University of Zurich[UZH]

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

Design and Prototypical Implementation of ...

Author
City, Country
Student ID: 00-711-999

Supervisor: ...
Date of Submission: January 1, 2006

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

ifi

Bachelor 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

Das ist die Kurzfassung...

Acknowledgments

Optional

Contents

Chapter 1

Introduction

1.1 Motivation

Big data blabla

1.2 Description of Work

1.3 Thesis Outline

Chapter 2

Related Work and State of the Art

Although the widest usage of the MapReduce programming model to date may be the one of Hadoop [2] with its centralised master-slave architecture, recently, there have also been a number of attempts to transport it to a more decentralised setting in an endeavour to support currently popular Cloud platforms, pervasive grids, and/or mobile environments. Such endeavours address the problem of Hadoop (and similar MapReduce implementations) being designed for dedicate environments and not supporting dynamic environments with high churn rates [12].

[2] is an open-source implementation of MapReduce (mainly implemented by Yahoo) enjoying a wide adoption and is often used for short jobs, where low response time is critical [17]. It's implementation closely resembles Google's [4], where master nodes manage a number of slave nodes. The input file resides in the distributed file system (HDFS [5]) and is split in even chunks replicated for fault-tolerance [17]. Output of map tasks are not replicated. [5]. task scheduler implicitly assumes cluster nodes to be homogeneous and tasks progress linearly to decide if a task should be re-executed (in homogeneous environments, execution should be more or less equal on each node). This makes Hadoop rather unrealistic for "office" settings, where commodity hardware may vary greatly in performance.

[5] explore the limitations of Hadoop over volatile, non-dedicated resources. They propose the use of a hybrid architecture where a small set of reliable nodes are used to provide resources to volatile nodes called MOON. They demonstrated that Moon's data and task replication design greatly improved the quality of service of MapReduce when running on a hybrid resource architecture with many volatile and only a small set of dedicated nodes. However, there tests only included homogeneous configurations across nodes, creating natural heterogeneity only through node unavailability.

[6] recognised the problem of centralised master-slave architectures to not cope well with dynamic cloud infrastructures, where nodes may join or leave the network at high rates. Thus, they introduce a peer-to-peer model to manage node churn but also master failures and job recovery in a decentralised way. Each node may become a master or a slave at any given time dynamically, preserving a certain master/slave ratio. Slaves are assigned tasks to perform by the masters, which handle management, recovery, and coordination.

To reduce job loss in case of master failures, each master may act as a backup master for a certain job, only executing it if the master primarily responsible for that job fails. Overall, the structure of different entities resembles very much the one of Hadoop, simply ported to a P2P setting. They were able to show that such an implementation provides better fault tolerance levels compared to a centralised implementations of MapReduce and only limited impact on network overhead.

In a very similar direction goes the PER-MARE initiative that aimed at proposing scalable techniques to support existing MapReduce data-intensive applications in the context of loosely coupled networks (e.g. desktop grids) [11]. The aim was two-fold to develop a MapReduce implementation for pervasive or desktop grids and to keep the implementation compatible to Hadoop's API while using a P2P environment. The hope was to be able to reuse existent applications over pervasive grids. The identified problem relied on the assumption that Hadoop and other MapReduce implementations are designed for dedicate environments and do not support environments with high node churn. The authors already hypothesised about using a DHT with controlled data replication as a solution to ensure fault tolerance without relying on full data replication. However, their first Prototype CONFIIT was abandoned [10] for the later CloudFIT implementation (see below) due to several problems that lead to an exponential increase in execution time with increasing data size, which made it unsuitable for the intended purpose.

CloudFIT [13, 14], which the authors advertise as a "Platform-as-a-Service (PaaS) middleware, allows the creation of private clouds over pervasive environments". The idea is to use private computers to set up a private "Cloud", and that MapReduce jobs are then distributed to this environment and executed in a P2P fashion. CloudFIT is, thus, built to support various P2P overlays and the authors show the performance of CloudFIT with both PAST and TomP2P against Hadoop. The results demonstrated CloudFIT to be able to achieve similar execution speeds as Hadoop while omitting the need of a dedicated cluster of computers. Data in CloudFIT is directly stored within the DHT of the corresponding overlay, such that it is replicated by a factor of k. Although such an implementation may not guarantee n-resiliency (meaning, replicating the data on each node), it is a reasonable assumption that fault tolerance may be ensured more than enough (provided at least a number of nodes stay alive and the complete dataset is replicated on these nodes) while drastically improving storage performance. Furthermore, not all the data is broadcasted but only the keys of each task, further reducing the network workload the authors showed in another study [14] to be one of the main problems in distributed environments for the performance of MapReduce tasks.

Chapter 3

Methods and Prototype

In this section, the used methods are laid out and the implemented prototype is described.

3.1 Methods used

3.1.1 MapReduce

At the very core of this thesis lies the idea of MapReduce, a programming model made popular by Google and presented in [4]. Having large data sets to analyse that may not be handled by only few computers, like the millions of Websites gathered by Google each day that have to be indexed fast and reliably, MapReduce provides a way of automatic parallelisation and distribution of large-scale computations. Users only need to define two types of functions: a map and a reduce function. Each computation expressed by these functions takes a set of input key/value pairs and produces a set of output key/value pairs. A map function takes an input pair and produces a set of *intermediate* key/value pairs. Once the intermediate values for each intermediate key I are grouped together, they are passed to the reduce function. The reduce function then takes an intermediate key I and the corresponding set of intermediate values (usually supplied as an iterator to the reduce function) and merges them according to user-specified code into a possibly smaller set of values. The resulting key and value are then written to an output file, allowing to handle lists of values that are too large to fit in memory.

A very simple yet often used MapReduce operation is to count words in a collection of documents, the so called *WordCount*. Below pseudocode demonstrates how one can count all the words in documents using map and reduce functions.

```
function MAP(String key, String value)
    // key: document name, value: document content
    for word w in values do
        EmitIntermediate(w, "1");
    end for
end function
```

```
function REDUCE(String key, Iterator values)
    // key: a word, values: a list of "1"'s
    int result = 0;
    for v in values do
        result += ParseInt(v);
    end for
    Emit(AsString(result));
end function
```

The map function splits the received text (*value*) into all corresponding words and for each word, emits its occurrence count (simply a 1). The reduce function then takes, for each word (*key* of the reduce function) individually, these occurrences (*values*) and sums them up, eventually emitting for each word the sum of occurrences (*result*). Although above pseudocode suggests inputs and outputs to be strings, the user specifies the associated types. This means that input keys and values may be of a different domain than the output keys and values (note that the *intermediate* keys and values are from the same domain as the output keys and values). Conceptually, this is expressed as follows:

map	(k1, v1)	→ list(k2, v2)
reduce	(k2, list(v2))	→ list(v2)

The traditional way to view the workflow in a MapReduce program is indicated by Figure 3.1. A master schedules execution and assigns tasks to workers, which then perform the actual map or reduce on the assigned data. Input data is partitioned into M splits with a size of around 16-64 MB per piece to be processed in parallel by different machines, and the intermediate key space is partitioned into R pieces, where R usually is the number of available workers for executing reduce but may also be specified by the user. To evenly distribute these pieces, a partitioning function like *hash(key) mod R* may be used. In such a setting, same intermediate keys may not be grouped together, which is done before the reduce action is performed. Additional combination in between may be applied to reduce the amount of data to be distributed among the workers.

3.2 Peer-to-Peer Overlay Networks

3.2.1 Definitions and Flavours

Peer-to-peer (P2P) systems and applications are distributed systems without any centralized control or hierarchical organisation, where the software running at each node is equivalent in functionality [15]. Instead of a central server (as present in Client/Server architectures [9]), many hosts (usually referred to as peers) that posses desired contents also handle user requests for them [3]. According to [9], a P2P network is given if the participants share a part of their own hardware resources (like e.g. processing power or storage capacity), which provide the service and content offered by the network and which are directly accessible by other peers. Peers are, thus, both resource (service and content) providers and requesters. There exist different flavours of P2P overlay networks but the

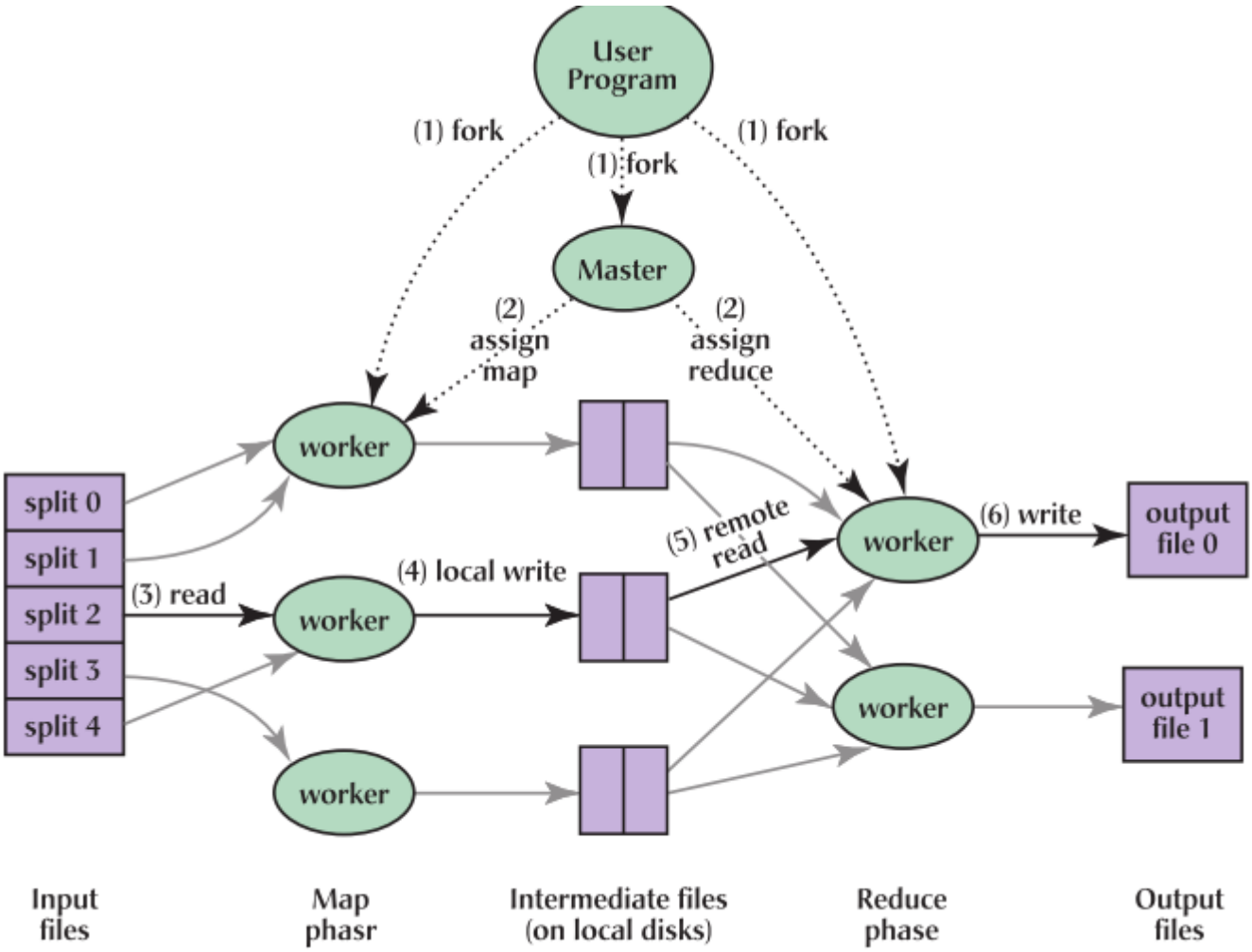


Figure 3.1: Execution overview in a typical MapReduce program

overall themes may be summarised as "pure" P2P and "hybrid" P2P systems. In a *pure P2P system*, any entity may be removed without the network suffering any service loss. A *hybrid P2P system*, on the other hand, requires a central entity to provide parts of the offered networks. An example of such a hybrid system was Napster [3]. The used P2P system in this work is TomP2P which will be explained shortly.

3.2.2 Discovering Content and Distributed Hash Tables

The definitions given above demonstrate the need for efficiently discovering content and/or services without using a central server. Central servers may provide O(*1*) lookup of a file's location in the P2P network but also suffers from the problem of being a single point of failure (SPOF), which made Napster very efficient for lookup but also vulnerable for lawsuits [3] and it was certainly not scalable at that time [8]. However, although being rather resilient to node crashes, without efficient lookup capabilities, P2P networks are *unstructured* and as there is no constraint on where files are placed in the overlay network, queries need to be flooded across the overlay to find the location of the desired content (requiring O(n) lookups). Such overlay networks like e.g. Gnutella were, therefore, not scaling. To overcome this, distributed hash tables (DHTs) were introduced, of which the Content-Addressable Network (CAN, [8]) may be one of the first such design outlines intended to be used in P2P networks. Keys are mapped onto values in a DHT, operations include (among others) typical hash table or hash map operations like put(*key*, *value*), get(*key*): *value*, etc. The idea is that the problem of P2P systems is not the file transfer process which is inherently scalable but finding the peer from whom to retrieve the file[8]. Thus, the indexing scheme needs to be scalable to make the whole P2P system scalable. Importantly, it requires no centralized control and there is no rigid hierarchical naming structure like e.g. in IP or DNS routing. DHTs make overlay networks *structured* and provide hash-table-like semantics on internet-like scales [8]. Efficient routing protocols (e.g. Chord[15], Kademlia[7]) 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 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[3]), they provide a good balance between node state and communication overhead. As TomP2P uses Kademlia as its DHT, its functionality will be briefly outlined next. For more detailed information, please consult the accompanied references.

title

Kademlia

Kademlia is a P2P DHT with XOR-based metric topology [7]. It combines provable consistency and performance, latency-minimising routing, and a symmetric, unidirectional topology. It minimises number of configuration messages nodes must send to learn about

each other, and configuration information spreads automatically as a side-effect of key lookups. Furthermore, nodes have enough knowledge and flexibility to route queries through low-latency paths. Kademlia contacts only O(log n) nodes while searching the system containing n nodes. Keys are 160-bit opaque quantities of which each participating computer is assigned one node ID in the 160-bit key space. Key-value pairs are stored on nodes with IDs "close" to the 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). A node's ID is a large random number intended to achieve uniqueness. 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 leaves 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 hight 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 by its ID. A Kademlia node stores contact information about each other to route query messages for nodes of distance 2^i and 2^{i+1} (called k-buckets). k-buckets are sorted by time last seen (least-recently (head) to most-recently (tail) Appropriate k-buckets are updated whenever a Kademlia node receives any message (request or reply) from another node.

TomP2P

The following descriptions are based on [16] and [1]. TomP2P implements, among others, a structured DHT and unstructured broadcasts. These are the main facilities that are used in the present implementation. Additional to the common put(key, value) and get(key): value methods, extended DHT operations that allow storing multiple keys for the same value (add(key, value), mimicking Multimap-behaviour) or the same key/value pairs by distinguishing different *domains*. Overall, there are 4 separate keys to distinguish a same value v: location key (the actual k in e.g. put(k, v)), domain key, the content key, and the version key. All domain, content, and version keys are zero if not specified per default. Of these keys, the presented implementation will only make use of location and domain key as these are all that is needed. All keys are of type Number160 in compliance with the aforementioned 160-bit key space of Kademlia, which TomP2P uses for routing. Keys are stored on the nodes with an ID closest to that key. TomP2P makes heavy use of futures to reduce blocking in systems. As each method call will immediately return, the corresponding futures need to implement a listener that specifies what shall happen it is called. Futures may also be cancelled and will, thus, not finish their intended use. Data replication factor is 6 per default.

How to connect peers How broadcast works Kademlia routing explained with iterative routing... References to slides...

3.3 Prototype

3.3.1 Discarded Initial Prototypes

After two failed attempts of implementing a working prototype (see appendix X), it was decided to redesign the prototype from scratch. Initially, a similar design concept as presented in [13] 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 implementing the prototype, it became more and more apparent that, to keep it as generic as possible, too many unknowns had to be guessed by the system. Furthermore, over-engineering lead to the fact that the prototype was not possible to test any more, and problematic assumptions about how to aggregate the data lead to a large number of DHT access calls which slowed down the whole system to an extent that it was not feasible for the execution of MapReduce jobs anymore. Therefore, it was discarded eventually and a new, simpler prototype, which will be presented shortly, was instead implemented. However, due to the pressing time, this last prototype only contained the bare minimum to proof a concept, and has, thus, much improvement and possible redesign to be done.

3.3.2 Final Prototype

To accommodate the fact that most of the knowledge about the data to process and the intended outcome of a MapReduce job lies with the user, the final prototype gives much more freedom of choice for how the data should be processed. However, this also requires the user to be familiar with the TomP2P framework. The prototype mainly provides a number of helper classes and interfaces to help the user implement a MapReduce job. In the next subsections, these helper classes and interfaces are outlined for users to get a better feeling of how to implement a MapReduce job on their own.

Rationale and Mode of Operation

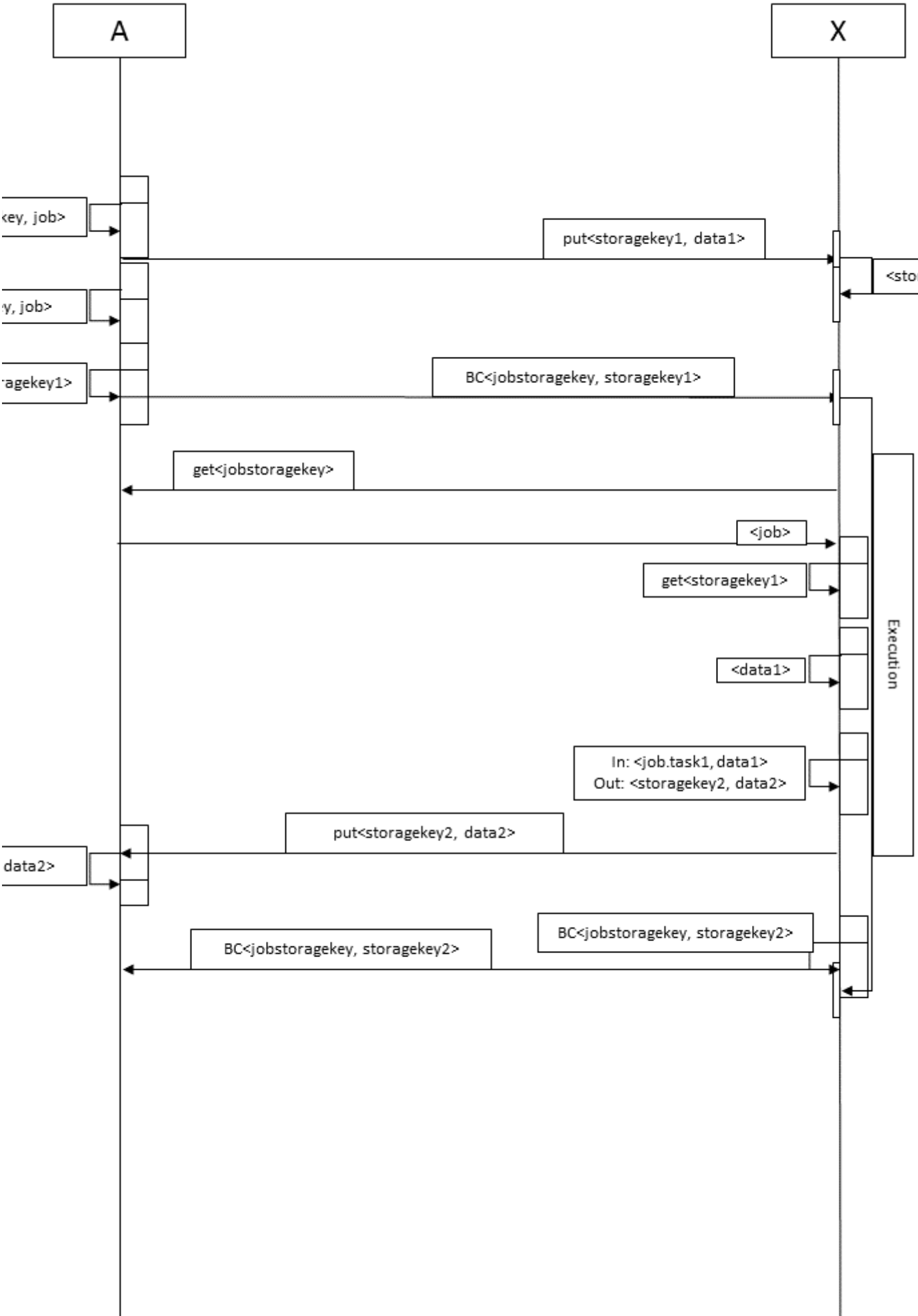
TODO
What remained from the initial prototypes is the conclusion that, to keep implementation simple and as generic as possible, there needs to be an abstraction for *both* map and reduce functions.
The basic idea is that the user defines a set of tasks to be executed in a certain order, such that task i+1 is executed after task i on the data produced by task i. How these tasks and data sets are divided is up to the user who defines the tasks. Thus, the user needs to take care of most of the implementation and the framework only provides certain helper functions. The basic interaction relies on DHT put/get functions and broadcasts. Put and get are used to distribute a task's data and results to and from the DHT, while broadcasts are exclusively used to inform other nodes about completed tasks, which may initiate the receiving node to start the next task defined in the broadcast message. All tasks are summarised in a job and the first task to be executed is intended to be a local

task that emits the data to the DHT. The first locally distributed broadcast will then cause other nodes to participate in the execution according to a user's specifications. A simple illustration of the main mode of operation is given in Figure 3.2. In this simplified view, there are two participating nodes, which are connected to the DHT. The fact that data is replicated on multiple nodes is discarded in this example. Node A is the one emitting a job and one data item and node B the one that will execute the first task of the job. Node A uses put(jobstoragekey, job) to store the job in the DHT. Depending on the jobstoragekey, the node with the closest ID to this key will receive and store it (see the description of Kademlia). In the illustration, the jobstoragekey was stored on node A. On the other hand, the data item is stored with put(storagekey1, data), which leads to the data being stored on node B. Once storing both job and data succeeds, node A emits a broadcast, which, in the ideal case, all connected nodes (A and B) receive. In this example, node A will not do anything on receiving a broadcast. On the other hand, node B will start execution of the first task of the job on the stored data. To do so, it will first retrieve the stored job from the DHT using the get(jobstoragekey) function. Jobstoragekey can directly be taken from the broadcast. In the example, the job is stored on node A, thus, it takes some time for the job to arrive at node B before execution may start. Secondly, after receiving the job (which may be stored on the execution node in case there are multiple executions needed in that job for performance reasons), the data for the next task needs to be retrieved. By using the same get(storagekey1) function as before, node B is able to retrieve data1 from the DHT, which in this example is also stored on node B. Once both job and data are received, node B will determine which task of the job needs to be executed. Here, as it is the starting point, node B executes task1 on data1. During execution, multiple new data items may be produced. For simplicity, only one data item data2 is created, which node B again stores in the DHT using a new storagekey2 and by invoking the DHT's put function. Again, the storage key determines on which node the data is going to be stored. Thus, in the case of data2, it is stored on node A. Once the execution completed on node B, a broadcast with the new storage key storagekey2 is emitted, such that all participating nodes receive it and according to the user's implementation, may or may not execute the next task for this storage key. Again, in the example, the jobstoragekey is sent as well but this is up to the user to be defined.

Reliability considerations: what may go wrong and how to avoid it

Task

The main extension point for a user to implement a MapReduce job is the abstract Task class (Figure 3.3). It provides only one method broadcastReceiver(input, dht) for the user to be extended. An extension could be a map or reduce function or any other included procedure to be carried out before or after the actual MapReduce job, like reading, writing, or sorting data items according to a users needs. Thus, Task is the embodiment of the idea of having only one abstraction for all procedures to be carried out during a MapReduce job. Two parameters need to be provided to the broadcastReceiver method: a NavigableMap<Number640, Data> that contains the user-specified data for this Task, and an instance of DHTWrapper. DHTWrapper provides the user with methods


A
X
key, job>
put<storagekey1, data1>
<storageke
y, job>
agekey1>
BC<jobstoragekey, storagekey1>
get<jobstoragekey>
<job>
get<storagekey1>
<data1>
Execution
In: <job.task1, data1>
Out: <storagekey2, data2>
put<storagekey2, data2>
data2>
BC<jobstoragekey, storagekey2>
BC<jobstoragekey, storagekey2>

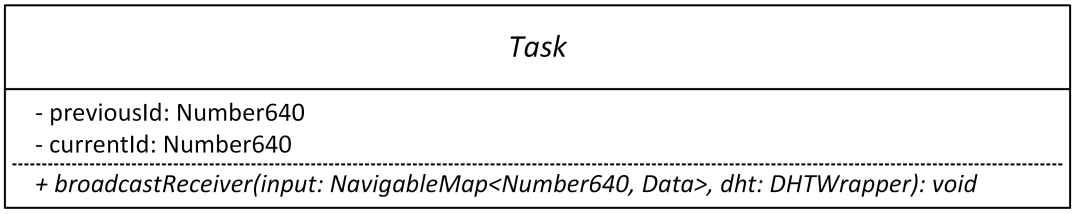


Figure 3.3: The Task class - the main extension point for implementing MapReduce jobs

to connect and disconnect, put, add, and get data from and to the DHT, and allows for direct emission of broadcasts to all the other nodes. The user-specified data may be anything that this Task needs to execute its job. In the case of a simple WordCount, the initial Task may be to read the data from the user's file system into the DHT. In such a case, this map would contain e.g. the path to the location on the file system, and the Task then would read the data, split it according to the user's specification, and uses the provided DHTWrapper to put the data into the DHT. A last step then is to send a broadcast out, such that other peers connected to the DHT get informed. The broadcast should only be emitted once the task truly finished. Although it is up to the programmer to emit half finished tasks, the reliability increases if the broadcast is only sent when the Task finished completely. If the user, however, knows that the equipment the MapReduce job should run on is reliable enough, Task may be extended to accommodate any performance-increasing measure the user may think of. Thus, the simple interface is actually very powerful for skilled users. Figure 3.3 also shows that a Task has two Number640 members, 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 initial Task, the previousId is intended to be null, such that the start of a procedure execution may be determined by the system.

Job

To define and start a MapReduce job, there exists one class simply called Job that allows the user to add defined tasks and start the job. Several helper methods like findStartTask or findTask according to the Task's id (see currentId in Figure 3.3) are provided as well as the actual start method. The start method takes the same input parameters as the Task's broadcastReceiver method. The intention behind this is that the start method simply looks for the first Task in its Task list (specified by the previousId being null as stated before) and then calls broadcastReceiver with these input parameters on the found first Task. Thus, the first task is a locally executed procedure. In the WordCount example, this task may read the data from the local file system and send it out to the DHT. A broadcast call will then initiate the first non-local procedure, where other peers connected to the DHT may execute the user-specified tasks. Very important are the two method serialize() and deserialize() provided by the Job class. As peers connected on other nodes may not know of the actual Task implementation provided by the user, they need to be serialised and transferred to these nodes. To do so, Job uses the helper class SerializeUtils, which provides two different sets of methods for serialising and deserialising both needed class files and actual instantiated Java objects defined by the user: serialize-

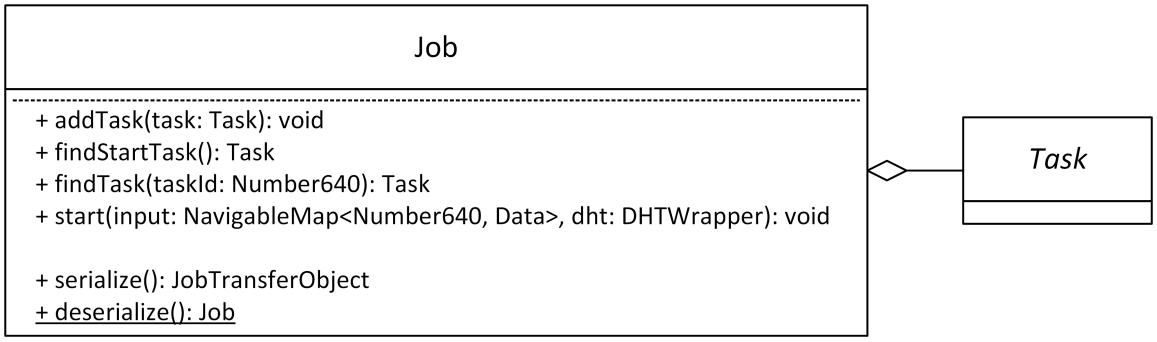


Figure 3.4: The Job class - starting point of any MapReduce job

ClassFile/deserializeClassFile turn a class specified by its Class<?> into a byte[] or back to their respective Class<?>, respectively. This allows for the user-defined Task class(es) to be transferred and instantiated on another node across the network without the other node having to know about the .class file(s). The other set of methods provides by SerializeUtils are serializeJavaObject/deserializeJavaObject. While serializeJavaObject turns an instance of the specified class into a byte[], deserializeJavaObject will convert this byte[] back to the object on another node across the network. However, deserializeJavaObject needs the beforehand serialized class files to reliably convert the object back on another node as the class may be unknown when the node receives it at runtime. Thus, SerializeUtils is one of the most important class for allowing the transfer and instantiation of beforehand unknown classes (like user-defined Tasks) on different nodes. Job.serialize() and Job.deserialize() both use two objects to store the serialized data: JobTransferObject and TransferObject. Both classes are simply used for transfer the specified data and contain the byte[] for class files and objects to be instantiated on the node that retrieve the job.

MapReduceBroadcastHandler

Another important class that has to be used in a MapReduce job is an instantiation of MapReduceBroadcastHandler. Although it is up to the user to define how this broadcast handler is implemented, the execution of a MapReduce job relies on the implementation of its super class' receive(message) method. This method is inherited from StructuredBroadcastHandler and makes sure that all nodes in the network are informed when a new message (e.g. a completed Task) is received. The idea is that, when a new job is submitted to the DHT, the broadcast handler is the instance to retrieve the job and execute the next Task in the MapReduce job for which this message was received. As an example, if the user specified that the map task should emit all words and ones for a file, the broadcast handler will then look for the corresponding next Task to execute (which will count all the ones for that file). Thus, the broadcast handler will, similar to the Job's start method, find the next Task and invoke its broadcastReceiver(input, dht) method. Thus, where Job.start represents the *local* starting point of a MapReduce job, MapReduceBroadcastHandler.receive(message) is the *remote* starting point for all Tasks afterwards.

Chapter 4

Evaluation

- 2 jobs –> Execution time - Disconnecting nodes –> effects - Throughput/messaging

Chapter 5

Summary and Conclusions

Bibliography

[1] Autoren: Titel, Verlag, `http://...`, Datum.

Abbreviations

AAA	Authentication, Authorization, and Accounting

Glossary

Authentication

Authorization Authorization is the decision whether an entity is allowed to perform a particular action or not, e.g. whether a user is allowed to attach to a network or not.

Accounting

List of Figures

List of Tables

Appendix A

Installation Guidelines

Appendix B

Contents of the CD

Bibliography

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

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

[3] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott 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.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51(1):107, jan 2008.

[5] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe 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.

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

[7] Petar Maymounkov and David 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.

[8] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[9] R. Schollmeier. [16] 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–, Washington, DC, USA, 2001. IEEE Computer Society.

[10] Luiz Angelo Steffenel. First Steps on the Development of a P2P Middleware for Map-Reduce. Technical report, 2013.

[11] Luiz Angelo Steffenel, Olivier Flauzac, Andrea Schwertner Charao, Patricia Pitthan Barcelos, Benhur Stein, Sergio Nesmachnow, Manuele Kirsch Pinheiro, and Daniel 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.

[12] Luiz Angelo Steffenel, Olivier Flauzac, Andrea CharÃ£o, Benhur Stein, Patricia Pitthan Barcellos, Sergio Nesmachnow, Manuele Kirsch-Pinheiro, and Daniel Diaz. Per-mare: Adaptive deployment of mapreduce over pervasive grids. In *8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, Compiègne, France, October 2013.

[13] Luiz Angelo Steffenel and Manuele Kirch Pinheiro. CloudFIT , a PaaS platform for IoT applications over Pervasive Networks. 2015.

[14] Luiz Angelo Steffenel and Manuele Kirch Pinheiro. Leveraging Data Intensive Applications on a Pervasive Computing Platform: The Case of MapReduce. *Procedia Computer Science*, 52:1034–1039, 2015.

[15] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari 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.

[16] K. Wehrle, S. GÃ¶tz, S. Rieche, and T. Bocek. Lecture notes in overlay networks, decentralized systems and their applications, March 2014.

[17] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.