigabytes of data, non-distributed indexing algorithms reach the limits of usability [WMB99]. MapReduce benefits from the distributed data processing, even for applications which are not related to search engines or information retrieval in general.

3.2 Development Goals

The development of a distributed Crawler and Indexer for the Hayoo! searchengine is one of the main ideas for writing this thesis. This practical oriented application is the starting point for defining the main problems which need to be investigated and solved.

The main goal of this thesis is to develop a distributed MapReduce system using the functional programming language Haskell. A distributed crawler and indexer program is only one possible application. The framework itself should be adoptable to all prob-

lems which can be solved by a single computer MapReduce library and independent from the Holumbus-Searchengine library.

Since there is only limited experience in building a distributed system in Haskell, it is worth investigating if the programming language supports the programmer in this task. If necessary appropriate solutions for distributed applications need to be developed, too.

A minor aspect is the practical application of Haskell. Although the first version of the language was developed in 1990, it is still not frequently used in commercial applications. Despite the expected decrease of development time [Hug89], functional programming languages in general are regarded as hard to learn because of the new perspective for problem solving. It will be interesting to evaluate if this claim is based on a true foundation and which of these two effects will dominate at the end of the development process.

3.3 Framework Structure

For an efficient development process, the overall structure of the software needs to be investigated before the beginning with the implementation. The implementation of the MapReduce system is the main goal of this thesis, but additional software components are needed to reach it.

Three major subsystems arose from the analyze of the whole project described in this thesis:

- Communication architecture
- Storage system
- MapReduce system

A distributed system requires the existence of a reliable and powerful communication mechanism between the system components. In detail, the communication system is responsible for the data exchange between concurrent processes and threads, respectively. It has to be investigated if there are existing implementations for the Haskell programming languages which could be used in this project or if a new implementation is necessary. The functionality and the features of the communication architecture determine the usability of the whole system.

The question of an effective data exchange between the nodes of the system can be expanded to the problem of data storage in a distributed environment. A mechanism

has to be developed to send input data and receive the output results. It has to be considered that the system creates intermediate data during the work cycles and it has to be defined where the data should be stored and how it can be accessed.

The MapReduce system is responsible for the distribution of the defined work among the mappers and reducers. It processes and controls the phases of the MapReduce sequence on multiple machines and passes the results back.

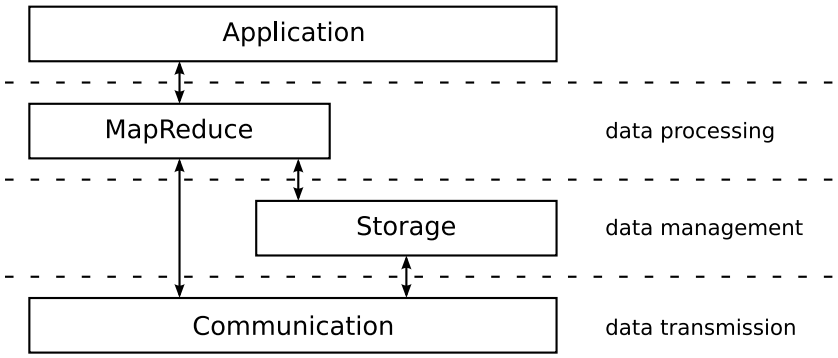


Figure 3.1: *The dependencies of the communication, storage and MapReduce system*

Figure 3.1 illustrates the dependencies between the three subsystems. Both, the MapReduce system and the storage system use the same communication architecture, while the MapReduce system needs the storage system for data access and storage. Although the communication and storage systems are the foundation for the distributed MapReduce framework and should be developed in respect of this purpose, they should be suitable for the use in other applications, too.

3.4 Requirements

3.4.1 General

Although each of the three subsystems has its own set of requirements, there are general guidelines for the implementation of the framework. The system should be as much as possible independent from the conditions of its environment. Some limitations have to be accepted because only a limited set of system platforms are available during the development process.

The software should run on common system platforms. In practice, the most likely operating systems will be the newest versions of Microsoft Windows and widely used Linux distributions. Apple MacOS is not regarded.

Since the change from 32-bit hardware architectures to 64-bit is still in progress, the software can be used in both environments. The support of distributed systems with a mixture of different hardware platforms is not guaranteed.

All computers are expected to be located in the same local area network (LAN), connected via a 100 MBit or gigabit ethernet. No special communication hardware is required. All components in the network are supposed to be trustworthy, so there is no special need for security mechanisms. The use of this system in an untrusted environment, like the Internet, or a communication between two networks is not part of this work.

In the implementation of the framework, foreign software and libraries should be used to save time and to benefit from their development. Although these components should not be used if they bring too much limitations or if they are not frequently maintained. This means that foreign libraries should always compile with the newest compiler versions and be supported by an active community. The use of proprietary software should be avoided if this implies additional costs for buying software licences.

Since the whole framework should be used to build distributed systems, it has to fulfil the four main requirements for a distributed system: openness, transparency, scalability and fault tolerance. The demand for those four main features leads to detailed requirements for each subsystem.

3.4.2 Communication System

The design of the communication layer has a direct effect on the main features of a distributed system.

The demand for openness is not very strong in this project in the sense that arbitrary programs should be able to be connected to the system. The MapReduce and storage systems which are created with this framework only consist of a distinct number of different components. Therefore the communication system does not have to be able to process data transmissions from external programs. An expansion to support arbitrary software would be an unnecessary complication.

As a good estimation, there will be two types of data exchanges in a MapReduce system. In the beginning and the end of a MapReduce phase there will be few data packages with much data to transport the input and output of the phase through the system. While processing a phase there will be a large number of small messages to control the computation progress. The communication system has to switch between these two use cases and should be able to handle both of them.

Fault tolerance is always a problem when talking about communication over a network. The whole communication process should not break down caused by a single corrupted connection. It is sufficient when errors are reported to the calling software module, error correction is not provided.

In most cases error correction needs special information about the content of the messages and limits the flexibility and openness of the library. The programmer should be allowed to use this library to build arbitrary distributed systems, not only a MapReduce architecture. Therefore the lack of an efficient error correcting strategy supports the design of an open system.

Transparency in the sense that the user is unaware of the internal structure is not the main goal of the transportation layer. This should be achieved by the applications and the user interface of the system. Although the communication layer can provide network transparency. This means that the programmer can use the same interface for inter and intra process communication. It makes no difference for him, if the data is transferred over a network connection or just copied from one memory cell to another.

Network transparency might simplify the process of implementing a distributed application. First, a single program is created which runs with concurrent threads on a single processor. The data exchange between the threads is handled by the communication library. The implementation and testing of the application take place on a single processor. After this step, the program can easily be split into smaller programs and distributed over multiple processors. Because of network transparency in the communication layer, only small changes in the application code have to be made. The supposed speedup is reasoned by the general observation that there are better programming techniques for building and testing a single program than a distributed application with lots of network communication.

Of course in some situations this kind of transparency is not wanted. Sometimes the programmer needs to be aware of the communication method. Therefore the interface for data exchange has to satisfy both needs, network transparency and network awareness.

3.4.3 Storage System

The MapReduce system should use the storage solution to read and write its input and output data. On the other hand, the implementation of the storage should not depend on the MapReduce application. It is likely that other distributed systems need a storage mechanism as well.

With such different use cases of the storage system, it is obvious that there will be no

general data format to store. Every application will have its own data type, like simple text files or arbitrary binary objects. The storage implementation needs to handle all of them, ideally without making any changes and recompiling of the system's source code.

The size of the data may vary between very small pieces of information and several megabytes for one single object. The storage capacity is not regarded to be a problem, because modern harddrives can store hundreds of gigabytes of data. But the transferrates of common network connections still allow only several minutes to copy multiple gigabytes from one computer to another. Since all the data is transferred between the storage and the distributed application, network bandwith is expected to be a critical ressource. The storage system has to support the distributed system in its data transfer and should not be an additional bottleneck [DG04]. Therefore the network communication and the data transfer should be reduced to a minimum.

The necessity for fault tolerance and correction applies especially to a storage solution. Data loss or corruption are severe errors and a great probability for their occurrence can make the whole distributed system unusable for practical applications. Therefore mechanisms for avoiding these problems are very important for the whole project.

As mentioned above it is planned to use the storage system in other applications as the MapReduce system. Therefore the programming interface of the storage system should support external programmers to use the system. The requirement for transparency is important in this context. The user of the storage system does not have to know any details about the physical location of the data and its transmission inside the system. If it is possible the data access functions should be similar to the functions for file handling from the Haskell standard library.

3.4.4 MapReduce System

The MapReduce approach is a very flexible and powerful mechanism for the processing of large data sets. Although the software of the original MapReduce approach [DG04] of Google Inc. is not published, the new framework should implement the general MapReduce concept as good as possible.

As described in the fundamentals chapter 2, the framework does not determine the actual algorithm which manipulates the data. With the help of the Haskell type parameters, it should be possible to add arbitrary transformation functions to the mapper and reducer processes, as long as they fulfil the abstract function type. This openness does not mean to bypass the strict type system of Haskell. The use of

interfaces with predefined types and the combined need for manual typecasts should be avoided. Instead of that, the user of this framework should be provided with an interface which ensures type safety.

The ability of the system to scale the processing of large data sets is elementary for using it in practical applications. To handle an increased amount of input data it is possible to expand the capacity of the system by adding components. Although this system would not be used in large clusters with hundreds or thousands of computers, it is planned to build a system using the computer pools at the University of Applied Sciences in Wedel, each of them with a size of approximately thirty machines. In a MapReduce system every new mapper or reducer has to be controlled in some way and produces a small amount of communication overhead. Therefore a system does not run exactly twice as fast when doubling the number of components [TS07]. The communication overhead has to be kept as low as possible.

Scalability in a MapReduce system can also be achieved by changing the granularity of the processed data. This leads to the question, how to split the data between the mappers. The grouping and sorting of the data between the map and reduce phase determines the size of the data packages. Even the ratio between the number of mappers and reducers can take a significant effect on the performance of the whole system and therefore the processing time. The system has to be as flexible as possible, because the programmer should be able to change all major parameters which determine the structure of the dataflow, for example the numbers of mappers and reducers and the algorithm for splitting and grouping the data. To fulfil the requirement of a simple and small interface, default values should be provided.

Fault tolerance is an important topic for a distributed system. Since the MapReduce framework encapsulates and hides all aspects of distribution from the application programmer, it has to deal with this aspect, too.

There are different reasons for fault occurrences in a MapReduce system and some of them depend on the practical implementation. The general but most common source for faults is the unavailability of one or more components of the system during the computation. The system has to handle this by adding redundancy when storing the result data or by recalculating the lost information. If this is not possible, an overall result should be computed anyway. Although in this case the overall result will not consider all inputs, it will be representive because in most of the MapReduce applications the amount of input data is so big, that a single missing data set does not affect the result significantly. In bigger MapReduce systems it is more unlikely that a computation will finish without any failures than the other way around.

As mentioned in the fundamentals chapter 2, transparency for a distributed system

means that the whole system appears to the user as a single software program. Applied to a MapReduce system, in general this means that the source code the application programmer adds to the framework is free of any signs of concurrent programming. It is not required that the user of the library does have any deeper knowledge of creating and running distributed systems. Therefore default values for all system parameters need to be provided. If the more experienced user really wants to change some system parameters, this should be possible, too.

4 Implementation

4.1 Development Environment

4.1.1 Haskell

The main reason for choosing Haskell [Has] in this project is the fact, that there already exists a related library in Haskell, the Holumbus-Searchengine. But beside this some aspects should be investigated when using a functional programming language.

It is interesting to evaluate the complexity of building distributed systems with Haskell. It has to be investigated how much effort it takes for the programmer to implement such a software.

Another focus lies in the time and space consumption of the programs. Despite general believe, it has been proved that programs written in a functional programming language are not always slower than those written in an imperative language [Ben]. But the program performance is still worth noticing.

There are different Haskell compilers and interpreters which might show some differences in handling the program source code. The Haskell compiler of choice is the Glasgow Haskell Compiler (GHC) [GHC]. It is the most common Haskell compiler and supports nearly all parts of the language definition. For this thesis the newest version of the GHC (6.10.1) was used.

4.1.2 System Platform

In general the software is designed with the aim of being independent from the underlying hardware and software architecture. The system is developed and tested only on x86 32-bit hardware running a Debian GNU/Linux system.

To be as independent from other software as possible, only libraries which are available via Hackage [Hac], the Haskell package database, are used in this project. The maintenance and support for this libraries is considered to be quite relyable, because the Haskell developer community has a great interest in using them.

4.2 Framework Structure

Holumbus [Hol] is the name of the framework, which is used and extended in this work. It provides libraries and tools to create applications in the area of information retrieval to the programmer, especially for building and using searchengines on individual input data.

Before starting this thesis the Holumbus framework only consisted of one library: Holumbus-Searchengine. During the development process of this work, new libraries were implemented: Holumbus-Distribution, Holumbus-Storage and Holumbus-MapReduce.

4.2.1 Holumbus-Searchengine

The Holumbus-Searchengine library was developed by Timo Hübel [Hüb08] and Sebastian Schlatt [Sch08]. It provides functions and tools for generating and querying individualized search indices.

One of the tools is an adaptable webcrawler and indexer for HTML documents. It is based on a non-distributed MapReduce implementation. The limited efficiency of this approach led to the development of the other three Holumbus libraries.

4.2.2 Holumbus-Distribution

Holumbus-Distribution consists of modules and tools for implementing distributed systems. Besides common data types and small helper functions, the main part of this library is the communication system.

The intention to separate these modules from the MapReduce library is to encourage the use of the communication system in other distributed Haskell programs.

4.2.3 Holumbus-Storage

Holumbus-Storage is needed to build a distributed storage system and access it from arbitrary application programs.

At the moment the storage system is only used in combination with the MapReduce system. But with the separation of the codebases the further developments of both systems will be independent from each other.

4.2.4 Holumbus-MapReduce

The Holumbus-MapReduce library contains all tools and modules for building an individualized distributed MapReduce system.

It also consists of a distributed crawler and indexer for the Holumbus-Searchengine library and other example programs which are described in the appendix of this thesis.

4.3 Communication System

4.3.1 Motivation

The first design of a distributed MapReduce system led to the conclusion that a general solution for the communication between two programs would simplify and speed up the development process.

Although Haskell provides access to the operating systems socket interface, it would not be feasible to implement a different communication protocol for every data exchange. A wrapper around the socket interface which deals with error handling and the writing and reading of messages from the socket channel could solve this problem.

Other functioncal languages like Erlang or LISP provide easy-to-use implementations for data exchange between multiple processes. Therefore it was investigated if there are comparable libraries for Haskell which could be used in this software.

There are three notable projects, Glasgow Distributed Haskell (GdH) [GdH; PTL01], Erlang-style Distributed Haskell [Erlb; Huc99] and port-based distributed Haskell [Por; HN00]. All these libraries implement a message passing interface and mailboxes for data exchange. Unfortunately, the development of these projects seems to have stopped a few years ago. The compatibility with newer versions of the Glasgow Haskell Compiler than version 5.00 is not guaranteed. Therefore these implementations do not fulfil the requirement of this project to be open for new language definitions. An own communication system for the MapReduce system has to be developed. All

projects mentioned above were used as examples and gave an inspiration for the actual implementation.

4.3.2 Concept

Although there are several theoretical approaches how to realize the data exchange between two processes, message passing seems to be the most reliable.

The use of shared memory with a locking mechanism in a distributed environment is possible, but many implementations in these area are still under research and therefore not ready to be used in real world applications.

In contrast, message passing is commonly used for communication between two computers or between different threads. For the second use case Haskell already provides the `Chan` data type from the module `Control.Concurrent.Chan`. Unfortunately, a channel can only be used between two threads which share the same address space and not between two processes or even two computers.

In the multiparadigm programming language Mozart/Oz [Moz], these two use cases of message passing are combined to a network transparent but also network aware implementation [RH04]. The communication in Mozart/Oz is realized with the help of so-called stream-port objects.

A stream in this context has the role of a mailbox. It collects all incoming messages and handles them to a thread for further processing. Port objects encapsulate the sending of messages to a stream. Every port is linked to a specified stream. The sender is able to decide, if the stream is located in the same address space or on a different machine. Depending on the result of this decision, the message is directly sent to the stream or via a network connection. Figure 4.1 illustrates the use of streams and ports to solve the common producer-consumer-problem. The consumer owns a stream and the two producers send their messages via the ports to it. The ports decide if the message can be transferred directly or via the socket interface.

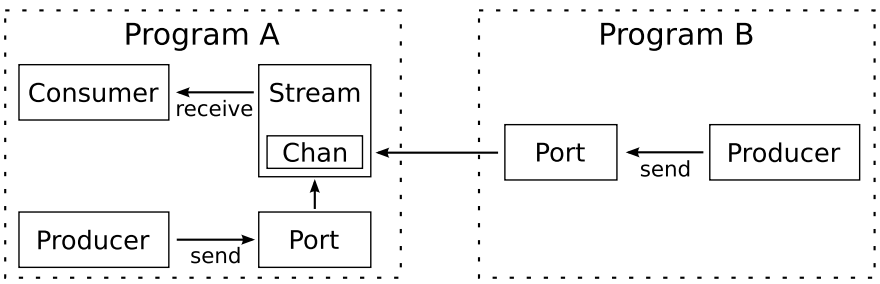


Figure 4.1: *The producer-consumer-problem with streams and ports*

To make this decision, the port needs to know where the stream is located. This data has to be provided when the port is created. To determine the location of a stream in the network the name or the IP address of its machine can be used. The name is more convenient because in some environments the IP address may change when restarting the computer.

When using the computer name as an indicator of the stream location, it has to be changed when the program with the stream runs on a different machine. This change can be done manually and without recompiling the applications via configuration files. This reduces the flexibility of the distributed system, especially the ability to react easily on structural modifications of the network. It would be easier if there is an automatic mechanism for getting the stream location.

The Erlang programming language [Erla] provides a solution for this problem. It uses a mailbox method for inter-process communication comparable to the stream-port concept [Arm07]. To be independent of hardware changes, the mailbox can be provided with a unique string identifier. The mapping between the mailbox names and its physical address is done by a registry application. This registry works like a DNS-server for mailbox names. Everytime a message is sent to the mailbox, its hardware address is resolved. If the hardware configuration changes, only the mapping has to be updated. The sender applications stay unchanged.

As shown with the implementation of the port-based distributed Haskell approach [SH01], the idea of named mailboxes can be transfered to the stream-port concept. The streams are provided with a unique name and a registry application for the name resolution is added to the network. In this application, the registry is called port-registry. To increase the flexibility of the system, stream names can be registered and unregistered at the port-registry at any time. Figure 4.2 shows its two main functions, adding new entries from a stream and giving them to requesting ports.

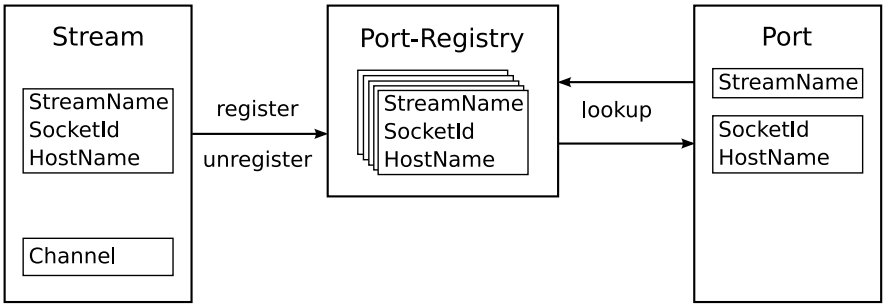


Figure 4.2: *The port-registry and its use for stream name resolution*

As described so far, all streams are able to receive messages from the same process or from different processes. Even if the mailbox is only used for internal data exchange, external processes might send data to it and manipulate the internal program state. In applications which deal with confidential data, such a scenario represents a severe security violation. The programmer needs the possibility of controlling the outside access to a stream. Therefore three different types of streams are introduced:

- Global streams
- Local streams
- Private streams

Global streams accept messages from all internal and external sources and send their name to the registry.

Local streams do not register their name at the port-registry but like global streams, they accept messages from all sources. When transmitting messages to a local stream, the sender needs to know the name of the destination computer and the socket number in addition to the stream name.

Private streams can only be used to communicate between two threads in the same address space. They do not accept messages from foreign processes and solve the above security problem. These streams fulfil the same functionality like the existing `Chan` data types in Haskell. The advantage of using a private stream is the opportunity to change them without big effort to a more open communication.

The distinction between the three stream types provides the programmer with the ability to control the origin of the incoming messages.

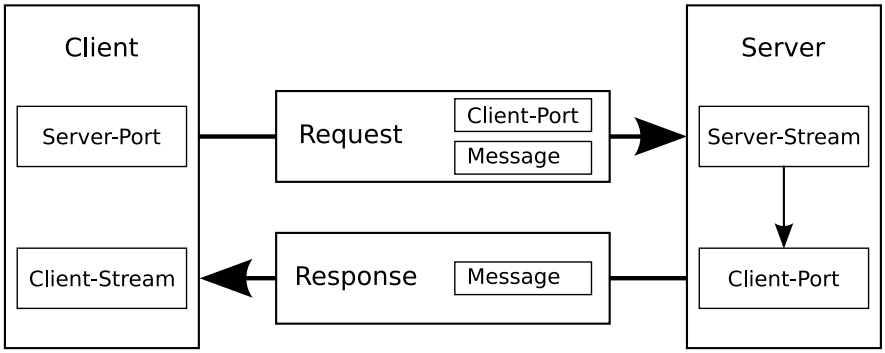


Figure 4.3: *Request-response communication with streams and ports*

So far, the stream-port concept only allows a one-way communication. The port sends requests to a stream but the stream cannot respond. But for a client-server

architecture a request-response mechanism is mandatory. Therefore a port data object has to be serializable. When the client sends its request to the server, the port for the response stream is attached to the message. Having information about the port, the server is able to send its response back to the client. Figure 4.3 shows this mechanism in detail.

4.3.3 Stream and Port Communication

The Haskell module `Holumbus.Network.Port` implements the `Stream` and `Port` data types. To provide the functionality as explained in the previous concept section, the data types shown in listing 4.1 are introduced.

```
type StreamName    = String
type BinaryChannel = Chan (Message ByteString)
data SocketId      = SocketId HostName PortNumber
data StreamType    = STGlobal | STLocal | STPrivate

data Stream a
   = Stream {
       s_StreamName :: StreamName
     , s_SocketId   :: SocketId
     , s_Type       :: StreamType
     , s_Channel    :: BinaryChannel
     }

data Port a = Port {
    p_StreamName   :: StreamName
  , p_SocketId     :: Maybe SocketId
  }
```

Listing 4.1: *Main data types for streams and ports*

Although the streams and ports hide the actual network transmission from the application programmer, they need an internal reference to the network environment. This link is given by `SocketId`. It consists of two values which are unique in a local network: the hostname and the socket portnumber. The hostname was chosen instead of the IP address because the Haskell network API does not provide an easy way to determine the current IP address of the system. In tests, only the loopback interface could be obtained and this information is worthless.

The `StreamName` is the reference to the receiver stream for the application programmer. Like the `SocketId` it is stored in the stream as well as in the port. Although the data type allows all possible strings as a valid stream name, for convenience only alphanumerical characters are allowed.

The stream also owns information about its `StreamType`. It is used to distinguish between the three different stream types and to control the external access to the stream as explained above.

The `BinaryChannel` is the queue for incoming messages. Although the type variable of the messages is set to `ByteString`, explicit encoding and decoding takes place when sending or receiving a message. Therefore, an exception might be raised when the stream receives a data package with the wrong type. These errors are inevitable, but the possibility of their occurrence is reduced to a minimum because of the type variables of the stream and port data types.

Most of the fields of the `Message` data type shown in listing 4.2 are for debug and logging purposes only. The two most important fields are `msg_Data` and `msg_Generic`. The first one contains the actual data which is transferred between the port and the stream. The generic field is used to transmit the response port in case the receiver of the message wants to answer the request. To make the use of this library much easier, the response port has to be decoded by the programmer at this application level. An automatic decoding mechanism would have increased the number of the type variables and could lead to an unfeasible typing overhead.

```
data MessageType = MTInternal | MTExternal

data (Show a, Binary a) => Message a = Message {
    msg_Type           :: ! MessageType
  , msg_Receiver       :: ! StreamName
  , msg_Data           :: ! a
  , msg_Generic        :: ! (Maybe ByteString)
  , msg_ReceiverSocket :: ! (Maybe SocketId)
  , msg_SenderSocket   :: ! (Maybe SocketId)
  , msg_Send_time      :: ! UTCTime
  , msg_Receive_time   :: ! UTCTime
  }
```

Listing 4.2: *The message data type transmitted between ports and streams*

The `SocketId` can be used by the port to determine how to send messages via the network to a stream. But it is not possible to decide if a stream is in the same address space as the port. Therefore the direct transmission of messages into the stream's queue is not possible and the major reason for the stream-port approach is not fulfilled.

Another problem is to decide if each stream has its own socket or if it should be possible to use the same socket for multiple streams. After the first experiences, the second approach seems more practically. When using streams and ports to implement a request and response mechanism, it is very likely that a large number of the response

streams will only be created to receive one message and closed after their work is done. If each of these response streams opens its own socket, this produces a large amount of overhead. Everytime such a stream is used, the operating system has to create, manage and close a socket. These operations, especially the creation, take much time. When using one socket for all response ports, the operating system only has to open it once and therefore the request-response handling becomes faster.

These two problems, the local access of streams and the economically use of sockets can easily be solved by the introduction of one global data structure in each program which uses streams and ports. In the current implementation, it is called stream-controller and shown in listing 4.3.

```
data StreamControllerData = StreamControllerData
    (Maybe SocketId)
    (Maybe GenericRegistry)
    Int
    (Map StreamName (BinaryChannel, PortNumber, StreamType))
    (Map PortNumber (ThreadId, HostName))
    (MultiMap PortNumber StreamName)

type StreamController = MVar StreamControllerData
```

Listing 4.3: *The stream-controller data type*

The stream-controller keeps maps of all used streams and opened sockets in the program. The mapping between sockets and streams is stored in the `MultiMap` data structure. For every created socket, a new thread is spawned. These threads listen for incoming data packages, read their destination stream and put them in the appropriate `BinaryChannel` queue. The access control for external messages of the different stream types is also done here.

If a port has to send data to a stream, it contacts the stream-controller. If the stream is local, it is registered at the controller and therefore the data can be written directly into the stream's channel. Otherwise it is sent over the network. So, the two problems mentioned above are solved.

Every global stream needs a reference to the port-registry program to register its name, therefore this information is stored in the stream-controller. At program startup the application programmer only has to set the link to the port-registry once, so every global stream in the program knows how to access the port-registry. This requires that there is only one stream-controller per program. To ensure this, a global reference to this data object is created via the `unsafePerformIO` function of the Haskell API.

The main goal of the Holumbus-Distribution library is to build a very powerful but easy-to-use communication mechanism. Therefore, the practical usability of the whole

concept depends on the design of an intuitive application interface. By looking at the stream and port operations in the listings 4.4 and 4.5 this goal has been reached. Most of the function names are self-explanatory.

The difference between the `readStream` and `tryReadStream` functions is that the first ones wait until a new message arrives while the second type of functions return immediately even if the queue contains no messages. The `tryWaitReadStream` functions are a mixture of both concepts. First, they wait for a given period of time given in microseconds until a message arrives. If this is not the case, they return with `Nothing`.

```
newGlobalStream      :: (Show a, Binary a)
                     => StreamName -> IO (Stream a)
newLocalStream       :: (Show a, Binary a)
                     => Maybe StreamName -> IO (Stream a)
newPrivateStream     :: (Show a, Binary a)
                     => Maybe StreamName -> IO (Stream a)

closeStream          :: (Show a, Binary a)
                     => Stream a -> IO ()

isEmptyStream        :: (Show a, Binary a)
                     => Stream a -> IO Bool

readStream           :: (Show a, Binary a)
                     => Stream a -> IO a
readStreamMsg        :: (Show a, Binary a)
                     => Stream a -> IO (Message a)

tryReadStream        :: (Show a, Binary a)
                     => Stream a -> IO (Maybe a)
tryReadStreamMsg     :: (Show a, Binary a)
                     => Stream a -> IO (Maybe (Message a))

tryWaitReadStream    :: (Show a, Binary a)
                     => Stream a -> Int -> IO (Maybe a)
tryWaitReadStreamMsg :: (Show a, Binary a)
                     => Stream a -> Int -> IO (Maybe (Message a))
```

Listing 4.4: *The main operations for the streams*

The two constructors need some information about the receiver stream because the ports are linked to a stream. The function `newPortFromStream` reads the necessary data from a stream directly while `newPort` only needs the name and an optional `socketId`. The first one should be used for the creation of ports from private streams while the second function can be used for local and global streams.

The difference between the two send functions is that the function `sendWithGeneric`

allows the programmer to set the generic data field in the message type. As mentioned before, this can be used to send a response port to the receiver and therefore realize a request-response communication architecture.

```
newPortFromStream    :: (Show a, Binary a)
                     => Stream a -> IO (Port a)
newPort              :: (Show a, Binary a)
                     => StreamName -> Maybe SocketId -> IO (Port a)

send                 :: (Show a, Binary a)
                     => Port a -> a -> IO ()
sendWithGeneric      :: (Show a, Binary a)
                     => Port a -> a -> ByteString -> IO ()
```

Listing 4.5: *The main operations for the ports*

4.3.4 Client-Server Communication

One major implementation goal of the MapReduce framework and the storage system is the handling of errors. Parts of the network may become unaccessible at some time. This problem can be divided into two subproblems: failure detection and correction. The last one depends on the overall system. If parts of the distributed storage system are not available, individual failure handling is necessary. For example in a MapReduce framework other actions need to be taken than in a distributed storage solution. But the problem of failure detection is the same in both systems. To save development time and avoid the duplication of source code, this functionality is extracted into a separated module, `Holumbus.Network.Communication`.

The general implementation should be suitable for both systems and provide a basic mechanism for the detection of connection failures and jump-in-points for own correction methods.

Figure 4.4 illustrates the network structure. It consists of two component types, a central server and many distributed clients. The clients are only able to interact with the server and not with each other. For a proper communication between one client and the server, a connection between the two components is established. A connection in this context means that the clients need to register themselves at the server before they can exchange data. Because of the registration, the server knows all clients located in the network.

The term “connection” is used on a very abstract level in this context. It does not mean that a physical TCP connection exists all the time between the server and the

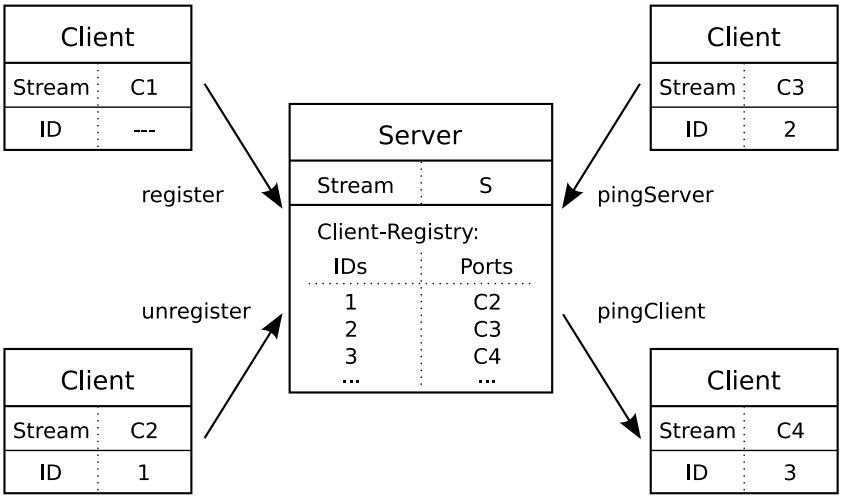


Figure 4.4: *Communication between the client and server components*

client. For every new request, a new TCP connection is established. This seems to add a performance overhead to the system because the creation of new connections and their handles by the operating system takes time. On the other hand, the number of handles which can be used in parallel is limited by the operating system. In practical environments, the number of clients is very high while the frequency for the data exchange between each client and the server is low. Keeping the number of concurrent connections as low as possible and taking a small amount of time for each request is more acceptable than the other way around.

Although the system uses a request-response mechanism, the implementation of session handling is possible, too. After the registration, the client gets a unique Id from the server. This number is used for identification and authentication. The server can store a state record for each client and a communication model which relies on more than just a single request-response processing can be established.

Both component types, the clients and the server, may become unaccessible, and both of these events should be recognized by the unaffected components. That means, if the server is disconnected from the network, the clients should recognize this and perform a reaction. In case that one or more clients are affected, the server has to handle this, too.

In order to do this, the server and the clients check if the communication partner is still responding to incoming requests. The server frequently sends ping-messages to each client. In case one client does not respond, an error counter is increased. If the value of the counter becomes higher than a previously defined threshold, the client is

unregistered from the system. A similar mechanism is used at the client-side. Each client periodically sends ping-messages to the server. If the server does not respond in time, the client assumes that it became unavailable. For a fast reorganization of the system, the clients immediately re-register when the server is available again. To eliminate the possibility of false error detection in case of one lost ping-message, a threshold is used, too.

The default value for the time interval between two ping-messages is set to five seconds and the tolerance threshold is three. That means a component is considered faulty if it is unable to respond to a ping-message within 20 seconds.

```
newServer
  :: (Binary a, Binary b)
  => StreamName -> Maybe PortNumber
  -> (a -> IO (Maybe b))                    -- request handler
  -> Maybe (IdType -> ClientPort -> IO ()) -- register action
  -> Maybe (IdType -> ClientPort -> IO ()) -- unregister action
  -> IO Server

closeServer
  :: Server -> IO ()

getClientInfo
  :: IdType -> Server -> IO (Maybe ClientInfo)

getAllClientInfos
  :: Server -> IO [ClientInfo]

sendRequestToClient
  :: (Show a, Binary a, Binary b)
  => ClientPort -> Int
  -> a                                      -- message to be sent
  -> (b -> IO (Maybe c))                    -- response handler
  -> IO c                                   -- response value
```

Listing 4.6: *Main functions of the server component*

Besides the failure detection, the server and client modules allow the application programmer to extend them with additional functions. The function `newServer`, shown in listing 4.6 needs several user-defined functions. The first one processes incomming messages and is needed to handle new types of requests. The last two function values are the listener functions for registering and unregistering a client. As mentioned before, the registration-process is done automatically between client and server, but it is very likely that additional actions have to be performed on both events. Especially the method for the unregister action is very important, because it is also executed when a client becomes unavailable in the system.

The two methods for getting more information about the registered clients, `getClientInfo` and `getAllClientInfos`, help the application programmer to access the clients. The information returned by these two functions can be used when calling `sendRequestToClient`. This method sends a message to a client and waits for a response and handles it. A timeout for the response can also be used.

```
newClient
  :: (Binary a, Binary b)
  => StreamName -> Maybe SocketId
  -> (a -> IO (Maybe b))                    -- request handler
  -> IO Client

closeClient
  :: Client -> IO ()

getClientId
  :: c -> IO (Maybe IdType)

getServerPort
  :: c -> IO (ServerPort)

sendRequestToServer
  :: (Show a, Binary a, Binary b)
  => ServerPort -> Int
  -> a                                      -- message to be sent
  -> (b -> IO (Maybe c))                    -- response handler
  -> IO c                                   -- reponse value
```

Listing 4.7: *Main functions of the client component*

The clients have a similar interface as the server. Although the `newClient` function in listing 4.7 requires only one additional function: the request handler. User-defined actions for registering and unregistering are not needed at the moment, but newer versions might provide this feature. With the help of the function `getClientId`, the programmer can check if the client is registered at the server. The function `sendRequestToServer` can be used to communicate with the server. It is the counterpart to `sendRequestToClient`.

4.4 Storage System

4.4.1 Motivation

In general, a MapReduce system gets a large amount of input data and produces few distributed output. Between the map and reduce phase intermediate results are

calculated and have to be stored and transferred. It is important to examine how the mappers and reducers can be provided with the right data.

During the implementation of the communication system, it was analyzed if there are already suitable third party software products. The first idea was to use a central file-storage technology like SQL-databases or FTP-servers. There are plenty of open source applications available for this technology. Most of them are easy to install and administrate in a private network. Implementing this technology would not limit the usability of the MapReduce framework.

Unfortunately this concept does not meet the requirement for reducing the network bandwith. Everytime a node in the distributed MapReduce system wants to read or write data, it has to send a request to the central machine. It is very likely that the server cannot handle the amount of traffic and will become a bottleneck for the whole system. The whole application will be affected, if the server is not reachable anymore.

These problems lead to the conclusion that instead of using a centralized architecture, distributed storage might be better for the MapReduce framework. In a distributed storage application, information is automatically spread across several nodes of the network. To cope with the likely failure of one node in the network, the data is stored redundantly on multiple nodes. If the MapReduce framework wants to access a distinct piece of data, it chooses a node which owns the desired information. Because of this, the network traffic is shared between multiple components and the storage is not a single point of failure any more.

There are only few distributed storage technologies, available and most of them are proprietary and therefore not suitable for this project. The use of distributed file systems like NFS [NFS] or GlusterFS [Glu] was considered but they are not independent from a special platform or third party products and therefore do not fit the requirements.

The two main MapReduce frameworks - Google MapReduce and Apache Hadoop - solve this problem by implementing their own distributed file system, the Google File System (GFS) [GGL03] and the Hadoop Distrbuted File System (HDFS) [Had]. The file systems are tightly coupled with the MapReduce libraries, which leads to a high performance.

All these arguments lead to the conclusion that it would be the best solution to implement an own file system for the Holumbus-MapReduce library. The term “file system” can be misleading, because these implementations are more a distributed system for storing data than a “classical” distributed file system which is controlled by the operating system. To avoid misunderstandings, the term “distributed storage” is used in this thesis.

4.4.2 Concept

Before talking about the structure of the distributed storage system, it is necessary to define what kind of data should be stored. Because there has to be some way to identify and get access to the stored data, it became clear that the system should store key-value-pairs.

```
type FileId       = String
type FileContent  = ByteString

data FileData     = MkFileData {
    fd_FileId             :: FileId
  , fd_Size               :: Integer
  , fd_CreationDate       :: UTCTime
  , fd_LastModifiedDate   :: UTCTime
  }
```

Listing 4.8: *Main data types for the storage system*

The key is a string and is called `FileId` whereas the value element is named `FileContent`. To be able to handle any kind of data as long as it is serializable, the `FileContent` is identical to the general `ByteString` data type from the Haskell bytestring library. As the idea of the storage system was borrowed from a file system based technology, the type `FileData` was introduced to save meta information for each piece of data in the system. The internal structure of this data type is shown in listing 4.8.

Figure 4.5 shows the general structure and communication scheme of the system. It consists of three independent component types:

- Interface
- Node
- Controller

The interface components enable the access to the storage system from other applications. Every program wanting to read or write to the system needs its own interface component. The interface manages the communication to the controller and the nodes, so that the storage system does not appear as a distributed system to the application programmer. Therefore the requirement for a transparent distributed system is fulfilled at least for this module.

The nodes are responsible for storing the data. The information is split between several nodes on different machines. To increase the overall storage capacity, it is

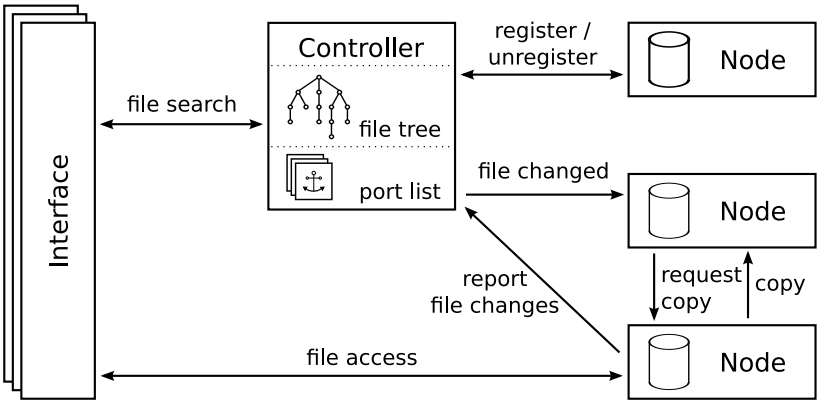


Figure 4.5: *General architecture of the storage system*

not expected that every node provides all data, but a subset of all information. It is possible that some key-value-pairs are stored on more than one node. The nodes get new data from the interface components and deliver key-value-pairs back if requested.

Because the interface has to ask every node directly for data, it has to know for every requested key-value-pair, on which nodes it is stored. Therefore the storage system needs a global controller. The controller component stores the information on which nodes a single piece of data is located. Every node has to register itself at the controller and send the names of its value-pairs to it.

Although the controller is a central component in the system, it is not expected to be a bottleneck for data access. As illustrated in figure 4.5, the interface only queries the controller for a reference to the relevant nodes. This amount of data is very small compared to the actual data, which is transferred directly between node and interface.

Since the existence of multiple copies of one key-value-pair leads to the problem of keeping the system in a consistent state, the nodes report any change to the controller. The controller then can order other nodes to get a copy of the new data or to refresh existing pairs.

A drawback of a central controller component is that it becomes a single point of failure. When is is not available, no data can be accessed or manipulated. Like the key-value-pairs, the global record of the controller needs to be saved redundantly, too. This is regarded as a minor problem since in this project the created networks are relatively small and their runtimes very short compared to commercial applications [DG04].

4.4.3 Interface

With the implementation of the storage system, it became clear that the user of this library should be very flexible in designing the physical structure of the system. It should be possible to install the nodes and the controller as single programs or to start them separately from the main application. For the MapReduce system this is not necessary because a storage node should be started on every computer on which a mapper and reducer program is running. Linking the storage node together with the worker programs of the MapReduce system will reduce the overhead of the inter-process communication since both of them will run in the same address space. It should even be possible to link the controller and one node together and create a non-distributed storage system.

```
mkStandaloneFileSystem :: FSStandaloneConf -> IO (FileSystem)
mkFileSystemController :: FSControllerConf -> IO (FileSystem)
mkFileSystemNode       :: FSNodeConf       -> IO (FileSystem)
mkFileSystemClient     :: FSClientConf     -> IO (FileSystem)
```

Listing 4.9: *Constructors for the storage system*

All these four use cases are provided by the four construction methods shown in listing 4.9. These constructors create new interfaces for the storage system. All of them provide the same functionality but consists of different internal components.

Listing 4.10 gives an impression about the actual implementation of the interface data type. It contains a link to the controller to be able to communicate with it. The optional reference to a node data type is used when a node component has to be located in the same address space as the application program. The typeclasses for the controller and the node make it possible to store the actual component or just a wrapper around a communication port.

The first option is chosen when the node or the controller should be located within the application program, the second one for inter-process communication.

```
data FileSystemData =
  forall c n. (ControllerClass c, NodeClass n) =>
  FileSystemData SiteId c (Maybe n)

data FileSystem = FileSystem (MVar FileSystemData)
```

Listing 4.10: *Main data types for the storage system's interface*

The Haskell language extension of existentially quantified types [HHJW07] is used to hide the real instantiation of these two values from the outside of the interface

data type. Otherwise the use of type parameters for the storage data type would be necessary. This would make the use of the interface more complicated.

The operations on the storage system are similar to those on ordinary files. A subset with the most common functions is given in listing 4.11. The function `getFileContent` gives the actual content of the requested key-value-pair while `getFileData` only returns the meta-data of the pair.

```
containsFile
  :: FileId -> FileSystem -> IO Bool

createFile
  :: FileId -> FileContent -> FileSystem -> IO ()
appendFile
  :: FileId -> FileContent -> FileSystem -> IO ()
deleteFile
  :: FileId -> FileSystem -> IO ()

getFileContent
  :: FileId -> FileSystem -> IO (Maybe FileContent)
getFileData
  :: FileId -> FileSystem -> IO (Maybe FileData)

getNearestNodePortWithFile
  :: FileId -> FileSystem -> IO (Maybe NodePort)
```

Listing 4.11: *Main operations on the storage system*

The function `getNearestNodePortWithFile` is used by the MapReduce system to optimize the distribution of work between the mappers and reducers. It queries the controller for the "nearest" node with a specific data object from the machine on which this function is called. The distance between the interface and the node is measured in three categories. The first and nearest one is applied when the interface and node are located in the same address space. The second category describes that both components are running on the same machine but in different processes, while the third category just indicates that node and interface are not on the same computer.

If the requested key-value-pair is not stored on the local machine a random node is chosen which contains the data.

As an optimization the physical structure of the network and the workload of the nodes could be regarded. This information could be used to introduce a more accurate distance function and reduce the network communication.

4.4.4 Controller

The main task of the controller is to keep a record of all nodes in the system and the key-value-pairs stored on them. When the nodes register at the controller, they send a list with all their keys to the controller. If a node becomes unavailable by error or normal shutdown, the controller has to delete all related entries. The main data structures for these tasks are illustrated in listing 4.12.

```
type FileToNodeMap       = Map FileId (Set NodeId)

data FileControllerData = FileControllerData {
    cm_FileToNodeMap :: ! FileToNodeMap
  }
type FileController      = MVar FileControllerData

data ControllerData      = ControllerData {
    cd_Server           :: Server
  , cd_FileController   :: FileController
  }
```

Listing 4.12: *Main data types for the storage system's controller*

The node management - automatic registration and unregistration - is done by the `Server` module from the communication system. Therefore, the controller only consists of two elements, the `Server` module and a map for storing the reference between `FileId` and a set of `NodeIds`.

4.4.5 Node

Since the controller uses the `Server` module of the communication system, the nodes have to use the corresponding `Client` data type. It automatically registers at the controller and checks periodically if it is still available. The physical storing of the key value pairs is done by the so called `FileStorage` data type. The node is actually providing only the bridge between the `FileStorage` and the `Client` module.

```
data NodeData = NodeData {
    nd_Client           :: Client
  , nd_Storage          :: FileStorage
  }

data Node      = Node (MVar NodeData)
```

Listing 4.13: *Main data types for the storage system's nodes*

As the name implies, in a file-storage, the key-value-pairs are stored directly to a physical file on the systems harddisk. The values of each pair are stored in their own

binary files, whereas the filenames are derived from the key-string. This solution is not very efficient. It is very likely that lots of files will be managed by the system while most of these will only store a small amount of data. Some file systems do not handle small files very well and consume for each file a distinct amount of data for meta information. Therefore, the files can take more space on the harddisk than their actual overall size. Another aspect is that it takes very long for the operating system to open and close a file. In general the reading and writing of a file is much faster. Therefore it is expected that a frequently used storage system takes most of the time to get the data and not to deliver it.

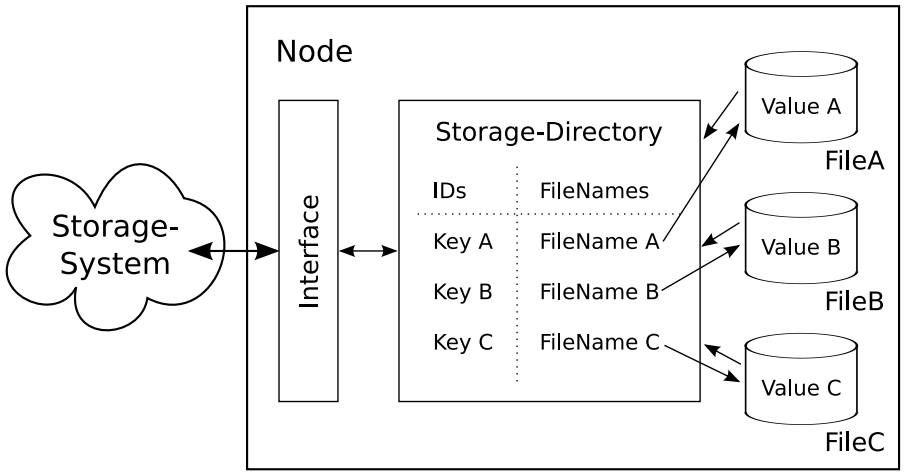


Figure 4.6: *Internal structure of one storage system node*

As described in [GGL03], this became a problem for the Google MapReduce system. Therefore the Google File System (GFS) stores multiple key-value-pairs in a single file, called a chunk. In general such a chunk is around 64MB in size. This reduces the time overhead for accessing multiple key-value-pairs in the same chunk.

Because the storage system was only implemented to support the MapReduce system, the implementation of a more sophisticated technology, like the GFS, is not scope of this work. To make the further development of the storage system easier and to provide the ability to choose between different physical storages, the `Storage` typeclass is introduced. Every new storage type has to implement this typeclass.

4.5 MapReduce

4.5.1 Architecture

In the fundamentals section it is explained, that the work is distributed between multiple mappers and reducers and that the framework takes care about the management of the MapReduce computation. This management can only be done by a system component which has information about all mappers and reducers in the system and their current work.

Google MapReduce and Apache Hadoop propose the introduction of a central process, called master. To be accessible in the MapReduce system, all mappers and reducers have to register at the master. The master splits and delegates the work to these components, whereas the mappers and reducers give feedback when their current work is finished. The efficient implementation of the master is crucial for the practical useability and performance of a MapReduce system.

Until now, the mapper and reducer components were strictly separated. During the implementation of this framework, it became clear that this separation makes no sense in the planned application. It is expected that the MapReduce networks do not exceed a size of thirty machines and are only computing one MapReduce operation at a time. With a strict separation of mapper and reducer computers, this would mean, that a huge percentage of the system will not do any work in one phase because it can only operate in the other phase. Of course, the mapper and reducer programs could be started on the same computer, but this increases the overall number of independent processes in the system and the communication overhead. Therefore, the mappers and reducers are combined together to so called workers. These can do both, map and reduce, and possible other phases. When delegating a piece of work to a worker, the master has to give it additional information of the current phase.

So far, the MapReduce network consists of one single master and multiple workers. But it is not clear, how the network should be accessed by the user. It is obvious, that the only access point to the system should be the master, since it controls the distribution process among the workers. The question is what mechanism should be provided to start new MapReduce computations and to get their results.

The first thought was to build the master with its own user interface. To start a MapReduce program, the user has to access the master computer and load his programs into the system. The definition of such a MapReduce program has to be in a human readable but easy to parse format. An individual XML-format would fulfil the requirements. Although this approach is feasible, it became clear that the existing

MapReduce implementation in the Holumbus-Searchengine library has a much better solution. It provides an API which can be called directly from any Haskell program. With this, it is much easier to start a MapReduce program and retrieve its result. As shown in the implementation of the crawler and indexer for the Hayoo! searchengine [Sch08], additional code is needed for the solution of complex problems, for example reading input and configuration data or doing manipulations on the output of the MapReduce program. The manual access via the master cannot provide this level of flexibility.

All these thoughts lead to the conclusion, that the distributed MapReduce framework has to be accessed from other programs like a normal function call. This requires the introduction of a third component in the MapReduce system, the interface. Figure 4.7 shows the interaction between all three components.

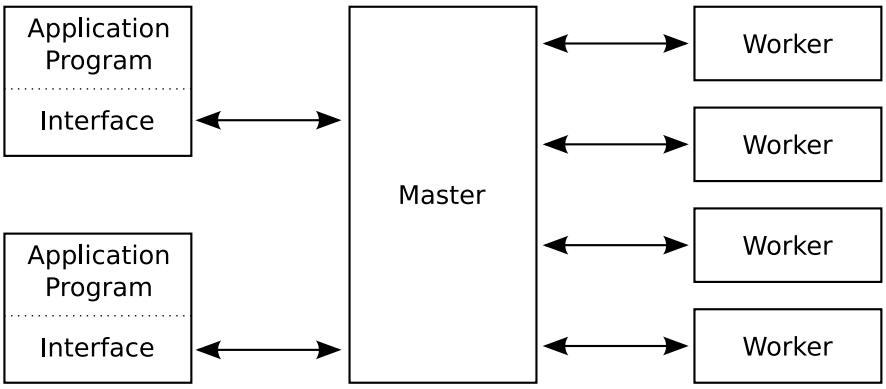


Figure 4.7: *Overview of the MapReduce system components*

The application program creates an interface component and uses it to transmit the MapReduce computation requests to the master. The master delegates the work to the workers which do the actual computation. When the overall result is known, the master reports it back to the interface component and therefore makes it available to the user program. The same master-worker-network can be used by different application programs for different computations, under the condition that the workers can handle all necessary computations.

It is not a coincidence that the architecture of the MapReduce system is comparable with the one of the storage system. As mentioned before, the storage system was developed with respect for the requirements of the MapReduce system. The master of the MapReduce system includes the controller of the storage system whereas the workers integrate the storage nodes. This means that both systems are tightly coupled and that each worker has a local storage space for its own. Intermediate data of the

MapReduce phases can be stored locally but is still accessible for the whole system. This can be used by the master to reduce the amount network traffic and exchange of huge data. The possible strategies are discussed in detail with the description of the master's internal structure.

To make the MapReduce system capable of handling new kinds of tasks, the new functions need to be spread through the network, so that every worker can execute the new programs. To integrate new functions into the MapReduce system, the application programmer has to compile them into the worker programs. Then the new workers need to be copied and started manually. This setup process for a MapReduce network is not feasible for a large number of machines, but at the current stage of development, it is acceptable. The master process is independent from the specific implementation of the map and reduce functions, so it does not need to be recompiled. The workers re-register at the master, so it does not even need to be restarted.

A better mechanism for the fast and stable spreading might be to integrate the distribution of new source code into the MapReduce system itself. This could be easily achieved if the used programming language provides a mechanism to load source code on the fly, without restarting the whole application. In such an environment, the master can copy the new code to each worker which will do the reloading.

Unfortunately, there is no stable mechanism in Haskell for the dynamically loading of new modules at this moment. There is a library called hs-plugins [PSSC04] which could do this, but its implementation depends strongly on the used compiler version. If a new version of the Glasgow Haskell Compiler is used, it is very likely that the hs-plugins library needs to be updated, too. Since the development process of hs-plugins tends not to be very fast, the usage of this library means an unnecessary limitation to the development of Holumbus-MapReduce. Maybe in future versions this will not be the case anymore.

4.5.2 Concept Enhancements

As described in the fundamentals section, the basic MapReduce concept only consists of two phases: map and reduce. Although this approach is very simple, it can be used to create complex manipulations on huge amounts of input data. In many MapReduce applications, the map function produces lots of intermediate data while the overall results of the reduce function are relatively small. That means between the two phases, it is very likely that large blocks of data need to be exchanged between the workers. Since network bandwidth is a critical ressource, it is suggested in [Läm06] to introduce an additional combine phase between map and reduce.

The combine function is like a locally applied reduce function. The idea is that the combine function reduces the intermediate data of the map function, so the amount of data which needs to transported to the reducers gets smaller. This only makes sense when no data needs to be transmitted between the workers before starting the combine phase. Therefore the combine function has to be executed on the same workers were the map functions are processed and only on the locally stored intermediate files. Since network bandwidth might be an issue in further applications of the Holumbus-MapReduce system, it is added to the actual implementation.

Besides the reduction of the network traffic, there is a strong requirement to design the framework to be as flexible to the application programmer as possible. Flexibility in this context means that the user of this framework can influence the distribution of the data in the sense that he is able to determine how the data is distributed between the workers. So far, it was said, that the MapReduce system automatically groups all key-value-pairs with the same key and sends them to one reducer. In many cases, this will be the desired functionality, but in some cases an individual grouping is necessary.

One of these use cases is the creation of multiple inverted index files for a distributed searchengine. There are several ways how to split the index files while the general creation algorithm stays the same [MWZ06]. For example, the index files can be split by input documents or by terms. Since the further versions of the Hayoo searchengine might become distributed, the Holumbus-MapReduce framework has to provide this flexibility.

To get control over the grouping, [Läm06] suggests to introduce two functions between two consecutive phases: partition and merge.

```
defaultPartition
  :: (Binary k, Binary v)
  => a -> Int -> [(k,v)] -> IO [(Int, [(k,v)])]
defaultPartition _ 1 ls
  = return [(1,ls)]
defaultPartition _ n ls
  = do
    return $ AccuMap.toList $ AccuMap.fromTupleList markedList
    where
    markedList = map (\t@(k,_) -> (modHash k,t)) ls
    modHash k = ((hash k) `mod` n) + 1
    hash k = fromIntegral $ Hash.hashString $ show $ encode k
```

Listing 4.14: *Default implementation of the partition function*

Partition is called at the end of a phase to assign the data to different workers. As shown in listing 4.14, an integer value is assigned to every key-value-pair in order to separate the data. The same number means that the pairs are assigned to the same

worker. The function is given the maximum number of the workers as a parameter, so it can be guaranteed that every worker is used in the next phase. The default use case is that the same key is always mapped to the same number, but different assignments are possible. If the mapping leads to a number bigger than the number of workers, a modulo calculation is performed. An `AccuMap` data type is used to group the key-value-pairs by their assigned numbers.

The calculation of a hash value for the partitioning is not very efficient and maybe not suitable for all applications. Therefore the default implementation of the partition function can be redefined by the programmer.

```
defaultMerge
  :: (Ord k, Binary k, Binary v)
  => a -> [(k,v)] -> IO [(k,[v])]
defaultMerge _ ls
  = return $ AccuMap.toList $ AccuMap.fromTupleList ls
```

Listing 4.15: *Default implementation of the merge function*

The merge function is needed at the beginning of a phase. The worker gets a list of key-value-pairs from the master, but it is not guaranteed that a key only appears once in the list. Therefore all values with the same key are stored in a single list on which the reduce function can be applied. In most applications, this default functionality given in listing 4.15 does not need to be overwritten.

The partition function works fine for assigning the key-value-pairs to the workers of the combine or reduce phase. But in the basic MapReduce model, there is no cycle before the map phase and so there is no partition function to control the distribution of the data over the mappers. The master would be the best place to handle this, but since it is independent from the actual worker functionality, the application programmer wouldn't be able to overwrite the default definition.

```
defaultSplit
  :: a -> Int -> [(k,v)] -> IO [(Int, [(k,v)])]
defaultSplit _ n ls
  = return $ AccuMap.toList $ AccuMap.fromList ps
    where
    ns = [(x `mod` n) + 1 | x <- [0..]]
    is = map (\a -> [a]) ls
    ps = zip ns is
```

Listing 4.16: *Default implementation of the split function*

In [YDHP07] this problem is solved by introducing a fourth phase to the MapReduce model. It is called split phase and is performed before the map phase. The split

function is executed by severval workers and it can be overwritten by the user. As shown in listing 4.16 the default implementation just performs a partitioning on the key-value-pairs data and assigns them to specific workers. The master still has to partition the data before the split function, but this can be done by a simple division into chunks of the same size.

Because the split phase is not part of the basic model and is not needed in many applications, the Holumbus-MapReduce framework should not rely on it. If the user omits a definition for the split phase, the master takes care of the splitting.

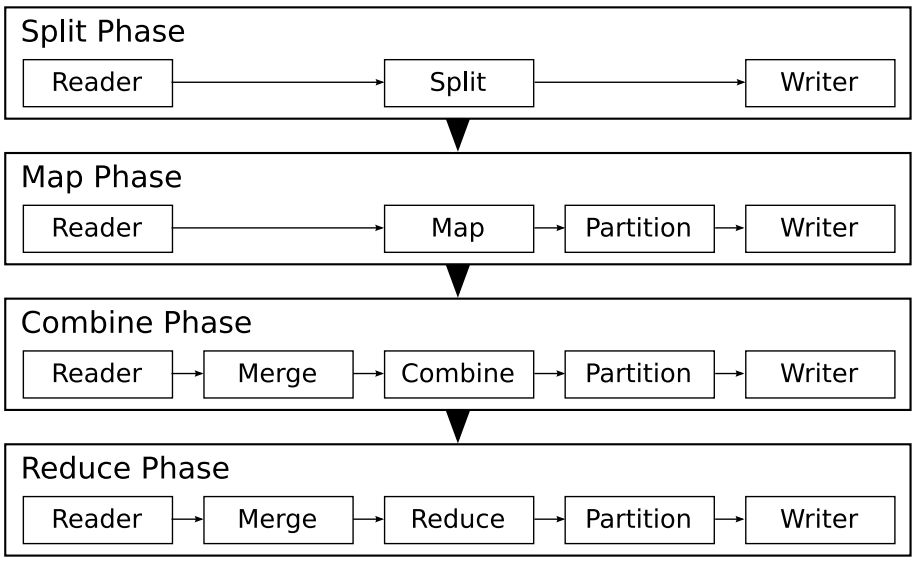


Figure 4.8: *Complete sequence of a MapReduce computation*

A MapReduce computation now contains of four phases: split, map, combine and reduce. As mentioned before, each of these sections can be omitted. By using only map and reduce, the enhanced MapReduce concept is identical to the one described in [DG04]. Figure 4.8 gives an impression about the sequence of the four phases and their internal structure.

The declarations of all functions necessary for the MapReduce processing are shown in listing 4.17. The combine phase is left out because, its interface is identical to the one of the reduce phase. The "action" functions declarations illustrate only the resulting types when sequencing all functions inside a phase. They cannot be defined directly by the user.

Although the MapReduce framework is a distributed system, for some applications there is the need of a global configuration state. The state only needs to be read-only but the user needs a mechanism to give parameters which are important for the

behaviour of the MapReduce program. This became obvious while implementing a distributed grep program. In this example, the global state is the regular expression on which each line of the input files should be tested. It doesn't make sense to recompile all the workers if the regular expression changes.

In a single process MapReduce system, the introduction of a global state wouldn't be a problem. It could be inserted into the map and reduce functions without even modifying the interface for the function definitions by the use of currying or other techniques. But in a distributed environment, the configuration has to be distributed, too. Therefore all functions in listing 4.17 have an additional parameter `a` in their declaration which represents the global configuration.

```
type SplitAction a k1 v1
  = a -> Int -> [(k1,v1)] -> IO [(Int, [(k1,v1)])]
type SplitFunction a k1 v1
  = SplitAction a k1 v1

type MapAction a k1 v1 k2 v2
  = a -> Int -> [(k1,v1)] -> IO [(Int, [(k2,v2)])]
type MapFunction a k1 v1 k2 v2
  = a -> k1 -> v1 -> IO [(k2,v2)]
type MapPartition a k2 v2
  = a -> Int -> [(k2,v2)] -> IO [(Int, [(k2,v2)])]

type ReduceAction a k2 v2 v3
  = a -> Int -> [(k2,v2)] -> IO [(Int, [(k2,v3)])]
type ReduceMerge a k2 v2
  = a -> [(k2,v2)] -> IO [(k2,[v2])]
type ReduceFunction a k2 v2 v3
  = a -> k2 -> [v2] -> IO (Maybe v3)
type ReducePartition a k2 v3
  = a -> Int -> [(k2,v3)] -> IO [(Int, [(k2,v3)])]
```

Listing 4.17: *The function interface for the MapReduce system*

4.5.3 Data Exchange

In the early development stages, when the storage system was not completely designed, the workers of the MapReduce system did not store the data locally. An approach with direct data transmission was implemented and is still available in the current version of the framework. When using this technique, the workers receive the input data directly from the master and send the results back to it. The consequence is that the master stores all intermediate results in memory. This is not feasible for real

applications but is helpful when testing the MapReduce system on small data sets for debugging.

```
data FunctionData
  = TupleFunctionData ByteString
  | FileFunctionData FileId
```

Listing 4.18: *Input and output type for a MapReduce task*

As mentioned before, the Holumbus-MapReduce system uses the Holumbus-Storage system. The data format in which the the data is stored in the physical files has to be a bytestring, but how the data is encoded from its original data structure is not determined by the storage system. To give the application programmer the possibility of controlling this transformation, reader and writer functions are introduced. The types of both functions are shown in listing 4.19. The user only has to implement the encoding and decoding of the key-value-pairs to and from a bytestring. The access of the storage system is still done by the MapReduce system.

```
type InputReader  k1 v1 = ByteString -> IO [(k1,v1)]
type OutputWriter k2 v2 = [(k2,v2)]  -> IO ByteString
```

Listing 4.19: *Types for the reader and writer functions*

This could be useful, if the intermediate files need to be readable by another program which uses a specific data format on its own. If this features is not needed a default implementation is provided for both functions.

```
data JobAction = JobAction {
    ja_Name               :: ! ActionName
  , ja_Output             :: ! TaskOutputType
  , ja_Count              :: ! Int
  }

data JobInfo   = JobInfo {
    ji_Description        :: ! String
  , ji_Option             :: ! ByteString
  , ji_SplitAction        :: ! (Maybe JobAction)
  , ji_MapAction          :: ! (Maybe JobAction)
  , ji_CombineAction      :: ! (Maybe JobAction)
  , ji_ReduceAction       :: ! (Maybe JobAction)
  , ji_NumOfResults       :: ! (Maybe Int)
  , ji_Input              :: ! [FunctionData]
  }
```

Listing 4.20: *Types for the MapReduce job definition*

When the system has to execute a new MapReduce program, a job has to be defined and sent to the master. An instance of the `JobInfo` record is automatically created by the interface component. As shown in listing 4.20 it consists of a short description for better debug output, the input data list (`ji_Input`), the global configuration state (`ji_Option`) and information about each one of the four phases. The `JobAction` record stores the name of the function set, the workers have to execute and also defines the maximum number of workers used for the the according phase (`ja_Count`). If no data is given for a specific phase, it will be omitted by the system.

The master will store this job definition and start immediately with its execution. For this, the job will be split into several tasks. A task defines a small package of work which is sent to a worker, executed and then sent back to the master added with the results of the computation. Listing 4.21 gives an impression about the general data structures of a task.

```
type TaskId          = Integer

data TaskType        = TTSplit | TTMap | TTCombine
                     | TTReduce | TTError

data TaskState       = TSIdle | TSSending | TSInProgress
                     | TSCompleted | TSFinished | TSError

data TaskOutputType = TOTRawTuple | TOTFile

data TaskData        = TaskData {
    td_JobId         :: ! JobId
  , td_TaskId        :: ! TaskId
  , td_Type          :: ! TaskType
  , td_State         :: ! TaskState
  , td_Option        :: ! ByteString
  , td_PartValue     :: ! (Maybe Int)
  , td_Input         :: ! (Int, [FunctionData])
  , td_Output        :: ! [(Int,[FunctionData])]
  , td_OutputType    :: ! TaskOutputType
  , td_Action        :: ! ActionName
  }
```

Listing 4.21: *Types for the MapReduce task definition*

The type `TaskData` is the main data container for a task definition. Every task is linked to its job (`td_JobId`) and has a unique number (`td_TaskId`) to help the master to keep track of the responses from the workers. The combination of the `TaskType` and `ActionName` is enough information for the worker to know which set of functions has to be invoked. The fields `td_Option`, `td_PartValue` and `td_Input` are the input

parameters for the actual computation while `td_Output` holds the result.

The `TaskData` type is used for both, starting a task at a single worker and receiving its result. As shown in the detailed description of the data structures of the master process, it is even used for the internal storage of the task information. The field `td_TaskState` is not needed by the workers at all, only by the master. Because all of this information is transmitted twice, this introduces a short amount of communication overhead. It would be better to introduce distinct data types for sending, receiving and internal storage. At this stage of development, a single data structure for all three scenarios is still feasible because changes to the underlying data exchange are very possible and the overhead is still very small.

4.5.4 Interface

In the normal application scenario, the interface, master and worker components of the MapReduce system will be located on different computers. In spite of this, during the development process the demand arose to become flexible regarding the distribution of the system. It is not a strong need, but the access mechanism for the storage system gives a good example how to design a flexible interface component for the MapReduce system, too. Therefore the approach of being able to compile the master or worker component directly into the application program is followed in the MapReduce library as well.

```
data DMapReduceData =
  forall m w. (MasterClass m, WorkerClass w) =>
  DMapReduceData SiteId m (Maybe w)

data DMapReduce = DMapReduce (MVar DMapReduceData)
```

Listing 4.22: *Types for the MapReduce interface component*

The data declaration in listing 4.22 gives a good impression about the strong similarities in the designs of the interfaces for the storage system and the MapReduce network. A `DMapReduce` object is a reference to a distributed MapReduce component. Such a data type can be created via the three constructors shown in listing 4.23.

The functions `mkMapReduceMaster` and `mkMapReduceWorker` create a new master and a new worker component. Both of them need a reference to the distributed storage system, while the list with the available actions is only required by the worker. The data structure created by `mkMapReduceClient` contains only a reference to an external master component. This is the kind of method most of the application programs will

use to access the system and start new MapReduce programs. Since the result types of all three functions are the same, all possible components share the same interface.

```
mkMapReduceMaster
  :: FileSystem -> DMRMasterConf -> IO DMapReduce

mkMapReduceWorker
  :: FileSystem -> ActionMap -> DMRWorkerConf -> IO DMapReduce

mkMapReduceClient
  :: DMRClientConf -> IO DMapReduce
```

Listing 4.23: *Constructors for the MapReduce components*

As said in the analysis chapter, the system's API should be very simple and easy to use. Therefore only three steps are required for the proper invocation of a MapReduce program:

1. Create a new interface component
2. Execute the MapReduce program
3. Shut down the interface

The first step is already shown in listing 4.23 while the other two are given in listing 4.24. To start a MapReduce computation and get the result, only the function `doMapReduce` has to be invoked. The first parameter, the `ActionConfiguration` determines which MapReduce program to use and guarantees type safety.

There are two possible ways for the user to provide the MapReduce system with input data. He can give the input data directly via a list of key-value-pairs or indirectly as a list of keys saved by the storage system. A mixture of both is possible, too.

The output type is defined with the help of the `TackOutputType` parameter. It can be either a tuple list with the actual values or the keys for the storage system.

For large sets of input and output data, the use of tuple lists is not very efficient, because the data is directly transmitted to the master. This will decrease the performance of the master process and increase the network traffic. It is only suitable for smaller applications with small amounts of input and output data.

For future versions, it might be possible not to transfer the tuple list directly over the network, but to save it automatically in the storage system. The handling of the input and output data could be integrated in the system's API. The detailed functionality of this approach is not very clear at this moment because more practical experiences are needed. Until now, the choice between the raw data and a list of storage ids seems the best.

```
doMapReduce
  :: ActionConfiguration a k1 v1 k2 v2 v3 v4 -- action to use
  -> a                                       -- options
  -> [(k1,v1)]                               -- input (tuples)
  -> [FileId]                                -- input (files)
  -> Int                                     -- number of splitters
  -> Int                                     -- number of mappers
  -> Int                                     -- number of reducers
  -> Int                                     -- number of results
  -> TaskOutputType                          -- result type
                                             -- (tuples or files)
  -> DMapReduce                              -- the MapReduce system
  -> IO ([(k2,v4)],[FileId])                 -- result list

closeMapReduce
  :: DMapReduce -> IO ()
```

Listing 4.24: *The application interface for the MapReduce system*

For the performance of the system and the distribution of the result set the definition of the first three integer values is very important. With their help, the user can specify the maximum number of workers in the split, map and reduce phase. Since the same workers are use in the map and the combine phase, the same number applies. The fourth integer value controlles the partitioning of the overall results and might be important for the further processing of the data.

The function `doMapReduce` will block as long as it takes to compute the overall result list. In case of an error which prohibits the calculation of a result, an exception is raised.

The use of the function `closeMapReduce` is not only essential for a good programming style. It helps to keep the system in a consistent state especially by shutting down a master or worker component and freeing limited system ressources.

It is to mention that the actual API consists of more functions than shown in this section. Although all of them are meant for debug purposes only and are not recommended for practical use. Among other things the master component can be switched into a single step mode. It means that every controlling step of the master has to be initiated manually. This is very useful to keep track of its internal state and debugging but not for normal use cases.

4.5.5 Master

An efficient implementation of the internal functionality of the master is very important for the practical usability of the whole system. The master is the core application

and implements the major management algorithm. A poorly designed data structure will not just decrease the performance of this component, but destabilize the entire system. Therefore a strict distinction between independent functional modules is required. Figure 4.9 gives a first impression about the master's main parts.

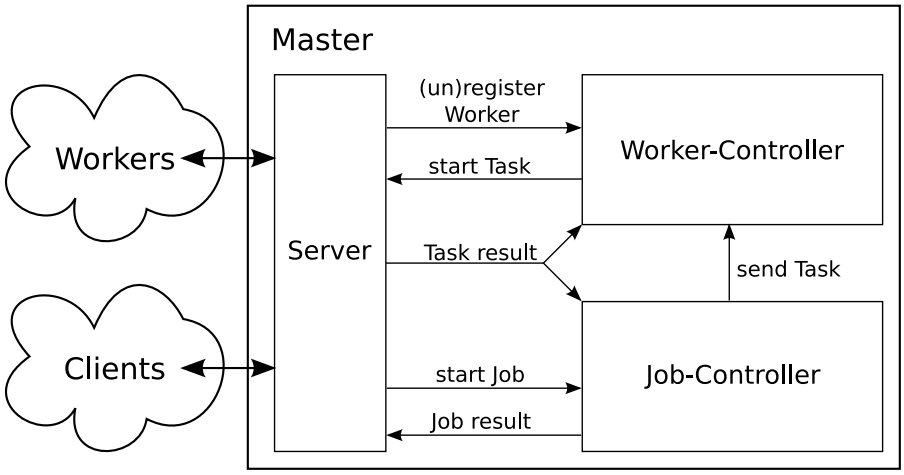


Figure 4.9: *The internal structure of the master component*

The master consists of three internal parts which interact with each other but are responsible to fulfil different roles. These three parts are:

- Server
- Worker-Controller
- Job-Controller

The server component is provided by the Holumbus-Distribution library. It is responsible for handling incoming requests and sending the responses. Since the workers implement the according client module, it is guaranteed that the workers automatically register at the master. Also ping messages are exchanged between the master and the workers. Therefore the failure of a specific worker is automatically recognized by the master and a reaction on this event can take place. As explained above, the server component keeps a map with information about all available clients in the network.

The job-controller is only responsible for managing the jobs and their tasks. But it has no knowledge about the available workers. The server component has this information but cannot be expanded to store the currently assigned tasks for each worker. That's why the worker-controller fills the gap between the server component and the job-controller. It keeps a record of which tasks are assigned to which worker.

To implement the storage of the many-to-many relationship between the tasks and the workers, two maps are used. With this information, the worker-controller is able to select the least busy worker and send new tasks to it. This selection can only be successful if the worker has implemented the desired action functions. Therefore a listing with all availabe actions for each worker is required, too.

```
type TaskToWorkerMap       = MultiMap TaskId WorkerId
type WorkerToTaskMap       = MultiMap WorkerId TaskId
type ActionToWorkerMap     = MultiMap ActionName WorkerId

data WorkerControllerData = WorkerControllerData {
    wcd_TaskToWorkerMap    :: ! TaskToWorkerMap
  , wcd_WorkerToTaskMap    :: ! WorkerToTaskMap
  , wcd_ActionToWorkerMap  :: ! ActionToWorkerMap
  }

type WorkerController      = MVar WorkerControllerData
```

Listing 4.25: *Types for the worker-controller*

The strict distinction between the job-controller and the worker-controller seems to add unnecessary complexity to the program design. On the contrary, it helps to structure the source code and adds more flexibility for future changes. Although the approach with introducing one master and many workers is very settled in this framework, it might be interesting for other projects to explore different designs. Because the job-controller is independent of the actual task sending and execution mechanism, it can still be used in other MapReduce systems without any changes. In early stages of development of the framework, a single computer MapReduce application was designed. In this program only the worker-controller and the server component are modified. The tasks are executed directly without the existence of any worker. Instead, the complex implementation of the job-controller remains untouched.

The job-controller module consists of the main algorithms of the MapReduce master. It is responsible for the job management and the distribution of the job into tasks. The tasks are given to the worker-controller and the results are transmitted back to the job-controller. If all tasks of the current job phase are completed, the job-controller switches to the next phase. When the entire job is finished, the results are reported back to the server module. To fulfil all these requirements, the data structure shown in listing 4.26 is introduced.

```
type JobMap                = Map JobId JobData
type TaskMap               = Map TaskId TaskData

type StateJobIdMap         = MultiMap JobState JobId

type JobIdTaskIdMap        = MultiMap JobId TaskId
type TypeTaskIdMap         = MultiMap TaskType TaskId
type StateTaskIdMap        = MultiMap TaskState TaskId

type TaskSendFunction      = TaskData -> IO (TaskSendResult)

data JobControlFunctions = JobControlFunctions {
    jcf_TaskSend         :: TaskSendFunction
  }

data JobControllerData     = JobControllerData {
    jcd_ServerThreadId   :: Maybe ThreadId
  , jcd_ServerDelay      :: Int
  , jcd_FileSystem       :: Maybe FileSystem
  , jcd_NextJobId        :: JobId
  , jcd_NextTaskId       :: TaskId
  , jcd_Functions        :: JobControlFunctions
  , jcd_JobMap           :: ! JobMap
  , jcd_TaskMap          :: ! TaskMap
  , jcd_StateJobIdMap    :: ! StateJobIdMap
  , jcd_JobIdTaskIdMap   :: ! JobIdTaskIdMap
  , jcd_TypeTaskIdMap    :: ! TypeTaskIdMap
  , jcd_StateTaskIdMap   :: ! StateTaskIdMap
  }

type JobController         = MVar JobControllerData
```

Listing 4.26: *Types for the job-controller*

The two most important data structures are the `JobMap` and the `TaskMap`. The first map stores all jobs known to the master while the second one holds all tasks. The key to both data types is a unique identification number. Because only the master can create new job and task data structures, it is very easy to fulfil this requirement. The next Id for each type is located in the record fields `jcd_NextJobId` and `jcd_NextTaskId`, respectively. They are incremented each time a new Id is needed.

It is to mention that at this point, there is no garbage collector for the job-controller. That means, new jobs and tasks can be created, but they are not deleted from the map. The reason for this lies in the increase of debugging and failure detection possibilities if this information is still available after a job is fully completed. For a very long runtime of the master, it could mean that the available memory of the process might be exceeded. The implementation of a garbage collector mechanism is very

straightforward. After the job is finished, all its data is deleted from memory. In case the data might really become important, it is worth to think about saving it on local disk permanently before removing it from the memory. To save even more memory, one could imagine to delete the data of a completed task. Since any worker might be unaccessible at some time, the results of each task might be lost after its completion. Therefore, the master should be able to start each task again, but without the task data, this would be impossible.

Although the efficient storing of the job and task data structures is important, most of the master's performance relies on the fast access and filtering of the jobs and tasks. As shown later, the jobs need to be filtered by their current phase, the `JobState`. For the tasks more criteria are needed for filtering: the related job (`JobId`), the type (`TaskType`) and the current processing state (`TaskState`). For a more complex filtering, the combinations of all criteria should be possible. Therefore, four helper data structures are introduced: the `StateJobIdMap`, the `JobIdTaskIdMap`, the `TypeTaskIdMap` and the `StateTaskIdMap`. Each of these data structures maps one filter criteria to a set of Ids. To get all Ids for one criteria value, a lookup has to be performed on the according map. To combine multiple filters, the results for the single lookups have to be combined by appropriate set operations.

Although this filter technique is very simple and fast, there is the problem of keeping the consistency. Each time a task is created or changes its internal state, all four maps need to be checked and updated. Besides this, there are techniques which will do both, store the data and allow filtering under the condition of obtaining consistency. The most common of these techniques is provided by SQL-databases.

There are different database bindings available for the Haskell programming language. The most promising is the SQLite [SQL] implementation. It creates a local database in a single file and thus does not depend on an additional database application. An SQLite database would automatically provide data consistency and persistence. The last feature is very important if the master should be able to resume its work after it was shut down. Maybe future versions of the master program will use such a technique. But at the moment it is the best to use simple and clearly defined data structures to get a good estimation about the complexity of the master's data structures and their access. The SQLite database will add a performance overhead, so it cannot be judged if the overall performance of the master is caused by the main algorithms or only due to a possibly slow database implementation.

Since the job-controller uses the worker-controller to assign tasks to a specific worker, there has to be a way to link these two software modules together. This mechanism is provided in the form of the `JobControlFunctions` record. The `TaskSendFunction` is

called by the job-controller when a new tasks is created and ready for sending. When the task is completed, the response is given via the two functions `setTaskCompleted` and `setTaskError` of the job-controller module.

The heart of the job-controller is the "controller loop". This is an infinite loop in which the scheduling of the jobs and tasks actually takes place. The fields `jcd_ServerThreadId` and `jcd_ServerDelay` are required to control the internal controller loop. The `threadId` is needed to stop the controller properly while the delay between the loops can be used to change the time interval between the loop cycles. No gap time would waste too much processing power because the program would only be busy with performing the loop. Since it is very unlikely that an event will occur in every loop cycle, the waiting time is introduced. Experiments have shown that a value of about 10ms is suitable for most applications.

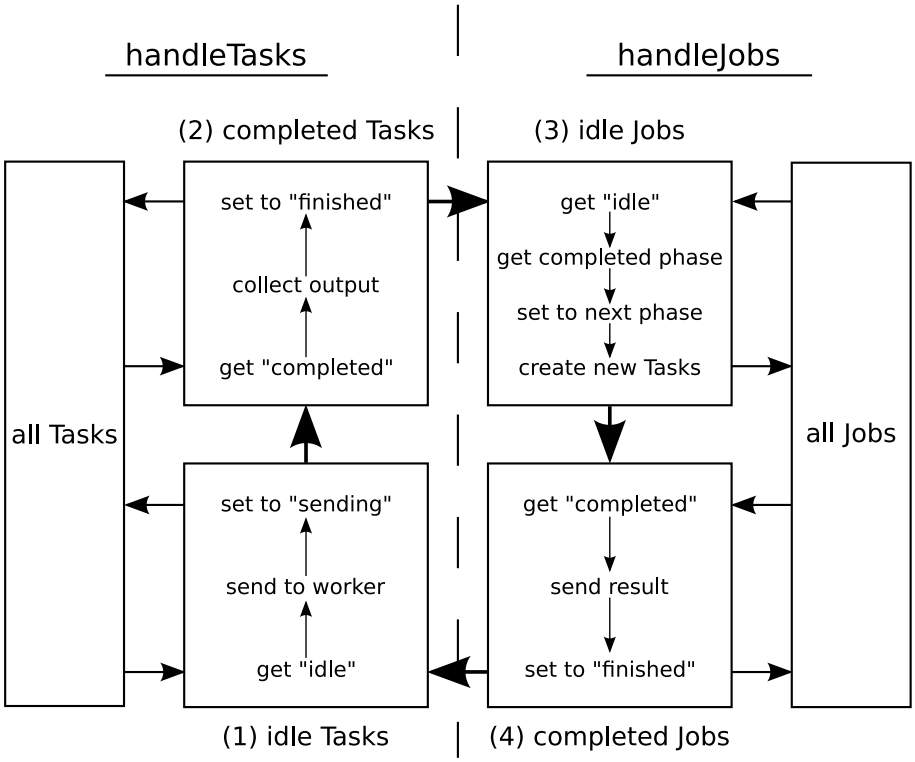


Figure 4.10: *The internal loop of the job-controller*

The detailed sequence of the controller loop is illustrated in figure 4.10. For a better structure, it can be divided into two sequential functions `handleTasks` and `handleJobs`. As the names indicate, the first function schedules the Tasks while the second one manages the jobs.

At the beginning `handleTasks` gets all jobs which are not finished and not flagged as erroneous. Then for these jobs, all tasks are looked up which are not assigned to a worker. Each of these tasks is given to the worker-controller and then sent to an appropriate worker machine. After that the tasks which are marked by the workers as completed are processed. The results of the tasks are added to the job data as the input values for the next phase. If the worker-controller marks one tasks as "not sent", it will be scheduled again in the next cycle. If a task is marked as erroneous by the worker-controller in the case the worker could not process it due to a runtime error, the error counter for the task will be increased by one. If the counter has not exceeded the limit of three processing attempts, the task will be sent again in the next cycle. Otherwise it will stay in the error state and never be processed again.

The function `handleJobs` analyzes if there are any jobs whose tasks of the current phase are completely finished. If there are any, the jobs are set to the next phase and the according tasks are created. The splitting of the input data among the tasks is very straightforward. Because of the partition functions at the end of the previous phase, every key-value-pair of the result list is bundled together with an integer value. These numbers are used to partition the key-value-pairs in the sense that all pairs with the same number are the input of one single task. When the tasks are created they are set to "idle" and are assigned in the next controller cycle.

After the new tasks are created, the function `handleJobs` searches for completed jobs. That means all tasks are finished and no phase is left. The result values of this job are looked up and sent back to the application programs. After that the job is set to the "finished" state and will never be scheduled again.

The controller loop maintains the main functionality of the master program. Although the scheduling progress relies on the actual processing and completion of the tasks. This is located in the counterpart of the master application, the workers.

4.5.6 Worker

Although the performance of each worker strongly depends on the added user functions, a clear and simple design of the software architecture is important, too. Figure 4.11 gives a detailed description about the internal structure of a worker program. Unlike the master, a worker only consists of two major components: the client and the task-processor.

The client is the counterpart of the master's server module. As said before, the clients automatically connect themselves with the server. If the server is unreachable, the client tries to register again. This means that the master and worker programs

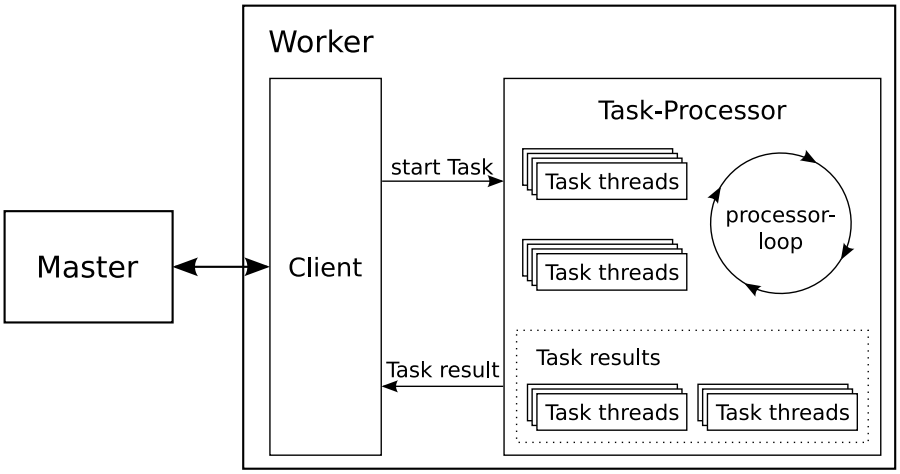


Figure 4.11: *The internal structure of the MapReduce workers*

can be started in any order. Therefore the initialization of the system is much easier and its ability to recover from failures is increased. If a worker crashes, it simply has to be restarted. The registering and resuming of the work is done by the framework. Beside this, the client receives new tasks from the master and passes them to the task-processor.

```
type TaskResultFunction      = TaskData -> IO Bool

data TaskProcessorFunctions = TaskProcessorFunctions {
    tpf_TaskCompleted      :: TaskResultFunction
  , tpf_TaskError          :: TaskResultFunction
  }

data TaskProcessorData       = TaskProcessorData {
    tpd_ServerThreadId     :: Maybe ThreadId
  , tpd_ServerDelay        :: Int
  , tpd_FileSystem         :: FileSystem
  , tpd_Functions          :: TaskProcessorFunctions
  , tpd_ActionMap          :: ActionMap
  , tpd_MaxTasks           :: Int
  , tpd_TaskQueue          :: [TaskData]
  , tpd_CompletedTasks     :: Set TaskData
  , tpd_ErrorTasks         :: Set TaskData
  , tpd_TaskIdThreadMap    :: Map TaskId ThreadId
  }

type TaskProcessor           = MVar TaskProcessorData
```

Listing 4.27: *Types for the task-processor*

The task-processor module handles the execution of the actual tasks and sends them back to the master. Listing 4.27 shows the internal data structure of the task-processor.

One single worker is able to process several tasks at the same time. This kind of implementation supports the future development of the framework and increases its flexibility. Since it is possible that the framework can handle multiple jobs in parallel, the situation might occur that a long running job blocks the whole system and prevents some shorter jobs from running because no workers are available.

The client module puts new tasks into the task-queue (`tpd_TaskQueue`) and the task-processor will start them as soon as possible. For a better control of the program's performance there is a maximum number of tasks a worker is able to process at the same time. It can be set by the field `tpd_MaxTasks`. If this number is exceeded the worker will not start any additional tasks. The tasks stay in the queue until some of the current tasks finish their work.

To be able to run several tasks concurrently and to react on external events while tasks are processing, for every task a new thread is spawned. The ids of the threads are stored in the data structure `tpd_TaskIdThreadMap`. With this data, it is possible to control the threads and keep a record of the actual processed tasks.

As explained above, the user compiles his application code into the worker program. To access this code, the task-processor uses the field `tpd_ActionMap`. The user actions are stored in a map data structure to access them by their name.

The `TaskProcessorFunctions` record builds the bridge between the task-processor and the master program. As explained above, the MapReduce system should be flexible in its structure. Although it is the normal use case, master and workers do not have to be separated in different processes. Therefore, the task-processor in general has to be independent from the actual mechanism to give feedback to the master. In the current implementation this is done by two functions which send the result back to the master via a port object.

Like the master's job-controller the task-processor also consists of a control loop. It is controlled with the two fields `tpd_ServerThreadId` and `tpd_ServerDelay`. The same thoughts for the delay value as in the job-controller loop apply here as well.

The control loop invokes two functions: `handleNewTasks` and `handleFinished-Tasks`. The first one starts new tasks from the queue until it is empty or the maximum number of concurrent tasks is exceeded. The actual calculation of the tasks result takes place in the function `performTask`. It reads all necessary data from the tasks and starts the appropriate action with the given input values.

When a task is finished or has caused an internal error, its result is put either in the `tpd_CompletedTasks` or `tpd_ErrorTasks` set, respectively. The function `handle-`

`FinishedTasks` checks these data structures in the controller loop. If they contain any data, the according `TaskResultFunction` is invoked and the data is sent back to the master.

Although the task-processor itself is very simple, it fulfils all requirements and provides all necessary features for the proper use in the MapReduce system.

5 Conclusion

5.1 Summary and Results

The main goal of this thesis is the implementation of a distributed MapReduce system in Haskell. This goal has been reached with the development of the Holumbus-MapReduce library. The implementation of the communication and storage systems has been necessary to create a stable foundation for the MapReduce system. With the creation of applications like the distributed webcrawler and indexer, it is shown that the design of the system and its interfaces are suitable for real world applications.

As explained before, the four main features of a distributed system are openness, transparency, scalability and fault tolerance. It is now to analyze if the system meets the special requirements or if there is room for optimizations and improvements.

The request for openness which means for this project not to strongly depend on third party programs has easily been met. During the implementation of the framework, it became obvious that there currently are no third party tools which would support the development process. For this reason, the development of the communication library and the storage system were necessary. Foreign Haskell libraries were used if required.

Transparency has been reached on multiple levels. The fact that the MapReduce system is a distributed one is hidden from the application programmer where possible.

Of course, he has to compile his code into the workers and the client, but the interface is not affected by the fact that it is distributed. The demand for network transparency in the communication system is also fulfilled. The implementation of the communication system with the stream-port concept has sped up the development of the entire project.

The question of scalability is very important for a MapReduce system which is designed to run on huge amounts of data. Therefore, many parameters which allow the application programmer to change the granularity of the processed tasks and influence the ability of scalability were introduced. In experiments, using twice as many system components led to an overall time saving of 45%. Since 50% is the maximum value, the loss of time for management overhead is acceptable. Unfortunately and due to the lack of further computing power, the experiments give only a short impression. A reliable performance measurement in a real world application is still missing. This will be done in further projects.

The mechanisms for fault tolerance are not completely implemented. The failure of single workers or processing errors during a task are handled by the master. Other possible faults, like the failure of the master or client causes the system to abort the current work. For smaller systems and processes, this is acceptable because it is comparably unlikely that one of these components fails. Even if so, it is no problem to restart the whole system assuming that the overall computation does not take too much time. For the creation of systems which are more relyable, additional program features have to be implemented.

5.2 Lessons Learned

The Haskell programming language or functional programming in general is still not expected to be common knowledge for every programmer. So before writing the libraries the experiences in these two topics existed only in the field of educational problems which are explained in the common literature about Haskell. Most of these examples only deal with “pure” code, that means functions which do not produce any side effects. Only few papers [Jon01] and books [Hut07; OSG08] deeply explain the proper use of monadic structures for “impure“ statements, like IO operations. Despite this, the learning of the language and its features took less time than expected. Even in the first week of development, productive code has been developed which in parts can still be found in the current version of the framework. More time was consumed by finding optimal patters for designing Haskell modules and their interfaces. Especially learning the proper use of typeclasses and the different ways of error propagation took lots of time and led to many changes to the resulting code.

This framework is the first part of the Holumbus framework which implements a distributed system, consisting of several different components which interact with each other. So no practical experiences in designing such a system in Haskell existed before this work.

In general, it is to say that it is possible in Haskell to build distributed systems and many language features support the programmer in this task. The strong type system of the Haskell language eliminates most of the problems with conflicting types while currying and generic typing add flexibility. The type safe implementation of the generic configuration for the MapReduce system would not have been possible without these features. Compared to non-functional languages like C or Java, Haskell encourages the programmer to use concurrent threads. With `forkIO` and the `MVar` data type, the creation of threads and their communication is pretty easy and less error-prone. During the implementation, no deadlock situation occurred which could be caused by concurrent threads. For the communication between separated processes, the `Data.Binary` library [Bin] was a great help. To be able to serialize data types, only a short typeclass definition has to be provided. In other languages, this mechanism needs a lot more coding.

Despite these advantages, it should be mentioned that other functional programming languages like Erlang or LISP provide better mechanisms for building distributed systems. Especially the existing of a message passing approach like in the Erlang language would have sped up the development process. But for that reason the Holumbus-Distribution library has been created, so that following projects can benefit from it. Another feature which is still not properly implemented in Haskell is the ability of loading new modules dynamically. Other languages, like Java, Erlang or LISP provide this functionality, while in Haskell it is not recommended to use the hs-plugins module in productive environments.

Regarding the MapReduce concept, many new experiences are collected. Although the general architecture is explained in published papers, the detailed functionality and internal structure of the master and worker programs is not part of the literature. The developed structure of the internal control cycles adds a new level of understanding to the whole concept. The partitioning and merging between two MapReduce phases was a topic of research during the development of the Holumbus-Searchengine, but the distribution of these functions required a deeper understanding of the MapReduce approach.

The additions to the basic MapReduce concept where made while implementing the distributed crawler and indexer. Especially the introduction of the global state shows that there might be a difference between the theoretical concept and building a real

world application. When looking at the example MapReduce programs, the concept has shown that in spite of its simplicity, it allows the creation of very complex and powerful data manipulations. Therefore the overall concept has proved its purpose.

5.3 Future Work

Despite all achieved goals, the developed MapReduce framework is still in the state of scientific work. Its use in a commercial environment is not recommended at this stage of development. In order to change this, additions to the framework are necessary.

5.3.1 Communication System

The communication system with the stream and port data types offers all necessary features for building a distributed system in Haskell. There are some weaknesses though, which might be eliminated in the future.

The concept of global streams makes it very easy to build a distributed system. The application programmer only needs the name of the global stream and the port-registry automatically provides a global lookup service for the name. The drawback is that the port-registry is a single point of failure. If it is not available, no global stream can be reached. There are different approaches to solve this problem.

First, in the current implementation every port always connects to the port-registry if it wants to send a message to a global stream. The ports do not have a cache for storing the mapping between stream name and physical address. A temporary failure of the port-registry can be handled by implementing such a cache. The question of how long the cache entries should be stored depends on the actual requirements of the implemented system, so the programmer should be able to set these parameters if needed.

Even with caching, it is required that the port-registry is the first program which has to be started in the system. This reduces the flexibility of the startup process. A global stream only tries to register once at the port-registry. If this fails, the stream is not listed in the registry and therefore not available. Even if the port-registry is shut down due to an internal error an restarted, the existing streams in the network will not repeat their registration. This could be solved when every stream has an internal status field which indicates registration. If it is not registered, it repeats the request until it is successfull. In addition, it could check periodically for the port-registry and renew its entry. With this solution the loss of messages remains possible, but the

whole system could be stabilized without the need for restarting every component in it.

Another interesting approach is the usage of broadcast messages. In case the port-registry is not available, the components of the system could send broadcast messages to each other. So every single process in the system is able to generate a list with all known global stream. This solution has many similarities with the topic of creating adhoc-networks for wireless communication. The algorithms developed in this field, especially for solving routing problems, may be a good starting point for the implementation.

5.3.2 Storage System

The storage system is tightly coupled with the MapReduce framework. It could be used by other applications, too. But for the use in commercial applications, some enhancements are needed. Since the implementation of a storage system was not the main goal of this thesis, this has to be done in future versions.

The concept of storing only small amounts of key-value-pairs into a single file, which is the current implementation, is not very performant. A new set of files are created for every single task a worker processes. It would be better to create bigger files with lots of key-value-pairs. This concept is implemented in the Google File System (GFS) [GGL03] and the Hadoop Distributed File System (HDFS) [Had]. For bigger data sets, the transport and retrieval of data are faster when multiple values can be received with only one request. In both projects, the big container files are called “chunks”.

When using “chunks” the system will need a locking mechanism since multiple nodes could write into the same chunk at the same time. The locking has to be controlled by a global component. The existing controller is predestined for this task.

If the controller is not working, the whole system could not be used. Therefore the question of handling controller failures arises. A quick restart of the controller program is necessary to reduce the downtime of the whole system. The internal state of the controller does not need to be stored on local disk. When the controller is restarted, it will automatically rebuild this information.

Another solution for handling controller failure could be the introduction of backup controllers. If the master controller is not available any more, one of the backup controllers takes over. With this mechanism, the failure of the original controller would not even be recognized by the system nodes. Of course when using this approach, the internal state of the controllers needs to be synchronized somehow.

It is even thinkable to eliminate the need for a global controller. In [Mül07] some experiments with a peer-to-peer storage solution in LISP have been done. It would be interesting to extend this research.

When dealing with large sets of input data, the question arises, how the data gets into the storage system. At the moment, this has to be done manually by the application programmer. Maybe a file explorer for the distributed storage system with an integrated upload and download function would simplify this task and encourage the use of this system.

5.3.3 MapReduce System

The main features of the MapReduce system are implemented in the current version, but for the reliable creation of a large system, further efforts have to be made.

First of all, the question of scalability needs more investigation. Until now only small networks have been created, so it will be interesting to see how the system's performance will evolve by an increasing number of components. Maybe some adjustments regarding the task granularity are required to reach an optimal task distribution among a large number of workers. Detailed attempts for optimization can only be given after further experiments.

On the other hand, in bigger networks, the question of fault tolerance has a more important role. Failures of the master or client components are not handled in the current system, although it is very likely that their occurrence may lead to a complete loss of the calculated results. Especially for long time running programs, it is not acceptable if all computations have to be repeated. Special actions need to be taken to minimize the loss of data and the downtime of the system.

This problem could be solved with simple solutions. The internal state of the master could be stored on local disk periodically. If the master quits running, it could be restarted and resume its work at the latest milestone. Therefore, only the work between the last saved state and the crash of the program has to be repeated. Of course, the whole system would still be out of order until the master program is restarted. To close this time gap, additional master programs could be introduced as backup. When the main master becomes unavailable, one of the backup masters jumps in. Therefore, the downtime of the system will be reduced to a minimum. Like the backup controllers in the storage system, the backup masters have to be synchronized with the main master. This could require a more complex solution.

The failure of the client component is not as severe as an error in the master. If the application program which has started the job is unavailable when the MapReduce

program is finished, the master could just store the result on local disk, so it could be manually retrieved later. This could be implemented together with the saving of the internal state of the master.

Another cause of failures which is not handled in the current system is the occurrence of stragglers. These are workers which are processing their tasks, but for some reasons their performance is very low. This could lead to the situation that the whole system is waiting for one single worker to be finished. To prevent this, the missing task should be given to multiple workers at the same time. In the Google MapReduce system, the master starts to reassign unfinished tasks when 30% of the tasks in the current phase are missing. As described, with this technique more than 20% of the processing time for one MapReduce program is saved in average [DG04].

Besides further investigation of the system's scalability and fault tolerance the software is missing some features which might improve its usability.

In the current implementation the MapReduce master does not delete any internal data. If a master processes many jobs, this could lead to memory problems. Therefore a garbage collector mechanism for the MapReduce master needs to be developed. This could be achieved just by deleting old data when a job is finished. Before the garbage collector starts its work the data might be useful for generating reports and storing them on local disk. These reports could be helpful for debugging and performing statistical evaluations.

The Google MapReduce system and the Apache Hadoop system provide another useful feature, the possibility to create counters. A counter is like a global integer variable and in each task, this variable could be increased. This is very useful to create statistical data, for example to get the exact number of executed tasks. It should be possible to create counter objects dynamically, while the master guarantees the consistency of the actual counter value. When the MapReduce program is finished, the values of all counters are reported to the user. The implementation of this feature does not require extensive changes. Only the task data type needs to be extended and some additional code for handling and storing the counter values has to be added to the master. The access to the counter objects should be similar to the reference of the storage system.

For a better usability it might be considered to implement a mechanism for the automatic distribution of new map and reduce functions. This would simplify the set up of new MapReduce systems. As mentioned before, the use of the hs-plugins library for loading new Haskell modules dynamically at runtime might bring too much limitations. Therefore other solutions need to be investigated, too. It might be possible to extract the map and reduce functions into separate programs and copy these auto-

matically to every worker in the system. During the MapReduce computation these small programs are invoked as independent processes of the operating system. Since all of these concepts add a new level of abstraction to the software, they might bring a loss of performance and lead to less type safety.

In addition to the automatic distribution of new functions, the administration of the system could be improved by redesigning its user interface. In the current implementation, the system can only be controlled with a command line interface. A web interface would simplify the system's management for unexperienced users and allow them to monitor the whole system through a central component. Other MapReduce frameworks like Disco [Dis] also provide a web interface for their administration.

5.4 Outlook

The next steps for this project are already planned. First, the source code of the implemented libraries will be published as open source software. With this step, a wider range of possible use cases is expected and maybe some important impulses for the further development are received from additional users.

Of course, the ideas for future work described in the previous section will be implemented. The most important task is to do more research about the system's scalability when running in a larger environment. In his Master's Thesis Sebastian Reese will investigate the use of this system in the five computer pools of the University of Applied Sciences in Wedel. Each of these consists of about 30 computers.

It is planned to create a high performance distributed crawler and indexer for the Holumbus-Searchengine. When being able to generate indexes over very large document sets, the searchengine itself needs to be extended. One of the goals is to split the index among multiple computers to create a distributed searchengine. Since there are only few research groups in this topic and a very limited amount of related literature, much research has to be done.

A

Example MapReduce Programs

To illustrate the flexibility of the MapReduce concept and give some ideas how to use the framework four example programs are introduced. Some of them are already shown in [DG04], but here they are implemented in Haskell. It is interesting to see how small but powerfull they are.

A.1 Word-frequency

The calculation of the word-frequency is the one of the most common examples when introducing the MapReduce concept. Word-frequency in this context means to count how often a distinct word appears in a collection of texts. For example it might be interesting to know how many times the word "Lord" is mentioned in the Bible or which word is mostly used in Shakespeare's plays. These statistics could be used to give a first measurement for the strength of expression of any literary text or to optimize text-based compression algorithms [MT02].

The input of the map phase is a list of textfiles. The whole content of each textfile is represented by one large string. It is split into single words with the help of the `words` function of the standard library. It just uses whitespaces as delimiter for the separation of distinct words. To retrieve a better result, a removal of all punctuation marks, a transformation to lower case letters and a word stemming might be useful.

```
mapWordFrequency
  :: () -> String -> String -> IO [(String, Integer)]
mapWordFrequency _ _ v
  = return $ map (\s -> (s,1)) $ words v
```

Listing A.1: *Map function of the word-frequency example*

As shown in listing A.1 for every word, a key-value-pair is created. The word represents the key while the value is set to "1". These pairs are partitioned and sent back to the master. The default partition function guarantees that all pairs with the same key are assigned to the same reducer machine. In the reduce phase each worker groups the key-value-pairs by their keys and then processes the reduce function in listing A.2.

```
reduceWordFrequency
  :: () -> String -> [Integer] -> IO (Maybe Integer)
reduceWordFrequency _ _ vs
  = return $ Just $ sum vs
```

Listing A.2: *Reduce function of the word-frequency example*

In the reduce function, the sum of all integer values for one distinct word is calculated. After that the number of occurrences for each word is known and passed to the user.

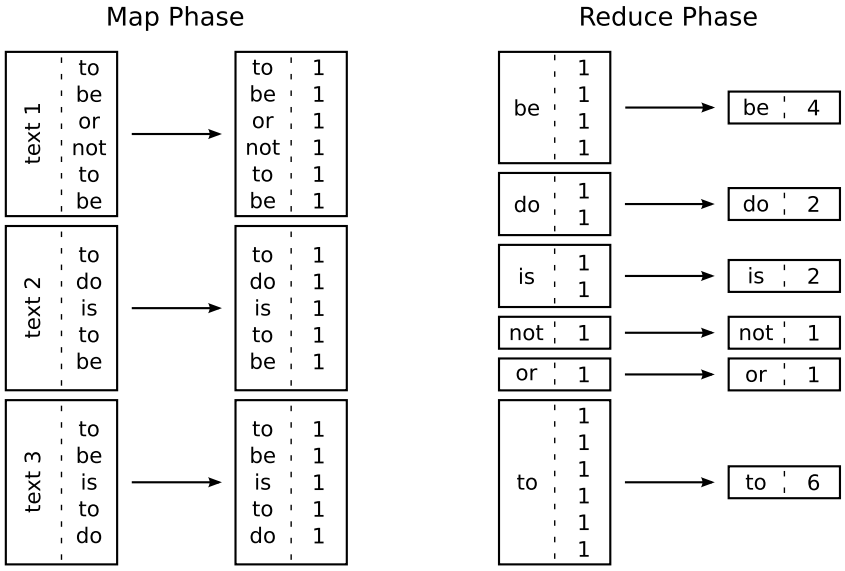


Figure A.1: *An example calculation of the word-frequency*

Since the mapping of each word to the number "1" is not very intuitive, figure A.1 illustrates the data transformation in the map and reduce phase.

For this simple example, there is no need to overwrite the default merge and partition functions. To control the actual distribution of the words among the reducers the partition function at the end of the map phase could be redefined.

The introduction of a combine phase is very straightforward. In this additional phase it is possible to calculate a word frequency for each mapper and merge them in the reduce phase. This will significantly lower the number of key-value-pairs which are processed by the reducers and therefore decrease the amount of data transmissions. Since the result types of the map and reduce phase are the same, the reduce function definition of listing A.2 can also be used in the combine phase.

A.2 Distributed Sort

Since the MapReduce framework groups the key-value-pairs between two following phases, this could be used to sort the data. To illustrate how his mechanism could be used, a list of names is sorted in this example.

```
mapSort
  :: () -> () -> (String, String)
  -> IO [(String, (String, String))]
mapSort _ _ v
  = return $ [(snd v, v)]
```

Listing A.3: *Map function of the distributed sort example*

In order to do this, the map function shown in listing A.3 takes a list of tuples as value element. The first element of a tuple represents the forename while the second one the surname. The map function just creates a new key-value-pair with the surname as key and the input tuple as value element.

```
reduceSort
  :: () -> String -> [(String, String)]
  -> IO (Maybe [(String, String)])
reduceSort _ k vs
  = return (Just vs)
```

Listing A.4: *Reduce function of the distributed sort example*

The default partition function at the end of the map phase takes care of the correct splitting of the data among the reducer machines. The default merge function at the beginning of the reduce phase groups the name-pairs with the same surname. Another feature of this grouping which is crucial for this example, is that the keys are sorted in ascending order. The reduce function just returns the input values, similar to the `id` function of the Haskell standard library.

The efficiency of this approach mainly depends on the computational complexity of the sorting algorithm in the merge function and the number of reducers.

The sorting is realized by inserting the key-value pairs into a modified version of the `Data.Map` data structure which can handle multiple entries with the same key. Inserting elements into this type of map and converting it into a sorted map has a time complexity of $O(n \cdot log(n))$.

The actual runtime can be decreased by adding more workers in the reduce phase. If only one machine is used, the sorting is not distributed. When using multiple reducers, the result consists of distinct sublists. To get a overall result, these lists have to be merged. With a smart implementation of the partition function at the end of the map phase, for all sublist the ranges of the key-values do not overlap. Then the lists only need to be concatenated which is a very fast operation compared to a complex merging.

The implementation of this distributed sort gives a good impression of the internal handling of the key-value-pairs. It is questionable if this implementation really is suitable for practical applications since the network communication will produce a very high overhead compared to the actual processing. Therefore further investigation of this application is necessary.

A.3 Distributed Grep

The distributed grep program has the same general functionality than the well-known command line program. It tests every line of the input files if it matches a given pattern which is defined by a regular expression.

The input for the map phase is a tuple list whereas the first element is the filename while the second one represents the file content in a single string.

```
mapGrep
  :: String -> String -> String
  -> IO [((),(String,String))]
mapGrep e k v
  = return $ mapMaybe (getMatch) $ lines v
    where
    getMatch l
      = if (isJust $ matchRegex (mkRegex e) l)
        then Just ((),(k,l))
        else Nothing
```

Listing A.5: *Map function of the distributed grep example*

The map function in listing A.5 needs a third input parameter, the regular expression for the pattern matching. To pass this value to the map function, the global configuration parameter is used. This shows that for the solution of some problems a global configuration is required and that its introduction really is an important enhancement of the MapReduce concept. The alternative would be to compile the pattern directly into the map function. The disadvantage of this approach is that the workers have to be recompiled everytime the pattern changes and that the whole system needs to be restarted.

The map function splits the file content into separate lines with the help of the `lines` function of the Haskell standard library. For every line it is tested if it matches the given pattern. If this is the case the line is passed to the reduce phase with additional information about its original filename.

```
reduceGrep
  :: String -> () -> [(String,String)]
  -> IO (Maybe [(String,String)])
reduceGrep _ _ vs
  = return (Just vs)
```

Listing A.6: *Reduce function of the distributed grep example*

The reduce phase just collects all lines and passes them to the user. Therefore the reduce function in listing A.6 does no data manipulation and only returns its input values.

The efficiency of the distributed grep program is determined by the number of workers in the map phase. The overall time saving results from the use of parallel grep processes. For the reduce phase only a small number of workers or even only a single machine is needed.

A.4 Distributed Webcrawler and Indexer

One of the main use cases of the MapReduce framework developed in this thesis is to build a distributed webcrawler and indexer for the Holumbus-Searchengine library. Therefore the existing webcrawler and indexer which is based on the non-distributed MapReduce implementation by Sebastian Schlatt needs to be transferred to the Holumbus-MapReduce framework.

Sebastian Schlatt came up with the idea of implementing a distributed webcrawler by using MapReduce. The general idea of this approach is illustrated in figure A.2. The crawler has a global configuration state, called “crawler-state”, which contains two lists. The first list stores the Uniform Resource Locators (URLs) of all websites

which need to be crawled (todo-list) while the second one contains the URLs of all sites which were already processed (done-list). Unless the todo-list is not empty, the workers in the map phase load the requested websites, extract their links and save them on local disk. The processed URLs and the links are collected in the reduce phase and with the help of the previous todo- and done-lists the crawler-state is updated. This is only successful, if the reduce phase is processed on only one single worker. A more detailed description of this approach can be found in [Sch08].

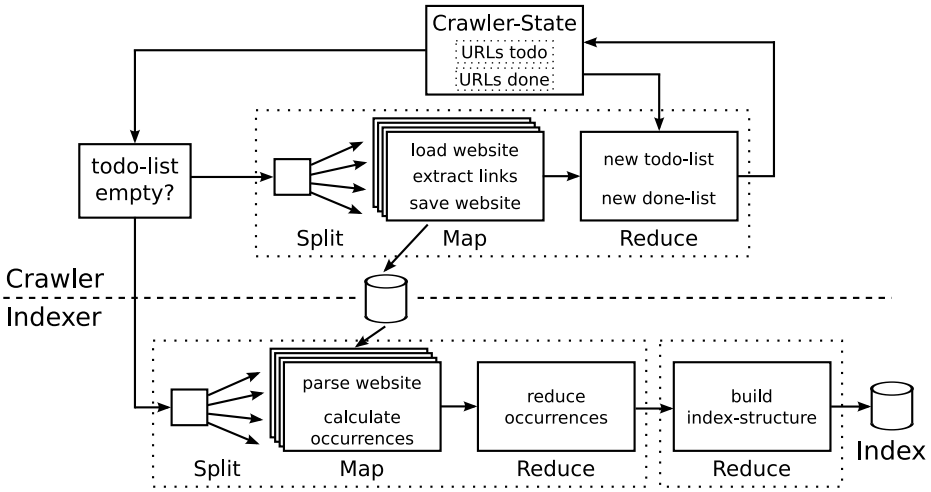


Figure A.2: *Action-sequence of the distributed crawler and indexer*

When the crawler is finished, that means the todo-list becomes empty, the indexer is started. It consists of two MapReduce programs.

In the first sequence, a so called "pre-index" is created. In order to do this, each website is loaded from local disk. After this word-occurrences are extracted from the content of the website. In general, these are pairs of distinct words and some information about their position inside the document. All the word-occurrences are collected in the reduce phase and the pre-index is build. Because this data structure is not suitable for online search, a second MapReduce has to be started. It only consits of the reduce phase and transforms the pre-index into a Holumbus searchindex data structure. In detail, the document processing is described in [Sch08] and the index construction in [Hüb08].

The challenge in this work was the transformation of the crawler and indexer codebase from the existing single process MapReduce implementation of Sebastian Schlatt to the distributed framework. Although the general concept has not changed, the interfaces of both systems are slightly different. The major problem was the fact that in the single process crawler and indexer abstract types like function parameters are

widely used for configuration purposes. Because in the distributed MapReduce system every data which needs to be transferred over the network has to implement the binary typeclass, abstract types cannot be used. This problem was solved by directly compiling the abstract types into the worker program. This means that the user is not able to change them without recompiling the worker code.

The distributed crawler and indexer program is tested for functionality but not for efficiency. At this stage of development it is not suitable for practical applications but it is a good starting point for further investigations and optimizations.

B Manual

The source code of this work is available at http://www.holumbus.org/. The website contains additional information about all parts of the Holumbus framework. This manual only describes the installation and general usage of Holumbus-Distribution, Holumbus-Storage and Holumbus-MapReduce.

B.1 Installation

Compiling the three libraries requires the current version of the GHC (6.10.1) [GHC] with the included implementation of the Haskell hierarchical libraries. In addition to this three additional packages are needed which can be downloaded from the Hackage database [Hac]:

- binary 0.4.4 or higher
- hslogger 1.0.7 or higher
- hxt 8.2.0 or higher

The build process is controlled by Cabal [Cab], a packaging system used for the distribution of Haskell libraries and programs. In general, Cabal is included with the GHC. To install the Holumbus libraries version 1.6 or higher is required.

The following commands process the configuration, compilation and installation of one library:

```
$ runhaskell Setup.hs configure
$ runhaskell Setup.hs build
$ sudo runhaskell Setup.hs install
```

They have to be executed in the root directories of Holumbus-Distribution, Holumbus-Storage and Holumbus-MapReduce. This order of the libraries is significant because of their internal dependencies.

If desired Cabal can also be used to generate the HTML documentation for the libraries. The following command only has to be executed for each of the three libraries:

```
$ runhaskell Setup.hs haddock
```

In addition to this, the GNU make tool is needed to compile the example programs. The names of the specific build targets can be found in the provided makefiles.

B.2 Holumbus-Distribution

After the installation is completed, the Holumbus-Distribution library can be used to build distributed systems. Although global streams can only be accessed from other programs if one instance of the port-registry program is running inside the network. Therefore the port-registry has to be started with the following command before all other components of the distributed system:

```
$ PortRegistry
```

The port-registry is a command line program and accepts all the instructions shown in listing B.1.

```
exit        –  exits the program
help        –  prints this help
lookup      –  gets the socket id for a port
ports       –  lists all ports
register    –  registers a port manually
unregister  –  unregisters a port manually
version     –  prints the version
```

Listing B.1: *Instructions for the port-registry*

When the port-registry is started, it creates the file `registry.xml` in the system's tmp-directory. It contains information how to access the port-registry and has to be copied to every machine which will be part of the distributed system.

The components which want to access the port-registry need to load the XML file by calling the function `newPortRegistryFromXmlFile` at program startup. After this the internal datastructures have to be updated by invoking `setPortRegistry`.

```
import Holumbus.Network.PortRegistry.PortRegistryPort
import Holumbus.Network.Port

main :: IO ()
main
  = do
    reg <- newPortRegistryFromXmlFile "/tmp/registry.xml"
    setPortRegistry reg

    gS <- (newGlobalStream "global")::IO (Stream String)

    msg <- readStream gS
    putStrLn msg

    closeStream gS
```

Listing B.2: *Example for the use of global streams*

The program in listing B.2 sets up a global stream with the name "global". Then it waits for an incoming message, prints out its content and terminates. Listing B.3 shows how to send messages to the stream.

```
import Holumbus.Network.PortRegistry.PortRegistryPort
import Holumbus.Network.Port

main :: IO ()
main
  = do
    reg <- newPortRegistryFromXmlFile "/tmp/registry.xml"
    setPortRegistry reg

    gP <- (newGlobalPort "global")::IO (Port String)

    send gP "Hello World"
```

Listing B.3: *Example for the use of global ports*

B.3 Holumbus-Storage

To integrate the Holumbus-Storage library into self-written applications, the following steps are necessary:

1. Start the port-registry program
2. Load the port-registry configuration
3. Create a storage instance (controller, node, client, standalone)
4. Use the storage system
5. Close the storage instance

The port-registry program has to be started only once inside the whole network while the other steps are required in every program which wants to access the storage system. Listing B.4 illustrates how to instantiate a stand-alone version of the storage system. The program creates a new file with initial data, reads its content and finally deletes the file.

```
import Data.Binary
import Holumbus.Network.PortRegistry.PortRegistryPort
import Holumbus.FileSystem.FileSystem

main :: IO ()
main
  = do
      p <- newPortRegistryFromXmlFile "/tmp/registry.xml"
      setPortRegistry p

      fs <- mkStandaloneFileSystem defaultFSStandaloneConfig

      createFile "myFile" (encode "Hello World") fs

      c <- getFileContent "myFile" fs
      case c of
        (Just bs) -> putStrLn $ "content: " ++ (decode bs)
        (Nothing) -> putStrLn "file not found"

      deleteFile "myFile" fs

      closeFileSystem fs
```

Listing B.4: *Example for the use of the storage system*

B.4 Holumbus-MapReduce

To set up a MapReduce system, the port-registry and the MapReduce master applications have to be started first. After the installation of Holumbus-MapReduce the master can be executed with the command:

```
$ Master
```

Like the port-registry, the master is a command line program. The available instructions for controlling the master are given in listing B.5.

The single step mode is for debug purposes only. If this mode is activated with `startC`, the master only processes the next controller cycle after the `step` command. With the help of the `debug` command, the change of the master's internal state can be monitored.

```
debug       -  prints the internal state of the master
exit        -  exits the program
help        -  prints this help
startC      -  switches from normal mode to single step mode
step        -  performs the next step in single step mode
stopC       -  switches from single step mode to normal mode
version     -  prints the version
```

Listing B.5: *Instructions for the MapReduce master*

After the master has been started, it is time to set up the workers. Listing B.6 gives an impression how a worker can be implemented. For the construction of a new worker instance the function `mkMapReduceWorker` needs information about the MapReduce actions the worker should be able to process. In this example, the map which contains the actions is named `wordFrequencyActionMap` and is declared in listing B.8.

```
import Holumbus.Network.PortRegistry.PortRegistryPort
import Holumbus.FileSystem.FileSystem
import Holumbus.Distribution.DMapReduce
import Holumbus.MapReduce.UserInterface

main :: IO ()
main
  = do
    p <- newPortRegistryFromXmlFile "/tmp/registry.xml"
    setPortRegistry p

    fs <- mkFileSystemNode defaultFSNodeConfig
    mr <- mkMapReduceWorker fs wordFrequencyActionMap
                            defaultMRWorkerConfig
```

```
    runUI mr "worker version 0.1"

    closeMapReduce mr
    closeFileSystem fs
```

Listing B.6: *Example for creating a MapReduce worker*

```
import Holumbus.Network.PortRegistry.PortRegistryPort
import Holumbus.Distribution.DMapReduce
import Holumbus.MapReduce.Types

main :: IO ()
main
  = do
    p <- newPortRegistryFromXmlFile "/tmp/registry.xml"
    setPortRegistry p

    mr <- mkMapReduceClient defaultMRClientConfig

    (ls, _) <- doMapReduce
                 wordFrequencyAction () textList []
                 1 5 1 1 TOTRawTuple mr

    putStrLn $ show ls
    closeMapReduce mr

textList :: [(String, String)]
textList
  = [("text1", "to be or not to be")
    ,("text2", "to do is to be")
    ,("text3", "to be is to do")
    ]
```

Listing B.7: *Example for creating a MapReduce client*

After each worker has registered itself at the master the MapReduce computation can be started. The according source code for the client is shown in listing B.7. Because the client has to know which MapReduce action should be executed, the function `doMapReduce` requires additional information. This is provided by the action configuration `wordFrequencyAction` in listing B.8.

The action configuration is a record type which contains all user-defined functions needed for the MapReduce processing. The configuration in this example is needed for calculation of the word-frequency, therefore it is labeled “WORDFREQUENCY”.

In the split phase the default implementation is used, that means although the programmer has not specified a function this phase will be executed by the system. In the map phase the function `mapWordFrequency` is processed while the reduce phase

invokes `reduceWordFrequency`. The combine phase is left out because no configuration is given. In the map and reduce phase the merge and partition functions are not overwritten, so the default implementations will be used.

```
wordFrequencyAction
  :: ActionConfiguration
       ()                                                  -- state
       String String                                       -- k1, v1
       String Integer                                      -- k2, v2
       Integer                                             -- v3 == v2
       Integer                                             -- v4
wordFrequencyAction
  = (defaultActionConfiguration "WORDFREQUENCY")
       { ac_Map     = Just mapAction
       , ac_Combine = Nothing
       , ac_Reduce  = Just reduceAction
       }
    where
    mapAction
      = (defaultMapConfiguration mapWordFrequency)
    reduceAction
      = (defaultReduceConfiguration reduceWordFrequency)

wordFrequencyActionMap :: ActionMap
wordFrequencyActionMap
  = KeyMap.insert (readActionConfiguration wordFrequencyAction) $
    KeyMap.empty
```

Listing B.8: *Action configuration for calculating the word-frequency*

Bibliography

A9c *A9 Product Search*. http://a9.com/, last checked: 21.02.2009

Ama *Amazon Elastic Compute Cloud (Amazon EC2)*. http://aws.amazon.com/ec2/, last checked: 21.02.2009

Arm07 Armstrong, Joe: *Programming Erlang: Software for a Concurrent World*. Pragmatic Programmers, 2007

Ben *The Computer Language Benchmarks Game*. http://shootout.alioth.debian.org/, last checked: 19.02.2009

Bin *Data.Binary*. http://code.haskell.org/binary/, last checked: 21.02.2009

Bit *Bittorrent Inc.* http://www.bittorrent.com/, last checked: 23.02.2009

BM96 Blelloch, Guy E. ; Maggs, Bruce M.: Parallel algorithms. In: *ACM Computing Surveys* 28 (1996), Nr. 1, pages 51–54

BP98 Brin, Sergey ; Page, Lawrence: The Anatomy of a Large-Scale Hypertextual Web Search Engine. In: *Computer Networks and ISDN Systems*, 1998, pages 107–117

BYRN99 Baeza-Yates, Ricardo ; Ribeiro-Neto, Berthier A.: *Modern Information Retrieval*. New York : ACM Press, 1999

Cab *The Haskell Cabal*. http://haskell.org/cabal, last checked: 22.02.2009

CAK+07 Cell, Mapreduce ; Architecture, B. E. ; Kruijf, Marc D. ; Sankaralingam, Karthikeyan ; Kruijf, Marc ; Sankaralingam, Karthikeyan: *Abstract MapReduce for the Cell B.E. Architecture*. 2007

CDK01 Coulouris, George ; Dollimore, Jean ; Kindberg, Tim: *Distributed Systems - Concept and Design, third edition*. Addison-Wesley, 2001

CKL+06 Chu, Cheng-Tao ; Kim, Sang K. ; Lin, Yi-An ; Yu, YuanYuan ; Bradski, Gary R. ; Ng, Andrew Y. ; Olukotun, Kunle: Map-Reduce for Machine Learning on Multicore. In: *NIPS*, 2006, pages 281–288

DG04 Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *OSDI '04: Proceedings of the 6th symposium on Operating System Design and Implementation*. Berkeley : USENIX Association, 2004, 137–150

Dij02 Dijkstra, Edsger W.: Cooperating sequential processes. New York, NY, USA : Springer-Verlag New York, Inc., 2002, pages 65–138

Dis *Disco: massive data – minimal code*. http://discoproject.org/, last checked: 23.02.2009

EM02 Eppler, Martin ; Mengis, Jeanne: *The Concept of Information Overload - A Review of Literature from Organization Science, Marketing, Accounting, MIS, and related Disciplines*. http://www.knowledgemedia.org/. Version: 2002

Erla *Erlang*. http://www.erlang.org/, last checked: 19.02.2009

Erlb *Erlang-style Distributed Haskell*. http://www.iist.unu.edu/~vs/haskell/dhs/, last checked: 20.02.2009

GdH *Glasgow distributed Haskell (GdH)*. http://www.macs.hw.ac.uk/~dsg/gdh/, last checked: 20.02.2009

GGL03 Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google File System. In: *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA : ACM, 2003, 29–43

GHC *The Glasgow Haskell Compiler*. http://haskell.org/ghc/, last checked: 22.02.2009

Glu *GlusterFS - GNU Cluster File System*. http://www.gluster.org/, last checked: 19.02.2009

Hac *HackageDB*. http://hackage.haskell.org/packages/hackage.html, last checked: 22.02.2009

Had *Apache Hadoop*. http://hadoop.apache.org/, last checked: 23.02.2009

Ham05 Hammerschall, Ulrike: *Verteilte Systeme und Anwendungen - Architekturkonzepte. Standards und Middleware-Technologien*. Pearson Studium, 2005

Has *Haskell*. http://haskell.org/, last checked: 22.02.2009

Hay *Hayoo!* http://www.holumbus.org/hayoo, last checked: 23.02.2009

HFL+08 He, Bingsheng ; Fang, Wenbin ; Luo, Qiong ; Govindaraju, Naga K. ; Wang, Tuyong: Mars: a MapReduce framework on graphics processors. In: *PACT '08: Proceedings of the 17th international conference on Parallel*

architectures and compilation techniques. New York, NY, USA : ACM, 2008, pages 260–269

HHJW07 Hudak, Paul ; Hughes, John ; Jones, Simon P. ; Wadler, Philip: A history of Haskell: being lazy with class. In: *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. New York, NY, USA : ACM, 2007, pages 12–1–12–55

HN00 Huch, Frank ; Norbisrath, Ulrich: Distributed programming in Haskell with ports. In: *Lecture Notes in Computer Science* Vol. 2011, Springer, 2000, pages 107–121

Hol *Holumbus*. http://www.holumbus.org, last checked: 23.02.2009

Hoo *Hoogle*. http://haskell.org/hoogle/, last checked: 23.02.2009

Hüb08 Hübel, Timo B.: *The Holumbus Framework - Creating fast, flexible and highly customizable search engines with Haskell*. 2008. – University of Applied Sciences Wedel – Master's Thesis

Huc99 Huch, Frank: Erlang-style Distributed Haskell. In: *In Draft Proceedings of the 11th International Workshop on Implementation of Functional Languages, September 7th 10th*, 1999

Hug89 Hughes, John: Why Functional Programming Matters. In: *Computer Journal* 32 (1989), Nr. 2, pages 98–107

Hut07 Hutton, Graham: *Programming in Haskell*. Cambridge : Cambridge University Press, 2007

Jon01 Jones, Simon P.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: *Engineering theories of software construction*, Press, 2001, pages 47–96

Läm06 Lämmel, Ralf: *Google's MapReduce Programming Model – Revisited*. 2006. – Accepted for publication in the Science of Computer Programming Journal

LV03 Lyman, Peter ; Varian, Hal R.: *How Much Information?* Version: 2003. http://www.sims.berkeley.edu/how-much-info-2003, last checked: 20.02.2009

LZW04 Lester, Nicholas ; Zobel, Justin ; Williams, Hugh E.: In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In: *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2004, pages 15–23

Moo65 Moore, Gordon E.: Cramming more components onto integrated circuits. In: *Electronics Magazine* 38 (1965)

Moz *The Mozart Programming System.* http://www.mozart-oz.org/, last checked: 19.02.2009

MT02 Moffat, Alistair ; Turpin, Andrew: *Compression and Coding Algorithms.* Dordrecht : Kluwer Academic Publishers, 2002

Mül07 Müller, Sebastian: *Sisyphos: Konzeption und Entwicklung eines Peer-to-Peer Server-Systems zur verteilten Speicherung, Organisation und parallelisierten Verarbeitung von Textdaten mit einer Programmierschnittstelle in Common Lisp.* 2007. – University of Applied Sciences Wedel – Diplomarbeit

MWZ06 Moffat, Alistair ; Webber, William ; Zobel, Justin: Load balancing for term-distributed parallel retrieval. In: *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval.* New York, NY, USA : ACM, 2006, pages 348–355

NFS *Network File System version 4 (NFSv4).* http://nfsv4.org/, last checked: 19.02.2009

OSG08 O'Sullivan, Bryan ; Stewart, Don ; Goerzen, John: *Real World Haskell.* O'Reilly, 2008

Por *Port-based Distributed Haskell.* http://www-i2.informatik.rwth-aachen.de/Research/distributedHaskell/index.html, last checked: 20.02.2009

PSSC04 Pang, André ; Stewart, Don ; Seefried, Sean ; Chakravarty, Manuel M. T.: Plugging Haskell In. In: *Proceedings of the ACM SIGPLAN workshop on Haskell.* Snowbird, Utah, USA : ACM Press, 2004, 10–21

PTL01 Pointon, Robert F. ; Trinder, Philip W. ; Loidl, Hans-Wolfgang: The Design and Implementation of Glasgow Distributed Haskell. In: *IFL '00: Selected Papers from the 12th International Workshop on Implementation of Functional Languages.* London, UK : Springer-Verlag, 2001, pages 53–70

Pyt *Python Programming Language – Official Website.* http://www.python.org/, last checked: 19.02.2009

RH04 Roy, Peter V. ; Haridi, Seif: *Concepts, Techniques, and Models of Computer Programming.* The MIT Press, 2004

RRP+07 Ranger, Colby ; Raghuraman, Ramanan ; Penmetsa, Arun ; Bradski, Gary ; Kozyrakis, Christos: Evaluating MapReduce for multi-core and multiprocessor systems. In: *In HPCA 07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, 2007, pages 13–24

Sch08 Schlatt, Sebastian M.: *The Holumbus Framework: Creating scalable and customizable crawlers and indexers.* 2008. – University of Applied Sciences Wedel – Master's Thesis

SH01 Stolz, Volker ; Huch, Frank: *Implementation of Port-based Distributed Haskell.* 2001

Sno92 Snow, C. R.: *Concurrent Programming.* Cambridge University Press, 1992

SQL *SQLite.* http://www.sqlite.org/, last checked: 21.02.2009

TS07 Tanenbaum, Andrew S. ; Steen, Maarten V.: *Distributed Systems - Principles and Paradigms, second edition.* Pearson Prentice Hall, 2007

TW06 Tanenbaum, Andrew S. ; Woodhull, Albert S.: *Operating Systems - Design and Implementation.* Pearson Prentice Hall, 2006

WMB99 Witten, Ian H. ; Moffat, Alistair ; Bell, Timothy C.: *Managing Gigabytes - Compressing and Indexing Documents and Images.* San Francisco : Morgan Kaufmann Publishers, 1999

Yah *Yahoo Launches World's Largest Hadoop Production Application.* http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html, last checked: 23.02.2009

YDHP07 Yang, Hung chih ; Dasdan, Ali ; Hsiao, Ruey-Lung ; Parker, D. S.: Map-reduce-merge: simplified relational data processing on large clusters. In: *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data.* New York, NY, USA : ACM, 2007, pages 1029–1040

Acknowledgments

I would like to thank some people who supported me during the creation of this work.

First of all, I would like thank Prof. Dr. Uwe Schmidt from the University of Applied Sciences in Wedel for the supervision of my thesis. Together with Timo Hübel and Sebastian Schlatt we have had friendly and productive discussions about the project which gave me new impulses and ideas.

Beyond this, Timo and Sebastian helped me with the first steps in the Holumbus framework. They were always open for questions and provided useful Haskell related information.

Sebastian Reese did a great job for providing the Holumbus framework with the official repository and server. I hope he will find the time to start with his thesis soon and bring the framework to the next level.

Special thanks to Simon Adler, Rebecca and Thorsten Ehlers, Ariane Köthe, Martin Schmidt and Florian Stumpf for the mental support, the right words at the right time and having the patience reading this work and giving important feedback.

Last but not least, I would like to thank the people who gave me the opportunity for my study, my parents. Without their help and support, especially in the hard times, I would not even have been able to start this thesis.

Thank you.

Affidavit

I hereby declare that this thesis has been written independently by me, solely based on the specified literature and resources. All ideas that have been adopted directly or indirectly from other works are denoted appropriately. The thesis has not been submitted to any other board of examiners in its present or a similar form and was not yet published in any other way.

Place, Date *Stefan Schmidt*