Scaling a Cloud Storage System
Autonomously

Master Thesis
Flavio Pfaffhauser

Supervised by:
Prof. Dr. Donald Kossmann
Dr. Tim Kraska

Systems Group, http://systems.ethz.ch/
Department of Computer Science
ETH Zurich

November, 2009 - March, 2010
Abstract

Data services in the cloud promise scalability and availability at a low cost. To achieve this goal, a variety of architectures have been proposed, such as Dynamo, BigTable or PNUTS. These architectures were copied in a number of open-source systems, for instance Cassandra, Voldemort or HBase. However, none of the systems allows for easily exchanging parts of the implementation, such as the replication protocol. Furthermore, none of the systems supports complex queries and transactions.

In this thesis we make a first step towards developing a modular cloud storage system, so that parts of it can be exchanged easily in order to support different workloads. As part of this modular architecture, the load balancing and cloud bursting component have been added to the new cloud storage system. Load balancing is the mechanism that spreads the load of the system evenly across all physical machines. Cloud bursting refers to the technique of automatically adapting the number of physical machines to the load of the system. Both mechanisms enable data services in the cloud to autonomously scale up and down while load increases or decreases.
Acknowledgements

First of all I would like to thank my colleagues Stephan Merkli and Raman Mittal. I really enjoyed working together with both of you. We were able to achieve a lot during the course of our master theses: we built an available and scalable cloud storage system that allows for easily exchanging its components. Furthermore, I would like to thank Dr. Tim Kraska, Simon Loesing and Prof. Donald Kossmann for their support and advice. Tim, my main advisor, cared a lot about the outcome of my master thesis and invested a lot of time in mentoring me. Thank you for your support.
# Contents

1 Introduction .................................................. 1
   1.1 Motivation and Problem Statement ....................... 1
   1.2 Contributions ........................................... 2
   1.3 Structure of thesis ...................................... 3

2 Cloudy2 Architecture ........................................ 5
   2.1 Overview .................................................. 5
   2.2 External Interface ........................................ 6
   2.3 Components .............................................. 6
   2.4 Services .................................................. 10
   2.5 Data Model (DPI) ......................................... 11
   2.6 Example Workflow ........................................ 13

3 Router .......................................................... 15
   3.1 Overview .................................................. 15
   3.2 Distributed Hash Table ................................... 16
      3.2.1 Hashing Strategy .................................... 18
      3.2.2 Partitioning Strategy ............................... 18

4 Protocol ....................................................... 19
   4.1 Overview .................................................. 19
   4.2 Simple Protocol .......................................... 20
   4.3 Quorum Protocol .......................................... 20
      4.3.1 Quorum Response Handler ........................... 21
      4.3.2 Normal Case Behavior ............................... 23
      4.3.3 Failure Case Behavior ............................... 24

5 Load Balancing ............................................... 27
   5.1 Overview .................................................. 27
   5.2 Load Calculator .......................................... 28
      5.2.1 Calculate load using DPI distribution ............... 28
      5.2.2 Calculate load using different measures ............. 28
   5.3 Load Balancer ............................................ 29
      5.3.1 Simple Load Balancer ................................. 29
      5.3.2 Maha Load Balancer ................................ 29
   5.4 Repartitioner ............................................. 31
      5.4.1 Distributed Hash Table Repartitioner ................. 31
      5.4.2 Failure Case Behavior ............................... 35
6 Cloud Bursting and Collapsing 37
  6.1 Overview ......................................................... 37
  6.2 Cloud Bursting Methods ........................................... 38
      6.2.1 Random Cloud Bursting ................................. 39
      6.2.2 Hotspot Cloud Bursting ................................. 39
  6.3 Cloud Bursting Executor ......................................... 39

7 Benchmarks 41
  7.1 Cloudy2 System Performance ................................. 41
      7.1.1 Read and Write Performance ............................ 42
      7.1.2 Client-driven versus Server-driven Routing .......... 44
      7.1.3 BDB primary index slow down ............................ 45
  7.2 Load Balancing Experiment ................................. 46
      7.2.1 Best Case Scenario ...................................... 47
      7.2.2 Worst Case Scenario .................................... 47
      7.2.3 Comparison ................................................. 48
  7.3 Cloud Bursting Experiments .................................. 49

8 Conclusion 51
  8.1 Summary .......................................................... 51
  8.2 Future Work ...................................................... 52
Chapter 1

Introduction

This chapter is an introduction to the topic of data storage systems in the cloud. It motivates our work and states the problem we want to solve. Finally, it describes the main contribution of our work: a modular data storage system in the cloud supporting different workloads.

1.1 Motivation and Problem Statement

Cloud Computing is a new paradigm that is based on the idea of providing virtually unlimited computing resources. These resources can be allocated and deallocated on-the-fly and users only pay for the resources they actually consume. This enables engineers to build highly scalable, highly available and fault-tolerant data storage services at a low cost. Such data storage services - which are designed as distributed systems from the beginning - allow data to be replicated among multiple data centers.

Many such large scale data storage systems have been built, for instance Dynamo [1], BigTable [2], Cassandra [3], Project Voldemort [4] or PNUTS [5]. Although the variety of different cloud storage systems is huge, it is still likely that some user’s needs are not perfectly covered by a single system. For example, if one would like to have the data model of HBase but the availability of Cassandra, then a completely new system needs to be built. Today’s systems are over-customized and force people to either adapt the application to the storage system or to start a new project. This also explains the huge variety of systems even in the presence of their complexity. Furthermore, by deciding for a particular setup, users narrow down the focus of their application (e.g., either analytical or transactional) and restrict the ability of the application to evolve over time.

To overcome this cloud storage system jungle, we started to build Cloudy2, a modular cloud storage system, that can be tailored and modified to support many different workloads:

- **Key-value store**: Storing keys associated with a value through a simple interface. This is usually needed in web services where data is stored and retrieved mostly by primary key and thus complex querying is not required. This use case is supported by most open-source implementations that are available.
• **Relational store:** Storing relational data for online transactional and analytical processing. This use case is required if more functionality is needed than provided by the key-value store. The relational store allows queries to be formulated using relational algebra. BigTable [2] and HBase [6] target a similar use case. Raman Mittal explores the relational store use case in his master thesis [7].

• **Stream processing:** Processing streams in the cloud. There are many stream processing systems available (e.g. Borealis [8]), but none of them is able to scale out. Stephan Merkli explores the streaming use case in his master thesis [9].

• **Storing graphs:** Storing graph structures, for instance social networks. Such applications become more and more popular, but the data storage tools are not optimized for that type of data.

• **Hosting applications:** Hosting applications in the cloud. In principle, hosting an application is nothing else but deploying it in a storage system that makes it highly available to its users.

We believe that all these use cases can be supported by a single cloud storage system that allows for exchanging parts of its implementation easily. For example, to support a stream processing workload, a different protocol and store are needed. We define one single data model called DPI for all these use cases (explained in detail in Section 2.5).

The particular focus of this thesis is load balancing and cloud bursting. Not one of the above mentioned open-source data storage systems supports both, reasonable cloud bursting and load balancing. Both mechanisms playing together enable a cloud storage system to scale out autonomously. The goal is not only to utilize every machine participating in the cloud in an optimal way, but also to minimize data movement between different machines as scaling out. One trend these days is to only pay for the computing and storage capacity that is actually consumed. This motivates our work on mechanisms (i.e. cloud bursting and load balancing) that make this methodology and business model possible. We believe that these mechanisms have to be an integral part of a data storage system in the cloud.

### 1.2 Contributions

The main contribution of this thesis is a modular data storage system in the cloud supporting different workloads. The modularity of the system allows to adapt it to be either a key-value store, a relational store or a stream processing system. Furthermore, load balancing and cloud bursting components have been added to the system. These components give the system the ability to scale up and down autonomously according to different workloads. The benchmarks in Chapter 7 prove the effectiveness of the system and illustrate its load balancing and cloud bursting mechanisms.
1.3 Structure of thesis

The remainder of this thesis is organized as follows:

- **Chapter 2** describes the architecture of Cloudy2, the first cloud storage system that allows for adaptability besides availability and scalability.

- **Chapter 3** describes the routing component of Cloudy2. The router is responsible for locating data in the system.

- **Chapter 4** describes the protocol of Cloudy2. It defines the replication and consistency guarantees of Cloudy2.

- **Chapter 5** describes the load balancing component of Cloudy2, which is responsible for spreading the load evenly across all the machines in the system.

- **Chapter 6** describes the cloud bursting component of Cloudy2, which is responsible for adding or removing machines from the system depending on the overall load. It is the component that makes Cloudy2 “elastic” and scale autonomously.

- **Chapter 7** contains the benchmarks of the Cloudy2 system performance, load balancing and cloud bursting.

- **Chapter 8** presents the conclusions of this thesis and lists topics for future work.
Chapter 2

Cloudy2 Architecture

This chapter introduces Cloudy2, a modular cloud storage system. Cloudy2 provides a highly flexible architecture for distributed data storage and is designed to support different workloads. Based on a generic data model, Cloudy2 can be customized to meet different application requirements. The overview section outlines the big picture of the Cloudy2 architecture, then the following sections describe the system design in more detail.

2.1 Overview

Cloudy2 is our implementation of a modular cloud storage system that allows for exchanging parts of it easily. Cloudy2’s main components are the external interface, the protocol, the router and the store.

![Diagram of Cloudy2 Architecture]

*Figure 2.1: Overview of the Cloudy2 architecture. Components are parts of the system that are exchangeable, services are not exchangeable. System parts highlighted in gray are explained in detail in this thesis.*
The implementation of all these main components plus the components shown on the right side of Figure 2.1 can be exchanged because they have been implemented using interfaces. The services in Cloudy2 are not exchangeable because they are not expected to be often individually configured by the user. One implementation of each of them should be sufficient.

The focus of this master thesis is cloud bursting (Chapter 6) and load balancing (Chapter 5). The router (Chapter 3) and the protocol (Chapter 4) components are described in detail because they play an important role in Cloudy2. The rest of the system is described in the remainder of this chapter. For more details on the store, the external interface, the messaging component and the gossiper please refer to the master theses of Stephan Merkli (Streaming in the Cloud [9]) and Raman Mittal (Query Processing in the Cloud [7]).

2.2 External Interface

Cloudy2 offers different interfaces to interact with it. The data model of Cloudy2, called DPI, is the form in which data moves through the system. Before interacting with Cloudy2 the client has to create a DPI which consists mainly of a key and a value. The DPI is described in detail in Section 2.5. All of the interfaces support the get, put and delete methods:

- void put(DPI)
- Set<DPI> get(DPI)
- void delete(DPI)

The following interfaces are implemented at the moment:

- **Java:** There are two clients written in Java available for interacting with the system. One of them uses server-driven routing and the other client-driven routing. The benchmarks described in Chapter 7 show that using client-driven routing is much faster.

- **REST:** The REST interface is based on HTTP and makes it possible to interact with Cloudy2 using HTTP requests.

- **MySQL:** The MySQL interface enables the client to use the standard MySQL [10] shell to interact with Cloudy2.

- **Thrift:** The Thrift interface was introduced due to the need of cross-language support when introducing the MySQL interface. MySQL is written in C/C++ and Cloudy2 is developed completely in Java. Apache Thrift [11] was used to close this gap.

2.3 Components

The components are the exchangeable parts of Cloudy2. Exchangeable means that a component can easily be replaced by another without breaking the system. To make that possible, each component has a clean interface and a specific task to fulfill. Components can only be replaced statically, before starting the system.
In a cluster, all endpoints (i.e. physical machines) running Cloudy2 must have the same component configuration, otherwise the behavior of the system is undefined. Different implementations of the same component may not be consistent with each other (e.g. the protocol sends messages that are not understood by another protocol).

All components are described in this section and for each of them there exists at least one working implementation. The development of Cloudy2 started with only a few main components (messaging system, router, store, protocol, gossiper and external interface). The current version of Cloudy2 is made up of about a dozen components. On one hand new components were created to reuse existing parts and to prevent duplication of code (e.g. load calculator). On the other hand running Cloudy2 on Amazon Web Services [12] and keeping the architecture modular at the same time required the creation of new components (e.g. bootstrapper, seed preparator).

### Protocol

The protocol component is the layer between the external interface and the rest of the system. It supports the three types of request, i.e. get, put and delete. It defines replication and consistency guarantees for the system. Chapter 4 describes the protocol in detail.

### Router

The router stores the routing table of Cloudy2 and gives other components access to it. The routing table saves which endpoint is responsible for which DPIs. In case the replication factor in the system is greater than one there can even be multiple endpoints that are responsible for a single DPI. These endpoints are returned in the form of a preference list, e.g. the first endpoint in the list is the main responsible endpoint, the other endpoints are the replicas. The router itself does not route, it just provides the necessary information to other components. Chapter 3 describes the router in detail.

### Store

The store is the storage layer of Cloudy2. The methods provided by the storage interface (get, put and delete) are invoked by the protocol. The store is described in detail in Raman Mittal's master thesis [7]. Currently there are two implementations of the store:

- **In-Memory Store:** The in-memory store is non-persistent and its implementation is based on a hash map.

- **Berkeley DB Store:** The Berkeley DB store [13] offers persistence and even indexing. Its source code is available as open-source from Oracle.

### Messaging

The messaging system sends messages between two different endpoints in Cloudy2. Endpoints have to register handlers for each message type so they can handle and
process messages of that type upon reception. Messages can be sent either reliably by using TCP connections or in a fire-and-forget style by using a UDP connection. Cloudy2 uses the messaging system of Apache Cassandra [3], version 0.5.0.

**Gossiper**

The gossiper is responsible for synchronizing the state information of every endpoint with all other endpoints. Cloudy2 uses a gossip-style protocol to propagate the state information of every endpoint to all other endpoints. Each endpoint has its own application state and every service or component can add its own state to the application state (e.g. the router adds the routing table, the statistics adds statistical information). The gossiper runs as a timer task on every Cloudy2 endpoint and periodically (every second) gossips the application state of its known endpoints to a randomly selected endpoint, to a seed (a seed is a designated endpoint that is allowed to bootstrap new endpoints) and to an unreachable endpoint. The implementation of the gossiper is based on the source code of Apache Cassandra [3].

Failure detection is included in the gossiper to detect endpoint failures. The implementation of the failure detector is based on the Phi Accrual Failure Detector [14]. The gossiper maintains lists of alive, unreachable and dead endpoints. If an endpoint is suspected to no longer be alive, the gossiper removes that endpoint from the list of alive endpoints and adds it to the list of unreachable endpoints. Alive endpoints are the ones for which the current endpoint is actively getting higher versions of the state information. An unreachable endpoint which does not respond or send its state information for more than certain threshold, is declared as dead.

**Leader Election**

The task of the leader election component is to acquire and release exclusive write locks. An identifier (i.e. a sequence of characters) determines the membership to a specific group, meaning that for one identifier the leader election component grants an exclusive write lock to one endpoint. Other endpoints requesting the lock have to wait for the release of the lock by the owner. If an endpoint currently holding the lock terminates (manually or due to a failure), the lock is automatically released and another endpoint waiting for the lock is the new owner and is informed thereof. Cloudy2 is using Apache ZooKeeper [15] for leader election. ZooKeeper uses TCP connections to determine when a node fails and it has to release a lock or assign it to another endpoint. The server instances of ZooKeeper are running replicated on several nodes in the cluster. They are separate system processes started by Cloudy2. If several endpoints try to acquire the lock at the same time the arrival order of their requests determines the order of lock ownership. The first endpoint holds the lock, the others have to wait. If the first endpoint crashes or releases the lock manually the next one will get the exclusive write lock.
2.3 Components

Bootstrap

The bootstrapper is used to bootstrap a new Cloudy2 endpoint. Whenever a new endpoint joins the system it invokes the bootstrapping method of the bootstrapper. The bootstrapper then acquires the bootstrap information from a Cloudy2 seed. A seed is a designated endpoint that is allowed to bootstrap new endpoints. The content of the bootstrap information depends on the implementation of the router (e.g. in case of a DHT, the bootstrapping endpoint is provided with a token). If the endpoint is unable to contact any of the seeds, it will fail to bootstrap. Currently two bootstrappers are implemented:

- **Local Bootstrapper**: Bootstraps endpoints in a local cluster.
- **EC2 Bootstrapper**: Bootstraps endpoints running on EC2 instances on Amazon Web Services [12].

Load Calculator

The load calculator calculates the load of the Cloudy2 system based on different factors (e.g. memory usage, disk usage, CPU usage and others). This load information is used by the load balancing and cloud bursting components to decide on how to balance the load in the system. The load calculator is described in detail in Section 5.2.

Load Balancer

The load balancer is responsible for distributing the load evenly among all endpoints in the Cloudy2 system. It is scheduled periodically (every two minutes) on every endpoint. Based on available load information provided by the load calculator it decides which load balancing step should be executed by the repartitioner. The load balancer is described in detail in Chapter 5.

Repartitioner

The repartitioner is invoked by the load balancer to execute load balancing steps. It will move the data from overloaded endpoints to underloaded endpoints. To do this the repartitioner needs to access the routing table in the router. Hence, its implementation is tightly connected to the implementation of the router. The repartitioner is described in detail in Section 5.4.

Cloud Burster

The cloud burster is the component that makes Cloudy2 elastic. It is a leader-driven task that is executed periodically (every five minutes) to check whether the overall load of the system is over a defined maximum or below a defined minimum load. In the first case cloud bursting would be triggered which would add a new endpoint to the system. In the latter case cloud collapsing would be triggered which would remove one endpoint from the system. The leader election component is used to elect a leader which executes the cloud bursting step. The cloud burster is described in detail in chapter 6.
Cloud Burst Executor

The cloudburst executor is instructed by the cloud burster to either start a new Cloudy2 endpoint or terminate an existing endpoint. The cloud burst executor is described in detail in Chapter 6.

2.4 Services

The Cloudy2 services are related to components but they are not exchangeable. They are inherent parts of Cloudy2 and empower the components to do their work. The reason why the services are not exchangeable is that the services are highly tied to the implementation of some lower-level system mechanisms and that they contain functionality that we believe does not need to be exchangeable.

Meta Data Handler

The meta data handler is responsible for synchronizing the meta data between the endpoints in the system. The meta data is stored as data within Cloudy2 and has a version number. The current version number of the meta data is transferred with each message sent between two endpoints. If an endpoint receives a message containing a higher meta data version number than it currently has, it immediately fetches the new meta data from Cloudy2. The meta data service is heavily used in the thesis of Raman Mittal and Stephan Merkli but not of great importance in this thesis.

Statistics

The statistics service collects system information (e.g. main memory usage, CPU usage and disk usage) and propagates this information using the gossiper. Therefore the statistics are not only providing information about the local endpoint, but include the values collected on remote endpoints too. The statistics are mainly accessed by the load calculator.

Version Control

Each DPI (i.e. the data model of Cloudy2, described in detail in Section 2.5) contains a field for the version. This version field is not a field like the others that can be set by the user. Instead, the protocol is in charge of setting the version for each DPI before processing it. Thereby it has two possibilities: it can either set a timestamp or it can set a version number. Timestamps are set according to the system clock, version numbers are maintained by the protocol itself (e.g. incrementing).

To increase the precision of the first mechanism, which is working with timestamps, each Cloudy2 instance maintains a clock skew table. This table contains estimations of the clock skews between the different endpoints in the cluster. Before comparing the versions, the timestamps are adjusted according to the known clock skew between the two involved endpoints. This clock skew table does not solve the problem of having a global clock in a distributed system nor does it give any boundaries on the clock skew as more sophisticated algorithms do [16],
but it is an approach to gain more precision for timestamps without having a measurable negative impact on the system performance.

It is important to note that Cloudy2 does not support versioned objects. Therefore there is always only one version (i.e. the latest one) of each DPI stored. The purpose of the version is to reject requests that have an older version than the one that is currently stored. The comparison of versions is performed for put and delete requests only. For get requests there is no reason to do that because get requests do not change the state of the store.

Data Streamer

After bootstrapping or load balancing steps, data has to be moved from one endpoint to another. This task is fulfilled by the data streamer. Moving data could always be done by sending requests through the protocol. But when a new endpoint bootstraps, a large amount of data has to be moved. Letting the protocol move such a large amount of data is not efficient and would clog the system. Because moving data is a regular, often occurring process in a highly loaded system we developed the data streamer. The data streamer opens a TCP connection between the the data sender and receiver and streams the data directly from one store to the other store without additional overhead by the messaging system or other components.

2.5 Data Model (DPI)

The DPI is the data model used in Cloudy2. It has two functionalities at the same time: It describes the data but might as well contain the data. It is used as a way of data exchange between the external interface and the internal components. We believe that the DPI is the right data model for all the described use cases in the introduction. Uniqueness is defined by the combination of the key and type field. Below you can find the BNF notation for the DPI:

\[
\text{DPI} = \langle \text{<" KEY "," TYPE "," VALUE "," LIFETIME ">} \rangle
\]

\[
\text{KEY} = \text{STRING} \mid \text{RANGE}
\]

\[
\text{RANGE} = ( \langle [" | (\text{"}) \text{STRING "," STRING (") | "]" \rangle \)
\]

\[
\text{TYPE} = \text{DOMAIN} \[ ":" \text{ TABLE [ ":" \text{ ATTRIBUTE ]]}
\]

\[
\text{LIFETIME} = \text{"0" | ... | "INF"
\]

\[
\text{VALUE} = \text{anything}
\]

\[
\text{DOMAIN} = \text{STRING}
\]

\[
\text{TABLE} = \text{STRING}
\]

\[
\text{ATTRIBUTE} = \text{STRING}
\]

\[
\text{STRING} = \text{any sequence of characters}
\]

In the remainder of this section each field of the DPI is described in detail:

**Key/Range**

The key and range have a quite complex syntax and semantic, because they can identify either a single DPI, denoted by a string identifier, or a range of DPIs. It is possible to either set the key or the range, but not both at the same time. A range is represented by a start and an end key. Both of them are string identifiers. All DPIs between these two specified keys
belong to this range (i.e. are returned in a get request / removed in a delete request). The range is represented as in its corresponding mathematical notation: ( or ) denotes exclusive starts or ends, [ or ] denotes inclusive starts or ends. The comparison of keys is based on their lexicographical order. The range (A,G] contains keys like A, B and G, but does not contain A or GA. By supporting ranges, we make range operations possible.

**Type**  
The type is a string that describes the domain associated with a DPI. A domain could be a generic class or a namespace to which the key of the DPI belongs to. For example, in the MySQL use case it could be a database name, table name or a combination of both.

**Lifetime**  
The lifetime can be defined between 0 and ∞. The lifetime defines how long a DPI lives. If one wants to store data only for a short time a smaller lifetime can be used. One can as well define a query that is continuously running for the period of the lifetime. This field is currently not used.

**Value**  
The value field contains the data as a binary object.

**Version**  
The version field is used to ensure consistency in the system and is only maintained by the system itself. It is not externally accessible. More information about how version control works in Cloudy2 can be found in Section 2.4.

**Hint**  
The hint field is used to transport additional information with the DPI, to for example, optimize certain queries. The hint is maintained by the system itself and not externally accessible.

As mentioned above, the fields version and hint are used internally and the field lifetime is not yet fully supported. The remaining fields key/range, type and value are set by the client. The key/range field has to be set in any request while value field is only required in put requests. the type field is optional.

**Examples**  
The next few examples explain how to use the data model. Several requests are encoded as a DPI in a triplet of the form <key/range,type,value>. To simplify the examples the hint and version fields were left out.

A get request with DPI<[a,d)> will return all DPIs where the keys start with a, b or c (start a is inclusive, end d is exclusive). The list below presents more concrete examples:

- **DPI<a,,"Hello World">**: A DPI containing a key and a value but no type represents a **put request** associating the key a with the value *Hello World*. a serves as unique identifier.
- **DPI<a,,>** : A DPI containing just the key and no type or value represents a **get request**. It could as well be a **delete request**. a serves as unique identifier.
2.6 Example Workflow

- DPI\langle a, d \rangle, > : A DPI containing a range exclusively a to inclusively d represents a range query, which can either get or delete data.

- DPI\langle a, customer, > : A DPI containing a key and a type but no value is a typed get request. It could as well be a typed delete request. The concatenation of type and key (customer:a) serves as unique identifier.

- DPI\langle (a, d \rangle, customer, > : A DPI containing a range with a type but no value is a typed range query, which can either get or delete data.

Throughout this thesis, the term key refers to the key of the DPI and acts as a unique identifier for a DPI. If additionally the type is defined then the unique identifier is the concatenation of key and type: key:type.

2.6 Example Workflow

In this section, an example workflow through the system is shown using concrete implementations of the different components. The following components are used:

- **Router**: Distributed Hash Table (DHT), which is described in detail in Chapter 3

- **Protocol**: Quorum Protocol with a replication factor of three and a read respectively write quorum of two. It is described in detail in Chapter 4.

- **Store**: BDB Store, which is described in Section 2.3.

- **External Interface**: Java External Interface, which is described in Section 2.2.

The step by step guide below and Figure 2.2 show an example workflow for a get request. The numbers in the figure correspond to the numbers in the step by step guide. The example workflow assumes a failure free environment. How failures are handled is explained in Chapter 4.

1. The client issues a get request to the Java external interface using the Java client with client side routing. Client side routing means that the request will be directly sent to the master endpoint for the request DPI. It is the one that is responsible for executing requests for this DPI.

2. The external interface forwards the get request to the quorum protocol.

3. The quorum protocol asks the DHT to provide it with the preference list for the given request DPI. A preference list of size three is requested because the replication factor is set to three. The protocol checks whether this endpoint is responsible for handling the request.

4. The DHT looks up in its routing table which endpoint is responsible for the DPI (Endpoint 8) and which endpoints are the two replicas (Endpoint 1, Endpoint 7) for the request DPI. It compiles all three endpoints into the preference list and sends it back to the quorum protocol.

5. Up to now, all steps were executed locally. Now, after receiving the preference list from the DHT, the quorum protocol sends out the get request to all the endpoints in the preference list using the messaging system.
Figure 2.2: Example workflow of a get request. The step by step guide above the picture explains in detail how the request is executed. The numbers in the picture show the chronological order of execution of the get request.

6. The quorum protocol on each endpoint receives the get request message and accesses the local store to retrieve the request DPI.

7. The store sends a set of DPIs back to the quorum protocol. In the case of a range query, the result set would be of size greater than one. In the case of a single key request, the size of the set is equal to one. If the query would not match any DPI in the store, then an empty result set would be returned.

8. The retrieved set of DPIs is sent back to the quorum protocol of the master endpoint. Because the read quorum is set to two, the master endpoint only waits for two responses to arrive. In this example we assume that Endpoint 1 and Endpoint 8 are the ones that satisfy the read quorum. The quorum protocol then resolves the two result sets participating in the quorum. In the failure-free case, both sets are equal and merged into one final result set.

9. The result set is then sent back to the Java external interface.

10. Finally, the result set is forwarded to the Java Client which completes the get request.
Chapter 3

Router

This chapter explains the routing component of Cloudy2. The router is responsible for routing the data to the physical machines, so called endpoints. It is a central component of Cloudy2, and its implementation can be exchanged thanks to the modular architecture of Cloudy2. In this chapter an implementation based on a distributed hash table is explained in detail.

3.1 Overview

The router holds the routing table of the Cloudy2 system and is responsible for providing these routing information to other components. The routing table is defined as mappings from range DPIs to endpoints. So in the context of the router, the data model of Cloudy2 is also used to specify the DPI ranges an endpoint is responsible for. The router functionality can be provided by different implementations, all of them implementing the following interface:

- `Map<DPI, Endpoint> getRoutingTable();`
- `Set<DPI> getPreferenceList(DPI dpi);`
- `Map<DPI, Endpoint> getDataSources(Set<DPI>);`
- `byte[] getBootstrapInformation(Endpoint endpoint);`

The router is used by the protocol to find all the endpoints that are responsible for a given request DPI. The responsible endpoints are provided to the protocol in the form of a preference list, which contains physically distinct endpoints that are known to be alive. The size of the preference list has to be provided. The router also provides the data sources to the data streamer, when an endpoint needs to stream data. And upon bootstrapping a new node, the router provides the bootstrap information to initialize the new endpoint. These interactions of the router with other components are summarized in Figure 3.1.

Each router on each endpoint in the system holds the complete routing table. This is possible because currently we do not support more than one thousand endpoints, so the routing table can fit into main memory as opposed to for example the Chord project [17] or big file sharing networks. These routing tables on different endpoints are synchronized using the gossiper. The gossiper only
synchronizes the state of the routers in certain intervals. Hence, it tries to approximate synchronization, but always lags a bit behind. This does not render the router incapable of routing. As a request is routed through the system, it sees “fresher” and "fresher“ routing information. Consider the scenario where a request is routed to an endpoint that is not responsible for it anymore. This endpoint detects that it is no longer responsible for this request because it has more up to date routing information. It then redirects the request to the endpoint that it thinks is currently responsible. Such scenarios can only arise if the routing table is modified at runtime as a result of load balancing or cloud bursting activities. It takes a short time to propagate the routing table update to all the endpoints in the system using the gossiping protocol. During this interval, scenarios like the one above can arise. This results in a higher response time because there is an additional hop when performing the routing request. If a client-driven routing is used, then the additional hop is only executed once. After that, the client remembers which endpoint to contact during the next request.

All routers support client and server-driven routing. Client-driven routing is faster because the client retrieves the routing table from the router and can route requests directly to the responsible endpoint. In server-driven routing, the client contacts any endpoint in the system and routing is then executed within the system. This causes additional overhead to complete a request (i.e. redirection and multiple hops).

### 3.2 Distributed Hash Table

The distributed hash table (DHT) implements a router using a skip list as the underlying data structure. It relies on consistent hashing and can use different hashing strategies (see Section 3.2.1) to route a DPI to the responsible endpoint. The output range of the order preserving hash function is treated as a circular space or ring (i.e. the largest hash value wraps around to the smallest hash value). Each endpoint in the system is assigned one or several tokens within this space, which represent the endpoint’s positions on the ring. Each DPI is assigned to an endpoint by hashing its partitioning information (e.g. its key, more details in Section 3.2.2) to yield its position on the ring and then walking the ring clockwise to find the first endpoint with a position larger or equal than the DPI's position.

**Figure 3.1:** The router offers its service to other components. It provides bootstrap information to the bootstrapper, preference lists to the protocol and data sources to the data streamer.
3.2 Distributed Hash Table

Figure 3.2: Example set up of a DHT with several endpoints and tokens ($T_i$). Some endpoints are assigned more than one token. For example Endpoint C is responsible for $T_2$ and $T_5$. Given a DPI the hash function will be applied to its partitioning information to get its position on the ring. Walking clockwise will to the next token defines the responsible endpoint. In this case Endpoint C. The preference list for this DPI and a replication factor of 3 is defined as [Endpoint C, Endpoint D, Endpoint B].

Thus, each endpoint becomes responsible for the region in the ring between itself and its predecessor endpoint. This is an important fact because the change of token assignments (i.e. because of load balancing and cloud bursting) of an endpoint only affect its close neighbors which will have to hand respectively take over DPIs.

As mentioned above, one endpoint can have several tokens assigned, which means that it is responsible for several regions in the ring. In Dynamo, this concept is called virtual nodes [1]. First of all, assigning several tokens to an endpoint is advantageous because the load on the system will be better balanced over the available endpoints in presence of non-uniform data. Secondly, it makes load balancing much more flexible because each endpoint can easily balance the load with any endpoint in the system.

Using the skip list it is very efficient to complete a routing request. Worst case complexity is $O(\log(n))$ and is derived from the underlying data structure. Each endpoint stores the assignments from tokens to endpoints in its skip list. For each routing request we can complete a simple search on the skip list to find the responsible endpoint.

A preference list of size N contains the endpoint that is responsible for the DPI plus its N-1 successor endpoints that are physically distinct and known to be alive. For range queries, all the endpoints contained in that range would be returned.

The router supports different hashing, partitioning and repartitioning strategies.
The implementation of these can be easily exchanged and is examined in more detail in the next sections.

3.2.1 Hashing Strategy

The choice of using an order-preserving hashing strategy was simple: it was necessary to support range queries. If we were using a random hashing strategy, the random hash function would “destroy” the order of the data by assigning random hash values to each DPI.

The drawback of using order preserving hashing is that a skewed data distribution does not get randomized anymore. So, a skewed data distribution leads also to a non-uniform data partitioning among the endpoints, thus requiring an explicit load balancing mechanism (explained in Chapter 5).

Currently there is an order-preserving hashing strategy implemented. It uses the hash function \( h(k) = k \) where \( k \) is the partitioning information described in Section 3.2.2

3.2.2 Partitioning Strategy

As partitioning information one can use either the key or a combination of the type and the key if one wants to make sure that DPIs of the same type are mapped adjacently onto the ring. For the simple key-value store use case, partitioning by key is sufficient.

Currently implemented are the following partitioning strategies:

- **Partitioning by key**, which simply uses the key
- **Partitioning by type**, which uses a combination of the type and the key (i.e. type_key)
Chapter 4

Protocol

This chapter explains the protocol component. The protocol acts as an intermediary between the external interface and the Cloudy2 system. It handles incoming requests and defines replication and consistency guarantees for the system. Two possible implementations are presented in this chapter, each of them providing different guarantees.

4.1 Overview

The protocol supports three kinds of requests: get, put and delete. It carries these requests out which makes it the most central component of Cloudy2.

Figure 4.1: Example get request run on a protocol with a replication factor of 3. An detailed example using the protocol is shown in Figure 2.2 in Chapter 2.

The external interface running on every Cloudy2 endpoint forwards requests to the protocol. With the help of the router, the protocol identifies the responsible endpoints and executes the requests on them. Each of these endpoints accesses its
store, enforcing the replication and consistency guarantees defined by the protocol. As an illustration Figure 4.1 shows an example workflow for a get request in a Cloudy2 system using a replication factor of 3. The protocol functionality can be provided by different implementations, all of them implementing the following interface:

- `Set<DPI> get(DPI dpi);`
- `void put(DPI dpi);`
- `void delete(DPI dpi);`

The remainder of this chapter presents two possible implementations of protocols. Both of them define concrete consistency and replication guarantees.

### 4.2 Simple Protocol

The simple protocol does not implement any replication as part of the protocol (i.e. it has a replication factor of 1). It coordinates incoming requests for any endpoint in the system. There is no master redirection active, so whichever endpoint receives a request also carries it out. To execute the request, it asks the router for the endpoint that is responsible for serving the request. Once the request is executed on the responsible endpoint, a response is returned to the coordinating endpoint, which returns the response to the client of Cloudy2. Because the simple protocol does not provide high availability and fault tolerance, it is not further explored in this thesis. It was thought to be a first implementation of a protocol on the way to the quorum protocol, which provides high availability and fault tolerance. The quorum protocol is examined in depth in the next section.

### 4.3 Quorum Protocol

To maintain consistency among replicas, Cloudy2 uses a consistency protocol similar to those used in quorum systems. The quorum protocol is based on ideas presented in the Dynamo paper [1] and is defined in detail in this section. It includes three parameters \(N/R/W\) which are configurable:

- **Replication factor N**: Number of endpoints that replicate a DPI.
- **Read quorum R**: Minimum number of endpoints that must participate in a successful get operation.
- **Write quorum W**: Minimum number of endpoints that must participate in a successful put/delete operation.

To achieve high availability and durability, the quorum protocol replicates each DPI at \(N\) endpoints. The \(N\) endpoints replicating a DPI are retrieved by asking the router for a preference list of size \(N\) for the given DPI. The first endpoint in the list is called the master and is in charge of the replication of the DPI on the \(N\) replicas. The protocol of the master sends out requests to the \(N\) replicas, only waiting for \(R\) respectively \(W\) successful responses from the \(N\) replicas before returning to the client of the Cloudy2 system. In this case, the quorum is said to be satisfied. If less than \(R\) respectively \(W\) request executions succeeded, then the
4.3 Quorum Protocol

Quorum would not be satisfied and the client would be notified that the request was unsuccessful. The quorum response handler (see Section 4.3.1) is in charge to enforce the quorum and for resolving the responses of the N replicas.

In absence of failure, choosing \( R + W > N \) guarantees strong consistency [18]. Choosing \( R + W \leq N \), even in absence of failure, only provides weak/eventual consistency. This means that after a DPI has been updated the system might not return the updated DPI. Eventual consistency is a special form of weak consistency. Eventually all accesses to a DPI will return the last updated value if no new updates are made to the object. There is a mechanism called “read repair” that repairs inconsistent replicas if they participate in the read quorum. Once all replicas participated in the read quorum - which will eventually happen - consistency is guaranteed, hence the notion of eventual consistency.

One can choose different values for \( N, R \) and \( W \) depending on the use case of the system [18]:

- **Read intensive case**: Systems that need to serve very high read loads often use \( R = 1 \) and \( N = W \).
- **Write intensive case**: Systems that need to serve very high write loads often use \( W = 1 \) and \( N = R \).

In Cloudy2 we generally use \( N = 3 \) and \( R = W = 2 \) to guarantee that for each data item there are always at least two consistent replicas in the system. By choosing \( R \) and \( W \) less than \( N \), consistency is traded off against higher performance and availability. Setting \( R \) and \( W \) to three would make each read and write more expensive.

The next section describes the quorum response handler. It is the part of the quorum protocol that checks whether the quorum is satisfied and resolves the \( R \) respectively \( W \) responses to a final response.

### 4.3.1 Quorum Response Handler

The quorum response handler handles the received responses from the \( N \) replicas. If the quorum is satisfied, the responses are resolved to a final response, which is then sent back to the client. In the case of a get request, the final response is a set of DPIs that matches the query and in the case of a put or delete request, the final response is a boolean indicating whether the execution of the request was successful. If the quorum cannot be satisfied, the quorum response handler throws an exception. The resolving processes for get and put/delete requests are described in the next two sub sections in detail.

#### Get Response Resolver

The get response resolver is called by the quorum response handler to resolve the received responses that satisfied the read quorum \( R \). Of the \( R \) responses, only the master endpoint returns a set of full DPIs plus a digest hash value for its result set. The replicas only respond with a digest hash value of the result set in order to decrease response time and bandwidth consumption in the system. If the digest hash values match, then the resolving process does not have to be executed. The
get response resolver can return the received set of DPIs from the master right away. In the other case, the resolving process works as follows (see Figure 4.2):

1. Compare the $R$ digest hash values. If they are the same, there is no need for further resolving and the DPIs retrieved from the master can be returned to the client. If the digest values mismatch, then each of the $R - 1$ replicas have to fetch a set of so-called digest DPIs. Digest DPIs solely contain the key, type and version field of the DPI. The value of the DPI is not fetched to decrease network bandwidth consumption.

2. Merge all the $R$ sets of DPIs (the full DPI set from the master and digest DPI sets from the replicas) into a final DPI set, which only contains the latest version of each DPI. Section 2.4 describes how version control works in Cloudy2. It is possible that a digest DPI ends up in the final DPI set if one of the replicas had a newer version of the DPI.

3. If there are digest DPIs present in the final DPI set, fetch the full DPIs from the corresponding endpoints so the latest version of each DPI can be returned to the client.

4. Schedule read repairs for the endpoints that did not have the latest version of the DPI.

5. Return the final DPI set, only consisting of full DPIs, to the quorum response handler which will forward it to the client.

The read repair manager is running as a background process and executes the scheduled read repairs detected in step 3 above.

**Figure 4.2:** Example resolving process of a get request with a read quorum of $R = 2$ where the digest hash values mismatched.

Figure 4.2 shows an example resolving process of a get request with a read quorum of $R = 2$ where the digest hash values mismatched. Endpoint 2 (EP2), the master endpoint, provides a set of full DPIs. Endpoint 1 (EP1), the replica, provides a
set of digest DPIs. All DPIs within this set are marked with a superscript D. The subscript of each DPI indicates its version. After merging the two sets and only keeping the latest version of each DPI, we end up with two digest DPIs in the final set. Endpoint 1 has to be contacted to get the full DPIs. Read repairs for DPI D and E have to be issued on endpoint 2 and for DPI C on endpoint 1.

Put and Delete Response Resolver

The put and delete response resolver is called by the quorum response handler to resolve the W received responses that satisfied the write quorum. The put and delete response resolver is much simpler than the get response resolver because it only has to check whether all the endpoints participating in the quorum successfully carried out the put or delete. If all of them are successful the client will be informed about the successful execution of the request.

4.3.2 Normal Case Behavior

Any endpoint in Cloudy2 is eligible to receive get, put and delete requests for any DPI. If a request for a DPI is received by an endpoint that is not the master for this DPI (that is the endpoint is not the first in the preference list retrieved from the router for this DPI), then the request is redirected to the master endpoint, which then sets the version of the DPI and carries out the request. Once the request has been redirected to the master endpoint, it sends out the requests to all the endpoints in the preference list including itself. The client receives the response as soon as the quorum has been satisfied. Figure 4.3 shows an example execution of a get request running on the quorum protocol. A detailed description of all execution steps is given in Section 2.6.

![Figure 4.3: Example workflow of a get request. The numbers in the picture show the chronological order of execution of the individual steps.](image-url)
4.3.3 Failure Case Behavior

Cloudy2 is built to be robust in case of failures: the bigger the cluster gets, the more probable are endpoint failures of temporary and permanent nature. Especially in large scale deployments of cloud storage systems, failures are a given, as shown in the Dynamo paper [1]. In our local test cluster failures are very improbable because the machines are very powerful. So we had to simulate failures to test the behavior of the system in case of failures. The failure case behavior has not been fully tested yet, but the failure scenarios shown in Table 4.1 were identified and are explained in detail in the remainder of this section.

<table>
<thead>
<tr>
<th>Failure</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporary failure</td>
<td>Sloppy quorum (and hinted handoff)</td>
</tr>
<tr>
<td>Permanent failure</td>
<td>Replication level assurer</td>
</tr>
<tr>
<td>Network partitions</td>
<td>Version Control</td>
</tr>
</tbody>
</table>

Table 4.1: Identified failure scenarios and their solution.

Temporary Failure

In the temporary failure scenario an endpoint is not reachable for a small period of time. After that, it will be reachable again. Ideally the system should still be available even though one endpoint fails for a short period. To account for this kind of failure, the ideal solution would be to use a “sloppy quorum” in combination with “hinted handoff” as described in the Dynamo paper [1]. Cloudy2 currently only uses a modified version of the sloppy quorum. If the master endpoint of a replication is unreachable, then the protocol will fetch an additional healthy endpoint from the router that will be included in the quorum. The reason why an additional healthy endpoint is included into the quorum is that a permanent endpoint failure is anticipated. In case one of the replica endpoints is temporarily unreachable, the system would not necessarily realize it because this replica endpoint would simply not participate in the quorum. Once the endpoint is reachable again, it will be read repaired (see Section 4.3.1). If too many replicas are unreachable, such that the quorum cannot be satisfied anymore, then the system will give an unavailable exception.

This paragraph and Figure 4.4 elaborate the case where the master endpoint fails in more detail: If the first endpoint in the preference list (master) is not reachable, then the request will be coordinated by the second endpoint in the preference list. It will fetch a preference list of size $N+1$. This preference list includes $N$ healthy endpoints plus the old master which could not be reached. So requests will be applied by $N$ healthy nodes in total. In case the the master would be reachable again, the request would as well be applied to its store, but because we anticipate permanent failure of the endpoint, this case is not very probable. Figure 4.4 illustrates the system behavior during temporary failure.

It would be of advantage to have hinted handoff in case of temporary failure. Hinted handoff is a mechanism where endpoint D in Figure 4.4 remembers that it was not intended to receive an update and it propagates the update to endpoint A as soon as it becomes available again. Currently hinted handoff is not imple-
Figure 4.4: Example illustrating a temporary failure with the DHT router. $N$ is set to 3. Endpoint A is not reachable, so the request is coordinated by endpoint B instead using a preference list of size 4.

Permanent Failure

If an endpoint fails permanently, it will be declared dead by the gossiper. Each endpoint keeps track of dead endpoints, and after a configurable threshold it is assumed that such an endpoint is dead forever. In this moment, a process called replication level assurer is triggered. The replication level assurer runs on each endpoint and checks whether it has to replicate DPIs in the range that was previously owned by the dead endpoint. If so, the replication level assurer starts streaming these DPIs from the current master endpoint of that range to make sure that each DPI is replicated $N$ times.

Network Partitions

Supporting network partitions is on the road map of Cloudy2. Currently, the behavior of the system in the case of network partitions is not tested. In presence of a partitioned network, Cloudy2 would be split into two independent partitions. Each partition would still be able to operate and receive requests. The endpoints in the other network partition would be declared as dead after a certain time. Once the network partition disappears and under the condition that at least one seed endpoint would be reachable, all endpoints would join again into the same Cloudy2 system. Upon re-joining the system, each endpoint streams the data it is responsible for. All the DPIs with the latest version would be kept but in case of network partitions, version numbers might not be compatible anymore. Version control is explained in detail in Section 2.4.
Chapter 5

Load Balancing

The load balancer is the component that distributes the load evenly on all endpoints in the system and is one of the main contributions of this thesis. The chapter first gives an overview over the load balancing component, then each of its parts is described in detail.

5.1 Overview

Load balancing involves different components: the load calculator, which calculates the load of an endpoint, the load balancer, which decides whether a load balancing step has to be executed and the repartitioner, which executes the load balancing step. Figure 5.1 visualizes the interactions between these different components. The load balancing runs completely decentralized, so each endpoint decides individually what the best load balancing step is. Load balancing is triggered when an endpoint is underloaded (light) or overloaded (heavy). In the first case, it tries to take over load, and in the second case, it tries to shed load to another endpoint that can still take over load.

![Figure 5.1: The load balancer interacts with the load calculator to get load information for any endpoint. After retrieving the load information, it makes a load balancing decision and tells the repartitioner to execute it. Then, the repartitioner changes the partitioning in the router component.](image-url)
The load balancer and repartitioner change the routing table and need to access router-internal information that is dependent on the router’s implementation. That is why, for each router, new repartitioner and load balancer implementations have to be provided. The components are described in detail in the next sections.

5.2 Load Calculator

In order to make a wise load balancing decision, there needs to be a way to calculate the load on the endpoints in the system. There is a statistics service in the system that tracks different measures and makes them accessible to the load calculator. Currently the statistics service is tracking the following measures:

- CPU usage
- Memory usage
- Disk usage
- Number of DPIs per token (i.e. virtual node)

Each endpoint knows the statistics of any other endpoint in the system. The statistical measures are gossiped using the gossiper component. The load calculator can be implemented using different calculation methods. The load balancer tries to balance the load best according to the chosen load calculator. If, for example, a load calculator based on disk consumption is chosen, the load balancer tries to balance the load in a way that the disk usage on all the endpoints in the system is about the same. The following subsections present a selection of these load calculators.

5.2.1 Calculate load using DPI distribution

The load calculator using the DPI distribution as a basis for the load calculation makes the load balancer distribute the total number of DPIs in the system as evenly as possible on all the endpoints.

5.2.2 Calculate load using different measures

This load calculator uses the following metrics that are available from the statistics service:

- **CPU Usage**: CPU usage of the endpoint averaged over the last minute.
- **Memory Usage**: Memory usage of the endpoint averaged over the last minute.
- **Disk Usage**: Current disk usage of the endpoint.
- **DPI Usage**: Current normalized number of DPIs that are in the store of the endpoint.

All the metrics are in the interval (0, 1). This load calculator combines the above listed loads with weights. The weight for each measure is configurable, and the default values have been chosen carefully. Empirical benchmarks have shown that
memory usage is usually the bottleneck in our test environment. That is why it is weighted with a factor of 4. The CPU was less of a bottleneck and is therefore assigned a weight of 3. The disk usage was never the bottleneck in our test environment. Hence, it is weighted with factor 1. The most critical factor is the distribution of the DPIs. It is assigned a weight of 5 so the load balancer would distribute the DPIs as evenly as possible in the system. If any of the metrics is above a defined maximum threshold (i.e. 0.8) then the load calculation is aborted and the metric is directly returned with its value. So weighting the disk usage with a factor of 1 does not mean that we have unlimited data storage. If it becomes the bottleneck and its value grows above the maximum threshold, then the bottleneck would immediately be reported. In all other cases the following formula is used to calculate the load. It calculates the weighted average of all the above mentioned statistical measures:

\[
\text{Load} = \frac{3 \times \text{CPU Usage} + 4 \times \text{Memory Usage} + 1 \times \text{Disk Usage} + 5 \times \text{DPI Usage}}{13}
\]

5.3 Load Balancer

Based on the load calculator, the load balancer decides whether a load balancing step is necessary. It is a completely decentralized process that runs periodically on each endpoint. Each endpoint individually decides what load balancing action to take. By default, load balancing is executed every two minutes.

5.3.1 Simple Load Balancer

The simple load balancer is based on a simplified version of the algorithm described in “Scalable Range Query Processing for Large-Scale Distributed Database Applications” [19]. It only works with the DHT implementation of the router because it balances the load of an endpoint with its neighbors. The concept of “neighbors” is only given if the endpoints are aligned on a ring structure. Because the Maha load balancer, described in the next section, contains all the functionality of the simple load balancer it is not described in more detail. The Maha load balancer extends the simple load balancer by the functionality of balancing the load with a random endpoint in the system.

5.3.2 Maha Load Balancer

The Maha load balancer borrows its name from the author of the “Scalable Range Query Processing for Large-Scale Distributed Database Applications” paper [19]. This load balancing algorithm was chosen because it relies as well on a DHT with order preserving hashing. This means that re-balancing the load preserves the order of the DPIs. The algorithm had to be modified because in Cloudy2 an endpoint can have several tokens to make the load balancing process more flexible. Our modified version of the algorithm accounts for this fact and was also influenced by the work of Karger et. al. [20] and Rao et. al. [21].

The basic idea is that underloaded endpoints migrate to zones in the DHT that are heavily loaded. The average load in the system is denoted as \( \bar{L} \). It averages the load of all endpoints in the system. Every endpoint \( e_i \) periodically checks the
load for all its tokens where the current load of token $t_i$ is denoted as $l_i$. If for a token $t_i$ the inequation $l_i > \varepsilon \bar{L}$ holds, then the token $t_i$ is overloaded (i.e. heavy), $\varepsilon$ is a constant with the assertion $\varepsilon > 1$. A token $t_i$ is underloaded if $l_i \leq \bar{L}$ holds. The default value for $\varepsilon$ is 1.2, that means that a token can hold 30% more load than the average load. $\max$ denotes the maximum load an endpoint can handle. It is a value in the interval $[0, 1]$ and by default 0.8. The load balancing algorithm works as follows:

**Case 1**: $\bar{L} \leq l_i \leq \varepsilon \bar{L}$, in which case $e_i$ does nothing to balance $t_i$. It is in a balanced state.

**Case 2**: $l_i > \varepsilon \bar{L}$, in which case $e_i$ tries to shed part of its load. Let $e_j$ denote the least loaded neighbor $e_{i+1}$ or $e_{i-1}$. There are two sub cases to consider:

a) If $(l_i + l_j)/2 \leq \max$, then $e_i$ and $e_j$ start a load balancing procedure. If $e_j = e_{i-1}$, then $e_j$ moves its token $t_j$ clockwise on the DHT and takes over $(l_i - l_j)/2$ of the DPIs of $t_i$ on endpoint $e_i$ (See Figure 5.2 for an example). If on the other hand $e_j = e_{i+1}$, then $e_i$ moves its token $t_i$ counterclockwise on the DHT ring and sheds $(l_i - l_j)/2$ of its DPIs to its successor (See Figure 5.3 for an example). Both tokens on both endpoints end up having $(l_i + l_j)/2$ of the load. The load balancer instructs the repartitioner to execute a moveData(range, $t_i$, $t_j$) request. Where range denotes the $(l_i - l_j)/2$ DPIs that $e_j$ takes over.

b) If $(l_i + l_j)/2 > \max$, then $e_i$ contacts an endpoint say $e_k$, that has announced itself as underloaded and hence can still take over tokens. Underloaded endpoints are known to other endpoints through the gos-siper. On $e_i$’s request $e_k$ takes over half of $t_i$’s load. In other words $e_k$’s new token $t_k$ is a predecessor of token $t_i$. See Figure 5.4 for an example showing how the repartitioner executes the createVirtualNode($e_k$, $t_k$) request.

**Case 3**: $l_i < \bar{L}$. There are two sub cases to consider:

a) $(l_i + l_j)/2 > \max$, in which case $e_i$ does nothing. This is very important given that if endpoint $e_i$ would remove its token $t_i$, all DPIs in that range would be handed over to its successor endpoint, which would become overloaded.

b) $(l_i + l_j)/2 \leq \max$, in which case $e_i$ tries to remove its token $t_i$. This is only possible if it has more than one token and the removal of the token would not bring two tokens of the same endpoint too close together. The load balancer tells the repartitioner to remove a virtual node by calling removeVirtualNode($t_i$).
5.4 Repartitioner

The repartitioner executes the load balancing decisions from the load balancer. It transfers DPIs between endpoints according to the decision of the load balancer. After successfully moving the data, it accesses the routing table in the router and updates it.

The repartitioner has the following three operations:

- **moveData** (DPI range, Endpoint from, Endpoint to); move DPIs in a range from an endpoint to another endpoint.
- **createVirtualNode** (Endpoint endpoint, DPI rangeDPI); create a new virtual node on an endpoint with the given range.
- **removeVirtualNode** (DPI rangeDPI); remove the virtual node with the given range from the local endpoint.

In the following section, the DHT repartitioner is introduced. It is a repartitioner for the DHT router.

5.4.1 Distributed Hash Table Repartitioner

The DHT repartitioner repartitions the DHT on behalf of the load balancer. It only works if the DHT is used as the routing component. For repartitioning, there are four cases to consider. They are discussed in the next sections. If a failure arises during the execution of any of these cases, the repartitioning step is canceled.

**Move Data to Predecessor**

If the load balancer instructs an endpoint to balance its load with the predecessor, it hands over the DPIs in a given range to the predecessor. Figure 5.2 shows how the repartitioning step works. All the DPIs in the range \((A, B)\) have to be moved from endpoint \(e_4\) to its predecessor endpoint \(e_1\). The following steps have to be executed (We assume a replication factor of \(N = 3\)):

1. Endpoint \(e_4\) sends a repartition message to predecessor \(e_1\) to stream the data in range \((A, B)\).
2. Predecessor \(e_1\) starts streaming data in the range \((A, B)\) from the data sources. While streaming data from this range, the current master \(e_1\) of the range forwards all updates as well to \(e_4\).
3. After finishing streaming, predecessor \(e_1\) sends a response message to \(e_4\) saying that it has streamed all the data from the range \((A, B)\) and updates its routing table.
4. Upon receiving the response, \(e_4\) updates its routing table with \(e_1\) being the new master for the token B. Updates to the range \((A, B)\) are not forwarded any longer.
5. As the last step, \(e_4\) instructs \(e_7\), the last replica of its replication group, to delete the range \((A, B)\).
Move Data to Successor

When the load balancer instructs an endpoint to balance its load with its successor, it hands over the DPIs in a given range to its successor. Figure 5.3 shows how the repartitioning step works. If all the DPIs in a range \((A, B]\) have to be moved from endpoint \(e_4\) to its successor endpoint \(e_7\), then the following steps have to be executed (We assume a replication factor of \(N = 3\)):

1. \(e_4\) sends a streaming message to the last replica in the replication group of its successor to stream \((A, B]\) to ensure the replication level. In this case \(e_1\) needs to stream \((A, B]\) from \(e_4\).

2. \(e_1\) starts streaming data in the range \((A, B]\) from \(e_4\). While streaming data from this range, the current master \(e_4\) of the range forwards all updates as well to \(e_1\).

3. After finishing streaming the new replica \(e_1\) sends a reply to \(e_4\) that it has completed the data transfer.

4. \(e_4\) updates the routing table on the successor endpoint \(e_7\), which is now the new master for the range \((A, B]\). \(e_4\) waits for the notification of \(e_7\) having successfully updated its routing table. After that, it updates its local routing table. Updates in the range \((A, B]\) are not forwarded anymore.

5. To complete the repartitioning step, \(e_4\) deletes the range \((A, B]\) from its store.
5.4 Repartitioner

Figure 5.3: Load balancing step with successor. Endpoint $e_4$ holding token $B$ is heavy and chooses to balance its load with its successor $e_7$. $e_4$ moves its token from $B$ to $A$ and therefore $e_1$ needs to stream all DPIs in the range $(A, B]$ from $e_4$ to ensure the replication level of 3. Endpoint $e_4$ is no longer responsible for replicating $(A, B]$ and can delete the range from its store.

Create Virtual Node

In this case, the load balancer instructs an endpoint to balance its load with an endpoint that is underloaded and can still create new virtual nodes (tokens). The underloaded endpoint takes over all DPIs in the given range. Figure 5.4 shows how the repartitioning step works. All the DPIs in range $(A, B]$ have now to be replicated by endpoint $e_1$. The following steps have to be executed to complete the repartitioning step (We assume a replication factor of $N = 3$):

1. Endpoint $e_4$ sends a streaming request message to $e_1$ instructing it to stream $(A, B]$.
2. $e_1$ starts streaming data in the range $(A, B]$ from $e_4$. While streaming data from this range, the current master $e_4$ of the range forwards all updates as well to $e_1$.
3. After finishing the streaming, the new replica $e_1$ sends a reply to $e_4$ that it completed the data transfer.
4. $e_1$ updates its routing table, saying that it now owns the range $(A, B]$.
5. Upon receipt of the reply $e_4$, updates its routing table. Updates to the range $(A, B]$ are not forwarded anymore.
6. To complete the repartitioning step, $e_4$ instructs the last replica of its replication group $e_7$ to delete the range $(A, B]$. 
Load Balancing

Figure 5.4: Balancing load with an underloaded endpoint that still can create tokens. $e_4$ picks the underloaded endpoint $e_1$ to take over part of its load. $e_1$ needs to stream $(A,B]$ to ensure the replication level of 3 before taking over token $B$. Endpoint $e_7$ is no longer responsible for replicating $(A,B]$ and can delete the range.

Remove Virtual Node

In this case the load balancer detects that a virtual node (token) can be safely removed without overloading a successor. The data in the range has to be replicated one more time before the virtual node can be removed. Figure 5.5 shows how the repartitioning step works. The following steps are executed when an endpoint is instructed to remove its virtual node $B$ (We assume a replication factor of $N = 3$):

1. $e_1$, the last endpoint in the replication group of $e_4$’s successor, has to replicate all DPIs in the range $(A,B]$ to guarantee the replication level. A streaming request message is sent to it.

2. $e_1$ starts streaming data in the range $(A,B]$ from $e_4$. While streaming the data updates to the range are forwarded to the new replica $e_1$.

3. After finishing the streaming, $e_1$ sends a confirmation to $e_4$.

4. Upon receipt of the confirmation, $e_4$ sends $e_7$ a notification that it is now the new master for all data in range $(A,N]$ (which includes $B$). $e_7$ updates its routing table and sends a notification back to $e_4$.

5. Only after $e_7$ reports its routing table update, $e_4$ removes $B$ from its routing table, stops forwarding updates to the range $(A,B]$ and deletes all data in that range from its store.
5.4 Repartitioner

- Stream \([A,B]\) from \(e_4\)
- Delete \((A,B]\) from \(e_4\)
- Update router
- Update router

**Figure 5.5:** Removing virtual node. Endpoint \(e_4\) holding token \(B\) is light and chooses to remove its token \(B\) because its successor \(e_7\) is able to take over all its load. \(e_1\) needs to stream all DPIs in the range \((A,B]\) from \(e_4\) to ensure the replication level of 3. Endpoint \(e_7\) is the new master for the range \((A,N]\) and \(e_4\) can delete the range \((A,B]\) from its store.

### 5.4.2 Failure Case Behavior

If during any of the repartitioning steps a failure happens, the repartitioning step is canceled so it looks to the system as if it had never started. Changes to the routing table in the router are only made if there was no failure during the transfer of data. The arising failure scenarios are described in the following list:

- **Handling multiple create virtual node requests:**
  Each endpoint can only create one virtual node at a time. This is enforced by using a lock that has to be acquired before executing a create virtual node request on an endpoint. If there is already a create virtual node request in progress on a particular endpoint, then this endpoint reports back to the requester that it cannot create another virtual node at the moment.

- **Too many repartitioning steps in a short time:**
  The repartitioner remembers when it executed the last repartitioning step. If it gets asked to participate for example in a create virtual node repartitioning step, it checks whether it is allowed to already execute a repartitioning step again. The default waiting time between two successive repartitioning steps is 30 seconds.

- **More than one virtual node of the same endpoint in one replication group:**
  There is a mechanism that avoids that an endpoint owns too many successive virtual nodes or virtual nodes in the same replication group. Upon each create virtual node and remove virtual node request, the endpoint checks that by executing the request, it is not creating a situation, where there is more than one virtual node of the same endpoint in the same replication group.
• **Streaming fails to replicate data:**
  If during any repartitioning step the replication of data fails, then the whole repartitioning step will be canceled, leaving the system in the same state it was before executing the repartitioning step. Replicating the data correctly is probably the most crucial point of the whole repartitioning step. If this fails, then the whole repartitioning step has to be canceled.

• **Endpoint participating in repartitioning fails:**
  If an endpoint participating in a repartitioning step fails temporarily or permanently, then the repartitioning step times out at one point, leaving the system in the same state it was before executing the repartitioning step.
Chapter 6

Cloud Bursting and Collapsing

Cloud bursting refers to the activity of changing the size of the cloud Cloudy2 is running on. Once the system reaches a critical level of load, new endpoints will be added. When the load decreases again, the opposite of cloud bursting will be triggered: cloud collapsing. Both functionalities are summarized by the term cloud bursting. In combination with load balancing, which distributes the load evenly in the whole system, cloud bursting makes a cloud storage system scale autonomously. The next section will explain cloud bursting and its connection to the load balancing component in detail.

6.1 Overview

The cloud bursting component makes Cloudy2 “elastic” and scale autonomously. Its goal is to keep the average system load between a lower and an upper bound (min and max). If the average system load exceeds one of the bounds cloud bursting (inclusion of new endpoints) respectively cloud collapsing (exclusion of existing endpoints) is triggered. The values for min and max are defined in the load calculator, which is also used by the load balancer. As a matter of fact, cloud bursting works nicely together with the load balancing component (see Chapter 5). Both components have clear spheres of action: The load balancer distributes the system load evenly on each endpoint as long as the load on the endpoint is between the lower bound min and upper bound max. The cloud burster gets only active, once the average system load $\bar{L}$ is greater than the max or smaller than the min value defined in the load calculator. At this point in time, the cloud burster can assume that the load balancer did its best to distribute the load evenly according to the defined load calculation. Both components act seamlessly together without interfering with each other. In more mathematical terms, the spheres of action are defined as follows:

- Load Balancer: $min \leq \bar{L} \leq max$
- Cloud Burster: $\bar{L} < min \text{ or } max > \bar{L}$

Cloud bursting is executed periodically on the “cloud bursting leader”, which is elected using Apache ZooKeeper [15] (how leader election works is described in
Section 2.3). Per default, cloud bursting is executed every five minutes. In each period all endpoints try to acquire the lock in order to become the cloud bursting leader. The one that manages to acquire the lock checks whether cloud bursting or cloud collapsing is necessary based on the average load in the cloud. In the case of cloud bursting, a new endpoint will bootstrap into the system while setting a portion of the data assigned (see Figure 6.1 for an example). Once the data has been streamed to the new endpoint it is ready to serve requests. In case of cloud collapsing, the endpoint that will leave the system has to make sure that its data gets replicated one more time (see Figure 6.2 for an example). After that it will shut itself down.

![Diagram of cloud bursting](image)

**Figure 6.1:** Example of cloud bursting in a Cloudy2 system using a DHT as the routing component and a replication factor of three. The newly added endpoint $e_4$ has to stream the data it is responsible for (highlighted in black) upon bootstrapping. It approximately takes over half the load of endpoint $e_1$.

The reason for using a leader election is to avoid the situation where multiple endpoints perform cloud bursting simultaneously. Imagine the scenario where we have a growing load on the system. The load balancer will distribute the load evenly on all endpoints. Eventually, all endpoints will be overloaded and trigger cloud bursting, resulting in a cloud of twice the size. To avoid this, cloud bursting is executed leader-driven. Using the leader-driven approach guarantees that the size of the cluster will increase or decrease steadily, instead of abruptly. The next sections explain the two existing implementations of cloud bursting.

### 6.2 Cloud Bursting Methods

This section presents two possible methods for cloud bursting and collapsing. The more sophisticated one is the hot spot cloud bursting which includes a new endpoint at the “hottest spot” and excludes endpoints from the “coldest point” in the system.
6.3 Cloud Bursting Executor

The cloud bursting executor executes cloud bursting and collapsing requests in different environments. There are two cloud bursting executors implemented:

**Local cloud bursting executor** executes cloud bursting and collapsing in the local cluster at ETH. There have to be machines present that listen on a specified port for cloud bursting and collapsing requests. In case of cloud bursting, these nodes will be contacted and the bootstrapping information...
is sent via TCP. In the case of cloud collapsing, the node will shut down the java Cloudy2 program and wait again for new cloud bursting requests. Unfortunately, we were not able to set up Eucalyptus [22] in the local cluster, which would have enabled us to simulate a cloud in the local cluster.

**EC2 cloud bursting executor** executes cloud bursting and collapsing requests on Amazon EC2 instances. In the case of cloud bursting, it will start a new instance of the same type as the previously started instances. The bootstrapping information is provided in the user data field of the instance request. In the case of cloud collapsing, the instance terminates itself.
Chapter 7

Benchmarks

The benchmark chapter evaluates the performance of Cloudy2 and justifies claims made in previous chapters. The chapter is structured as follows, first the system performance is evaluated, then the performance of the load balancer and cloud bursting components. All the benchmarks were run in the ETH internal cluster consisting of machines equipped with 24 GB of main memory and a 16-core Intel Xeon processor (2.27 GHz per core).

The discussion and interpretation of every benchmark is directly given within each section.

7.1 Cloudy2 System Performance

The Cloudy2 system performance section evaluates the system performance of Cloudy2. First the read and write performance of the system is benchmarked and then the client-driven routing approach is compared to the server-driven approach. To finish up, there is a simple benchmark that shows the system slow down when the BDB store index grows bigger.

<table>
<thead>
<tr>
<th>Fix parameters</th>
<th></th>
<th>Variable parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Java Virtual machine</strong></td>
<td>Heap Size of 16 GB</td>
<td></td>
</tr>
<tr>
<td><strong>Protocol</strong></td>
<td>Quorum protocol with $N = 3, R = 2, W = 2$</td>
<td></td>
</tr>
<tr>
<td><strong>Store</strong></td>
<td>BDB store with cache size of 5 GB</td>
<td></td>
</tr>
<tr>
<td><strong>External interface</strong></td>
<td>Java</td>
<td></td>
</tr>
</tbody>
</table>

| Cluster size | 3 / 10 | |
| Request type | Read / Write | |
| Request size | 1 KB / 10 KB / 100 KB / 1 MB | |
| Request rate | 25 / 50 / 100 / 200 threads | |
| Routing approach | Client-driven routing / Server-driven routing | |

Table 7.1: Tests have been run with all possible combinations of the parameters shown in the table.

Cloudy2 was tested with different configurations and parameters. All combinations of the parameters shown in Table 7.1 have been benchmarked. In every test,
latencies and throughput were measured. The remainder of this section shows the most interesting results extracted from these tests.

### 7.1.1 Read and Write Performance

In this section the read and write performance is evaluated. Table 7.2 shows the read and write performances for a 3-node cluster and Table 7.3 for one with ten nodes. DPIs of size 1 KB, 10 KB, 100 KB and 1 MB have been written to and read from Cloudy2.

<table>
<thead>
<tr>
<th></th>
<th>Requests/sec</th>
<th>MB/sec</th>
<th>avg latency (ms)</th>
<th>99.9\textsuperscript{th} percentile (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB read</td>
<td>33039.94</td>
<td>33.04</td>
<td>2.86</td>
<td>9</td>
</tr>
<tr>
<td>10 KB read</td>
<td>18017.37</td>
<td>180.17</td>
<td>5.31</td>
<td>27</td>
</tr>
<tr>
<td>100 KB read</td>
<td>2979.56</td>
<td>297.96</td>
<td>30.86</td>
<td>110</td>
</tr>
<tr>
<td>1000 KB read</td>
<td>265.22</td>
<td>265.22</td>
<td>290.85</td>
<td>1014</td>
</tr>
<tr>
<td>1 KB write</td>
<td>20006.4</td>
<td>20.01</td>
<td>4.93</td>
<td>29</td>
</tr>
<tr>
<td>10 KB write</td>
<td>3153.4</td>
<td>31.53</td>
<td>31.31</td>
<td>228</td>
</tr>
<tr>
<td>100 KB write</td>
<td>436.09</td>
<td>43.61</td>
<td>220.43</td>
<td>457</td>
</tr>
<tr>
<td>1000 KB write</td>
<td>39.75</td>
<td>39.75</td>
<td>2040.17</td>
<td>5944.15</td>
</tr>
</tbody>
</table>

Table 7.2: Read and write performance with different data sizes in a 3-node cluster. In total 3 GB of data have been written to Cloudy2 and read out again by 90 threads concurrently with client-driven routing. This makes 30 threads per node.

It can be observed that the average latencies remain the same as we scale out from three to ten nodes. In some cases, such as the writes for big values in the 10-node cluster, they even improve. This improvement is due to the fact that ETH could not provide enough machines to run the tests and Cloudy2 on different machines. So each machine was sending requests to the system while being part of the system as well. That means that some DPIs did not have to be sent over network at all. Hence, there is a speed-up for the writes in the 10-node cluster.

In each test the 99.9\textsuperscript{th} percentiles were calculated to get a feeling of how most of the response times behave. Averages are interesting but to be able to really judge the system performance, 99.9\textsuperscript{th} percentiles were measured. We think it is the right measure to show what response time 99.9% of the users of Cloudy2 get. The 99.9\textsuperscript{th} percentile latencies are always significantly higher than the average latencies, indicating that the variance in the response times seems to be quite high. Java garbage collection could be one of the reasons for these extreme outliers. Future work on Cloudy2 should aim at decreasing the variance of average latencies. Another observation is that the 99.9\textsuperscript{th} percentiles are generally higher in the 10-node cluster than in the three node cluster. This is because of the limitation that we had to send the requests from the same machines as Cloudy2 was running on. There were not enough machines at our disposal to run the tests on different nodes. Clearly, if each machine executing a test is also running Cloudy2, the probability for outliers increases.
7.1 Cloudy2 System Performance

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Requests/sec</th>
<th>MB/sec</th>
<th>avg latency (ms)</th>
<th>99.9th percentile (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB read</td>
<td>78 005.73</td>
<td>78.01</td>
<td>3.74</td>
<td>26</td>
</tr>
<tr>
<td>10KB read</td>
<td>63 425.85</td>
<td>634.26</td>
<td>4.07</td>
<td>49</td>
</tr>
<tr>
<td>100KB read</td>
<td>88 190.00</td>
<td>881.90</td>
<td>25.93</td>
<td>164</td>
</tr>
<tr>
<td>1000KB read</td>
<td>591.91</td>
<td>591.92</td>
<td>268.94</td>
<td>1335.37</td>
</tr>
<tr>
<td>1KB write</td>
<td>45 618.72</td>
<td>45.62</td>
<td>6.41</td>
<td>134</td>
</tr>
<tr>
<td>10KB write</td>
<td>90 385.51</td>
<td>90.39</td>
<td>31.39</td>
<td>218</td>
</tr>
<tr>
<td>100KB write</td>
<td>1289.52</td>
<td>128.95</td>
<td>192.59</td>
<td>445</td>
</tr>
<tr>
<td>1000KB write</td>
<td>142.13</td>
<td>142.13</td>
<td>1287.09</td>
<td>3804.06</td>
</tr>
</tbody>
</table>

Table 7.3: Read and write performance with different data sizes in a 10-node cluster. In total 10 GB of data have been written to Cloudy2 and read out again by 300 Threads concurrently with client-driven routing. This makes 30 threads per node.

Another interesting point is that read latencies are significantly lower than write latencies. This is because in the read case only the digest hash value of the DPIs are retrieved from the two replicas. The master is the only node that retrieves the full DPIs from the store. In the write case, the full DPI is sent in any case to all replicas, which explains the big difference in the latencies. There is much more network traffic and the differences get more extreme the bigger the values get.

The number of requests per second in the 10-node cluster are in general 2 to 3 times better, which underlines the almost linear scale out of Cloudy2 independently of how many machines are added. Cloudy2 supports up to almost 100 MB/s read throughput and 15 MB/s write throughput. The huge difference arises because the tests were run with the quorum protocol and a replication factor of 3.
So the write case is much slower because the data has to be pushed to every endpoint, while in the read case the data has only to be pulled from the master endpoint.

Figure 7.1 shows boxplots for the Cloudy2 configuration that achieved the minimal response time, that is writing and reading 1 KB DPIs. Interesting, again, is that the average is usually higher than the median of the response times because there are quite a few outliers on the upper end. These outliers have not been plotted in the box plots. But in general the plots for reading and writing 1KB DPIs in the clusters of different sizes look very similar, which underlines the almost linear scale out that Cloudy2 supports.

![Boxplots](image)

(a) Writing 100 KB DPIs  
(b) Reading 100 KB DPIs

**Figure 7.2:** Boxplots of response times of reading and writing 100 KB DPIs in a three node and ten node cluster. When reading and writing 100 KB DPIs the highest throughput was achieved.

Figure 7.2 shows boxplots for the Cloudy2 configuration that achieved the maximal throughput, that is writing and reading 100 KB DPIs. With 100 KB sized DPIs, the difference between read and write latencies are already very drastic. Reads have 7 to 8 times lower latencies but as well 7 to 8 times higher throughput.

### 7.1.2 Client-driven versus Server-driven Routing

As described in Chapter 3, Cloudy2 supports client-driven and server-driven routing. In client-driven routing the client periodically downloads the routing table, so it can contact for every request directly the responsible endpoint. This avoids additional hops on the server side to find the responsible endpoint.

Table 7.4 shows the results at the 99.9\textsuperscript{th} percentile and averages for reads and writes of size 100 KB in a three node cluster. Client-driven routing reduces the latencies by at least 120 milliseconds for the 99.9\textsuperscript{th} percentile and decreases the average latencies by about 40 milliseconds. The latency improvement results from the client-driven routing approach that eliminates the overhead of an extra network hop that may happen when a request is assigned to a random endpoint. Another interesting point is that read latencies are significantly lower than write latencies. This is because in the read case only the digest hash value of the DPIs
7.1 Cloudy2 System Performance

<table>
<thead>
<tr>
<th></th>
<th>99.9th percentile read latency (ms)</th>
<th>99.9th percentile write latency (ms)</th>
<th>Average read latency (ms)</th>
<th>Average write latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>server-driven</td>
<td>239</td>
<td>807</td>
<td>69.1</td>
<td>278.27</td>
</tr>
<tr>
<td>client-driven</td>
<td>110</td>
<td>457</td>
<td>30.86</td>
<td>220.43</td>
</tr>
</tbody>
</table>

Table 7.4: Performance of client-driven and server-driven routing approaches. The table shows the 99.9th percentiles and averages for reads and writes of size 100 KB in a three node cluster.

is retrieved from the replicas. Only the master retrieves the full DPIs. In the write case the full DPI (100 KB) is sent to all \( N \) replicas, which explains the big difference in the latencies.

7.1.3 BDB primary index slow down

Table 7.5 shows that the average read latencies increase as the primary index of the BDB store grows. Three tests have been run, each of them writing 4 GB of data into a ten node Cloudy2 cluster and then reading it out again. All data ended up on the same endpoint because all the DPI keys were prefixed with the same string (i.e. “tpcw”). The BDB store cache size was configured as 5 GB.

<table>
<thead>
<tr>
<th></th>
<th>Total of 4 GB data</th>
<th>Total of 8 GB data</th>
<th>Total of 12 GB data</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg read latency (ms)</td>
<td>1.41</td>
<td>5.04</td>
<td>6.09</td>
</tr>
</tbody>
</table>

Table 7.5: Three tests have been run on a 10-node cluster. Before each test, the total data size in the system has been increased by 4 GB and then all the data has been read. The average read latencies have been calculated.

The first test was hitting the empty BDB cache in comparison to the latter ones. Average read latencies grew by a factor of 4.3, while the total data size only grew by a factor of 3. The average write latencies remained around the same. To support a lot of data per endpoint (e.g. several 100 GBs), the BDB store may need reconfiguration, or there should be one BDB store for each replica an endpoint owns. So, if the replication level is set to three in a system, it might make sense to have three BDB stores running on every endpoint. This could be an interesting point to explore in future work (see Chapter 8.2).
### 7.2 Load Balancing Experiment

The load balancing experiment evaluates the load balancing component of Cloudy2. The goal of the experiment is to show the effectiveness of the load balancing mechanism exposed to different request distributions. The performance in the worst case scenario is compared to the performance in the best case scenario to show how effective the load balancing mechanism is. The detailed experiment configuration is listed in Table 7.6.

<table>
<thead>
<tr>
<th>Java Virtual machine</th>
<th>Heap Size of 16 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol</td>
<td>Quorum protocol with $N = 3, R = 2, W = 2$</td>
</tr>
<tr>
<td>Store</td>
<td>BDB store with cache size of 5 GB</td>
</tr>
<tr>
<td>External interface</td>
<td>Java</td>
</tr>
<tr>
<td>Cluster size</td>
<td>10</td>
</tr>
<tr>
<td>Total data size</td>
<td>5 GB (in total 15 GB with replication factor of 3)</td>
</tr>
<tr>
<td>Request size</td>
<td>100 KB</td>
</tr>
<tr>
<td>Request rate</td>
<td>175 threads</td>
</tr>
<tr>
<td>Routing approach</td>
<td>Client-driven routing</td>
</tr>
<tr>
<td>Experiment duration</td>
<td>2 hours</td>
</tr>
<tr>
<td>Request type</td>
<td>80% Read and 20% Updates</td>
</tr>
<tr>
<td>Request distribution</td>
<td>Uniform / Skewed (“tpcw_customer”)</td>
</tr>
</tbody>
</table>

Table 7.6: Tests have been run with all possible combinations of the parameters shown in the table.

Cloudy2 is initialized with ten endpoints, each having one token. All the tokens are uniformly distributed on the DHT ring, as if the system was expecting a uniform workload. The range of the ring is defined as $[0 - 9A - Zu - z]^*$. Two experiments have been run:

- **Best case scenario**, where the workload is uniformly distributed matching the initial distribution of the tokens in the DHT ring. The data gets loaded uniformly on all endpoints and after that a two hour uniform request load consisting of reads and updates is executed.

- **Worst case scenario**, where the request load is heavily skewed. All DPI keys are prefixed with the string “tpcw_customer”\(^1\). In this scenario, the data is initially only loaded on one single endpoint which is responsible for keys with the prefix “tpcw_customer”. It has to handle the whole request load while all other endpoints have no load at all.

In every test, the combined average read and write latencies, the variance of the load in the system and details of each load balancing step have been measured. All measurements have been grouped and averaged into 30 second intervals because otherwise the data could not be visualized reasonably. The load balancer was configured to run periodically every two minutes and $\varepsilon$ was set to 1.5 (i.e. each endpoint can have 50% more load than the average without being heavy). The

\(^1\)The prefix tpcw_customer has been chosen because in Raman Mittal’s thesis [7] the TPCW benchmark was run on Cloudy2 using partitioning by key examining the load balancer’s performance in the worst case scenario.
7.2 Load Balancing Experiment

load calculator weighted disk usage with a factor of 1, CPU usage with 3, memory usage with 4 and DPI distribution with 5 as described in Section 5.2.

7.2.1 Best Case Scenario

In the best case scenario, Cloudy2 was tested under a uniform request distribution. Figure 7.3 shows the outcome of this experiment. The test has been run for two hours. As expected, during the whole two hours no load balancing step has been executed.

![Figure 7.3: Best case scenario](image)

In this setting, there is no need for load balancing because the tokens were distributed uniformly and the request load is uniform too. So the point of this test is to show that the average latencies stay constant even with activated load balancing. Load balancing does not have a negative impact on the system. Figure 7.3 shows that the average latency stays around 65 milliseconds and the variance at 1.8%.

7.2.2 Worst Case Scenario

In the worst case scenario, Cloudy2 is tested under a highly skewed request distribution (all keys are prefixed with “tpcw_customer”). Figure 7.4 shows the outcome of this experiment. The test has been run for two hours totaling in 89 load balancing steps, where the average load balancing step duration was 3.08 seconds. The time includes the initiation of the load balancing step and the time used by the repartitioner to transfer the data to another endpoint.

By the end of the test, all endpoints shared the load evenly and all the tokens started with the prefix “tpcw_customer” (i.e. the initial hotspot disappeared). After 15 minutes of load balancing activity, the average latency already decreased from over 200 milliseconds to 120 milliseconds. After an hour of load balancing,
the average latencies settled at around 100 milliseconds and stayed in this region for the second hour of the test. In total this is equal to an improvement of factor 2.5 compared to the highest average latency measured in the beginning of the test. Also the variance in the loads on all the 10 endpoints decreased continuously from 20% to around 5%. The first load balancing steps had the biggest impact on the average latency. Later load balancing steps decreased the average latency too but not as drastically anymore because the load was already distributed much better.

The worst case scenario has also been run with the parameter $\varepsilon$ set to 1.2. In this case, the load balancer was more sensible and balanced the load at a finer granularity. It was able to reduce the average latency to 80 milliseconds already after an hour but twice as many load balancing steps were executed in total.

7.2.3 Comparison

To conclude the load balancing experiment, the worst and best case scenarios are compared to each other. The results turned out as expected, the load balancer reacts effectively to a non-uniformly distributed request load and manages to bring the average latencies down very quickly. In the worst case scenario the system was in a balanced state after 70 minutes of load balancing. And from the best case scenario we can conclude that load balancing does not have a negative impact on system performance.

In both scenarios, the system was exposed to a read and update workload that is typical for the key-value store use case. The load balancer works also in different use cases. The load calculation formula (see Section 5.2) and the $\varepsilon$ value in the load balancer can be adapted to support other workloads. This offers the possibility to tailor the load balancing behavior to the user’s needs. For instance, increasing the $\varepsilon$ value in the load balancer would make the system execute load balancing steps less often but transfer bigger chunks of data at once. This shows...
7.3 Cloud Bursting Experiments

The cloud bursting experiment evaluates the cloud bursting component of Cloudy2. The goal of the experiment is to show that Cloudy2 is able to scale out and in, according to the overall load on the system. The detailed experiment configuration is listed in Table 7.7.

<table>
<thead>
<tr>
<th><strong>Java Virtual machine</strong></th>
<th>Heap Size of 16 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Protocol</strong></td>
<td>Quorum protocol with $N = 3$, $R = 2$, $W = 2$</td>
</tr>
<tr>
<td><strong>Store</strong></td>
<td>BDB store with cache size of 5 GB</td>
</tr>
<tr>
<td><strong>External interface</strong></td>
<td>Java</td>
</tr>
<tr>
<td><strong>Initial Cluster size</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>Total data size</strong></td>
<td>10 GB (in total 30 GB with replication factor of 3)</td>
</tr>
<tr>
<td><strong>Request size</strong></td>
<td>100 KB</td>
</tr>
<tr>
<td><strong>Request rate</strong></td>
<td>Varying from 30 to 240 threads</td>
</tr>
<tr>
<td><strong>Routing approach</strong></td>
<td>Client-driven routing</td>
</tr>
<tr>
<td><strong>Experiment duration</strong></td>
<td>2 hours</td>
</tr>
<tr>
<td><strong>Request type</strong></td>
<td>80% Read and 20% Updates</td>
</tr>
<tr>
<td><strong>Request distribution</strong></td>
<td>Uniform</td>
</tr>
</tbody>
</table>

Table 7.7: System configuration for the experiment.

Cloudy2 was initialized with five endpoints each having one token. All tokens were uniformly distributed on the DHT ring. After that 10 GB of data was loaded into the system. For the next two hours uniformly distributed requests were sent to the system (80% reads and 20% updates). The number of simultaneous requests varied between 30 and 240 in order to stimulate cloud bursting and collapsing. Cloud bursting was using the random cloud bursting method and was configured to collapse at an average system load of 0.3 and to burst at 0.6. Load balancing was enabled too, using the Maha load balancer implementation. It was configured to run periodically every two minutes and $\varepsilon$ was set to 1.5 (i.e. each endpoint can have 50% more load than the average without being heavy). The focus of this experiment is to measure cloud bursting though and because the load distribution was uniform, only a few load balancing steps were executed. The load calculator weighted disk usage with a factor of 1, CPU usage with 3, memory usage with 1 and DPI distribution with 1. All measurements have been grouped and averaged into 30 second intervals because otherwise the data could not be visualized reasonably.

Figure 7.5 shows the result of the experiment. There are two peak loads during the experiment, which lead to cloud bursting and collapsing activity. As expected, when the average load on the system increases cloud bursting adds new endpoints, and once the load decreases again endpoints are removed. The cloud bursting and collapsing activity always lags behind the average system load. Initially the system only reacts after approximately five minutes of exposure to the load. From start to completion of a cloud bursting step, it takes about 3 to 5 minutes. After
that the endpoint is ready to serve requests. In the case of cloud collapsing, the delay is shorter: after 1 to 2 minutes the cloud collapsing step is completed. There is one strange case in the second peak where one more cloud bursting step should have been executed because the average system load is higher than the cloud bursting threshold. After consulting the logs, it turned out that leader election failed around that time. In the next cloud bursting period the leader was elected again.

The experiment proves that cloud bursting and collapsing enable a data storage in the cloud to adapt the number of physical machines according to the system load. During the experiment a few problems arose: first of all, the leader election service ran riot by electing three cloud bursting leaders in several experiments that are not shown here. The three leaders concurrently added new endpoints to the system resulting in a cluster size of three times the initial size. Secondly, defining the weights in the load calculation formula turned out to be very tedious. To find the correct weights that stimulate cloud bursting the best, many experiments had to be run. If the workload would change again slightly, the same tedious work would have to be done again. Future work (see Section 8.2) should simplify this process and also solve the problems with the leader election.
Chapter 8

Conclusion

This chapter provides a summary of the work in this thesis and highlights the most important conclusions. Furthermore, it describes how the research community can benefit from Cloudy2 and lists points for future work.

8.1 Summary

This thesis presented Cloudy2, one of the first cloud storage systems that allows for adaptability besides availability and scalability. Having the adaptability in place, Cloudy2 enables the research community to use the system as a research platform to try out new algorithms and different component implementations in a very quick and efficient manner. There is no need to build yet another cloud storage system to simply try out a new idea.

The addition of the load balancing and cloud bursting components allows the system to scale autonomously up and down based on the current system load. Cloud bursting and load balancing work nicely together. Both components have clear spheres of action. Load balancing distributes the load evenly on all endpoints in the system. Cloud bursting adapts the number of physical machines in the cluster to the current system load. As a concrete implementation of the load balancer, the maha load balancer has been presented which allows for configuring its balancing sensibility. Hotspot cloud bursting is an instance of the cloud bursting component and allows for bootstrapping a new machine at the “hottest” spot in the system. The activity and sensibility of both components can be customized to different workloads through the load calculation component. The benchmarks presented in Chapter 7 underline the effectiveness of these mechanisms.

Cloudy2 is still under development and we do not claim that it is flawless. But the work presented in this thesis and the theses of Raman Mittal [7] and Stephan Merkli [9] is a first step towards a new open-source cloud storage system that allows an application to evolve over time without needing to replace the storage system.
8.2 Future Work

Cloudy2 and its components are still in the development stage. There are various functionalities and performance enhancements that could not be implemented in the scope of this thesis. This section describes what could be part of the future work on Cloudy2.

Service Level Agreement

At the moment, it is quite tedious for a user of Cloudy2 to define the weights in the load calculation formula. Changes to the formula have an effect on average response times because the formula is used to trigger load balancing and cloud bursting. So, instead of defining weights in a formula, a user of Cloudy2 would much rather like to specify a Service Level Agreement (SLA), which is a formal contract between the user and Cloudy2. An example SLA could be that Cloudy2 will provide a response within 300 milliseconds for 99.9\% of its requests at a peak client load of 300 requests per second. Having defined a SLA the user knows what he can expect from the system. The system then has to cloud burst and load balance to meet the requirements of the SLA.

Improve Robustness

Future work should improve and extend the mechanisms that are in place to handle temporary and permanent failures and also network partitions:

Temporary Failures

At the moment, Cloudy2 only uses a modified sloppy quorum as described in Chapter 4. To make Cloudy2 always writable, even in temporary failure scenarios, the hinted handoff mechanism should be added. It allows the system to temporarily write data to endpoints that are not responsible for storing it. Later, when the responsible endpoint becomes available again, the updates will be propagated to it. The Dynamo paper [1] describes such a mechanism.

Permanent Failures

In order to support permanent failures, Cloudy2 uses the replication level assurer as described in Chapter 4. It replicates the data of the failing endpoint to a new endpoint to ensure that each data item is replicated $N$ times, but it does not guarantee that the replicas are synchronized. The data is simply streamed from an alive endpoint of the replication group with the possibility of streaming outdated data. What is missing is a replica synchronization protocol, for example based on Merkle trees [1], that does not replicate possibly outdated data when assuring the replication level.

Improve Testing Environment

Testing distributed systems is extremely difficult. To test Cloudy2 we are using simple methods, such as unit tests and scalability tests. Building a test environment for complex failure scenarios and performance measurements is left as future work. It would be useful if Cloudy2 would be built and benchmarked once every
night, so that developers can see whether their changes resulted in increased or decreased system performance. These benchmarks should be run automatically and with different component and parameter configurations. It should be very easy to expose Cloudy2 to different workloads.

**Improve Store Performance**

The Berkeley DB store in the current configuration, as the benchmark in Section 7.1.3 has shown, does not scale well with the amount of data loaded into the system. Future work should examine if there are better suited stores available or if BDB can be tuned to support a larger amount of data without causing the primary index look-ups to become slower.

Another problem arises with the BDB store when data needs to be streamed from one endpoint to another one. The data streamer takes a long time to stream a range from the store of one endpoint to another endpoint while concurrently requests are sent to the system. During the streaming, the data streamer needs to iterate through the store, but this seems not to be supported well at peak request loads. This drastically increases the duration of load balancing and cloud bursting steps. Furthermore, it decreases system performance because iterating over the whole store seems to be very “expensive”.
Bibliography


