University of Zurich^UZH

Design, Prototypical Implementation and Traffic Evaluation of MapReduce on TomP2P

Oliver Zihler
Kloten, Switzerland
Student ID: 06-729-420

Supervisor: Dr. Thomas Bocek, Patrick Poullie
Date of Submission: May 4, 2016

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

This master thesis presents the design and implementation of a prototype for executing MapReduce jobs on a peer-to-peer system (TomP2P) with high user implementation flexibility and without master-slave architecture. Instead of map and reduce functions, users define a number of *tasks* that are chained together for as many steps as needed until a final result is obtained. The prototype is completely decentralised: every node contains the same code and can start MapReduce job executions without any central entity required to assign work. In cases of node crashes, fault tolerance measures allow other nodes to resume processing of failed tasks without additional coordination mechanisms needed. Data storage is provided as a customised distributed hash table, and communication is limited to one type of broadcast messages to inform nodes about executable tasks. The design is evaluated for produced traffic, where a word count solution is implemented and distributed to 3 - 6 nodes. The analysis covers both single job executions run from one machine as well as two jobs submitted simultaneously from two nodes. The initial evaluation shows constant average traffic per node with certain user-controllable deviations.

Zusammenfassung

Diese Masterarbeit präsentiert das Design und die Implementierung eines Prototypen zur Ausführung von MapReduce Jobs auf einem Peer-to-Peer System (TomP2P) mit hoher Implementierungsflexibilität für Nutzer und ohne Master-Slave Architektur. Statt Map- und Reducefunktionen definieren Nutzer eine Anzahl *Tasks*, welche dann zu beliebig vielen Schritten verkettet werden können, um ein gewünschtes Resultat zu erreichen. Der Prototyp ist komplett dezentralisiert: jeder Knoten führt denselben Code aus und kann MapReduce Jobausführungen starten, ohne eine zentrale Einheit zur Zuweisung von Arbeit zu benötigen. Im Falle von Knotenabstürzen erlauben Störungstoleranzmassnahmen anderen Knoten die Bearbeitung fehlgeschlagener Tasks wiederaufzunehmen, ohne zusätzliche Koordinationsmechanismen zu benötigen. Datenspeicherung wird als angepasste verteilte Hashtabelle angeboten und Kommunikation ist limitiert auf einen Typen von Broadcastmitteilung, um Knoten über ausführbare Tasks zu informieren. Das Design wurde evaluiert bezüglich produziertem Traffic, wobei eine Lösung zum Zählen von Wörtern implementiert und auf 3 - 6 Knoten ausgeführt wurde. Die erste Analyse deckt die Bearbeitung von einem einzelnen Job gestartet von einer einzigen Maschine aus ab. Die zweite Analyse befasst sich mit der gleichzeitigen Ausführung zweier Jobs durch 2 Maschinen. Die Evaluierung zeigt konstanten durchschnittlichen Traffic pro Knoten mit gewissen durch den Nutzer kontrollierbaren Abweichungen.

Acknowledgments

I would like to thank everyone that helped me in the creation of this thesis.

First and foremost, I want to thank Thomas Bocek for his constant advices, guidances, and especially patience during the implementation of this prototype.

Furthermore, I thank Prof. Dr. Burkhard Stiller for the possibility to write this thesis at the Department of Informatics of the University of Zurich.

Last but not least, I thank my family and friends for their constant support and patience. Special thanks go to David and Frida for inspiring discussions during lunch breaks and working hours. Let us continue to bring the hammer to work in the time to come.

Contents

Chapter 1

Introduction

1.1 Motivation

MapReduce is a paradigm for analysing big data made popular by Google in the early 2000 [16]. It is inspired by map and reduce primitives of functional languages: a user defines two procedures, map and reduce, where map takes an input key-value pair and produces a set of intermediate key-value pairs, which are then passed to the user-defined reduce function that accepts an intermediate key and its associated values and merges them to form a possibly smaller set of values. The application of MapReduce has been shown for a wide range of domains, including machine learning problems, extraction of properties from web pages, large graph computations, etc.

Several open source adaptations of MapReduce exist, of which a popular example is Apache Hadoop [10]. Apache Hadoop relies on a centralised master-slave architecture similar to Google's implementation [16] that, although being resilient to slave disconnections, does not handle master crashes or joins and leaves of nodes well in general [20, 27]. It thus best requires a dedicated environment of homogeneous computers for a seamless operation. However, in many cases, such dedicated hardware may simply be too expensive to obtain. Alternatively, offices frequently provide a vast range of personal notebooks and desktop computers that could instead be re-purposed to additionally run MapReduce jobs but lie idle. The recent emergence of so-called desktop grids [13] or private cloud infrastructures [30] are a clear sign of the pursuit to utilise these resources to capacity. Unfortunately, unexpected node disconnections are not unlikely in these heterogeneous environments. Thus, a centralised master-slave architecture may simply not suffice.

An idea to overcome limitations associated with centralised systems is to instead employ decentralised peer-to-peer (P2P) networks [12], which are commonly used as distributed shared resource providers (e.g. Box2Box [17]). Participants (*peers*) offer a part of their resources, like disk storage or computing power, to provide a service to the entire network of connected peers. P2P systems are inherently able to handle high rates of node joins and leaves without corrupting data nor halting running executions. Additionally, key-value based storage facilities in the form of distributed hash tables (DHTs [14]) often utilised in P2P systems offer a complete distributed file system deployable on heterogeneous hard-

and software. As DHT's key-value-based lookup and storage concept may directly be adapted to the also key-value-based MapReduce paradigm, their effective combination is only a reasonable step in the endeavour of achieving a more fault-tolerant and easy-to-deploy MapReduce system.

Although there have been several attempts to enable MapReduce on P2P systems (compare Chapter 2), such endeavours often try to either impose MapReduce as described by Google [16] directly on a P2P overlay network, e.g. by employing a dynamic master-slave architecture that can handle master failures [20] or by reusing interfaces of existing implementations (like Apache Hadoop) and abstracting the underlying file system to employ a P2P network instead [30]. However, the complexity of the designs may restrict extensibility and future improvements. Furthermore, users are often limited to the implementation of map and reduce functions only, a circumstance that may restrict the degree of freedom and choice as every problem needs to be mapped onto these two functions. An abstraction and simplification of the concept should allow users to focus more on the actual problem to solve instead.

1.2 Description of Work

The thesis covers the design and implementation of a prototype that enables users to write MapReduce jobs and distribute them on TomP2P. Evaluation covers the analysis of network traffic produced for the execution of one MapReduce job and a comparison of one to two job executions. Furthermore, based on challenges and issues encountered during both design and evaluation phases, a comprehensive overview is given for future work and extensions of the prototype.

1.3 Research Questions and Thesis Goals

The thesis explores the following research questions and goals:

- **RQ 1** *How can a completely decentralised MapReduce environment be implemented on top of a P2P overlay network with high user flexibility, little coordination and communication overheads, and a distributed hash table as main storage facility?*

As an initial evaluation of the prototype, it is focused on the produced traffic for an increasing number of connected nodes. Overall traffic per node should not noticeably increase if more nodes are added, as more network accesses most likely result in longer execution times. Thus, a first sub research question to explore is:

- **RQ 2.1** *Can traffic be kept constant for single-job executions if more nodes are added to the system?*

Beyond that, traffic should also not differ if two jobs are executed instead of one, where the file sizes of one job is twice the file size of the two-job execution. Thus, the second evaluated sub research question is:

- **RQ 2.2** *Can traffic be kept constant if two jobs are executed at the same time with the same overall file size to process as an equivalent single job?*

1.4 Thesis Outline

Chapter 2 presents related work of MapReduce implementations with a focus on adaptations of the paradigm to P2P systems. Chapter 3 introduces the theoretical background, methods, and libraries used to design and implement the prototype, which is covered in Chapter 4 (design) and 5 (implementation). Chapter 6 presents an analysis of traffic produced between nodes when running MapReduce jobs to investigate the two sub research questions, and Chapter 7 summarises the work and provides discussion points and conclusions based on design, implementation, and evaluation. Furthermore, limitations and future works to improve the prototype are outlined.

Chapter 2

Related Work

The next section examines MapReduce implementations found in literature with a special focus on P2P adaptations to emphasise problems and limitations of existing systems.

2.1 Literature

Besides Apache Hadoop [10] with its centralised master-slave architecture, recently, there have also been a number of attempts to port MapReduce to a more decentralised setting. The intention is to more appropriately support private cloud platforms [30], pervasive [29] and desktop grids [13], and/or mobile environments characterised by heterogeneous devices and a high churn rate. Such endeavours address the problem of Hadoop (and similar MapReduce implementations) being designed for dedicated hardware and not supporting dynamic environments with high churn rates [28], as failures may corrupt or even halt execution.

Apache Hadoop can be seen as a direct adaptation of the MapReduce programming model presented by Google [16]. Master nodes manage a number of slave nodes, which they assign tasks to execute. Input files reside in an own distributed file system (Hadoop Distributed File System HDFS) and are split into even chunks, which are replicated for fault-tolerance. Hadoop YARN provides a framework for scheduling and cluster resource management on which MapReduce is based. The task scheduler implicitly assumes cluster nodes to be homogeneous and tasks to progress linearly to decide if re-execution is needed (in homogeneous environments, nodes are assigned similar workloads). This makes Hadoop less suited for commodity hardware often found in offices, which may vary greatly in performance and reliability and where node crashes are common.

Lin et al. [18] explore limitations of Hadoop over volatile, non-dedicated resources. The authors propose the use of a hybrid architecture with multi-dimensional, dynamic replication. A small set of reliable, dedicated nodes are employed to provide resources to less reliable nodes (called MOON, Mapreduce On Opportunistic eNvironments), augmenting Hadoop's HDFS that only provides static data replication. They demonstrate that MOON's data and task replication design greatly improves the quality of service of

MapReduce when running on a hybrid resource architecture with many volatile and only a small set of dedicated nodes.

Marozzo et al. [20] improve the reliability of Hadoop's master nodes by introducing a decentralised P2P model (based on JXTA) that manages node churn, master failures and job recovery. MapReduce is provided through Hadoop. Each node may become a master or a slave at any given time dynamically, preserving a certain master-to-slave ratio. To limit job loss in cases of master failures, each master may act as a backup master for a certain job, only executed if the primary job master fails. Evaluations show better fault-tolerance levels compared to a centralised implementations and only limited increase in network overheads.

The PER-MARE initiative [28] proposes scalable techniques to support existing MapReduce applications in the context of loosely coupled networks. The goal is to develop a MapReduce system for desktop grids and to keep the implementation compatible with Hadoop's API. The network layer uses a token-passing P2P system with full data replication on every node. The authors already hypothesise, though not employ, the use of a DHT for storage to ensure fault tolerance without relying on full data replication. However, their first prototype CONFIIT is eventually discarded [26] for the later CloudFIT implementation (see below) due to several problems that led to an exponential increase in execution time with increasing data size, which makes it unsuitable for the intended purpose.

Tran et al. [32] implement MapReduce on a hybrid super-peer Gnutella P2P network. The intention is to exploit leisure resources with high heterogeneity for the execution of MapReduce jobs. The system employs a Master-Slave architecture similar to [20] and [10]: when a peer wants so solve a distributed problem, it sends messages to other peers, which can accept the request and then form a group of peers. The sender becomes the master and all other peers its slaves. By additionally providing super peers, the authors manage heterogeneity and increase scalability of the Gnutella network by limiting the number of incapable peers in query routing activities. Every peer group maintains a list of backup slaves to replace failed slaves. Master failures are not discussed in their work and they only reference [19] (conference paper of [20]). Thus, a certain master-node failure tolerance can be assumed.

CloudFIT [30, 29], CONFIIT's successor, uses desktop computers to set up a private cloud, where MapReduce jobs can be distributed to and executed with the help of several P2P overlay networks. It abstracts the underlying storage facility to support various P2P systems. The authors show the performance with both PAST and TomP2P against Hadoop. The results demonstrate CloudFIT to be able to achieve similar execution speeds as Hadoop while omitting the need of a dedicated cluster of computers. Tasks are executed in a random order on each node to avoid every machine executing the same task at the same time. Data is directly stored within the DHT of the corresponding overlay and replicated several times to assure a certain degree of fault tolerance. As MapReduce procedures produce output keys and values, only keys are sent to nodes via broadcast, whereas values remain in the DHT.

Name	**Architecture**	**Failure Prevention**	**Implementation Freedom**
Hadoop [10]	Centralised master-slave, HDFS	Multiple master and slave nodes, slave re-execution	Map & reduce primitives
MOON [18]	Hadoop with improved HDFS to handle heterogeneity	Dedicated nodes with little downtime	Map & reduce primitives
Marozzo et al. [20]	P2P for network, Hadoop for MapReduce, master-slave	Backup masters (& slaves)	Map & reduce primitives
CONFIIT [28]	P2P (token ring), Hadoop API	Full replication, dynamic node joins and leaves	Map & reduce primitives
Tran et al. [32]	P2P (Super peer Gnutella, no DHT, data from FTP server), master-slave	Backup slaves (& masters), super peers	Map & reduce primitives
CloudFIT [30]	P2P (Pastry or TomP2P), DHT for storage, Hadoop API	Replication factor, dynamic joins and leaves	Map & reduce primitives
This thesis	P2P (TomP2P), DHT for storage, own API	Replication factor, dynamic joins (and leaves), task resubmissions	generic Task (Map, Reduce, or any other functionality)

Table 2.1: Summary of important dimensions from reviewed literature.

2.2 Positioning the Prototype

Many of the systems encountered in literature, as summarised in Table 2.1, add improvements to Hadoop to provide better support for node heterogenity and master failures or node crashes in general. However, in almost all cases, the implementation is kept close to Hadoop and it is often tried to reuse as much of it as possible (which is definitely a valid approach due to Hadoops well-maintained API and active development community). Additionally, in cases of own implementations, the master-slave architecture is often followed, too. CloudFIT is the system that most closely resembles the presented design as it uses also a DHT as storage facility and broadcasts to send keys to connected nodes. However, the presented implementation is much closer to the underlying P2P system and does not abstract it as CloudFIT does, and customised DHT functionalities are implemented instead of reusing existing ones. The Hadoop API is not employed in any way, either. Furthermore, much more implementation freedom is given to the user, which is not restricted to specifying map and reduce functions only.

Chapter 3

Background

This section introduces important theoretical concepts and used libraries to provide a base for understanding the design and implementation of the prototype.

3.1 MapReduce

MapReduce [16] provides a way of automatic parallelisation and distribution of large-scale computations suited for datasets to big to fit in memory. Users define map and reduce functions. A map function takes an input key-value pair on which it conducts user-defined operations that eventually emit a set of *intermediate* key-value pairs. All values for each intermediate key I are grouped together and then passed to the reduce function. The reduce function uses I and merges the corresponding values according to user-specified code into a possibly smaller set of values. The resulting keys and values are finally written to an output file. Simplified, each computation expressed by these functions takes a set of input key-value pairs and produces a set of output key-value pairs. An exemplary pseudocode to count words expressed in these two functions is depicted in Listings 3.1 and 3.2.

Listing 3.1: map function

```
void map(String k1, String v1) {
    // k1: file name, v1: content
    for(word in v1) {
        emit(word, 1);
    }
}
```

Listing 3.2: reduce function

```
void reduce(String k2, List<int> v2){
    // k2: word, v2: list of 1s
    int sum = 0;
    for(one in v2){
        sum = sum + one;
    }
    emit(k2, sum);
}
```

Map splits the text (*v1*) of a document specified by its file name (*k1*) into all corresponding words and for each word (*word*), emits its occurrence count (*1* for every encounter of the word). The reduce function then sums up all occurrences (provided as a list or iterator of 1s, *v2*) of each word (*k2*) individually, eventually emitting the overall sum of occurrences (*sum*) for every word (*k2*) in the whole dataset. Users can specify associated types as

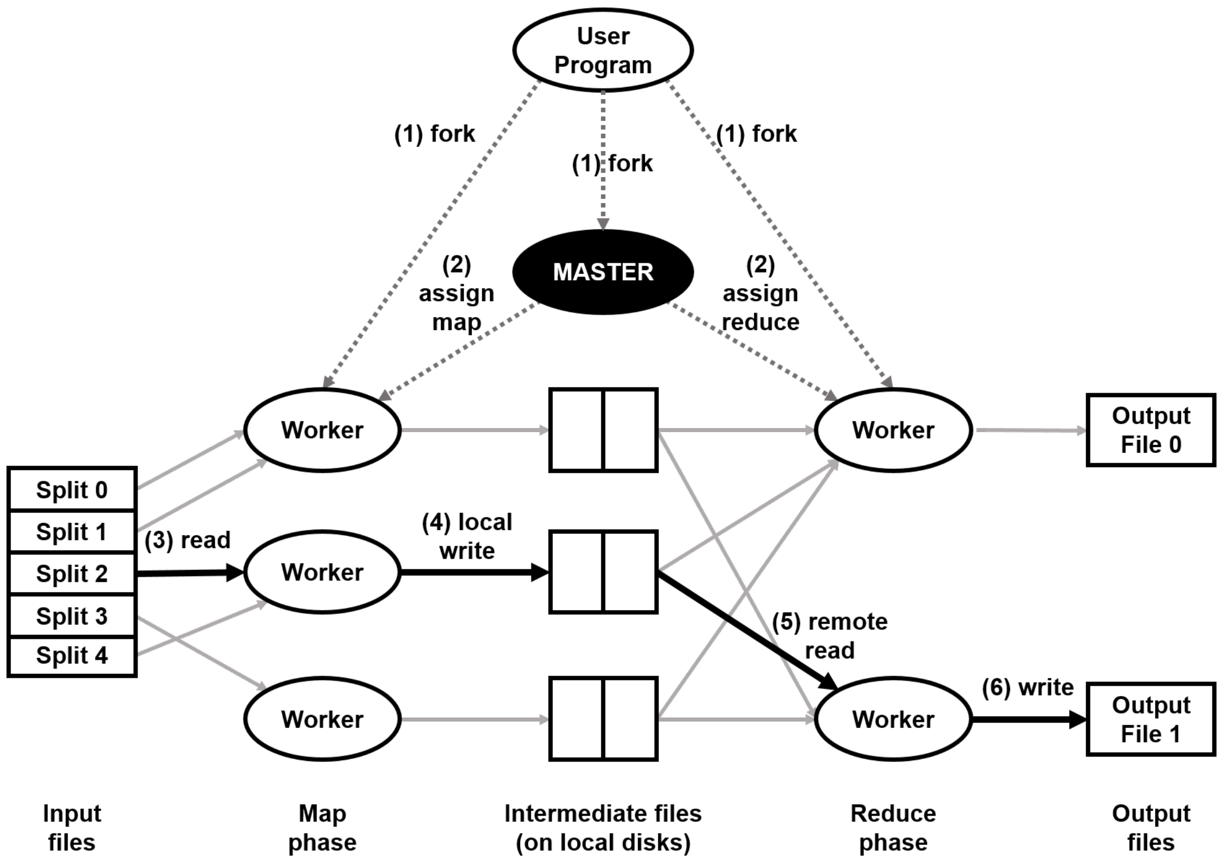


Figure 3.1: Conceptual execution of a MapReduce program. Adapted from [16].

needed (in the example strings and integers). Only *intermediate* keys and values need to be from the same domain as final *output* keys and values.

The conventional workflow of MapReduce [16] is indicated by Figure 3.1. (1) input files are partitioned into M splits with a size of around 16 - 64 MB per piece and many copies of the program are started on a cluster of machines. (2) a master schedules execution and assigns tasks to workers, which read the content of the input data and pass it as key-value pairs to the user-defined map function. Intermediate key-value pairs emitted by the map function are then buffered in memory, periodically written to local disks, and eventually partitioned into R regions using a partitioning function (4). R usually is the number of available workers for executing reduce. To evenly distribute these pieces, a partitioning function like *hash(key) mod R* can be used. Locations of buffered pairs on local disk are passed back to the master to forward them to the reduce workers. (5) reduce workers read buffered data from the local disks of map workers. When all intermediate data for a partition is read, a reduce worker sorts the intermediate keys to group all same keys together. Then, it iterates over the data and passes each intermediate key and the corresponding set of intermediate values to the user-defined reduce function (6). The output is then appended to the final output file for this reduce partition. Before emitting the data to the network after the map function finished, local aggregation is commonly applied (called *combiners* in Hadoop [10]). Combiners typically execute the same code as reduce but only locally on the node emitting data. For example, if a word is encountered three times, map without combiner emits <word, 1>, <word, 1>, <word, 1>. Using a combiner, the emitted data is already reduced to <word, 3> before it is sent over the network. Although it makes no difference to the succeeding reduce function which result of the two possibilities it eventually processes, it may drastically reduce the amount of data sent and consequently, speed up execution. However, not every application allows the use of combiners. Hadoop, furthermore, aggregates map, reduce, and combiner functions together with other configuration settings in a job. Jobs can be linked and executed after another, until a final processing result is achieved.

3.2 Peer-to-Peer Overlay Networks

The following section introduces selected topics and principles of the vast range of peer-to-peer (P2P) systems and applications important for the presented prototype.

3.2.1 Definitions and Flavours

P2P overlay networks are distributed systems typically without a central server handling user requests and responds with according resources (as present in client-server architectures [24]). Instead, many hosts (peers) possess desired resources (e.g. files) and handle user requests to them as well [14]. A P2P network is given if participants share a part of their own hardware resources (like e.g. processing power or storage capacity) to provide services and contents that are directly accessible to other peers [24]. Thus, peers are both resource (service and content) providers and requesters. Nowadays P2P systems are not easily classified and different authors choose to characterise them in various ways, e.g. according to their data indexing scheme, their level of decentralisation, or their level of structure [12]. In an attempt to summarise different concepts, one can distinguish *unstructured* P2P, which consist of *centralised* P2P systems (a central entity is needed to provide a service in the form of e.g. an index to files), *pure* P2P systems (without central entities, any terminal can be removed without losing functionality), and *hybrid* P2P systems (with dynamic central entities called "super" peers). *Structured* P2P systems, on the other hand, typically make use of a distributed hash table (DHT). As the presented design makes use of such a DHT, its purpose and functionality is outlined in the following in favour of other forms of P2P types.

3.2.2 Discovering Content with Distributed Hash Tables

One of the most important tasks of P2P systems is the efficient lookup and retrieval of content and/or services. Although systems with a central server may provide O(*1*) lookup of a file's location in a P2P network, they suffer from the problem of being a single point of failure and possible scalability issues [22]. On the other hand, although being rather resilient to node crashes, without efficient lookup capabilities, P2P systems may need to flood queries across the network to find the location of desired content (requiring $O(n)$

lookups) as there is no restriction on where files reside. This severely limits scalability, too. DHTs offer a trade-off between these two extremes. They enforce a certain structure on the P2P overlay network and provide hash-table-like semantics on internet scales [22]: keys are hashed using a hash function like SHA-1, and every key is mapped to one or more values. Typical DHT operations include those present in common hash tables like put(*key*, *value*), get(*key*): *value*, etc. Efficient routing protocols reduce lookup complexity to O(log n) while lowering node state (the routing entries to other nodes) to O(log n) as well. Although there are some limitations to DHTs (e.g. a high churn rate requires O(log n) repair operations, keyword searches in DHTs by default may only handle exact-match lookups, and many queries may not require an exact recall as they are mostly for well-replicated files and thus, do not justify the overhead of a DHT [14]), they provide a good balance between node state and communication overhead flooding and centralised P2P systems can hardly achieve. Popular DHT implementations include Chord [31], CAN [22], Pastry [23], or Kademlia [21]. As TomP2P employs an XOR-based iterative routing similar to Kademlia [9] and is used in the implementation of the prototype, its principal way of functioning will be briefly outlined next.

Kademlia

Kademlia is a P2P algorithm and DHT with XOR-based metric topology [21]. In a system of n nodes, searches using Kademlia only require O(log n) peers to be contacted. Keys are 160-bit opaque quantities. Every connected node is assigned one of these keys as a unique ID. Key-value pairs are stored on nodes with IDs "close" to that key. Closeness is defined as the bitwise Exclusive OR (XOR) distance of two nodes' IDs (d(x, y) = x $\oplus$ y, where x and y are two 160-bit identifiers). Thus, distance is not used in a geographical sense as "neighbour" nodes may be spread around the world and are only logically considered close in the overlay network due to their small XOR distance. Kademlia treats nodes as leafs in a binary tree, with each node's position determined by the shortest unique prefix of its ID. In a fully-populated binary tree of 160-bit IDs, the magnitude of the distance between two IDs is the height of the smallest subtree containing them both. If the binary tree is not fully populated, the closest leaf to an ID is the leaf whose ID shares the longest common prefix. For every node, the binary tree is divided into a series of successively lower subtrees that do not contain the node. Every node knows at least one node in each of its subtrees if the subtree contains a node. By successfully querying the known node, contacts are found in lower subtrees until the lookup finally converges to the target node. Thus, any node can locate any other node using its ID.

3.2.3 TomP2P

TomP2P [9] features XOR-based iterative routing similar to Kademlia and can therefore be classified as a structured P2P system. Keys and node ID's are 160-bit quantities, and key-value pairs are stored on ID's close to that key. This also means that there are 2^{160} different peers possible. Furthermore, values can be stored for the same location key by distinguishing it using one or more of three additional 160-bit keys: domain-, content-, and version key. These three keys are optional per default. Domain keys are also used in the implementation of the prototype, which will be outlined in Chapter 5. Moreover, TomP2P uses Futures to reduce blocking in systems. As each method call will return immediately, the corresponding Futures need to add a listener that specifies actions conducted once the procedure completes. The concept of Futures will also be used and extended in the implementation of the prototype, making it a necessary feature to be familiar with as a possible user of the system, see explanations found at [9].

Broadcasting TomP2P provides structured broadcast messaging based on [15]. Often, broadcasting (sending messages from one to all) is accomplished through flooding: all nodes send received messages to all other nodes they know. Consequently, many messages will be received multiple times and duplicates are simply discarded. Although simple in principle and rather resilient to node failures, such an approach produces a lot of unnecessary network traffic. However, in structured P2P systems like TomP2P, the inherent topology of the network can be used for a fast and efficient delivery of broadcast messages. As nodes can be reached in logarithmically many steps due to the Kademlia routing scheme, a broadcast message will reach all nodes in logarithmic time. The method is cost-efficient as there are no duplicate messages. Problems may arise when packets get lost: not only will single nodes but a complete subtree miss out on a broadcast message. Thus, to achieve an acceptable balance between too many duplicates and a possible loss of a whole subtree that does not receive a broadcast message, a relaxation of the kademlia-based routing principle can be employed: In every subtree, not only a single but multiple nodes are selected to be responsible for forwarding a message. Although the method introduces duplicate messages again (with associated additional network traffic), it also effectively decreases the probability of skipping a subtree.

Chapter 4

Prototype Design

4.1 Design Goals and Challenges

This section sheds light on the goals of the prototype and what overall challenges have to be solved to achieve these goals.

4.1.1 High Flexibility for End Users

An often encountered limitation of current MapReduce systems is that they restrict implementation possibilities for users to map and reduce functions (see Chapter 2). Limiting users to two functions, although relieving them from overcomplicated implementations, also limits their freedom of choice on how to solve problems. For example, in Apache Hadoop, if a problem solution requires the implementation of multiple different functions executed after another, users need to define multiple MapReduce jobs and chain them together. However, how these two functions in every job are invoked is entirely hidden by the system and users can only influence certain parts through own configuration parameters. Ultimately, users are completely dependent on the developers of these frameworks to provide appropriate configuration mechanisms. Conversely, in this prototype, the goal is to *transfer responsibility for the execution from the system back to the users.*

Besides actual processing, different environments may also require varying *quality assurance measures.* If a MapReduce job is run on a dedicated environment of high-end computers, one execution may well suffice. On the other hand, an implementation run on a set of office notebooks may suffer from many crashes and associated data corruptions, requiring more sophisticated control measures. However the circumstances, users need to be able to decide for themselves what the needed measures are as they know the environment, data, problem, and solutions better than any system could guess without becoming overly complicated.

Regarding opportunities to alter the given system environment, a skilled user should be able to *exploit peculiarities* of the underlying P2P network if desired (instead of hiding it as in e.g. [30]). This allows for implementations system developers may not have thought of before and thus, possible realisations of more innovative ways to solve data processing problems.

Lastly, the prototype should work on as many different operating systems and hardware sets as possible to allow for heterogeneity, unlike e.g. Apache Hadoop that requires best a Linux environment [2] for deployment and may require non-negligible effort to be adapted to other operating systems [3].

4.1.2 Storing Results

Where Apache Hadoop relies on an own file system called HDFS [10], the intended prototype in this thesis explores the feasibility of a DHT as storage facility for data emitted during MapReduce jobs. As both map and reduce functions take key-value pairs as input and produce key-value pairs as output, this behaviour needs to be abstracted and mapped onto the DHT directly that also relies on key-value-based storage operations.

An important data storage challenge is to distinguish different output results if multiple nodes execute on the same dataset. As DHTs in their basic form simply map a key to a value, users should be provided with facilities to easily distinguish multiple key-value pairs from different nodes. Consider the word-count example outlined in Chapter 3.1: if a node simply uses the hash of a word to store the associated counts, every time the same word is emitted, an already existing count for the same hash would be overwritten. Checking the DHT beforehand for already stored data is not feasible in a distributed environment due to slow network access and thus, has to be avoided.

4.1.3 Communication

Communication in MapReduce as outlined in Chapter 3.1 requires central entities (masters) to assign tasks to workers (slaves). However, the goal is to avoid such an architecture: every node should work autonomously and be able to execute and finish a whole job on its own if the complete dataset can be accessed. Generally, every map or reduce procedure to execute can be seen as an isolated entity. One of two functions is invoked on a set of input keys and values and produces a set of output keys and values. The output keys and values of the previous task then become the input keys and values for the succeeding task. Therefore, nodes only need to be informed about what task to execute and on which data. They do not need to know which other nodes execute the same task, nor if any other node failed during execution. Thus, coordination should be as small as to the extent where no node depends on the completion of other nodes. Consequently, the failure of one node should *not affect* the execution of the whole job and allow for other nodes to take up failed executions. On the other hand, every node executing a certain part of a MapReduce job should also be able to *profit* from already finished datasets and not execute on the same data all over again if it is not required by the user. Conclusively, the prototype's design needs to allow nodes to *benefit from the execution of other nodes without depending on them*, to only use a *limited number of messages* to do so to avoid an unnecessary and complex message handling, and *to avoid employing any form of master-slave architecture.*

4.1.4 Resilience and Fault Tolerance

One of the main reasons for implementing MapReduce on top of a P2P system, which was also one of the main goals pursuit in the analysed literature in Chapter 2, is its inherent ability to be more resilient to node crashes than centralised systems. This distinguishes it from implementations like Apache Hadoop, where master failures may cause the whole system to stop [19]. Instead, in a P2P overlay network and thus, the presented prototype, it needs to be possible to remove *any* node without the execution being halted and a joining node has to be able to immediately start participation without having to be assigned work by any central entity.

4.2 Design Overview

In the following, the proposed design as a result of aforementioned goals and challenges is outlined to provide a conceptual overview for understanding the prototype's way of functioning. As both the solutions to achieve fault tolerance and avoiding clashes due to equal key hashes are best explained with the help of the actual implementation, they are omitted in the upcoming design overview and instead detailed in Chapter 5.

4.2.1 Task as an Abstraction for Map & Reduce Functions

Addressing above outlined challenges requires a maintainable yet simple design. First of all, an abstraction for both map and reduce functions can already decrease complexity as there is only one procedure type the system needs to handle. Both functions take input- and produce output keys and values (see Listings 3.1 and 3.2 of Section 3.1) with the sole difference that map takes only *one* value and reduce an *iterator* of values as input. However, also an iterator of values can contain only one value, rendering the differences between map and reduce interfaces redundant. Thus, in this prototype, the two procedures are summarised to a single function termed *task*. A user defines an ordered list of tasks such that task_{i+1} is executed after task_i. It is entirely up to the user how many tasks are chained and executed after another until the final result is achieved.

Tasks, moreover, are not restricted to data processing only: responsibilities that are usually assumed by the system need to be directly implemented by the user. Typical operations are e.g. aggregating and/or sorting intermediate data emitted by the map function before passing it to the reduce function. However, if such a behaviour is not required, it can as well be omitted. An example for a user-defined aggregation task is given in Chapter 5.4 (ReduceTask). Similarly, local reading of data and final writing of results can be realised with a task specified according to user needs. For instance, instead of storing the data in the DHT or writing it to a local file system, the final results could be printed to the screen or stored in an online database, etc.

4.2.2 DHT for Storage

The property of MapReduce to take input- and produce output keys and values allows for an almost direct mapping onto DHT operations. A task *gets* the input values from the DHT using an input key, executes the intended calculations on this data, and finally *puts* output keys and values back into the DHT. The user is able to define input and output keys and values as required by the problem to solve. For example, simply emitting every word (key) and occurrence count (value) as depicted in Listings 3.1 and 3.2 of Section 3.1 may not be the only or most performant way to count words in documents. Once keys and values are emitted by a task, the underlying P2P system takes care of an appropriate distribution of the data on all nodes as well as its replication (as defined by the user). What needs to be communicated eventually is the key(s) produced by the task so that other connected nodes can retrieve associated data items and execute the next task on them.

4.2.3 Broadcasts for Communication

Communication between nodes is accomplished through the use of broadcasts as described in Section 3.2.3. Broadcasts are effectively received by every connected node. Therefore, they allow for a simple communication and coordination scheme, where every node can react to a received task execution request. However, the user is ultimately responsible for sending a broadcast. It can also be decided to omit the emission of a broadcast or to send multiple different or same broadcasts if the task requires it. Broadcasts are intended to only be submitted once a task finishes and should contain the produced output DHT keys such that nodes receiving the broadcast may retrieve the data and the actual task to be invoked on that data. How a task is transferred to a node is up to the user as it may either be stored and retrieved to and from the DHT or directly be transferred within the broadcast message. The latter especially makes sense in the case of small tasks, whereas for larger tasks, it may be required to only send keys to nodes to retrieve the corresponding tasks directly. Only invoking broadcasts on finished tasks also reduces the number of status messages and thus, further lowers complexity.

4.2.4 Prototype Workflow Overview

The basic principle of data storage and retrieval, execution of a task, and communication with broadcasts is illustrated in Figure 4.1. A task is sent to a participating node using broadcasts together with a key to find the data in the DHT (1). This allows for every node in the network to be informed about executable tasks. Once the task and key for the data are received, a node uses that key to retrieve the data from the DHT (2), starts task execution on the retrieved data item (3), puts the result data back into the DHT using a possibly new output key (4), and finishes execution, normally by sending a broadcast message containing all output keys to produced data items and information about the next task to execute on that data (5). In summary, the basic task execution relies on only three operations: DHT *put* and *get* to distribute, store and retrieve data items, and *broadcasts* to communicate between nodes.

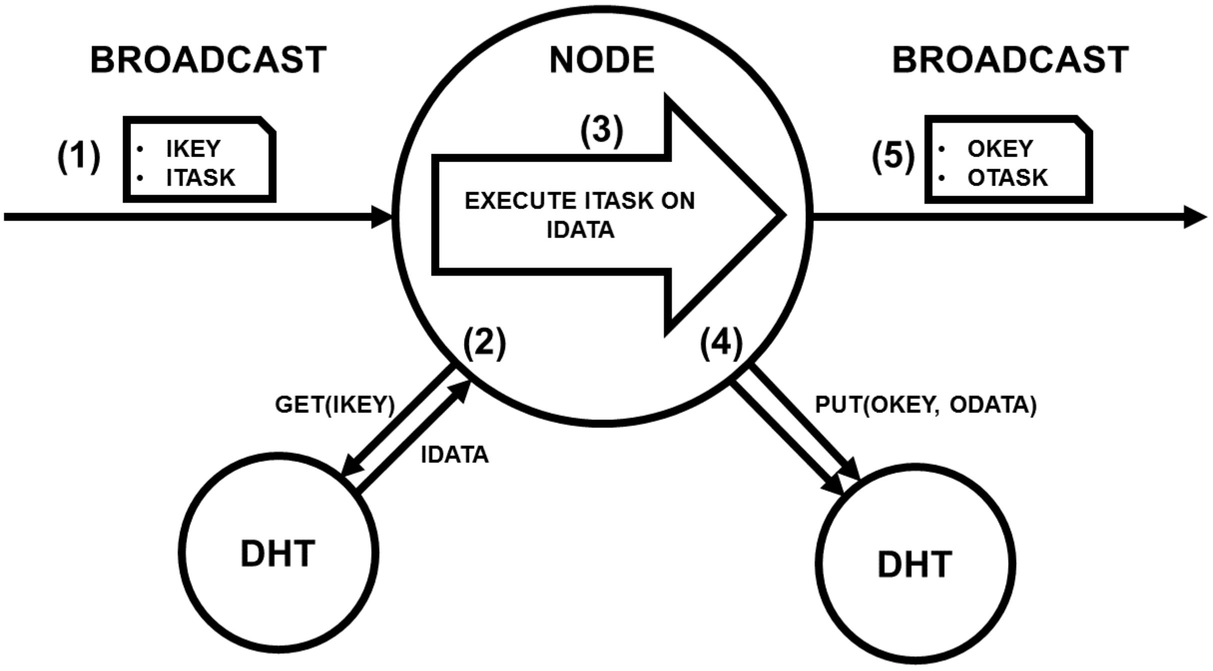


Figure 4.1: MapReduce job execution abstraction in the presented design.

4.2.5 A Note on Discarded Prototype Designs

After two failed attempts of implementing a working prototype, it was decided to redesign it from scratch. Initially, a similar design concept as presented in [30] was pursuit, who implemented their MapReduce engine based on the idea that the underlying P2P network should be replaceable and, consequently, the user should not have to care about the actual implementation details. However, during implementation of the prototype, it became more and more apparent that too many unknowns had to be guessed by the system to keep it as generic as possible. Furthermore, over-engineering harmed testing and modification of the prototype severely. Ultimately, problematic assumptions about how to aggregate data led to a large number of DHT access calls, drastically slowing down the whole system to an extent, where it was not feasible for the execution of MapReduce jobs anymore. Therefore, it was discarded eventually in favour of a much simpler implementation presented here. The new design still bares some commonalities of the initial designs, like the use of broadcasts for communication, to only communicate finished executions, and the abstraction of map and reduce functions to only a single task. Overall, the current and final implementation is a proof of concept and has thus much improvement and possible redesign to be done before becoming a viable alternative to other MapReduce implementations.

Chapter 5

Prototype Implementation

In the following, the actual realisation and implementation of the concepts presented in Chapter 4 are outlined. First, an introduction of the classes and interfaces providing users with tools to define own MapReduce jobs is given. To allow for a better understanding of these concepts, a possible (but not the only) implementation of a word count example is illustrated, which is also used in the evaluation of the system in Chapter 6. Then, the so-defined job is submitted conceptually to inform the reader about mechanisms and classes invoked internally and how certain concepts mentioned in Chapter 4 are realised. This also includes some notes on encountered interesting challenges and how these were resolved. As the prototype is under ongoing development, refactoring may alter the interfaces and implementations to some extent. However, the basic principles and implementation possibilities for users are intended to remain constant.

5.1 Frameworks and Programming Language

The entire prototype is implemented in Java JDK 8u20 [5]. TomP2P 5.0-Beta8 provides P2P functionalities [9]. Many internally invoked classes are customised extensions of TomP2P to allow for adapted DHT operations. Users are not only required to be familiar with the Java programming language but also with the concepts of TomP2P to implement own MapReduce jobs. An introduction to TomP2P can be found at [9]. The prototype is published on https://github.com/ollyblink.

5.2 User Extension Points

There are two extension points users need to implement: Firstly, the abstract class *Task*, and secondly, the interface *IMapReduceBroadcastReceiver*.

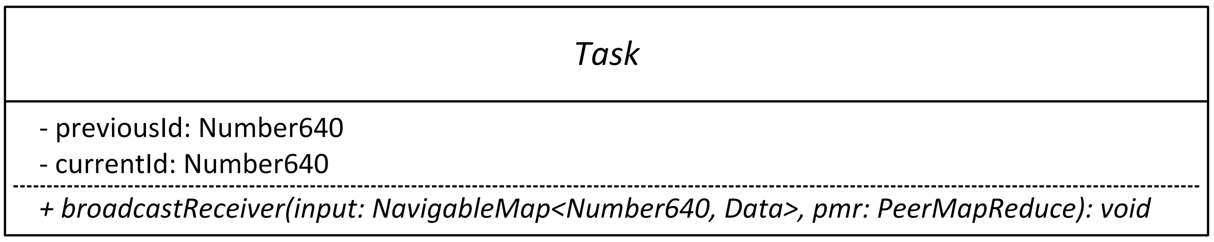

Task
- previousId: Number640
- currentId: Number640
+ broadcastReceiver(input: NavigableMap<Number640, Data>, pmr: PeerMapReduce): void

Figure 5.1: The Task class - the main extension point for implementing MapReduce jobs

5.2.1 Task

Task (Figure 5.1) is the embodiment of the idea of having only one abstraction for all procedures to be carried out during a MapReduce job as outlined in Section 4.2. It provides one method broadcastReceiver() for the user to be extended. The name of the method implies that a Task is intended to be invoked by the system on receiving a broadcast. A typical extension of Task is a map or reduce function but also any other action to be conducted before or after the actual MapReduce job, like e.g. reading, writing, or sorting data items. Two parameters need to be provided to the broadcastReceiver() method: a Java NavigableMap containing the user-specified data for this task, and an instance of PeerMapReduce providing users access to the P2P network. The user-specified data may be anything that a task needs for execution. In the case of a word count, an initial, locally executed task may require the file location as an input to read the data from the user's file system and distribute it to the DHT, whereas the input to a map task may be a key to the data stored in the DHT. Additional input can easily be defined here, like e.g. a dictionary that allows for only counting actual words in a file and no other tokens. When a task finishes, in many cases, a broadcast needs to be sent out, such that other connected peers get informed and can start execution of the next Task in the chain. To do so, users can directly access a TomP2P Peer instance wrapped inside PeerMapReduce (compare Figure 5.4). Figure 5.1 additionally shows that Task has two identifiers, previousId and currentId. These two instance variables need to be set by the user to specify which task comes after which. Especially in the case of the first executed task, the previousId needs to be *null* for the system to determine it to be the start of a job execution (see Figure 5.6 for an example). An ID may simply be a random TomP2P Number640 key and is only used to distinguish and link tasks. Its usage will be outlined in Section 5.4.

5.2.2 IMapReduceBroadcastReceiver

Another extension point for the user is *IMapReduceBroadcastReceiver* as presented in Figure 5.2, without which a MapReduce job cannot be started. Its intention is to allow a user to specify the actions taken when a broadcast is received on a node. This also includes starting the next task by invoking Task.broadcastReceiver(). The Message instance passed as a parameter is the complete broadcast message received, and the additional PeerMapReduce parameter is provided to be passed to the Task's broadcastReceiver() method. Allowing users the flexibility to decide how tasks are executed also

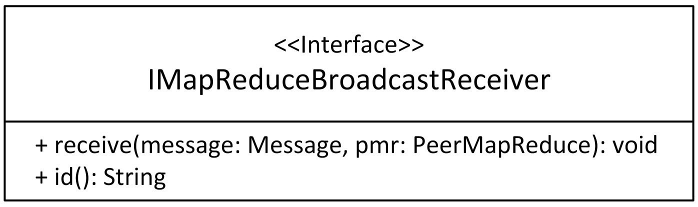

<<Interface>>
IMapReduceBroadcastReceiver
+ receive(message: Message, pmr: PeerMapReduce): void
+ id(): String

Figure 5.2: The IMapReduceBroadcastReceiver interface - the actions to be taken when a broadcast is received on a node.

enables them to intentionally *ignore* received messages. Therefore, although ideally, all broadcast messages sent are also received, it does not mean that every message needs to cause an action on the receiving node. Additionally, Figure 5.2 shows that an id() method needs to be implemented that defines a subclass-wide ID string to be able to distinguish one IMapReduceBroadcastReceiver from another. This has to do with the fact that the transmitted object and class files resulting from Task and IMapReduceBroadcastReceiver implementations are not the same when received multiple times, although they contain exactly the same information (this is related to the way in which the JVM loads classes that may also result in problems with automatically created equals methods. Details are beyond the scope of this work, although some additional insights are provided in Section 7.2.1). Thus, when a receiver object is added to a node's list of user-defined receivers, it may happen that the same receiver is added multiple times although the user's intention was to add it only once. By specifying a subclass-wide ID, e.g. the class's name, the current implementation assures that every node only contains one instance of the IMapReduceBroadcastReceiver extension.

5.3 Helper Classes

The prototype provides a number of helper classes and interfaces to facilitate users the implementation of MapReduce jobs. Some of these classes are required while others offer optional facilities to be used during implementation of Task or IMapReduceBroadcastReceiver extensions.

5.3.1 Job

All user-defined Tasks and IMapReduceBroadcastReceivers need to be aggregated in a Job as shown in Figure 5.3 to be able to start a MapReduce processing. The start() method takes the same input parameters as the Task's broadcastReceiver() method. On invocation, it retrieves the first Task (whose previousId is null, see Figure 5.6) and passes the input to it. Therefore, the first Task is *locally* executed on the node starting the Job. A typical local Task is reading data from the file system and storing it in the DHT. A broadcast call will then initiate the first non-local procedure, where other peers connected to the network start execution. As peers on other nodes do not know of the actual Task and IMapReduceBroadcastReceivers implementations, they need to be serialised

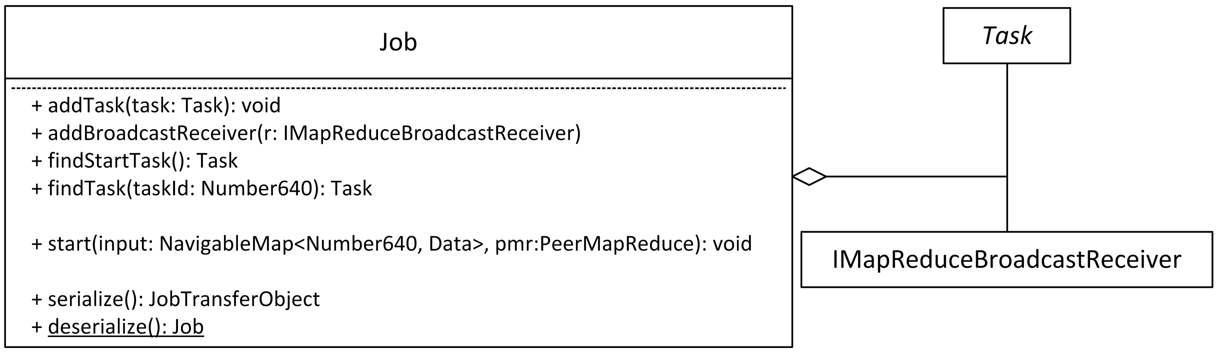

Job
+ addTask(task: Task): void
+ addBroadcastReceiver(r: IMapReduceBroadcastReceiver)
+ findStartTask(): Task
+ findTask(taskId: Number640): Task
+ start(input: NavigableMap<Number640, Data>, pmr:PeerMapReduce): void
+ serialize(): JobTransferObject
+ deserialize(): Job
Task
IMapReduceBroadcastReceiver

Figure 5.3: The Job class - aggregation and starting point of any MapReduce execution.

and transferred to and then deserialised and instantiated again on these nodes. The corresponding methods provided by Job are *serialize()* and *deserialize()*. Not only do these methods serialise all the instantiated Java objects and internally specified declared and anonymous classes but also all their corresponding class files. Job.serialize() returns a JobTransferObject containing the complete serialised job, which may be put into the DHT or sent via broadcast directly, however the user intends to execute a job.

5.3.2 PeerMapReduce

PeerMapReduce (Figure 5.4) is the main network interaction class that provides users with the possibilities to connect and disconnect to and from the overlay, put and get data to and from the DHT, and submit broadcasts to all nodes, including itself. It wraps an instance of TomP2P's Peer class that can be configured beforehand according to user requirements. A certain resemblance to TomP2P's PeerDHT, which also provides DHT functionalities to the user, is intended. The peer has to be directly accessed to emit broadcasts to other connected nodes. The put() and get() methods offer various parameters. First of all, two Number160 keys, *location-* and *domainKey*, allow users to distinguish storage keys to avoid clashes, which is demonstrated with examples in more detail in Section 5.4. Besides the actual *value* to store in the DHT, the put() method provides a *nrOfExecutions* parameter. It offers users the possibility to specify how many times a value can be retrieved for execution by internally restricting the number of accesses to that data item. As the system does not know about other nodes but instead relies on

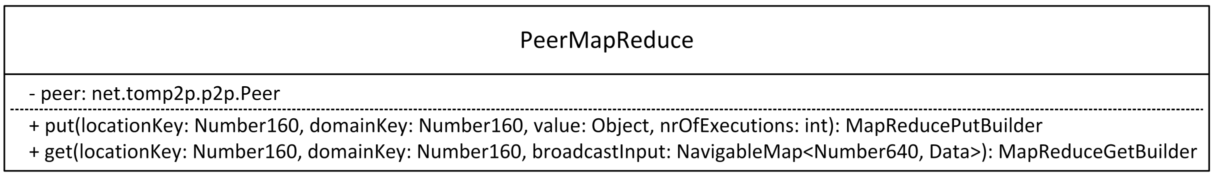
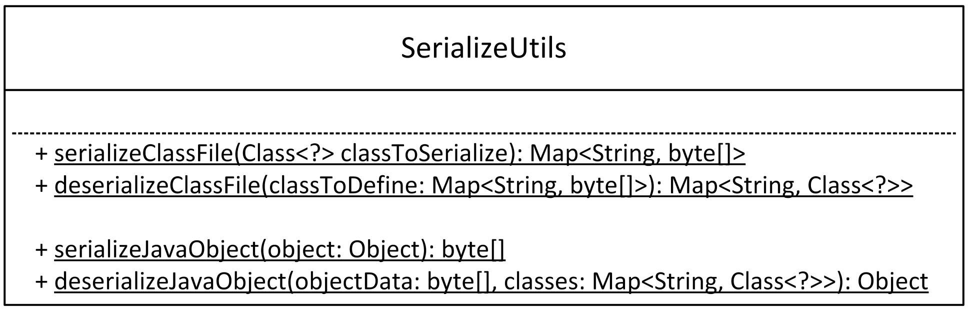

PeerMapReduce
- peer: net.tomp2p.p2p.Peer
+ put(locationKey: Number160, domainKey: Number160, value: Object, nrOfExecutions: int): MapReducePutBuilder
+ get(locationKey: Number160, domainKey: Number160, broadcastInput: NavigableMap<Number640, Data>): MapReduceGetBuilder

Figure 5.4: PeerMapReduce - a user's connection to the overlay network. It provides DHT-like put() and get() methods as well as direct access to an instance of TomP2P's Peer class to allow for the establishment of an overlay network and the submission of broadcast messages.



```
+ serializeClassFile(Class<?> classToSerialize): Map<String, byte[]>
+ deserializeClassFile(classToDefine: Map<String, byte[]>): Map<String, Class<?>>

+ serializeJavaObject(object: Object): byte[]
+ deserializeJavaObject(objectData: byte[], classes: Map<String, Class<?>>): Object
```

Figure 5.5: SerializeUtils provides methods to serialise and deserialise both class files and Java objects.

broadcasts that all peers receive and immediately process, also all data items may be accessed in parallel from all nodes. Consequently, this could result in a lot of unnecessary network traffic and waste of resources if all nodes execute all tasks at the same time, although only a restricted number of executions of every data item is required. Therefore, even though all nodes may *try* to access a data item at the same time, only some will actually receive it while others will not. However, the nodes that did not receive it may instead be granted access to other data items, effectively executing all tasks eventually. The get() method provides, besides location- and domainKeys to distinguish the actual data item to retrieve, an additional *broadcastInput* parameter. The parameter's type corresponds to Task.broadcastReceiver()'s *input* parameter. On every invocation of get(), a Task is intended to pass the input received via broadcast to the method. The parameter is used for an internal mechanism of fault tolerance explained in more detail in Section 5.5. Internally, if the node fails before emitting a broadcast (e.g. it goes offline due to crash), the system will notice and re-emit the broadcast with the same input again as before to guarantee the execution of all tasks eventually.

5.3.3 SerializeUtils

Instances of Job use the helper class SerializeUtils to serialise user-defined Task class(es) and objects as well as instantiations of IMapReduceBroadcastReceiver, such that they can be transferred to all nodes without them having to know about the actual implementations. On reception, SerializeUtils may then be used to deserialise the whole Job again and execute the current Task. Thus, a user needs to be familiar with the provided methods to effectively serialise and deserialise a Job. Furthermore, SerializeUtils may be used to transfer user-defined extensions of other classes and interfaces, provided these super classes or interfaces are available on all nodes that receive the extensions. To be able to instantiate transferred objects on a node, first, deserializeClassFile() needs to be invoked to make the class files available. Only then may deserializeJavaObject() be used, as it takes these deserialised class files as a second parameter ”classes” to correctly instantiate the serialised object specified in the objectData parameter as can be seen in Figure 5.5.

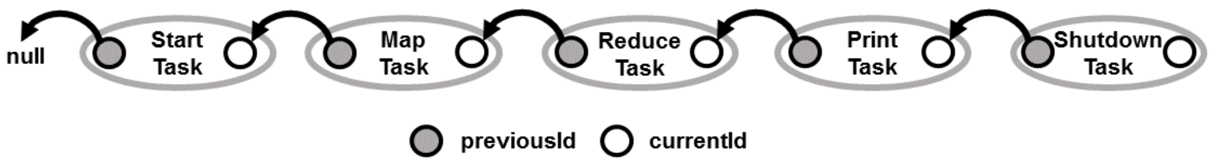


Figure 5.6: Task chaining. The order of task execution is enforced by specifying the previousId of a task to be the currentId of the preceeding task. StartTask has no previousId (thus, null), which is required to specify a Job's starting point.

5.3.4 NumberUtils

To simplify the generation of Number640 keys and removing the need to define every hash themselves, users are provided with a NumberUtils class. Two overloaded methods allSameKey() with either String or Integer input parameter will create a Number640 key consisting of location and domain keys as hashs of these input parameters, whereas content and version key are set to Number160.ZERO. This is especially useful when adding input values to the NavigableMap of the Task's broadcastReceiver() input parameter. Additionally, NumberUtils provides some publicly accessible constants that may directly be used within Task or IMapReduceBroadcastReceiver instances to simplify the consistent use of the same keys on the input NavigableMap, like CURRENT_TASK or NEXT_TASK. However, users are not required to use all of them but knowledge of their existence may facilitate the implementation of own Tasks or IMapReduceBroadcastReceivers.

5.4 Usage Demonstration: Counting Words

In this section, an actual implementation of a complete MapReduce job to count all words in a set of files is outlined to allow potential users a better understanding of how to utilise the prototype. It consists of five Task extensions, namely, StartTask, MapTask, ReduceTask, PrintTask, and ShutdownTask, executed in this order. The succession is enforced by specifying the previousId of each task to be the currentId of the task to be executed beforehand, see Figure 5.6. The complete implementation can be found in the package net.tomp2p.mapreduce.examplejob.

5.4.1 StartTask

StartTask reads all files locally and emits them individually to the DHT, each finished with a broadcast causing a MapTask to be started on that file on every receiving node. Location keys are hashes derived from file names, while the domain key is a hash of the peer's ID. This facilitates distinguishing results emitted from the same nodes. Both location and domain keys are sent as broadcasts for the succeeding MapTask to be able to retrieve the data items from the DHT, a principle followed in all the tasks and depicted in Figure 4.1 of Section 4.2.4. Additionally, every MapTask needs to be executed twice (a user's choice), thus the PeerMapReduce's put() method receives 2 as the nrOfExecutions parameter. The serialised job is sent with every broadcast as well but could have been stored in the DHT instead, however fits the processing best.

5.4.2 MapTask

MapTask uses the received location and domain keys to access the DHT using PeerMapReduce.get() and additionally, passes the complete input for other peers to resume execution in case a node fails. Furthermore, because all files can only be accessed twice as specified by the user, it may happen that a MapTask will not retrieve the data. In such cases, a MapTask simply returns and finishes without any execution or broadcast. On receiving the data for the input domain and location keys, however, it generates a Java Map<String, Integer> result object for the received file's content, where the key corresponds to a token/word and the value to the number of times this token occurred in that file. Thus, if two nodes are connected and two files emitted, four MapTasks will be executed, generating four Java Map instances containing the word count for each file. Before a token is counted, a user could specify to only do so for actual words or to use any measure of word alteration like e.g. stemming to reduce the size of produced data items, according to the user's needs. The maps are then stored in the DHT using put(), again with a nrOfExecutions parameter set to 2 to avoid executing the upcoming ReduceTask too many times, and a broadcast emitted. The broadcast message, again, contains the location and domain keys for the result Map for that file and node. The location key remains the same as before (a hash of the file name), whereas the domain key is a hash of the peer ID with an additional random number to avoid clashes from other executions of the same node with the same domain and location keys. An implementation like that minimises network access and transfers a much larger overall dataset compared to a word count where for each word, a 1 is emitted as shown in Listing 3.1. Moreover, it effectively links map and local combine functions as explained in Chapter 3.1.

5.4.3 ReduceTask

ReduceTask acts as an aggregation point for MapTask results. Before aggregating over all files, the specified number of MapTask results for every file needs to be received as broadcast in the form of location and domain keys. As long as this number is not reached, ReduceTask temporarily stores received MapTask result keys and finishes without aggregation or broadcast emission. This demonstrates the usage of a Task as an aggregation point *without emitting any broadcast* as briefly noted in Section 4.2.1. Only when all required keys for successfully finished MapTask executions are received, the ReduceTask will retrieve the corresponding Java Maps from the DHT, aggregate them in an own Map<String, Integer> with the key corresponding to a token/word and the value to the overall count of that word in all files, store the result in the DHT, and eventually emit a broadcast informing nodes about where to find the final word count results. Again, the location key is a hash of the file name and the domain key a hash of the peer ID combined with a random number to distinguish this aggregated result from other peers' executions. The reason why two executions of each MapTask are required is to demonstrate a user's ability to incorporate *quality assurance measures* (see Section 4.1.1) as needed: before finishing, ReduceTask could additionally retrieve *both* result sets from the DHT, aggregate them individually, calculate a hash on each result to determine if they are the same, and only emit the broadcast if this was indeed the case. In the case of three executions, a majority vote could determine which aggregation result to finally emit if e.g. two of three hash values were equal. As there is always a trade-off (e.g. longer execution times when executing every task three times instead of once against a possibly false execution when only carried out once), a user is able to choose such measures according to importance of correctness, expected node crashes, etc. In the current implementation, however, it is only chosen to retrieve one result set and aggregate the data before emitting the broadcast without any measure of quality assurance to keep overall execution time in the evaluation (Chapter 6) in a reasonable range.

5.4.4 PrintTask & ShutdownTask

PrintTask retrieves a ReduceTask's result from the DHT and writes it out to the local file system. The execution is again limited to 2. The last broadcast emitted causes the receiving nodes to shut down after a certain amount of time, specified in **ShutdownTask**. Two broadcasts from different PrintTasks need to be received before the shutdown is initiated, assuring that the results are actually saved twice as a minimal reliability measure in case one of the writing nodes' disks gets corrupted right after execution. ShutdownTask also emphasises the need for users to take care of a graceful disconnection of every node from the overlay network and thus, the possibility to directly access and influence the underlying P2P system as proposed in Section 4.1.1. The time offset allows for all nodes to receive the required messages before the shutdown is actually initiated to avoid unwanted execution disruptions.

5.4.5 ExampleJobBroadcastReceiver

A very basic implementation of the IMapReduceBroadcastReceiver interface (Figure 5.2) is chosen that only finds the next task to execute and calls the corresponding broadcastReceiver() method with the received input data. However, this implementation also needs to take care of Job deserialisation. Storing a Job for later usage within the IMapReduceBroadcastReceiver is a requirement in the presented implementation because the ReduceTask needs to aggregate received results. If the Job was not stored initially, every new deserialisation of it would result in a new Job object replacing the current one. This would erase all aggregated results stored within the ReduceTask due to related reasons briefly outlined in Section 5.2.2 for the required ID. Without such dependencies, the Job would not have to be stored necessarily.

5.4.6 Starting a MapReduce job

Assuming a user has defined these five Task extensions and an appropriate IMapReduceBroadcastReceiver as outlined before, a Job needs to be instantiated and the objects added to it. Furthermore, on each connected node, a PeerMapReduce instance has to be running, generating an overlay network using TomP2P's Peer instances. Connection between nodes is established in the same way as in other TomP2P applications by specifying the IP address and port of the Peer instance of a participating node (the bootstrapping node) to which all other nodes connect to. Additionally, the initial input for the StartTask has to be defined. The example word count takes as an input the location of the files to process and the number of files for the ReduceTask to know how many keys to expect. A job is then simply locally executed by calling Job.start() with the specified input for the StartTask as seen in Figure 5.3. Listing A.1 in Appendix A additionally provides a view closer to the actual implementation although some details are omitted.

5.5 Under the Hood

This section briefly describes the mechanisms invoked when a user calls different methods like get() or put(), and how basic reliability in the form of broadcast resubmissions and task scheduling are implemented. Furthermore, some interesting details on certain problems encountered during implementation and their corresponding solution are outlined.

5.5.1 Acquiring and Releasing Items

When a user calls put() or get() on PeerMapReduce, a MapReducePut- or GetBuilder, respectively, is returned, see the return types of the corresponding methods in Figure 5.4. These two classes resemble TomP2Ps Put- and GetBuilder but require additional parameters to allow for restricting data item access and resubmitting a broadcast in case of a mid-execution node failure, as mentioned in Section 5.3.2. When the methods are executed, a new instance of DistributedTask is created that eventually invokes methods of TaskRPC to either put or get data items. The current implementation only uses memory storage but could be extended to employ disk storage instead. Handling a PUT request is implemented straightforward by directly putting the data into the storage. In case of a GET request, it is first of all however necessary to assure that a data item can only be accessed the number of times specified by the user in PeerMapReduce's put() method (see Figure 5.4 and Section 5.3.2). This is enforced by incrementing a counter and only returning the value if it is smaller than the user-specified number of possible accesses. Secondly, and more importantly, every node containing the item in its storage facility is required to allow access again to this scarce data item if a node that was granted access to it crashes during execution of the corresponding task. In the case where the requesting node is also the responding node, no measure needs to be taken because if it crashes during task execution, the data item is lost anyways. Only when the requesting node is not the same as the responding node, a connection close listener is added for the connection of the requesting node that is called when the responding node notices its disconnection. Once the connection is closed, this listener will release the data item by decrementing the counter and emitting the broadcast again using the input provided when the user called the get() function. The close listener is removed when the broadcast of the finished task for this data item is received.

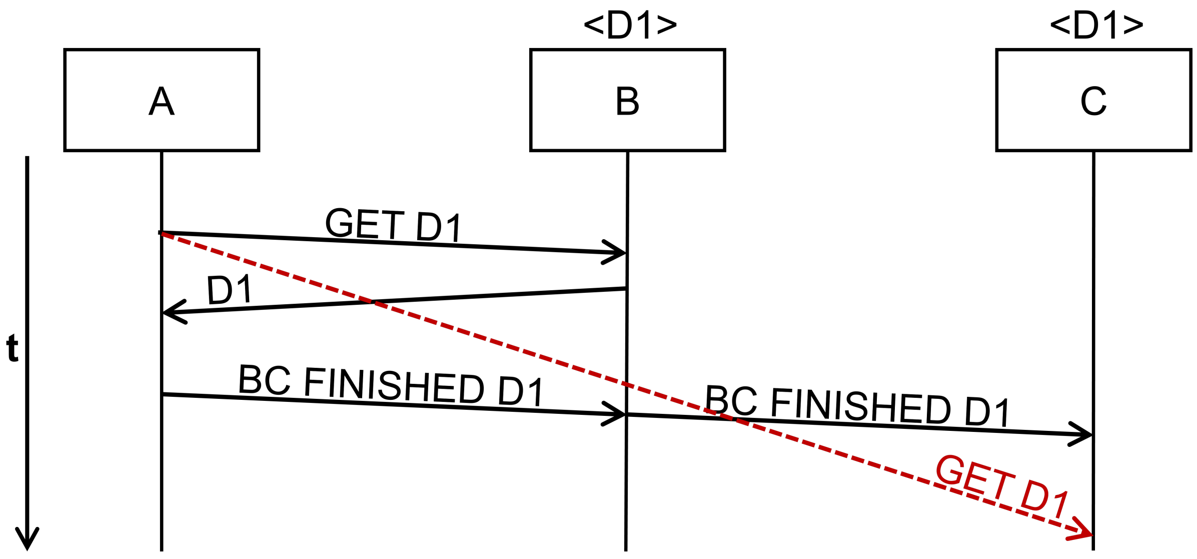


Figure 5.7: Although both B and C contain data item D1 requested by A, only one replication needs to be queried and retrieved to start execution on A. However, the execution is finished and a broadcast sent quicker than the second GET request arrives at C. Thus, C would ignore the broadcast and add a listener to the connection to A for data item D1 although the execution of D1 was already finished, see the descriptions in Section 5.5.2.

5.5.2 Network Latency Effects on Replicated Data

In the next part, some interesting issues and their resolution are outlined, which additionally realised a simple and effective random order scheduling.

Broadcast before GET

When a node puts data into the DHT, there will usually be multiple replications of the same data item on various nodes to avoid data loss in cases of node crashs. However, when PeerMapReduce.get() is called, not all nodes containing the same data item are visited at the same time necessarily to invoke the counters on them, depending on the chosen configuration. Instead, a subset of the replications of a data item is retrieved and, using a voting scheme, it is decided if the corresponding data item can actually be retrieved or not. Other nodes containing further replications of the same data item may only be updated afterwards although another node may already have finished execution on that data item. A simplified version of the problem is depicted in Figure 5.7, where two replications exist but only one needs to be checked and retrieved to start execution. This eventual consistency has the big advantage of speeding up retrieval. A disadvantage is that a node may already have finished execution on the data item and broadcasted the result again before all nodes holding a replication of that data item are actually informed about the previous GET request on it. In that case, a node that has *not* received the GET request yet but instead a broadcast that the task on the same data item is finished, will eventually add a connection close listener to the connection for a data item to the requesting node once this outdated GET arrives, although the task execution on that data item was already successfully finished. Thus, when a broadcast message is received but there is no connection close listener to remove on that item, a node will store this information locally. Every received GET request then, before sending the data item to the requesting node, checks if it has stored a received request for this item by the requester already and if so, refuses to respond this node with the data item. Only then, the local information about the too early received broadcast message is removed, enabling the same requesting node to try to access this data item again. Consequently, every data item can only be accessed once by every node at the same time, not multiple times, which is a limitation of the current prototype.

Majority Vote to Avoid Neutralisation

Configurations for replication and collection can be adjusted so that the problem outlined in the section before can entirely be avoided: e.g. if there are three replications, execution can only be started if all three items were queried. Although with such a setting, no broadcast for a finished item may be received before the GET to that item is processed, accesses to data items by different nodes may *neutralise* each other due to the counter on that data item. This behaviour could be observed in preliminary experiments that are noted in Appendix B. An extreme example is depicted in Figure 5.8. D1 is replicated three times on nodes A, B, and C, and the user defined in the put() method that D1 should only be granted access twice. As all nodes will immediately start execution of a task on broadcast reception, all data items will be queried in parallel by all nodes. However, access is only granted if all three data item replicas allow it, meaning their access counter is smaller than 2 before incrementing. Thus, before deciding if access can actually be granted, all data item replicas need to be queried to see if this is indeed possible. If only one data item was accessed already more than twice, it should not be possible to access it once more. As the local GET request from A, B, and C will most likely be granted immediately as there is no slow network access needed, the corresponding access counters of D1 will be incremented from 0 to 1. However, before these three nodes are able to check if they can indeed access the data item also on the respective other 2 replicas, D is granted access to D1 on all 3 replicas as the counter on each can be incremented from 1 to 2. The GET requests of the other three nodes arrive much later at the respective peers containing the replicas and are thus denied access as the access counters were already incremented to the maximum of 2 due to node D's request. This leads to the undesired effect that D1 is only processed once, although it should have been processed twice. To overcome this, the restriction that *all* replicas of a data item need to be accessible was discarded and instead, a relaxed majority vote is established: if a data item is granted access equalling or more than $\lfloor N/2 \rfloor$ times, where N is the number of replicas, access to it is still granted for that request. In the outlined example, thus, the data item is still received if at least one replica is accessible ($\lfloor 3/2 \rfloor = 1$). Unfortunately, the majority vote now leads to the problem that all four nodes are able to execute on the respective data item although only two should be able to do so. A solution to still enforce a user's execution restrictions is explained in the next subsection.

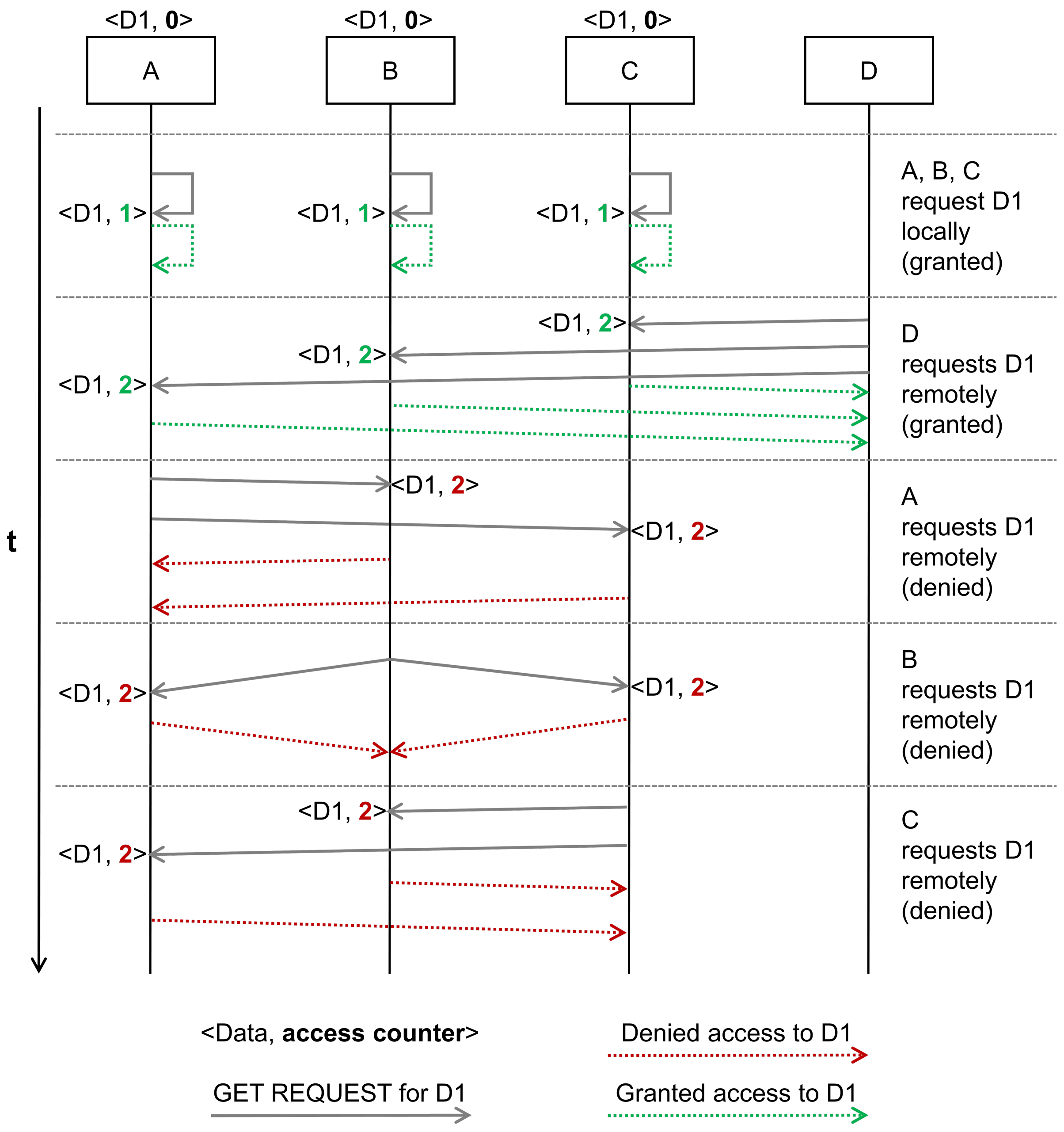


Figure 5.8: An example where GET requests neutralise each other, leading to the undesired effect that data item D1 is only executed once instead of twice as the user intended. See the corresponding descriptions for a detailed explanation in Section 5.5.2.

Stabilisation and Scheduling with Random Waiting Time

The issue of too many granted accesses is effectively resolved by adding a user-defined, variable *waiting time*. Whenever get() is invoked, it waits a random time between 0 and a user-specified number of milliseconds before actually executing the GET request. Thus, all nodes requesting the same data item will wait a different time until they start the retrieval process. If the difference between two access times is large enough for all nodes with the same data item to be visited, the restriction of accesses can effectively be enforced to the user-specified nrOfExecutions in the put() method. Preliminary evaluations showed a much more stable execution and only few additionally granted item accesses when using waiting times of around 3 - 7 seconds. Another advantage is the more likely *random order* in which data items will be accessed and processed by different nodes. The resulting randomised scheduling is similar to the one described in [30]. This simple measure effectively lowers false accesses and thus, leads to better node utilization as nodes do not execute too many unnecessary tasks and it is less likely for one node to retrieve most data items while others retrieve none. In most cases, this waiting time leads to the desired effect that all replicas are queried in the same order and thus, counters are incremented on all data items for the same node requesting it. In the example of three replications, it indeed could be seen that most requesting nodes either got three granted and zero denied replies or the other way round. However, variations could still be observed in certain cases, like granted twice and denied once or vice versa, such that the majority vote of the previous section decided to grant more accesses than required. Nevertheless, the additionally executed tasks were only few, which is acceptable because at least the previously often occurring neutralisation of accesses as exemplified in Figure 5.8 could effectively be suppressed as conducted tests showed.

Chapter 6

Evaluation

The following experiments are carried out as a preliminary evaluation of the system's behaviour. The prototype is not compared to other implementations like Hadoop yet. As initial experiments shown in Appendix B and also, other studies (e.g. [29]), already confirmed network traffic to be one of the main reasons for slowdowns, the conducted evaluation covers only an analysis of (measurable) traffic and not execution speeds. The pursuit goal is to keep average traffic measured per node constant with an increasing number of nodes as stated in the introduction and if not, to explain why there is a non-constant traffic.

6.1 Software and Testing Environment

Up to six commodity notebooks are used for the evaluation as shown in Figure 6.1. The software for all nodes is started directly from Eclipse Mars (Release 4.5.1) [6]. All notebooks share their resources with other applications but are assigned up to 1 GB of memory for the execution of jobs. The experiment only utilizes in-memory instead of external storage. Nodes are connected using a Netgear Ethernet Switch (ProSafe Plus 8-port Gigabit). Certain statistical analyses and visualisations utilise IBM SPSS 23 [4] and Microsoft Excel [8].

6.1.1 Configuration Issues

Before running the experiment, initial trials revealed problems with timeouts of connections in cases of too many and too large files. Conducted analyses revealed that when too many connections are opened at the same time and multiple files are transferred that have an overall size larger than a certain amount, some of the connections may not transfer any data at all and thus, TomP2P's default waiting time of around 5 seconds is not enough to reset the corresponding timeout. With increasing file sizes, also these timeouts had to be increased steadily. Thus, for the final experiment, all values regarding timeouts are set to Integer.MAX_VALUE. However, although it might be a feasible workaround for these

#	Name	OS	RAM	Disk (Type)	CPU
1	ASUS X53S	Win 10 (64 Bit)	4 GB	111 GB (SSD)	4 * 2 GHz
2	Lenovo T61	U 14.04 LTS (64 Bit)	4 GB	153.2 GB (HDD)	2 * 2.2 GHz
3	Lenovo T410	U 14.04 LTS (64 Bit)	8 GB	483.8 GB (HDD)	4 * 2.67 GHz
4	Lenovo W500	U 14.04 LTS (32 Bit)	4 GB	310.8 GB (HDD)	2 * 2.53 GHz
5	Lenovo T61	U 15.10 LTS (64 Bit)	2 GB	123.8 GB (SSD)	2 * 2.2 GHz
6	Lenovo T61	U 14.04 LTS (64 Bit)	2 GB	243.9 GB (HDD)	2 * 2.5 GHz

Table 6.1: Hardware used during experimentations. # indicates which computers are participating in the executions: machines 1 - 3 are used for 3 node executions, 1 - 4 for 4, 1 - 5 for 5, and 1 - 6 for 6. Win denotes Microsoft Windows, U Ubuntu.

first experiments, other solutions need to be found in further development stages of the prototype.

6.2 Traffic Analysis for 1 Job Execution

The first experiment focuses on average traffic measured per node when only one job is submitted from machine #1 (Table 6.1) in the cases of 3, 4, 5, and 6 connected peers. The intention is that average traffic per node does not increase with additional nodes so that there is an advantage in adding more nodes to the execution. Thus, the hypothesis this experiment evaluates is: *the average traffic per node remains constant for the execution of one job with an increasing number of nodes.* MiB and similar notations are the base-2 equivalent of MB etc. with base 10, see the abbreviations list at the back of this document.

6.2.1 Experimental Setup

The used job is the same as explained in Section 5.4. A dataset of 12 MiB of texts with an overall of 282487 different tokens compiled from the Gutenberg Project [1] is split into 24 parts of 0.5 MiB. Thus, StartTask conducts 24 DHT-PUT operations for every file split that will be executed twice each (in the best case), resulting in 48 MapTasks. These are then combined twice in the ReduceTask, and finally written out to the file system twice in the PrintTask. Results with more than 48 finished MapTasks or other irregularities are discarded from evaluation to achieve a coherent image of the produced traffic in an overall ideal case (no node crashs). The small dataset size was chosen to avoid overloading the JVM's assigned memory as only in-memory storage is currently employed in the prototype and to lower the time to conduct the evaluation. Preliminary experiments with up to 100 MiB however were also successful. Every dataset put into the DHT is replicated 3 times, meaning that 3 nodes will contain the same data, and 3 parallel requests are conducted. Eventual consistency (see Section 5.5.2) is thus disabled. After removing faulty experimental job runs (around 1 per 10 executions with 4 or more nodes, none in the case of 3 nodes), the remaining measurements consist of 10 runs for 3 nodes, 12 runs for 4 nodes, 11 runs for 5 nodes, and 20 runs for 6 nodes. Traffic per node is analysed by directly parsing the debug log output of TomP2Ps Decoder class. This is the most accurate representation of traffic per node as every message can be associated with its request type and therefore, all PUT, GET, BROADCAST, and other messages and their sizes can effectively be distinguished from each other. Traffic can only be captured when requests are sent from one to another node. Thus, requests that are directly handled by the issuing node itself cannot be measured. Finally, data for each number of participating nodes (3 to 6) is averaged for every message type.

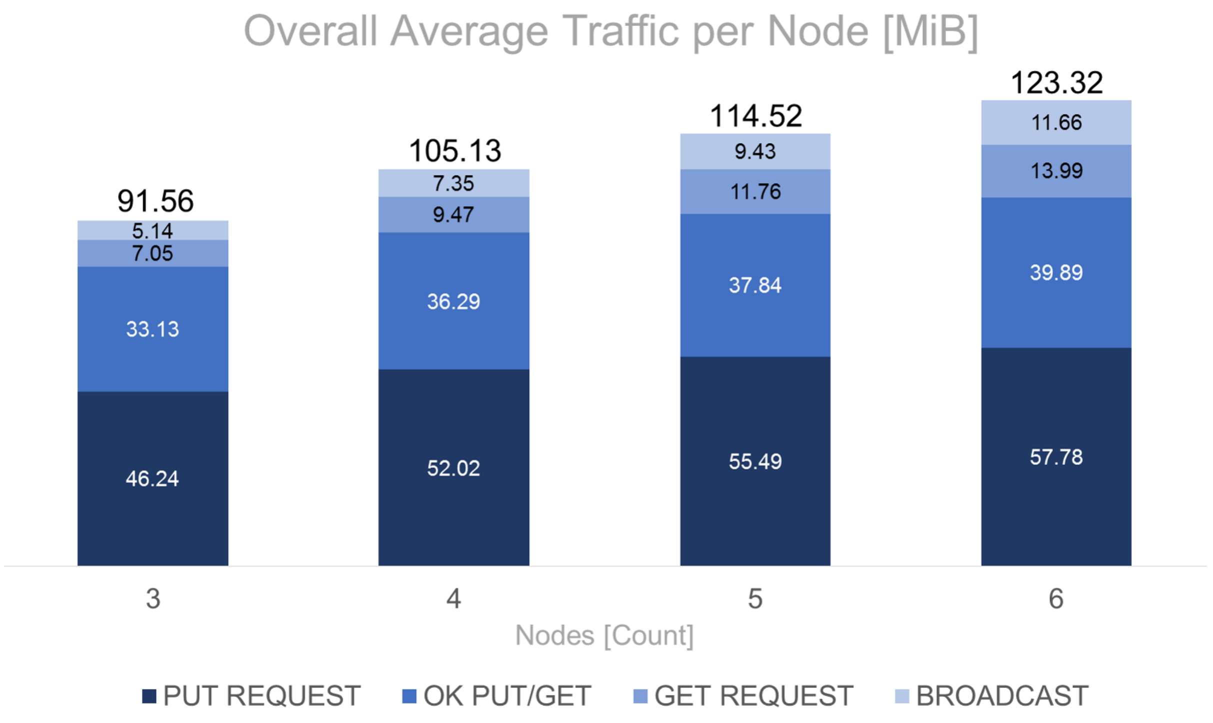


Figure 6.1: Overall average traffic per node for 3 - 6 connected machines.

6.2.2 Results

Figure 6.1 shows an overall summary of the log-parsed traffic per node for 3 to 6 connected machines, based on data summarised in Appendix C. The composition includes GET and PUT requests and responses and the overall sum of broadcast message sizes in MiB. Other traffic components that only contributed magnitudes below 0.01 MiB are not included (e.g. ping- or neighbour-related messages) as they do not alter the overall traffic in a noticeable manner. It can be seen that captured traffic per node is much larger than 12 MiB. Even in the case of 3 nodes, the summed-up overall traffic is 91.58 MiB. This is a consequence of the job's design and replication of data. Furthermore, there is an increase from 91.58 (3 nodes) to 123.4 MiB (6 nodes) visible. Effectively, all message types increase with sizes (together with the number of messages, see Table C.1). The small standard deviation very close to 0 listed in the same table also indicates significantly different overall traffic sizes with increasing numbers of nodes. In the following, important traffic parts are analysed to explain these increases.

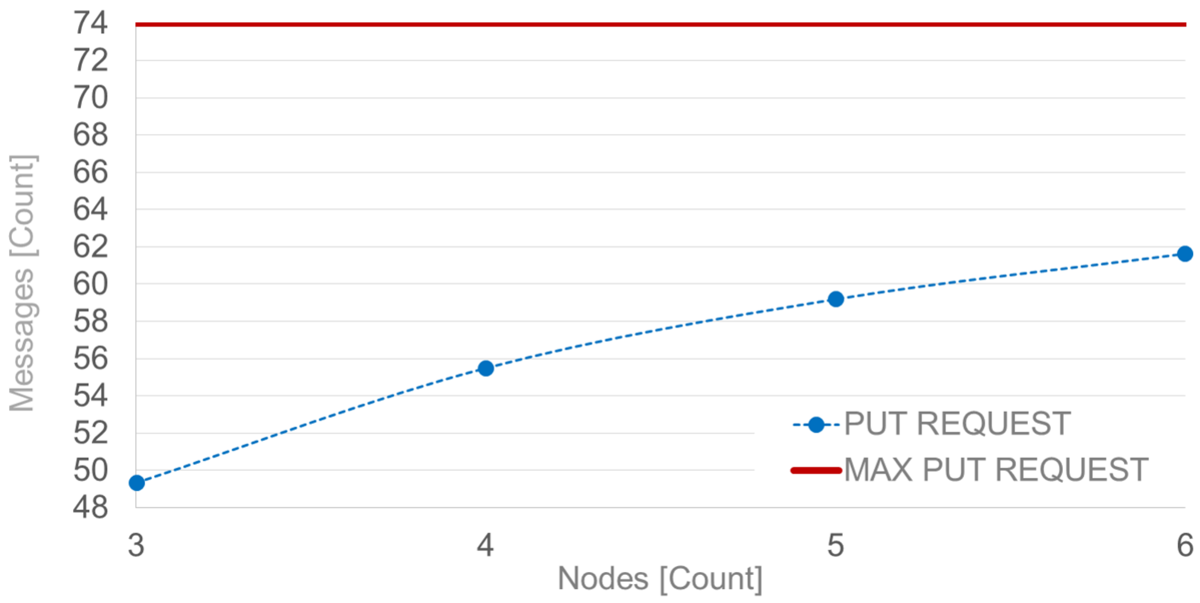


Figure 6.2: PUT requests increase with every node added to the execution. However, the maximal number of PUT requests (74) is never reached, and the curve flattens out, presumably approximating this maximum value progressively with every additional node.

Seemingly Increasing PUT Request Traffic

Figure 6.1 shows traffic sizes for PUT requests to increase from 46.24 to 57.78 MiB when expanding the number of executing nodes from 3 to 6. However, the increase needs to be analysed in terms of number of PUT request *messages* sent and not in sizes of MiBs to explain the increasing traffic, which is depicted in Figure 6.2. The number of PUT request messages correspondingly increases with every added node. However, the increased traffic is indeed a consequence of the DHTs attempt to distribute the files among all connected nodes more evenly and the fact that traffic is not measured in case the data is stored on the requesting node itself. To proof this, the maximal number of put requests needs to be calculated for that job: 24 in the StartTask, 48 in the MapTask, and 2 in the ReduceTask. Thus, maximally 74 PUTs are possible on one node if every PUT request is directed onto that node. This maximal average number of PUT requests per node is also drawn in Figure 6.2 as a red line. Analysing the *non-averaged* number of PUT requests on every node individually reveals that *no* node handles more than 74 PUT REQUEST messages in any case, regardless of the number of nodes connected. Therefore, it can also be seen in Figure 6.2 that the *averaged* number of PUT request messages (and thus, the overall size as shown in Figure 6.1) does *not linearly* grow with every additional node but instead, *approximates* the maximal number of 74 messages. Conclusively, the increasing sizes can fully be attributed to the way in which traffic is measured and not an actual increase of traffic, supporting the overall pursued goal of achieving constant average per-node traffic with increasing number of nodes.

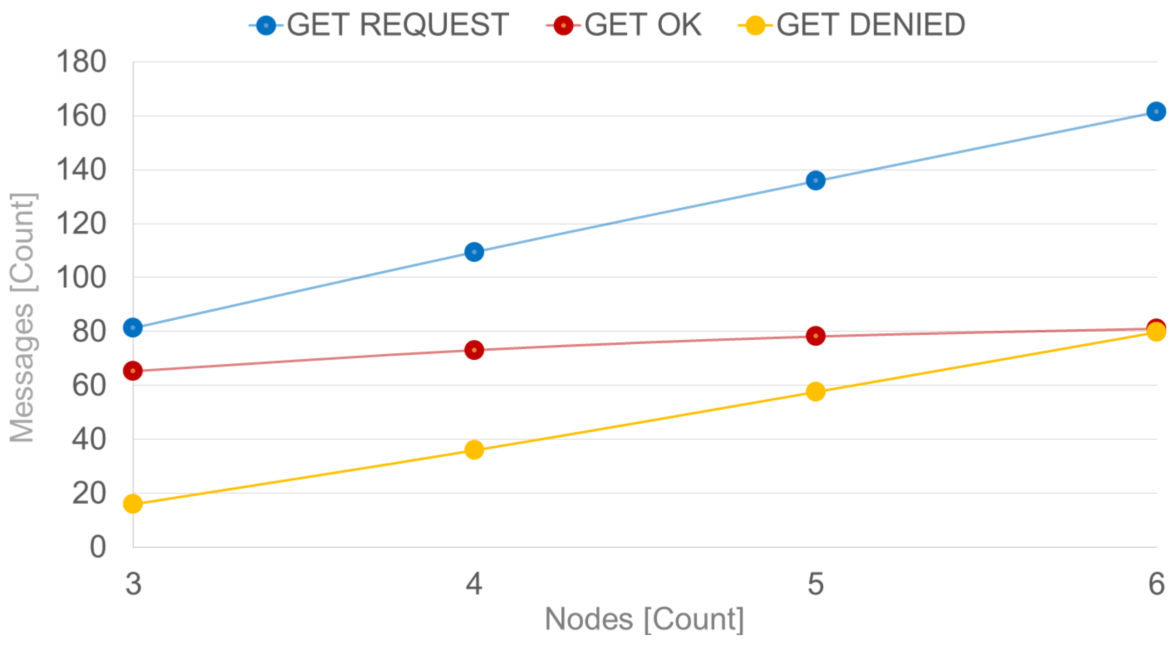


Figure 6.3: GET request messages and number of granted (OK) and denied GET requests for 3 to 6 nodes.

Effective Limitation of Granted GET Requests

Evaluation of averaged number of GET request messages per node reveals an effective limitation of the number of times a computation node can acquire the data for a per-broadcast received task. In Figure 6.3, with every added node, the number of GET REQUEST messages increases as they all listen to incoming broadcasts and will try to acquire the data to execute the corresponding task. However, also the number of DENIED messages (meaning, too many nodes tried to access the item already) increases with the same magnitude. Even more importantly, the increase in GET OK messages is *not linear* like GET REQUEST and DENIED messages but approximates a certain upper bound of around 80 messages as can be observed in Figure 6.3. Although there are cases where one node may still send a granted message (as the increasing number of GET OK messages with more nodes implies, which consequently results in an increased likelihood of more nodes accessing the same data item), tasks are only rarely executed more than they were intended by the user. However, it is not a perfect solution as there indeed still are cases where more than the minimal number of retrievals are undertaken but it is effectively limited by the additional waiting time (Section 5.5.2). Nevertheless, the trade-off is acceptable as it is more important that a job does not halt due to neutralisation than that the additionally produced traffic harms execution performance. The result still supports the goal of constant average per-node traffic with increasing number of participating nodes, taking into account possible deviations due to imperfect access restriction mechanisms.

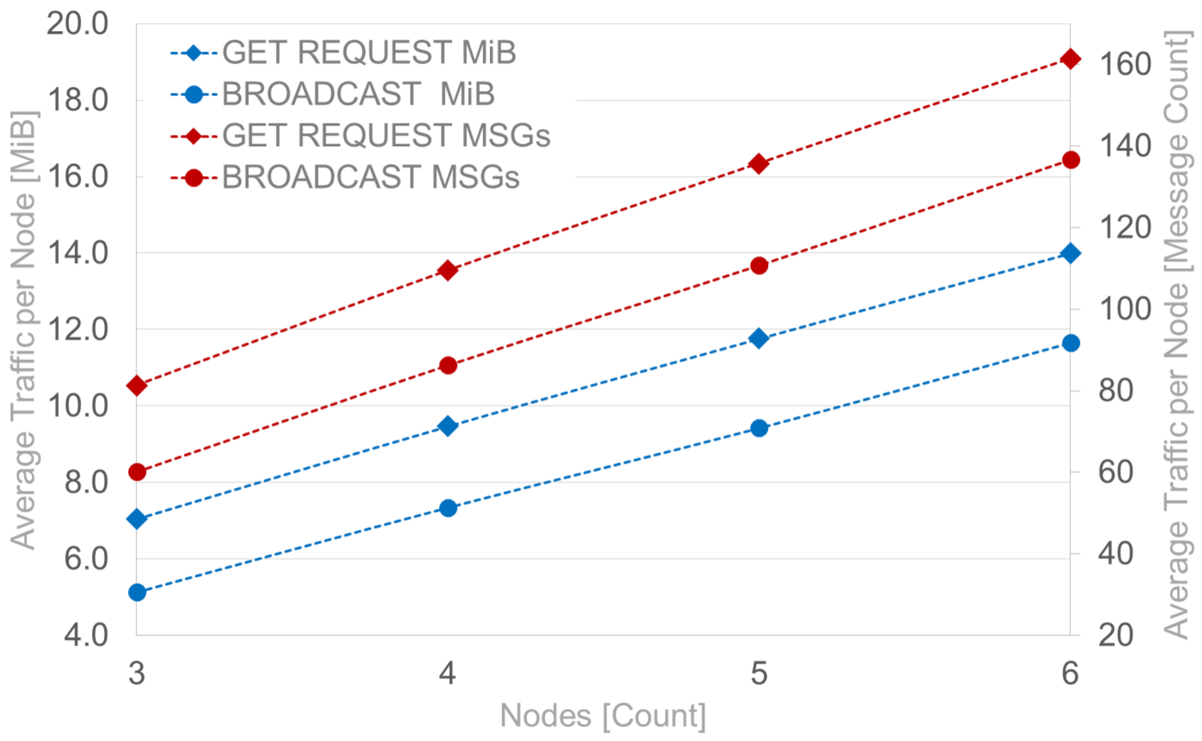


Figure 6.4: GET requests and BROADCAST messages for 3 to 6 nodes.

Broadcast Input Data and GET Requests

The presented implementation of the word count example sends on every successful PUT operation (and in some other cases) a broadcast containing various items. On reception, these broadcasts cause on every node the emission of a GET request for the MapTask. Thus, as Figure 6.4 clearly shows, both messages and corresponding message sizes increase almost linearly and equally for both types of messages with increasing node count. Although the larger number of messages is of no surprise (as every additional node will try to retrieve the data, more requests overall will be received on nodes containing the data, and broadcast messages are submitted more times to avoid the loss of messages, see Section 3.2.3) problematic is the increase in size: with every additional node, over 2 MiB of extra traffic is produced. The problem can be found in the broadcast input used to distribute information about next tasks to execute: such a message is, on average, around 0.085 MiB. The main part of data transferred can be attributed to the input parameter (see Figure 5.1), of which the serialized job is the largest part. Analysis of the job's serialised form (Java Job, Task, and IMapReduceBroadcastReceiver objects and all resultant class files) revealed a size of around 71.3 KiB. Although on its own, a job is not that large, the *cumulated* traffic increases noticeably: when only 3 nodes are connected, overall traffic per node produced by broadcasts consists of an average of around 60 messages, equalling 5.14 MiB. However, as visible in Figure 6.4, this number increases steadily with every added node, eventually equalling an average of around 136 messages with an overall size of 11.66 MiB. Consequently, the same increase in average per node traffic can also be seen for GET requests: on every call to PeerMapReduce.get() (see Figure 5.4), the same

#	**Split size [MiB]**	**File size [MiB]**	
		1 job	2 jobs
A	0.5	2	1
B	1	4	2
C	0.5	4	2
D	1	8	4

Table 6.2: Used datasets for evaluating the traffic of one and two jobs. Each configuration is run for 3, 4, 5, and 6 nodes, resulting in 16 data points.

input data received as broadcast is passed and transferred as a means of enabling the same broadcast to be emitted again should a node fail during execution (as explained in Section 5.3.2) and thus, making the corresponding data available to the execution of other nodes again. As *all* additional nodes try to acquire the data in the MapTask (although only a limited number of nodes will actually receive it), also the number and sizes of GET request messages increase. Consequently, the user needs to be aware of the fact that a larger input may also increase broadcast and GET request sizes. Post-evaluation experiments indeed show that GET request messages may become irrelevant (< 0.01 MiB) for the overall average per-node traffic when the serialised job is removed from the input. However, although messages increase with every added node in case of broadcasts and GET requests, it does not add a significant amount of traffic to the overall execution, still supporting the goal of having overall constant averaged traffic per-node with increasing number of nodes.

6.3 Comparing Traffic of 1 to 2 Jobs

The second experiment focuses on traffic generated on every node when *two* jobs are launched from two different machines in cases of 3, 4, 5, and 6 peer. This is compared to the same amount of traffic produced when a single job is launched with the same overall amount of data to process. The goal is to show that produced traffic for the same overall dataset sizes stays the same, regardless of the number of jobs needed during execution and if not, an explanation shall be found in the parsed data. Thus, the hypothesis this experiment investigates is: *the overall mean traffic per node is the same for one job execution that has the same amount of data to process as two jobs.* For this experiment, insights presented in Section 6.2 is reused and referenced. Additionally, to avoid potential problems with GET request messages, as they were shown before to increase overall traffic as a side effect of passing broadcast input with every GET operation, the feature is disabled.

6.3.1 Experimental Setup

The same job outlined in Section 5.4 is used for every submission. Datasets chosen for one job execution are always double the size of those chosen for the execution of two jobs, see Table 6.2. Split sizes, however, are the same in both cases. Replication and parallel requests remain 3. Faulty experiment runs (with too many executed MapTasks) are removed and not used in the evaluation to keep traffic comparable. As experiment 1 showed that messages and sizes only differ marginally with every run (small standard deviation, see Appendix C), it was decided to obtain only 2 correct (in terms of number of MapTasks executed and produced output) runs for each job, file size, and split size. These two runs for each configuration are then averaged for the final evaluation. Again, traffic data is directly parsed from TomP2P's Decoder output with the associated limitation as explained in Section 6.2.1 that requests sent and directly handled by the same node cannot be measured.

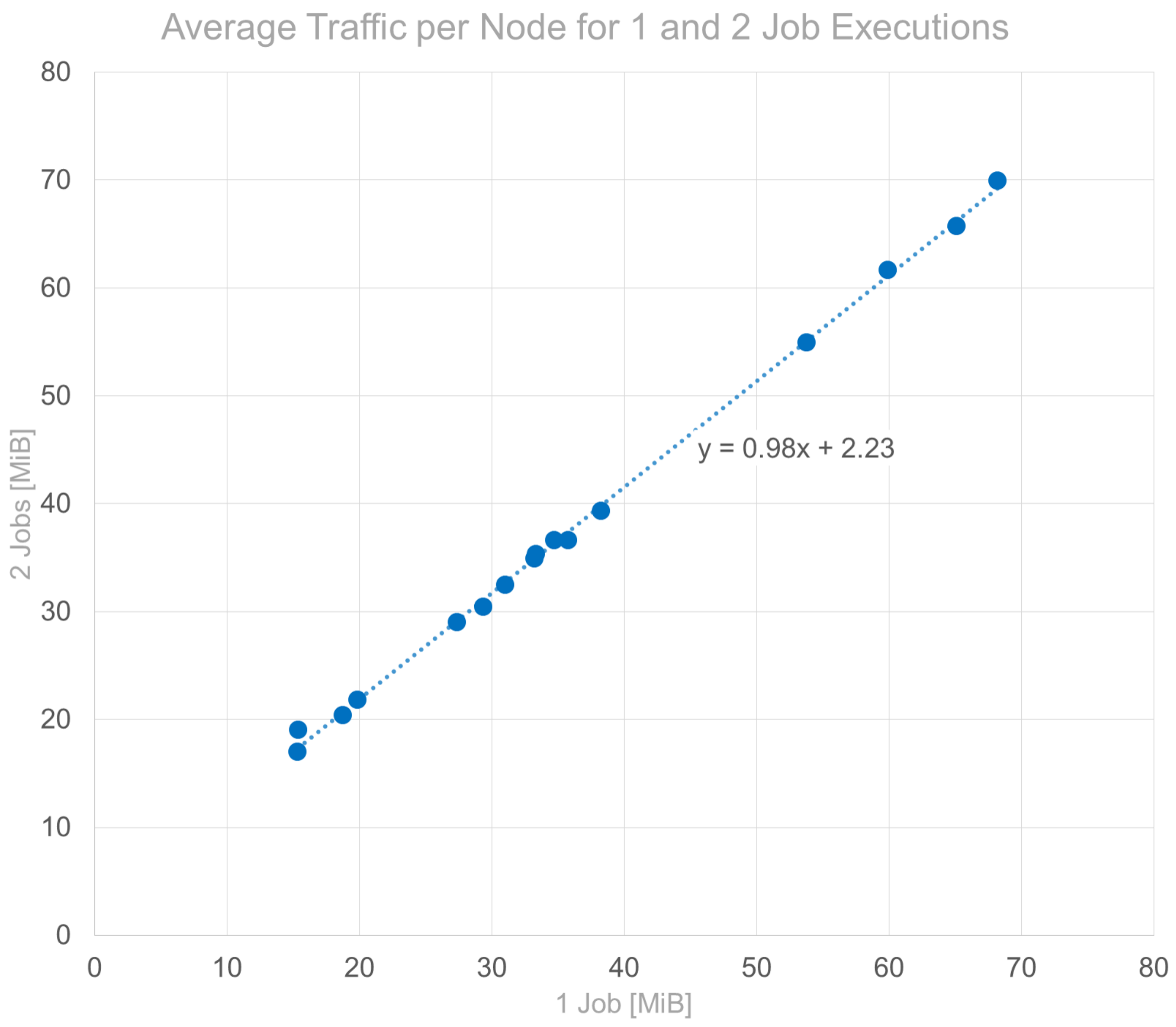


Figure 6.5: Cumulated overall per-node traffic in MiB for one and two job executions of same split and file sizes as presented in Table 6.2, plotted against each other. See Table 6.3 for the actual values.

6.3.2 Results

Figure 6.5 shows a clear linear correlation of mean overall per-node traffic with increasing file sizes. Spearman rank sum correlation test [25] also confirms this correlation to be

Config.	**1 job [MiB]**	**2 jobs [MiB]**	**Δ [MiB]**
A-3	15.36	17.02	+1.66
A-4	15.43	19.01	+3.58
A-5	18.80	20.37	+1.58
A-6	19.88	21.79	+1.91
B-3	27.41	29.03	+1.62
B-4	31.04	32.46	+1.42
B-5	33.26	34.93	+1.67
B-6	34.72	36.64	+1.92
C-3	29.39	30.45	+1.06
C-4	33.38	35.32	+1.93
C-5	35.78	36.63	+0.85
C-6	38.29	39.35	+1.06
D-3	53.82	54.93	+1.11
D-4	59.95	61.64	+1.68
D-5	65.13	65.73	+0.59
D-6	68.23	69.92	+1.70

Table 6.3: Mean overall traffic per node for the execution of same dataset sizes for one and two jobs. In the Config. column, A to D refer to column # in Table 6.2, and 3 to 6 specify the number of nodes. Δ indicates the difference in traffic sizes between one and two jobs.

statistically significant (p-value < 0.0001) with a very high positive correlation coefficient of 0.997, indicating an almost perfect positive correlation (1.0) as expected. Also the estimated linear line in the plot has a slope close to 1 (0.98). This shows that the overall traffic is almost the same between one and two jobs (for example, if the traffic produced by a single job is 60 MiB, traffic measured for two jobs is also very close to 60 MiB). Additionally, Table 6.3 shows similarly to experiment 1 how traffic sizes increase within split and file size configurations when more nodes are added. On closer analysis, the same problems already revealed in experiment 1 can be observed as well and are thus not investigated further. However, besides these explained differences *within* job execution configuration, the calculated linear line in the plot also indicates an offset of 2.23 MiB *between* them, meaning that for every single-job mean overall traffic value, the estimated corresponding two-job value is off by +2.23 MiB. This offset can also be seen in Table 6.3 in column Δ. Depending on the configuration, it ranges from 0.59 to 3.58 MiB. The increase is systematically positive for two jobs compared to one job (although the magnitude may vary) and stays between the specified range even though the overall dataset sizes increase. Analysis of the individual messages show a problem in the configuration of the job *executions*: for two jobs, ReduceTask is executed *four times*, not twice, which results in two additional put operations. The same applies to the PrintTask that retrieves the result data overall four times, not only twice, as each job dictates it to be executed twice. Thus, the overall systematic positive increase can be attributed to these additional two operations, supporting the hypothesis outlined initially that the mean overall per-node traffic remains constant regardless of the number of jobs executed at the same time.

Chapter 7

Summary and Conclusions

This chapter takes a look back at the initially formulated research questions, summarises the achievements, and gives an outlook for future work to improve the presented prototype.

7.1 Research Questions and Thesis Goals Revisited

Overall, the initially formulated thesis goal of designing and implementing a *completely decentralised MapReduce environment on top of a P2P overlay network with high user flexibility, little coordination and communication overheads, and a distributed hash table as main storage facility* (**RQ 1**), could be achieved. The system allows users to implement a data processing chain without being restricted to only map and reduce functions. The high flexibility enables the implementation of any additional method or feature required before, during, or after processing, which could be demonstrated to be effective with a simplified word count example. The evaluation also shows that the system is not restricted to a small number of nodes but may likely be deployed on more than the presented 6 machines. Additionally, experiments with Ubuntu 14.04 LTS (32 and 64Bit), Ubuntu 15.10 LTS, and Microsoft Windows 10 confirm the prototype's intended effortless deployability on various java-enabled platforms. As data and jobs can be stored and transferred either through broadcasts or DHT calls, it can be easily accomplished that new nodes may join and immediately start execution, thus contributing to the overall finishing of a job without any additional communication or coordination needed. Broadcasts, while kept small, provide a feasible way of communication and allow a very loose coupling with only few dependencies between executing nodes and tasks. If such dependences are still required, users can implement them in the task extensions. Furthermore, the restriction to only send broadcasts on completed executions simplifies coordination tremendously. The number of executions per data item can effectively be limited by internal mechanisms, which additionally create randomised data accesses as a side effect. Thus, nodes are enforced to execute different data items concurrently without requiring any other form of coordination or assignment of tasks, avoiding a master-slave architecture. Nevertheless, there are still some issues to resolve and more improvements need to be undertaken before the prototype becomes a viable alternative to established MapReduce frameworks like Hadoop [10], which is why Section 7.2 provides guidelines for and further discussion points on potential extensions of the prototype to incrementally improve its performance.

Another goal of this thesis was *to keep traffic constant for single-job executions if more nodes are added* (**RQ 2.1**) but also *if two jobs are executed at the same time with the same overall file size to process as an equivalent single job* (**RQ 2.2**). As evaluation results show, although traffic increases can be explained in many cases, it is still not completely constant overall. There are certain factors that may lead to larger overall traffic users need to be aware of as they may directly influence them with their implementation behaviour:

1. As a consequence of more connected nodes, also the number of average GET requests and broadcast messages per node increases. If a job is sent with every GET request and broadcast, the amount of traffic produced may quickly add up to some MiB due to the serialised objects and class files within the job. A strategy to overcome this is to put the whole job into the DHT and send only the key to it on every broadcast. Furthermore, larger split sizes resulting in fewer keys also decrease the overall number of broadcast messages and GET requests. Nevertheless, in the context of overall traffic, GET requests and broadcast messages have only a marginal impact compared to PUT request (storing data items) and GET response (retrieval of data items) operations as shown in Figure 6.1, the latter being effectively limited with measures described in Section 5.5.2.

2. Although the time offset allows for a better enforcement of user-defined execution restrictions, the majority vote intended to avoid sudden system interruptions may lead to superfluous executions that consequently result in a higher overall per-node traffic.

3. In cases of multiple job executions, the same job executed twice in parallel with half the dataset size of a single job (but overall the same amount of data) may not be the same due to a different number of task executions. It is the responsibility of the user to take care of such problems.

Moreover, the time offset enforced on GET requests, although limiting additional unwanted executions, restricts optimisations due to the randomised execution times: if a user knows certain nodes to have a better performance than others, it cannot be guaranteed that these nodes will also execute more tasks. However, according restrictions may also be implemented in the task directly (e.g., a user can identify a node by accessing the Peer's ID and choose to only execute a certain task on a specific node).

Besides, although not directly assessed here, a note on execution speeds is in order. Network access is one of the principle factors that harms the system's execution speed, which was already emphasised in other publications (e.g. [29]) and one of the main reasons why centralised systems like Apache Hadoop and its HDFS can achieve such high performances. Preliminary experiments depicted in Appendix B also indicate increased running times for multi-node executions that require network access compared to single-node executions that only store and retrieve data locally. Also Google's MapReduce implementation takes the location information of input files into account and attempts to schedule a map task on a machine that actually contains a replica of the input data [16]. Proposed optimisations in Section 7.2 are a first step towards achieving performance levels closer to these frameworks.

7.2 Future Work

During design, implementation, and evaluation, several limitations of the presented prototype became apparent. Besides the urgently needed resolution of some issues, the proposed additional future works provide a guideline in the endeavour of making the prototype a more feasible alternative to other implementations like e.g. Hadoop [10].

7.2.1 Encountered Issues and Limitations

There are several issues in the current implementation to be resolve as soon as possible that are outlined next. Furthermore, some limitations and points of awareness are listed for potential users to consider.

Timeouts

As outlined in Section 6.1.1, there is a problem regarding resets of connection timeouts. The current implementation makes use of Integer.MAX_VALUE to overcome this problem. However, with larger datasets that take much longer to execute, the same problem as for smaller datasets can be expected. Additionally, nodes may run out of connections to open if too many are established at the same time. Therefore, an urgent future work is to identify the reason why the system does not reset the timeouts and solve it accordingly.

Multiple Simultaneous Task Executions on a Node

At the moment, every node can only retrieve every data item once to avoid problems with eventual consistency measures as explained in Section 5.5.2. Only when a broadcast for a completed task of this data item is received, the same node is able to try to access the same item again. Thus, in a next development step, it should be evaluated how a node can distinguish multiple accesses to the same data item at the same time.

Filtered Broadcast Messages

A further encountered issue is that broadcast messages may be received multiple times on a node, leading to superfluous executions of the same task. This behaviour is observed systematically for nodes that are not the sender of a message. So far, it is hypothesised that this behaviour can be attributed to the way in which broadcasting is implemented in

TomP2P's broadcasting facilities (see Section 3.2.3). It is currently suppressed by storing received messages for its contained storage key and ignoring messages that contain the same key. Consequently, every node may only execute tasks on each storage key once. This limits the possibility for executing tasks that were resubmitted due to node crashes. Thus, timeouts on the storage key need to be introduced so that new messages containing the same key can be executed again after a certain amount of time.

Deserialisation Issues

A note of awareness for users is that class files and objects of user-defined extensions sent to nodes and then deserialised again may result in new class files and objects, see the brief explanations in Section 5.2.2. A class that is checked to be of equal type as another one using getClass() will not be recognised as such although it indeed is the same class. This could be observed for implementations of IMapReduceBroadcastReceiver, where the same instance was added multiple times although it was only intended to be added once. A subclass-wide identifier (as explained in Section 5.2.2) makes sure that only one and not multiple instances are added on the same node with every broadcast received. Similarly, if a task in a job relies on the temporary storage of data, as e.g. ReduceTask does in the presented implementation, an existing job may be overwritten and all contained data items lost if the same job is deserialised and assigned multiple times due to the repeatedly received broadcast messages. Thus, users need to be aware that every deserialisation on a node yields new class files and objects and take according precautions to avoid unexpected problems.

Thorough Testing & Evaluation

Currently, only most important classes are tested, like TaskRPC, DistributedTask, and SerializeUtils without which the current prototype would not work. Included in these tests are also some of the other classes like connection listeners or PeerMapReduce. However, some of the above mentioned issues may also be found through more thorough testing, thus requiring more test cases for both classes and components. Moreover, the datasets used in these initial evaluations are very small. Further executions with larger datasets and possibly more nodes at the same time are thus required. Additionally, although implemented and tested using test cases, releasing data items and resubmission of associated broadcasts as a consequence of node crashes during execution are not yet assessed in a real situation as the current evaluation only covers an ideal case without node crashes.

7.2.2 Data Clean-Up and Storage

The current prototype lacks facilities to remove intermediate or unused results and simply stores everything from start to end in the memory-based storage object. As the evaluation revealed, already small datasets of only 12 MiB may produce an overall PUT traffic per node that can, depending on the chosen configurations, increase by several magnitudes (in the presented setting, up to five times larger, see Figure 6.1). Moreover, the use of disk-based storage to enable the execution of much larger dataset sizes needs to be implemented and tested. To avoid overloading nodes with outdated datasets, timeouts on the data should to be added, such that the system automatically removes unused data sets after a certain amount of time it was not accessed. It has to be noted that the current implementation would allow users to implement such facilities themselves if they intended to do so by accessing the storage object through the PeerMapReduce class.

7.2.3 Reducing Traffic and Execution Time

A clear problem in the prototype, which tremendously harms execution time, is the current use of network accesses on every PUT and GET request. Local execution may allow for much shorter processing time and smaller traffic overheads if most GET operations are executed directly on the own storage and instead of accessing the network. Only PUT operations would be necessarily required to distribute the data onto the different nodes. Execution could look as follows: first, the StartTask reads the local files and distributes them to the DHT using the same PUT requests as before and finishes with a broadcast for every file. Instead of trying to acquire the data through DHT-based GETs, however, every node tries to execute successive MapTasks by finding the data on the own *local* storage first instead. If it does not find it, the execution for the received storage key is simply ignored or at least postponed for when there are still tasks to execute but the locally stored datasets are exploited. Only then could a peer start to access the DHT using *distributed* GET requests. Such an implementation would also further ease the need for synchronisation of the datasets before granting or denying their access (see majority vote and random time offsets mentioned in Section 5.5.2) as parallel requests may only rarely need to be handled. Additionally, this could contribute to reducing execution time as well. Again, users are able to implement such a behaviour to some extent themselves by directly accessing the storage object instead of using PeerMapReduce.get().

7.2.4 Simplified Task Implementation

In an earlier prototype, Oracle Nashorn [7] was used to allow for the implementation and transfer of JavaScript functions instead of Java classes. Most importantly, serialisation and deserialisation of both Java class files and objects could be simplified as only a set of Java String representations needs to be sent to nodes. Thus, the already existent components of this engine should be ported to and tested on the new prototype.

Bibliography

[1] Free ebooks by Project Gutenberg. https://www.gutenberg.org. Accessed: 2016-04-11.

[2] Hadoop MapReduce Next Generation - Setting up a Single Node Cluster. http://tinyurl.com/hadooplin. Accessed: 2016-04-30.

[3] Hadoop2OnWindows. http://tinyurl.com/h24win. Accessed: 2016-04-30.

[4] IBM SPSS - IBM Analytics. http://tinyurl.com/ibmspss23. Accessed: 2016-04-11.

[5] Java SE Development Kit 8 - Downloads. http://tinyurl.com/java8oracle. Accessed: 2016-04-15.

[6] Mars Eclipse. https://eclipse.org/mars. Accessed: 2016-04-11.

[7] Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM. http://tinyurl.com/oraclenashorn. Accessed: 2016-04-21.

[8] Spreadsheet Software Programs - Excel Free Trial. http://tinyurl.com/huvub24. Accessed: 2016-04-11.

[9] TomP2P, a P2P-based key-value storage library. http://http://tomp2p.net. Accessed: 2016-02-23.

[10] Welcome to Apache Hadoop! https://hadoop.apache.org. Accessed: 2016-02-23.

[11] Wireshark - Go Deep. https://www.wireshark.org. Accessed: 2016-04-16.

[12] G. Camarillo. Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability. http://tinyurl.com/p2poverview. Accessed: 2016-04-20.

[13] E. Cesario, N. De Caria, C. Mastroianni, and D. Talia. *Grids, P2P and Services Computing*, chapter Distributed Data Mining using a Public Resource Computing Framework, pages 33–44. Springer US, Boston, MA, 2010.

[14] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like P2P systems scalable. *Proceedings of the 2003 conference on Applications technologies architectures and protocols for computer communications SIGCOMM 03*, 25:407, 2003.

[15] Z. Czirkos and G. Hosszú. P2P based intrusion detection. *Infocommunications Journal*, 64:3–10, 2009.

[16] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] A. Lareida, T. Bocek, and S. Golaszewski. Box2Box-A P2P-based File-Sharing and Synchronization Application. *Proceedings of the 13th IEEE Conference on Peer-to-Peer Computing (P2P\ '13)*, pages 2–3, 2013.

[18] H. Lin, X. Ma, J. Archuleta, W. Feng, M. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM.

[19] F. Marozzo, D. Talia, and P. Trunfio. A Framework for Managing MapReduce Applications in Dynamic Distributed Environments. In *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 149–158, Feb 2011.

[20] F. Marozzo, D. Talia, and P. Trunfio. P2P-MapReduce: Parallel data processing in dynamic Cloud environments. *Journal of Computer and System Sciences*, 78(5):1382–1402, sep 2012.

[21] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-addressable Network. *SIGCOMM Computer Communication Review*, 31(4):161–172, aug 2001.

[23] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[24] R. Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pages 101–102, Washington, DC, USA, 2001. IEEE Computer Society.

[25] C. Spearman. The Proof and Measurement of Association Between Two Things. *American Journal of Psychology*, 15:88–103, 1904.

[26] L. A. Steffenel. First Steps on the Development of a P2P Middleware for Map-Reduce. Technical report, 2013.

[27] L. A. Steffenel, O. Flauzac, A. S. Charão, P. P. Barcelos, B. Stein, G. Cassales, S. Nesmachnow, J. Rey, M. Cogorno, and C. Souveyet. MapReduce Challenges on Pervasive Grids. 10(11):2192–2207, 2014.

[28] L. A. Steffenel, O. Flauzac, A. S. Charão, P. P. Barcelos, B. Stein, S. Nesmachnow, M. Kirsch-Pinheiro, and D. Diaz. PER-MARE: Adaptive Deployment of MapReduce over Pervasive Grids. In *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 17–24. IEEE, oct 2013.

[29] L. A. Steffenel and M. Kirsch-Pinheiro. Leveraging Data Intensive Applications on a Pervasive Computing Platform: The Case of MapReduce. *Procedia Computer Science*, 52:1034–1039, 2015.

[30] L. A. Steffenel and M. K. Pinheiro. CloudFIT, a PaaS platform for IoT applications over Pervasive Networks. In *3rd Workshop on CLoud for IoT (CLIoT 2015)*, sept 2015.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, pages 149–160, 2001.

[32] H. M. Tran, S. V. U. Ha, T. K. Huynh, and S. T. Le. A feasible MapReduce peer-to-peer framework for distributed computing applications. *Vietnam Journal of Computer Science*, 2(1):57–66, 2014.

Abbreviations

B	Bytes
CPU	Computing Processing Unit
DHT	Distributed Hash Table
GB	Gigabyte in decimal, 10^9 B
GHz	Gigahertz, 10^9 Hz
GiB	Gigabyte in binary (gibibyte), 2^{30} B
HDD	Hard Disk Drive
KB	Kilobyte in decimal, 10^3 B
KiB	Kilobyte in binary (kibibyte), 2^{10} B
MB	Megabyte in decimal, 10^6 B
MiB	Megabyte in binary (mebibyte), 2^{20} B
OS	Operating System
PB	Petabyte in decimal, 10^{15} B
PiB	Petabyte in binary (pebibyte), 2^{50} B
P2P	Peer-to-Peer
RAM	Random Access Memory
SSD	Solid State Drive
TB	Terabyte in decimal, 10^{12} B
TiB	Terabyte in binary (tebibyte), 2^{40} B

List of Figures

List of Tables

Appendix A

Exemplary Job Submission

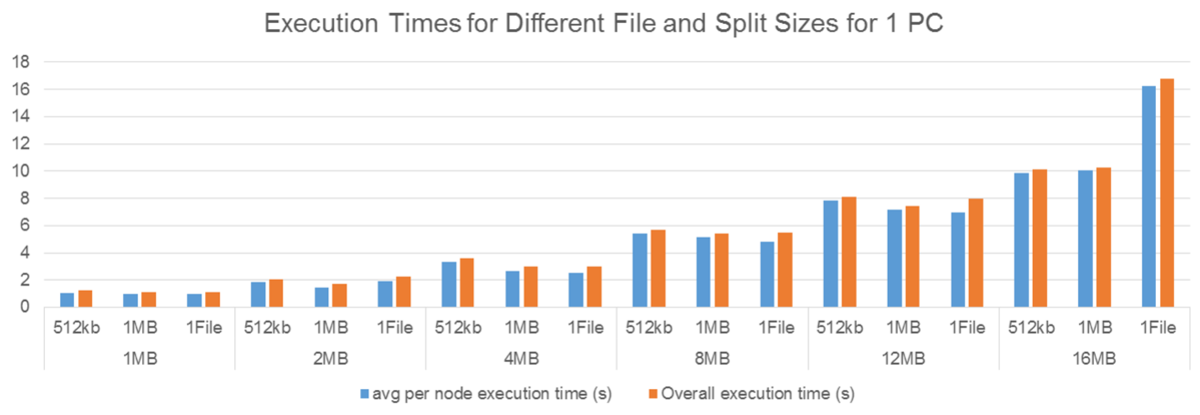
Listing A.1: Simplified Java code to define a MapReduce job

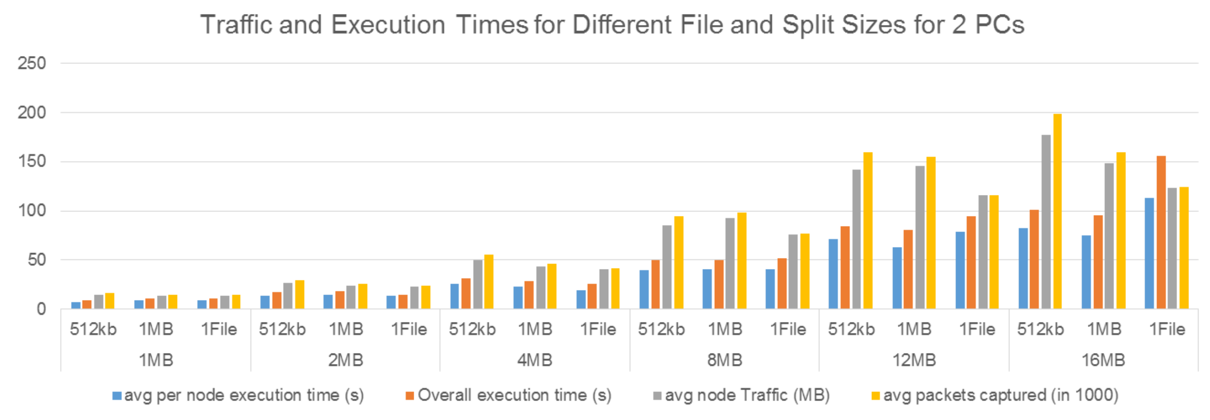
```
// Instantiate all tasks and broadcast receiver
Task startTask = new StartTask(null, NumberUtils.next());
Task mapTask = new MapTask(startTask.currentId(), NumberUtils.next());
Task reduceTask = new ReduceTask(mapTask.currentId(), NumberUtils.next());
Task printTask = new PrintTask(reduceTask.currentId(), NumberUtils.next());
Task shutdownTask = new ShutdownTask(writeTask.currentId(), NumberUtils.next());
IMapReduceBroadcastReceiver receiver = new ExampleJobBroadcastReceiver();

// Add all tasks and broadcast receiver to a new job
Job job = new Job();
job.addTask(startTask);
job.addTask(mapTask);
job.addTask(reduceTask);
job.addTask(printTask);
job.addTask(shutdownTask);
job.addBroadcastReceiver(receiver);

// Define the input for the first locally executed task (StartTask)
NavigableMap<Number640, Data> input = new TreeMap<>();
input.put(NumberUtils.allSameKey("DATAFILEPATH"), new Data(filesPath));
input.put(NumberUtils.allSameKey("NUMBEROFFILES"), new Data(nrOfFiles));

// Connect to the overlay
PeerBuilder peerBuilder = ... //User configuration of the peer
PeerMapReduce peerMapReduce = new PeerMapReduce(peerBuilder);

// Start a MapReduce job
job.start(input, peerMapReduce);
```

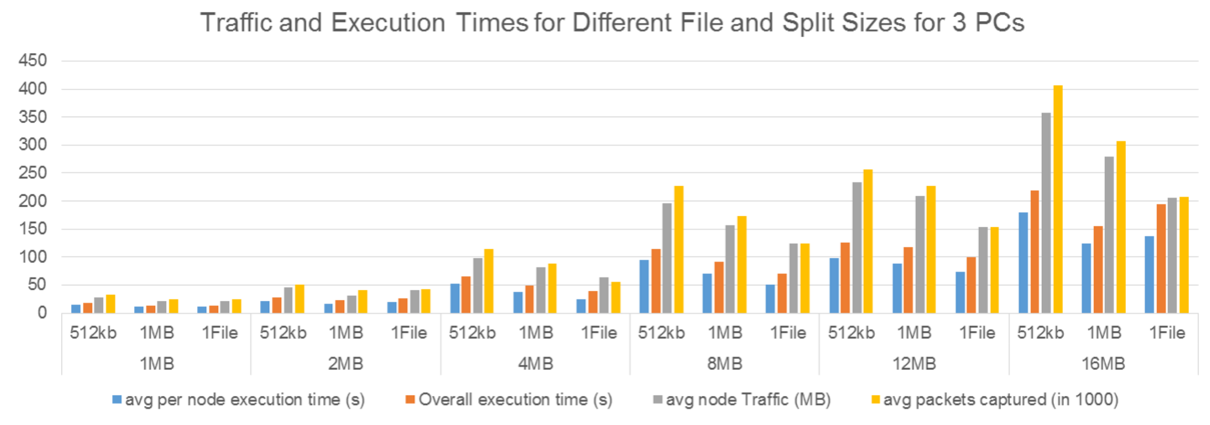
Appendix B

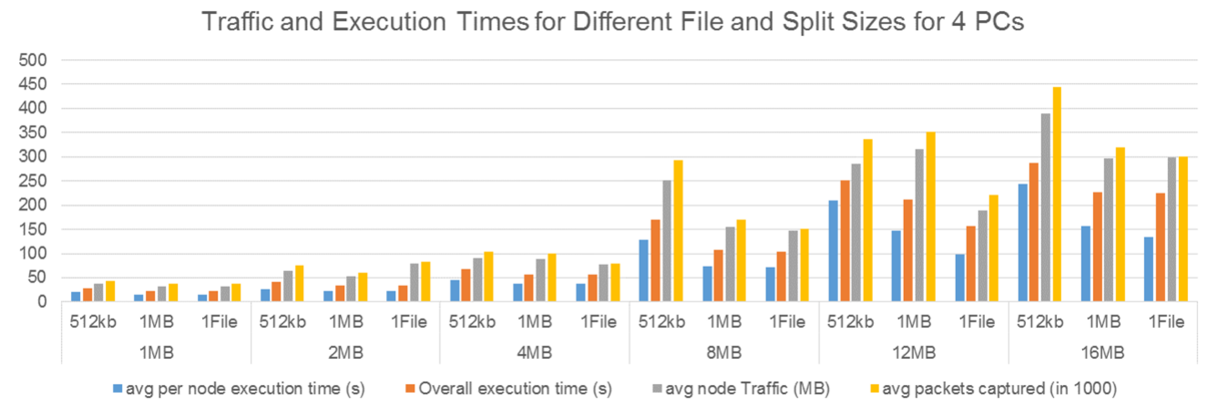
Initial Experiments

Before the actual experiments, runs were conducted for a multitude of file and split sizes, which are depicted below. These charts show average per-node traffic and execution times for 1 to 4 nodes. Traffic was captured using Wireshark [11] instead of the Decoder-class-parsed output used in the final evaluation and thus, covers not only TCP/UDP traffic but also IP headers etc. Furthermore, network access was provided through a wireless mobile phone hot spot (Samsung Galaxy S6 edge), which is noticeably slower than the used Ethernet link in the final evaluation. An increase in traffic sizes could be observed when adding more nodes, which could be explained by the problematic eventual consistency features outlined in Section 5.5.2. Furthermore, stability could not be guaranteed always and the prototype often halted mid-execution for the same reasons. This led to adaptations like the user-defined waiting time and majority voting scheme, which allowed the whole system to run much more stable and with fewer unintended additional executions as explained in Section 5.5.2, too.




Traffic and Execution Times for Different File and Split Sizes for 2 PCs
250
200
150
100
50
0
512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File
1MB
2MB
4MB
8MB
12MB
16MB
avg per node execution time (s)
Overall execution time (s)
avg node Traffic (MB)
avg packets captured (in 1000)


Traffic and Execution Times for Different File and Split Sizes for 3 PCs
450
400
350
300
250
200
150
100
50
0
512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File
1MB
2MB
4MB
8MB
12MB
16MB
avg per node execution time (s)
Overall execution time (s)
avg node Traffic (MB)
avg packets captured (in 1000)


Traffic and Execution Times for Different File and Split Sizes for 4 PCs
500
450
400
350
300
250
200
150
100
50
0
512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File 512kb 1MB 1File
1MB
2MB
4MB
8MB
12MB
16MB
avg per node execution time (s)
Overall execution time (s)
avg node Traffic (MB)
avg packets captured (in 1000)

Appendix C

Overall Mean Traffic Per Node

The following table features overall mean traffic per node, decomposed into different message types for 3 to 6 nodes as both MiB values and number of messages sent. GET OK only features message counts, as OK response messages could only be extracted as a combination of PUT and GET. Thus, sizes in MiB could not be reliably differentiated between the two response types. However, as all PUT requests are expected to result in an OK PUT reply, the number of GET OK messages could be estimated by subtracting the PUT request messages from the overall OK messages for PUT and GET.

		MiB				**#Messages**			
Msg. Type	**#Nodes**	**3**	**4**	**5**	**6**	**3**	**4**	**5**	**6**
PUT requests	mean	46.24	52.02	55.49	57.78	49.33	55.50	59.20	61.64
	median	46.24	52.02	55.49	57.80	49.33	55.50	59.20	61.67
	σ	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.06
	min	46.24	52.02	55.49	57.63	49.33	55.50	59.20	61.50
	max	46.24	52.02	55.49	57.80	49.33	55.50	59.20	61.67
GET requests	mean	7.05	9.48	11.76	13.99	81.33	109.50	136.00	161.42
	median	7.05	9.48	11.75	14.02	81.33	109.50	136.00	161.67
	σ	0.01	0.02	0.02	0.07	0.00	0.00	0.00	0.77
	min	7.04	9.45	11.74	13.74	81.33	109.50	136.00	158.33
	max	7.06	9.51	11.79	14.03	81.33	109.50	136.00	161.67
Broadcast	mean	5.16	7.43	9.46	11.67	60.37	87.16	111.16	136.75
	median	5.18	7.51	9.30	11.67	60.67	88.25	109.20	137.00
	σ	0.29	0.34	0.31	0.20	3.42	3.90	3.57	2.29
	min	4.70	6.82	9.16	11.33	55.00	80.25	107.80	133.00
	max	5.52	7.99	10.05	12.02	64.67	94.00	118.20	140.83
PUT/ GET OK	mean	33.13	36.31	37.95	39.82	114.67	128.64	137.08	142.68
	median	33.13	36.07	37.84	39.73	114.67	128.75	137.20	142.67
	σ	0.00	0.55	0.80	0.81	0.00	0.23	0.30	0.33
	min	33.13	35.82	36.68	38.13	114.67	128.25	136.60	142.00
	max	33.13	37.27	39.75	41.41	114.67	129.00	137.60	143.33
GET OK	mean					65.33	73.14	77.88	81.04
	median					65.33	73.25	78.00	81.00
	σ					0.00	0.23	0.30	0.31
	min					65.33	72.75	77.40	80.33
	max					65.33	73.50	78.40	81.67
GET DENIED	mean	0.00	0.00	0.00	0.00	16.00	36.00	57.62	79.75
	median	0.00	0.00	0.00	0.00	16.00	36.00	57.60	80.00
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.76
	min	0.00	0.00	0.00	0.00	16.00	36.00	57.60	77.00
	max	0.00	0.00	0.00	0.00	16.00	36.00	57.80	80.33
PUT/ GET NOT FOUND	mean	0.00	0.00	0.00	0.00	0.00	0.36	0.50	0.62
	median	0.00	0.00	0.00	0.00	0.00	0.25	0.40	0.50
	σ	0.00	0.00	0.00	0.00	0.00	0.23	0.32	0.37
	min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	max	0.00	0.00	0.00	0.00	0.00	0.75	1.00	1.50

Table C.1: Overall mean traffic per node decomposed into different message types for 3 to 6 nodes in both MiB and number of messages sent. σ denotes the standard deviation.