ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Streaming in the Cloud

Master Thesis

Stephan Merkli

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

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

September, 2009 - February, 2010

Abstract

Streaming applications are becoming more and more a commodity. They are used to monitor servers, for fault detection, for message filtering and many other scenarios. However, many scenarios just require one or two continuous queries for which the burden of installing and maintaining a full-blown stream system is often an overkill. The goal of this thesis is to build a new streaming system, Stormy, which offers streaming capabilities as a service.

Ideally, such a system would automatically adapt the required resources to the load, is fault-tolerant and always consistent. Current streaming systems, however, do not even meet the first requirement as they are designed for a fixed machine setup. In this thesis, we explore how to transfer well-established cloud-techniques to build a new kind of streaming system, which provides exactly those properties. Therefore, Stormy is co-developed with Cloudy2, a new componized key-value store. Cloudy2 allows exchanging components without the need to touch the other components. This design provides the opportunity to adjust and configure Cloudy2 and Stormy to the individual user's specific needs.

Acknowledgements

My gratitude goes to my supervisors Dr. Tim Kraska and Simon Loaesing for their help and support during the whole project and the thesis. Special thank goes to Prof. Donald Kossmann for the discussions in our meetings that provided us with helpful information and let us focus on the important aspects. Due to our great team I really enjoyed working on our project so I would like to thank the other student members of the Cloudy2 team, Flavio Pfaffhauser and Raman Mittal for their great co-work that made this thesis possible. Last but not least I would like to thank Sabrina Edler and Michael Schaufelberger for proofreading my thesis. Sabrina has made available her support in a number of ways, especially her effort with the thesis figures should be mentioned.
Thank you all very much!

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Contributions . . . 2
- 1.4 Structure of this Thesis . . . 3

2 Cloudy2 **5**
- 2.1 Requirements . . . 5
  - 2.1.1 Architectural Requirements . . . 5
  - 2.1.2 Component Requirements . . . 6
- 2.2 Architecture . . . 7
  - 2.2.1 Overview . . . 8
  - 2.2.2 Data Model (DPI) . . . 9
  - 2.2.3 Components . . . 10
  - 2.2.4 Services . . . 16
  - 2.2.5 Sample Workflows . . . 17
- 2.3 Conclusion . . . 20

3 Stormy **21**
- 3.1 Streaming and Complex Event Processing as a Service . . . 21
  - 3.1.1 Query and Stream Model . . . 21
  - 3.1.2 Problem Statement in Detail . . . 22
- 3.2 System Design and Architecture . . . 25
  - 3.2.1 Design Goals . . . 25
  - 3.2.2 Basic Design . . . 26
- 3.3 Normal State Scenarios . . . 36
  - 3.3.1 State Description . . . 36
  - 3.3.2 Master Protocol Overview . . . 36
- 3.4 Performance Scenarios . . . 40
  - 3.4.1 Range Split Ups . . . 41
  - 3.4.2 Load Balancing . . . 42
  - 3.4.3 Cloud Bursting . . . 43
  - 3.4.4 Master-Handover . . . 44
- 3.5 Failure Scenarios . . . 45
  - 3.5.1 Introduction . . . 46
  - 3.5.2 Lost Event . . . 47
  - 3.5.3 Transient Failure . . . 47
  - 3.5.4 Permanent Failure . . . 49

3.6 Lessons learned . . . 50
3.6.1 Stormy on Top of Cloudy2 . . . 50
3.6.2 Query Engine . . . 50
3.6.3 Java & Memory Leaks . . . 51

4 Performance Experiments and Results 53
4.1 The Linear Road Benchmark . . . 53
4.1.1 Overview . . . 53
4.1.2 Benchmark Architecture . . . 54
4.1.3 Implementation . . . 55
4.1.4 Measures . . . 56
4.2 Experiment Scenarios and Measurements . . . 56
4.2.1 Scenario 1: Scale the Number of Benchmarks . . . 57
4.2.2 Scenario 2: Scale the Number of Queries . . . 58
4.2.3 Scenario 3: Benchmark in the Presence of Failures . . . 58
4.3 Results . . . 59
4.3.1 Problems . . . 59
4.3.2 Adapted Test Scenarios . . . 60
4.3.3 LR Benchmark Load Distribution . . . 60
4.3.4 Adapted Scenario 1: Results . . . 61
4.3.5 Adapted Scenario 2: Results . . . 63
4.3.6 Discussion . . . 64

5 Conclusion 65
5.1 Summary . . . 65
5.2 Future Work . . . 65
5.2.1 Multi-Query Optimizations . . . 66
5.2.2 Simulation of Transient Failures . . . 66
5.2.3 Full LR Benchmark . . . 66

Chapter 1

Introduction

An insight to our motivation behind the work and an outlook to our two main contributions – a componized cloud system for different workloads and a streaming service in the cloud – are given in this chapter.

1.1 Motivation

In the past most companies stored their data within their own system on a single server that was backed up by several replication servers. On one hand these servers were very expensive, on the other hand the company itself was in charge of maintaining their infrastructure. These days companies have started to go a different way.
Some web service providers stopped buying own hardware and instead completely swapped out their data to cloud storage systems provided by other companies such as Amazon with their S3 storage [4]. New services have arisen due to the existence of such storage systems [10]. Other providers have developed their own cloud storage systems to fulfill their specific needs and are running these systems within their infrastructure. Examples are Facebook [20], Google [15] and Yahoo [8].
The way data is stored has evolved from the storage on a single high-end server to the distributed storage on several low-end machines. The advantage for companies running an own cloud storage system is the reduced costs of the acquisition of new hardware because these systems are usually designed to run on off-the-shelf hardware. New, upcoming web services with a fast increasing user base can rely on the scalability of cloud storage systems and additionally benefit from the "pay-as-you-go" cost model that only charges them for the resources they actually use. This prevents them to be overwhelmed by the costs of a system supporting high workloads even if they do not require yet. The future seems to belong to cloud storage systems. We have designed Cloudy2, a modular cloud system for different workloads.

One special type of cloud service is concerned with streaming. Streaming thereby is not related to streaming multimedia content but instead refers to content that is processed to generate new one. Examples are different incoming RSS feeds (streams) that are combined and produce new RSS feeds. Other streaming applications watch the payments in a web-shop or analyze server loads and inform the

system administrator as soon as a load peek is detected. Quite a few stream data management systems are available, among them Aurora [1], Borealis [2] (based on Aurora), IBM Stream Core [18] and TelegraphCQ [7]. This thesis presents a new one, Stormy, that is built on our cloud storage system and tries to overcome drawbacks that some of the existing systems provide.

1.2 Problem Statement

Apache Cassandra [29], Project Voldemort [26], Tokyo Tyrant [16] and Scalaris [27] are just some of the existing implementations of cloud storage systems [19]. Each of them was designed to fulfill a specific need. What we need is an architecture that serves different workloads and therefore fulfills various specific needs. One important requirement of a system that tries to solve different problems is its extensibility. We do not assume that one system fits all needs, but we believe that one system architecture can be used as a base to implement other systems, each of them serving different workloads.
Traditional database systems provide a rich interface to query the stored data. Some of todays cloud storage systems only provide a simple interface to store and retrieve data based on a single key [26, 17]. This limits the functionality and especially the performance for example when aggregation of stored values is desired. If a cloud storage system does not support aggregation it has to be performed manually after retrieving all data. Cloudy2's architecture does not limit the way how data can be retrieved. By running MySQL [24] on top of our cloud storage system while delegating some parts of the query processing to lower levels a rich query interface with acceptable performance is possible [22].

The main topic of this thesis focuses on the streaming service Stormy. Applications for stream processing have almost completely different requirements to its underlying architecture than an SQL-like query engine has. Some of the existing database streaming systems have one major drawback, they do not scale. These systems were designed for a fixed machine setup. The design of a fault-tolerant system under the assumption that the cluster size is fixed is comparable simple to the design of a scalable, fault-tolerant system.

1.3 Contributions

This leads us to the subjects of this master thesis that builds a scalable streaming system on top of an existing cloud storage system. Because previous evaluations of existing cloud storage systems made us realize that there are no existing systems that completely fulfill our needs for serving different workloads, we designed Cloudy2. Other architectures as for example Apache Cassandra were in an evolved and tested state but did not provide any flexibility for extensions. Project Voldemort provided this flexibility (e.g. exchangeable storage layer) but lacks of essential features such as server side routing and dynamic cluster sizes.

Due to the absence of an adequate system our first step was the building of a new modular cloud storage system according to our needs. This system can easily be modified in a way as working with plug-ins to create other services than its default one. The default implementation provides a highly-scalable key-value

storage (chapter 2). In a second step we extended our previous product to become a streaming system (chapter 3). The last contribution involved the running of different benchmark scenarios to measure the performance of our streaming service (chapter 4).

1.4 Structure of this Thesis

This thesis is structured as follows:

- **Chapter 1** introduces the reader to the main topic of this thesis and states the problem and our solution.
- **Chapter 2** describes the architecture of Cloudy2, our approach of a modular cloud storage system, and explains a typical data and control workflow inside the running system by two examples. The chapter ends with a short conclusion about Cloudy2 in general and mentions the advantages of testing a modular-designed system.
- **Chapter 3** focuses on the streaming service named Stormy. This chapter shows how Stormy was developed on top of Cloudy2 and thoroughly describes the new components added to support the operations required by a streaming application.
- **Chapter 4** contains a description of the three benchmark scenarios that have been used to measure the performance of Stormy. Furthermore the results of these benchmarks are presented and the appropriate conclusions are drawn.
- **Chapter 5** presents the conclusion of this thesis and lists the topics for future work resulting from the current limitations of Cloudy2 and Stormy.

Chapter 2

Cloudy2

Cloudy2 is not just another cloud storage system. This chapter explains the design of Cloudy2 and presents its flexibility based on the different implementations that transform Cloudy2 in a system for serving different workloads. Section 2.2.5 describes two typical data workflows in Cloudy2, a put and a get request. The chapter ends with a conclusion about the advantages of testing a modular architecture.

2.1 Requirements

Cloudy2, as stated in the first requirement of the architecture, is a componized system. Before describing each of the requirements, two terms need to be defined: cluster and configuration. A cluster references a group of Cloudy2 instances that interact with each other. Instances of different clusters do not interact. Each cluster has a configuration that specifies which implementations of the components are used. This configuration determines the behavior of the system.

2.1.1 Architectural Requirements

Listed below are the requirements the architecture of Cloudy2 has to fulfill. These requirements make no demands on the components, only the infrastructure provided to load and run the components is considered. The requirements for the components are described in section 2.1.2. The requirements below are also concerned with the developing phase, so they do not only list the requirements that need to be fulfilled when running Cloudy2 in an productive environment.

Modularity: The most important aspect of Cloudy2 is its modularity. A component implementation can always be replaced by another implementation of the same component without influencing other components. A configuration file specifies which components to use and which settings these components should have access to. This configuration file is written prior to the start of a Cloudy2 instance so that replacing components at runtime (i.e. choosing another configuration at runtime) is not supported. Running instances are only supposed to interact properly with other running instances using the same configuration, i.e. with instances that use the same implementations of the components.

Testing & Debugging Infrastructure: This requirement states that the architecture should provide the possibility to run tests that verify single components as well as the interactions between components. Facilitating the setup of multiple clusters with different configurations is another important requirement for running distributed tests during development. This involves providing the opportunity to determine a specific configuration based on an assigned identifier or another way to identify single configurations. Testing usually shows the presence of bugs. To fix these efficiently, an evolved logging service helps to backtrack these bugs to errors in the source code. For performance improvements, profiling tools should be available (e.g. JProfiler [11] for Java).

Programming Language: The architecture must be implemented in a programming language that we are sufficiently proficient in or at least are able to introduce ourself efficiently. The programming language should provide libraries and tools to facilitate the development of distributed applications. A large user base and active development is desired to be able to solve problems quickly. The language chosen should be widely known so that possible projects that serve as a base for a component are either implemented in that language or at least provide a library to access it.

Implementability: The interfaces of the components and their tasks should be defined in a way that at least one implementation can be developed and other, different implementations are feasible and justifiable. This should prevent the overwhelming of the architecture with a huge amount of different components for which only one possible implementation exists. In such a case it would not be justifiable to create a separate component; this part of the project should be directly integrated to the architecture itself.

Hardware: Cloudy2 should run on off-the-shelf hardware thus not requiring high-end server farms. Depending on the workload the configuration is targeted at, the system might require a huge amount of main memory, a lot of disk space or several fast CPUs, but the architecture should not be designed in a way to prevent running it on off-the-shelf hardware.

2.1.2 Component Requirements

As opposed to the architecture, which will be built from scratch, the implementations of some of the components may be based on existing source code. This makes additional demands on these existing parts to enable a seamless integration. The components are providing the necessary code to successfully run Cloudy2. That is why the typical requirements of distributed systems are listed in this section and not in the architecture part. The requirements for the components, their interaction and the overall system behavior are stated as follows:

Open Source License: When using existing software for building Cloudy2 components, this existing software must have an open source license that permits the release as well as the non-disclosure of the newly created source code. The license should provide the opportunity even for commercial usage without the necessity to pay license fees. One open source license that fulfills these requirements (under some additional constraints) is the 3-clause BSD

license [23]. GPL [13] requires that the derived work is available under the same copyleft, thus requiring the publishing of the source code.

Incremental Scalability: At least one provided configuration of the components allows incremental scalability, e.g. the load balancing part should not require to double the size of the cluster to handle a slightly increased load. This prevents the administrator of a cluster from unwanted costs that arise when running Cloudy2 on Amazon EC2 instances for example. Scalability at all is of course a requirement. The system should scale "indefinitely". The scalability should only be limited by the assumption that the complete routing information fits on every node.

Documentation: Existing as well as newly created source code should be clearly documented to provide better readability and understandability. Each component must describe how it solves the problem of its underlying task. If a component has special requirements or its function is bound to some assumptions, these requirements and assumptions must be explicitly declared.

Supported Operations: Besides the typical operations for data storage and data retrieval based on a unique identifier, range-based data retrieval should be supported too. In case of a range query several information units should be returned, as opposed to the normal case where at most one is returned. Range based retrieval must at least provide the possibility to specify a start and an end identifier. More sophisticated types of ranges are permitted as long as the default range type is supported.

Availability: A cluster of Cloudy2 instances should be always available. It is not required that a single instance is always accessible, but a cluster as a whole should never deny a request. The system should always try to return the most recent data and if not available currently, at least return stale data instead of no data at all.

Different Consistency Levels: The architecture of Cloudy2 should not add limitations that would deter the implementation from a specific consistency level, e.g. strong consistency. At least one configuration must support eventual consistency (e.g. Dynamo [9]).

Maintainability: A running cluster should require almost no manual intervention in a productive environment. The normal behavior includes failures of physical machines so that these failures are observed automatically and the corresponding actions are taken without additional intervention.

2.2 Architecture

This section provides a brief overview of Cloudy2's architecture and illustrates the function and the interactions between the components involved in the data workflow. The explanations and figures use the term endpoint which references an independent physical or virtualized machine running Cloudy2. Such a machine is represented by its unique IP address (local or global). It is not possible to run several instances of Cloudy2 on the same machine when all are using the same IP address, because they would be considered as the same instance. Sometimes the term node or server is used in place of endpoint.

2.2.1 Overview

The Cloudy2 system is mainly designed of 3 parts: external interfaces, components and services (see figure 2.1). An external interface acts as a link between the outside and the Cloudy2 system. Such an interface handles requests from a client (outside the system) and is in touch with the system usually through the protocol (i.e. forwarding the request to the protocol component and returning the response to the client). For a more elaborate explanation of the interactions within the Cloudy2 system see the sample workflows in section 2.2.5.

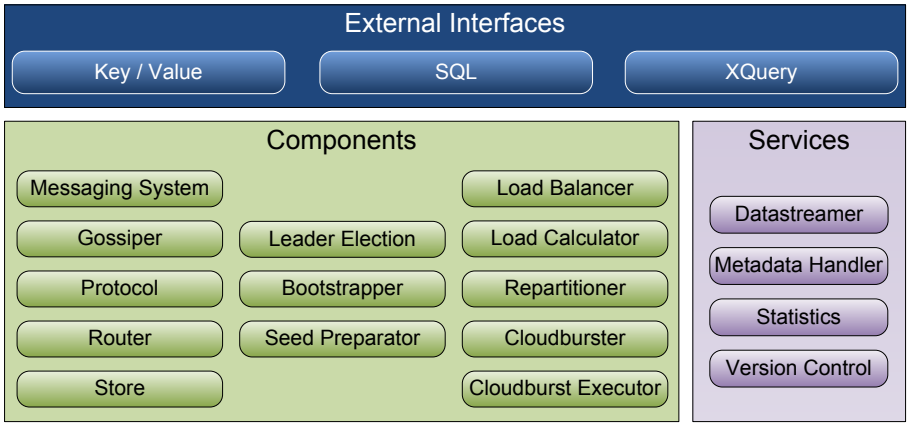

Figure 2.1: Cloudy2 Architecture Overview

Cloudy2 has three built-in external interfaces for request handling: a REST-interface (HTTP), a Java-interface[1] and a Thrift-interface. Besides the interfaces for request handling there are interfaces for administrative and maintenance operations (e.g. change of replication levels, adjustment of logging level, viewer for log files). Additionally, all parts of the system can register their implementation in JMX, making them viewable and modifiable at runtime. This feature is often used during the development to test new settings without the need to restart the machine or the whole cluster. In a productive environment the usage of JMX for maintenance is discouraged.

The components with their implementations and the services are explained in the sections 2.2.3 and 2.2.4.

A data storage system usually supports at least two main operations – the first operation stores data, the second operation retrieves stored data. Besides these two operations that are named **get** for data retrieval and **put** for data storage

[1] To access the Java interface Cloudy2 provides a corresponding client written in Java. This client has built-in support for client side routing. For the initialization one server has to be specified. This server is contacted to retrieve the routing table. Afterwards requests are not always send to the initial specified server but are directly routed to the appropriate endpoints. The routing table is updated at a regular interval (default 2 minutes). Client side routing improves the request performance because no additional hop is necessary that would otherwise be caused by a request redirection to the responsible endpoint.

in Cloudy2, our system supports a third operation to delete stored data. This operation is simply called **delete**. Each operation needs a way to identify data, data is not just a bunch of bytes. The data has a clearly defined structure, the one we use in Cloudy2 is explained next.

2.2.2 Data Model (DPI)

The DPI is our data structure passed along with the three main operations listed above. But the DPI does not only describe the data requested, it also contains the response data. It is **the** data structure used within Cloudy2, between the external interfaces and the components and in between the components. The DPI is the level of granularity how data is stored persistently. The list below gives an overview of its fields.

Fields

- **Key / Range:** Either a key or a range can be used. The key is the main identifier that makes a DPI unique. Different DPIs with the same key are considered as pointing to the same requested data. When no key is specified, two types of ranges can be used, normal ranges and prefix ranges. The next paragraph explains the differences and the semantics.
- **Type:** The type serves as an additional identifier enabling an RDF-like structure[2]. Depending on the implementation of the components, the pair (key, type) uniquely determines a DPI.
- **Lifetime:** The idea behind the lifetime is to treat queries as data. To save data persistently, a lifetime of ∞ is set. A usual query uses a lifetime of 0, therefore issuing a non-continuous query. To have continuous (persistent) queries and volatile data, the lifetime values are inverted. This field is not yet used and its value is ignored.
- **Value:** The data is contained in the value field.
- **Version:** The version is a field maintained by the system itself, it cannot be set manually. The field contains version information (e.g. timestamp or version number) and the endpoint who set the version. Its purpose is to provide a base so that a protocol can guarantee a specific consistency level.
- **Hint:** The hint is also a field maintained by the system. It is used to transport additional information with the DPI that is used within the different components.

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 default implementation of the components expects these fields to be set as follows: for get and delete requests the key / range field is mandatory and the type field is optional. Put requests additionally require the data field to be set. Ranges in put requests are not supported by default

[2]Resource Description Framework [31]: In RDF a resource can be represented as a tuple of subject, predicate and object. Such a resource can be represented in Cloudy2 as key, type and value.

because its semantic is not well defined. Either a range put request would just overwrite existing DPIs in the range or it would set all keys in the range to the value specified. While the first case could be implemented quite efficiently, the second one would lead to additional overhead in a get request too; this is another reason why we do not support range put requests.

Ranges

A DPI does either contain a key or one of two range types. The normal range specifies a start and an end key. 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 *aa*, *b* and *g* (end is inclusive), but does not contain *a* (because the start is exclusive) or *ga*. The second type of range is the prefix range represented by `|prefix|`. A prefix range contains all keys starting with the prefix specified, e.g. the prefix range denoted by `|b|` contains keys starting with the letter b: *b* itself, *ba*, *bz*, *bdddd* and others.

Examples

The next few examples provide a clarification of the data model which might not be that obvious from the fields listed above. The following syntax is used to describe a DPI (if not stated differently): `DPI<[key or range], [Type], [Value]>`. `DPI<id, table, data>` is a DPI with the key *id*, the type *table* and the value *data*. A DPI that can be used for deleting all keys starting with *tmp* (no type) is denoted by `DPI<|tmp|,,>` (or the short form `DPI<|tmp|>`). A get request with `DPI<[a,d)>` will return all stored DPIs whose keys start with a, b or c (start *a* is inclusive, end *d* is exclusive). The list below presents a more concrete example workflow, where the first term specifies the type of the request and the second the DPI sent along with the request. We assume that no DPI is stored yet.

1. `put DPI<mykey,, mydata>` is a put request that stores a new DPI with the key *mykey* and the data *mydata* (no type).
2. `put DPI<myotherkey, atype, somedata>` stores the given DPI.
3. `put DPI<mysecondkey, mytype, myotherdata>` stores the given DPI.
4. `get DPI<mysecondkey>` returns the DPI inserted in the last step.
5. `get DPI<|my|>` or `get DPI<(a,z)>` returns all the three DPIs inserted before.
6. `delete DPI<||>` is a full range delete request. This request deletes all stored DPIs because the prefix range || contains any key.

2.2.3 Components

The components (figure 2.2) 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. The replacement of components has to take place before running Cloudy2. In a cluster, all endpoints running Cloudy2 must have the same component configuration, otherwise the behavior of the system is undefined. This is due to the fact that two 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).

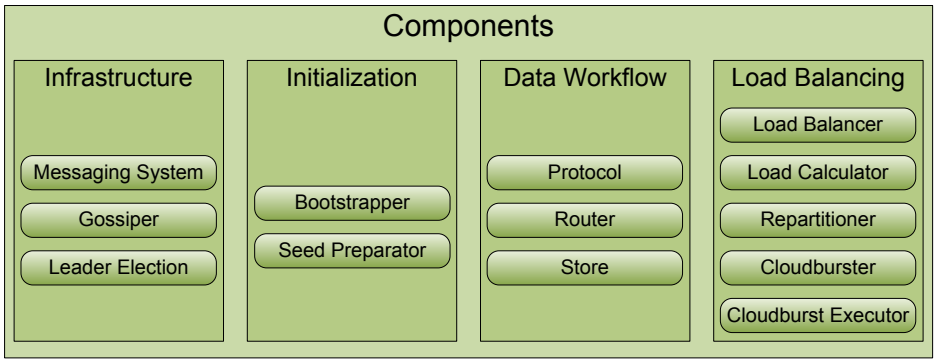

Figure 2.2: Cloudy2 Components

All components are described below and for each of them at least one working implementation exists; the implementations are listed right after the component explanation. The development of Cloudy2 started with only a few main components (messaging system, router, store, protocol, gossiper, load balancer). The current version of Cloudy2 is made up of about a dozen components. On one side new components were created to reuse existing parts and to prevent the duplication of code (e.g. load calculator). On the other side running Cloudy2 on Amazon AWS and keeping the architecture modular at the same time required the creation of new components (e.g. bootstrapper, seed preparator).

This thesis does not explain the components related to load balancing (i.e. load calculator, load balancer, repartitioner, cloudburster and cloudburst executor) in detail because that topic is part of Flavio Pfaffhauser's thesis [25]. Both groups, the load balancer in conjunction with the repartitioner, and the cloudburster in conjunction with the cloudburst executor, access the load calculator to retrieve the current load information. The first two components are responsible for balancing the load by changing the positions of endpoints, the second two components take care of an overloaded or underloaded cluster. Some more explanations can be found in the respective component descriptions.

Messaging System

The messaging system is independent of any other component. It offers the ability to send messages between the different endpoints in the cluster. On the receiving side a handler can be registered to react to the incoming message of a certain type. Messages can be sent either reliable by using TCP connections or in a fire-and-forget style by using a UDP connection.

- **Cassandra:** The evolved messaging system of Apache Cassandra is able

to handle a high load when correctly set up. We have ported the newest available version (0.5.0).

Gossiper

The gossiper provids general information about each node, like specific information for the router (e.g. a token in the ring in case of a router using a DHT). Each node has its own application state, the gossiper takes care that the application state is made known to the other nodes. All parts (services & components) in Cloudy2 can add their own data to the application state (e.g. the router as mentioned above or the statistics providing load information about the local endpoint). The gossiper is also in charge of notifying other components when a node fails or a new node joins the cluster.

- **Cassandra:** This implementation is based on the gossiper and failure detection source code from Apache Cassandra. Some parts of the code were modified or replaced (e.g. bootstrapping) to fit into the modular architecture of Cloudy2.

Leader Election

The leader election is a general component whose task is to acquire and release exclusive write locks. It is named leader election because exclusive write locks can be and are used for electing a unique leader in a group of endpoints. An identifier (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.

- **ZooKeeper:** Apache ZooKeeper [28] is a centralized service providing different sub-services that are used by distributed applications. ZooKeeper uses persistent connections to determine when a node fails and it should assign the lock 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 second one according to the arrival order will get the exclusive write lock.

Router

The router is used by the protocol to find all the endpoints that are responsible for a DPI. These endpoints are returned in a list ordered according to the preference, 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 (usually only the protocol is interested in this information).

- **Skip List Router:** The skip list router implements a distributed hash table (DHT) using a skip list as the underlying data structure. Each DPI is mapped to a position (token) on a ring. This mapping is done by using an order preserving hash function to hash some parts of the DPI (usually the key). An endpoint is assigned to one or more tokens on that ring. The responsible endpoints for a DPI are the endpoints whose token is a successor of the token assigned to the DPI (i.e. the token calculated by hashing the DPI key).

Protocol

Replication and consistency (partly) are assured by the protocol. This component is the layer between the external interfaces and the rest of the system, especially the store. The protocol supports the three types of request, i.e. get, put and delete.

- **Simple:** The simple protocol does not support replication (i.e. it has a replication factor of 1). It forwards an incoming request to the first endpoint in the preference list returned by the router. The endpoint to whom the request is forwarded is responsible for storing the DPI packed with the request.

- **Quorum:** The quorum implementation of the protocol is based on Dynamo [9]. Some parts of the source code are taken from Apache Cassandra. Incoming requests are always forwarded to the first endpoint in the preference list ("master" redirection). The protocol assures eventual consistency.

Store

The store builds the persistent layer. The methods provided by the storage interface (get, put and delete) are invoked by the protocol.

- **In-Memory:** The in-memory store is simply using an internal hash map to store the single DPIs. It was developed for testing reasons, but can, for small DPI sizes, be used in productive systems.

- **BDB:** Berkeley DB provides a Java interface that is accessed by this store implementation. The data is kept in the cache and saved to an internal persistent store (local hard disk).

Bootstrapper

In the starting phase of Cloudy2, that is when a new endpoint comes up, the endpoint's bootstrapper makes sure that the new endpoint receives the information necessary to become a member of an existing cluster or that the endpoint is in a new cluster if it is the first one.

- **Default:** When running Cloudy2 in a normal network cluster, the default bootstrapper should be used. It tries to contact the seeds provided by the seed preparator, one after the other, until a contact was successful or the system start fails otherwise.

- **EC2:** The EC2 bootstrapper was developed to start Cloudy2 on EC2 instances provided by Amazon. Because the IP addresses of the seeds are not known prior to the start of the cluster, the EC2 bootstrapper uses the dynamic hostnames provided by the EC2 seed preparator. The EC2 bootstrapper provides some additional handling necessary only on EC2.

Seed Preparator

The seed preparator's task is to return the endpoints that are seeds. These endpoints are contacted by new bootstrapping endpoints to join the cluster.

- **Default:** The default seed preparator returns the seeds as written in the configuration file (hardcoded).

- **EC2:** In an EC2 cluster on Amazon the seeds are determined based on their launch index. The EC2 seed preparator takes care of assigning dynamic hostnames to the first few endpoints starting.

Load Balancer

The task of the load balancer is scheduled at a regular interval (default 2 minutes). When the load balancer is invoked it calls the load calculator to retrieve the load status of the whole cluster and of the endpoints in its vicinity. Based on these load values the load balancer decides either to execute a load balancing step (by passing the control to the repartitioner) or it does nothing because the load is already evenly balanced across the endpoints in the cluster.

- **Simple:** The simple load balancer tries to balance the load with its direct neighbours. The implementation is based on a paper written by Maha Abdallah and Hung Cuong Le [3].

- **Maha:** This load balancer is an extension of the simple load balancer and additionally supports the creation and removal of virtual nodes to balance the load.

Load Calculator

This component calculates, based on different factors (e.g. memory consumption, CPU usage and others), the load and reduces it to a single value. This value is used by the load balancer to decide if a load balancing step is necessary.

- **CPU:** The CPU load calculator only considers CPU usage to calculate the load value.

- **Key Distribution:** The key distribution (number of DPIs per endpoint) gives a far more exact picture of the real load in the system.

- **Combined:** This load calculator uses several different factors and weights them to determine the load value.

Repartitioner

The repartitioner is invoked by the load balancer and changes the partitioning of the data placement. Its implementation is tightly bound to the router's implementation.

- **Token:** If using the skip list router implementation for the router component, this repartitioner must be used. It moves the tokens on the ring to different places.

Cloudburster

As the load balancer the cloudburster is a task scheduled for regular execution (default 5 minutes). These two components are not related to each other, the cloudburster works indepdently of the load balancer. The cloudburster decides, based on the average load in the cluster, if a new machine should be added or an existing one should be removed from the cluster. All endpoints execute the cloudburster task but only one endpoint is allowed to invoke the cloudburst executor to perform the requested operation. The leader election component is used to satisfy this constraint. Without this requirement it might happen that the cloudburster task on several endpoints decides to do cloud bursting and that several new machines are added at the same time even if one new machine would be sufficient to handle the load. What differs between the implementation of the cloudburster component is the position (routing information) where the new machine is started in the cluster.

- **Random:** The new endpoint is started at a random position in the cluster. The load balancer takes care of balancing the load later on.
- **Hotspot:** To improve the performance and decrease the necessary load balancing steps that occur when using the random cloudburster, the hotspot cloudburster starts a new instance at the heaviest (most active, hottest) position in the cluster.

Cloudburst Executor

The cloudburst executor starts a new Cloudy2 instance or terminates an existing instance. In case of cloud bursting, the cloudburster calls this component and passes data placement information along with the request. The new bootstrapping endpoint uses this data placement information to announce its position to the router.

- **Dummy:** The dummy implementation is used for testing purposes. It just ignores the request for cloud bursting or cloud collapsing, therefore it can also be used in networks of fixed size.
- **Local:** In a local network that does not support adding new machines, the local cloudburst executor is used with a cloudburst listener located on other machines in the network to implement a form of dynamic machine allocation.
- **EC2:** The EC2 cloudburst executor starts or collapses Amazon EC2 instances.

2.2.4 Services

The services (figure 2.3) are related to components but they are not exchangeable. They are inherent parts of Cloudy2. The exchangeable components can access the services for fulfilling their task. The services support the components in their work. The reason why the services are an inherit part of the system and 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.

Figure 2.3: Cloudy2 Services

Datastreamer

After bootstrapping or load balancing steps, DPIs have to be copied or moved from one endpoint to another. This task is fulfilled by the datastreamer. Moving data could always be done by sending requests through the protocol. From the perspective of the endpoint requiring data the requests are a remote get, local put(s) and a remote delete (for data moving, in case of copying there is no delete request). Usually, especially when a new node bootstraps, a large amount of data has to be moved. Letting the protocol move such a large amount of data is not that efficient. Because moving data is quite a regular, often-occurring process in a high-loaded system we developed the datastreamer. The datastreamer opens a TCP connection between the involved endpoints and streams the data directly from one store to the other store without additional overhead by the messaging system or other involved components.

Metadata Handler

The metadata handler is responsible for assuring that the metadata explicitly provided by the components is synchronized between all involved endpoints in the cluster. The metadata is stored as one DPI. The metadata has a version number which is transferred with each message sent between two endpoints. If an endpoint receives a message containing a higher metadata version number than it currently has, it immediately fetches the new metadata.

Statistics

The statistics service collects system information like memory consumption, CPU usage and free disk space. These values are stored in the application state that is exchanged with other nodes by the gossiper. Therefore the statistics are not only providing information about the local endpoint, they include the values collected on remote endpoints too. These values are mainly requested by the different implementations of the load calculator that is used by the load balancer to decide when and how to do the load balancing steps.

Version Control

Each DPI 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 that is working with timestamps, each Cloudy2 instance maintains a clock skew table. This table has 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 [21], 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 (the newest one) of the DPIs stored. The purpose of the version is to reject request that have an older version than the one that is currently stored. To clarify this, assume that a protocol processes two incoming requests for the same DPI key. It sets the version for both of them, DPI_1 has the version number 1, DPI_2 has the version number 2. The requests are forwarded to the responsible endpoints (e.g. master & replicas) in two separate messages. The final result should be that the stores on the responsible endpoints only contain DPI_2, because it is the newest one. In case of message transfer delays it might occur that one replica receives first the message for DPI_2 and then the message for DPI_1. Without versioning DPI_2 would be overwritten with the older DPI_1. With versioning the put request compares the version of the DPI already in the store (DPI_2) with the version of the DPI that is going to be stored (DPI_1). Because the version of the stored DPI is newer than the one of the request DPI the version control mechanism will prevent the storing of the new DPI and throws an exception (more precisely obsolete version exception) to inform the sender thereof. The comparing of version 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.

2.2.5 Sample Workflows

This section illustrates two workflows that typically occur, a request to put data and a request to get the stored data. For this illustration we assume that Cloudy2 was configured to use the DHT router and our implementation of the quorum protocol. The quorum settings (N/W/R) are 3/2/2, that is 3 replicas and a read and write quorum of 2. Other protocols might show a different behavior, but the basic workflow is almost always the same. Namely: a client is issuing a request (DPI) to the external interface. The external interface forwards the request to the protocol. The protocol contacts the router to get the preference list so that it knows where the requested DPI is located. The protocol then contacts the

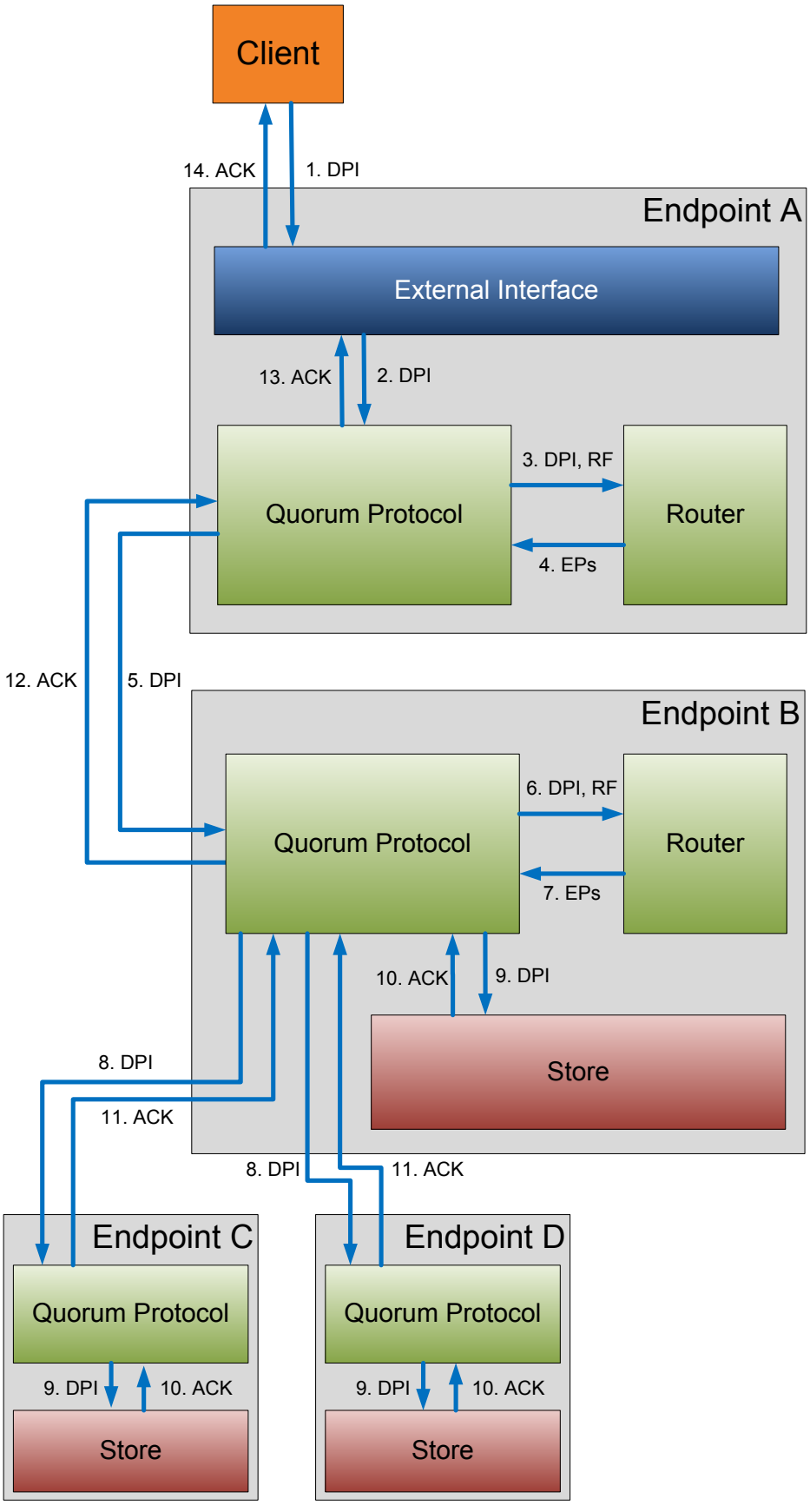

Figure 2.4: Sample Workflow

protocols running on other (remote) endpoints. These protocol instances then forward the request to their local stores (for storing the DPI persistently). Figure 2.4 illustrates the workflow for the put request. The workflow for the get request is quite similar, differences are noted in the explanation.

Put Request

The client issues a put request for `DPI<key, value>` to the endpoint A (step 1). The put request is received by the external interface of endpoint A and then forwarded to the quorum protocol running on that endpoint (step 2). The first task of the quorum protocol is to invoke the router to get the preference list for this DPI (step 3, RF: replication factor passed to the router). The router returns a list with the endpoints B, C and D (step 4, EPs: list of endpoints). The quorum protocol checks if it is the endpoint in charge of the given DPI. The endpoint is in charge if it is the first in the preference list (therefore being the endpoint to whom the DPI is assigned based on the partitioning used by the DHT router). Because in this case endpoint A is not in charge of the DPI, it forwards the DPI to the one that is, i.e. endpoint B (step 5).

The quorum protocol of endpoint B receives the redirected request (step 5). As before, the protocol asks the router for the preference list (step 6). Assuming that no new endpoints joined or existing endpoints leaved the cluster, this preference list will be the same as with endpoint A (step 7). The quorum protocol now sets the version (timestamp) and sends two messages to the replicas C and D (step 8). This message contains the DPI and the type of request (put in this example). After sending the messages to the replicas, the protocol forwards the request to its local store (step 9) and receives an acknowledgement (step 10, ACK: acknowledgement). The same procedure is executed on the replicas on reception of the request message (step 9 and 10 on endpoint C and D). The replicas send the acknowledgements to endpoint B (step 11). Endpoint B, who sends the two messages, waits for two acknowledgements (quorum setting W=2). This means that it waits for only one acknowledgement from a replica because it already got an acknowledgement from its local store. After having received this acknowledgement, the quorum protocol informs endpoint A, who was the sender of the redirect message, about the successful execution of the request (step 12). As soon as endpoint A receives the acknowledgement it informs the client about the successful execution of the operation (step 13 and 14).

Get Request

The workflow for the get request is quite similar. Steps 1 to 9 are the same as for a put request. What differs in the remaining steps is that the replicas do not just send an acknowledgement, but instead send the requested DPIs along with their responses (i.e. replace the "ACK" in figure 2.4 with "result")[3]. The endpoint in charge is waiting for two successful responses (read quorum R=2). As soon as it receives them, it is starting the resolving process. This process compares the versions of all the received DPIs with each other and returns the

[3]In the latest implemention of the quorum protocol the replicas only send digests. This increased the performance drastically because not all the DPIs have to be fetched from the store and transfered to the endpoint in charge.

newest one. Assume that in the previous example of the put request the endpoint in charge (endpoint B) received the responses from itself and from the replica C and that the message to endpoint D was lost. In this get request now endpoint B receives the responses from itself and from endpoint D. The DPI returned from endpoint D (if any) is outdated (because it missed the last put request). Therefore the reconciliation process will return the DPI from endpoint B as the newest. Additionally, the quorum protocol schedules a read-repair for endpoint D so that this endpoint gets the newest one too. The remaining steps are as described above, with the difference that the acknowledgements are the requested DPIs.

2.3 Conclusion

Using system testing [14] to evaluate a distributed system means to verify the behavior by accessing the system as a whole. Using this testing level for a distributed system is quite painful when tracking back failures during the execution to errors in the program source code. This is one of the reasons why tests for systems in general and especially for distributed system should start at the smallest possible level, usually unit tests, and continue up to the highest level, system testing. Testing levels that involve user acceptance or similar tests are not considered. We followed that principle for Cloudy2, started with verifying single code sections within one component, tested the component functionality, evaluated the interactions between different components and finally observed the behavior of a whole Cloudy2 cluster.

The Cloudy2 components usually have dependencies on other components through their interfaces. For that reason testing one component for its functionality involves internal calls from this component to other components. The modular architecture of Cloudy2 enabled us to remove these dependencies by replacing the corresponding components with dummies. These dummy components have no real functionality. They implement the component interface and return semantically valid values if asked for by the component under test. This facilitates the tests tremendously because an error in one component does not disturb the testing of others.

We believe that the described architecture of Cloudy2 is able to handle different workloads. Different implementations and configurations will show that (e.g. Stormy and Raman Mittal's thesis project [22]). Depending on the requirements of a future system changes to the existing interface might be inevitable; we observed that by ourselves during the development. We hope that other systems will be developed with Cloudy2 as a base and thereby preventing developers from reinventing the wheel again and again.

Chapter 3

Stormy

This chapter introduces the reader to the query and stream model used in Stormy and describes how Stormy was designed and implemented on top of Cloudy2. The functioning of Stormy is illustrated by a few workflow scenarios that cover the normal states, states where the load is actively balanced and failures.

3.1 Streaming and Complex Event Processing as a Service

It is essential that the terms query, stream and event are clearly defined. The reader has to know the concept of stream processing before the internals of the implementation can be explained. The following section describes these terms and the concept in a very general way. Because Stormy is using MXQuery [12] as its query engine, the examples given for illustration purposes are using XML as data and XQuery as the query language. Nevertheless the concepts explained are valid for other implementations too.

3.1.1 Query and Stream Model

A stream is a notion of a transport channel whose task is to forward events. An event contains a data block that can be processed by a query. The data block must be complete meaning that it contains valid data according to the data model used by the query engine. For XQuery an event is an XML fragment, for example `<a>content</a>`, whereas `<a>content` would not be an XML fragment because it is not well-balanced (the W3C document [32] provides the exact terminology).

A query can subscribe to a stream. Events arriving on one of the input streams of a query are forwarded to that query and consumed by it. After processing, the query itself might produce output events that are put on the query's output stream. A query has exactly one output stream but can have several input streams. This means that the query can subscribe to several different input streams and therefore react on all incoming events of these streams. In most cases a query in stream processing systems is a continuous query. When the query is registered it starts processing events until the system is terminated or the query is deregistered again.

Because each query itself has an output stream, a query can register to another query's output stream. This allows to build arbitrarily stream graphs. Figure 3.1 shows an example of such a stream graph. Streams are represented by directed arrows; the tail is where events are put in, the head is where they are put out. Queries are represented by rectangular boxes. Outgoing arrows of a box represent the query output stream, incoming arrows the query input streams. The outgoing arrows are labeled with the query identifiers because each query only has one output stream. Not all streams are output streams of queries; stream I and J in the example do not have a starting point. Events are put in to such a stream from a source outside the stream processing system, for example from an RSS feed. On the other hand the events of a stream can also be forwarded to a target outside the system; illustrated in the example by the streams C and E. The events of stream E will only be sent to an external target, whereas the events of stream C will additionally be consumed by the query E.

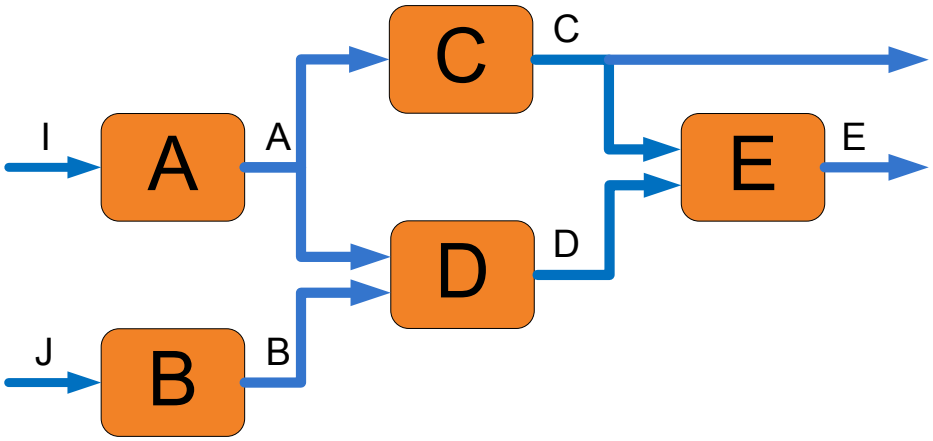

Figure 3.1: Example Stream Graph

Beside input streams, queries and outputs there is a fourth element called store (not related to the store described as a component of the Cloudy2 architecture). There are two types of stores, internal stores and shared stores. Both are a caching mechanism for the query engine. The internal store is bound to one query and only this query stores and retrieves data. Shared stores, however, are common (shared) caches used by several queries to retrieve data. A shared store, similar to a query, subscribes to one stream and is fed with the events only from this stream. For the rest of this thesis the two terms *shared store* and *internal store* are used to reference the fourth "stream" element, the term *store* is connected with the store component of Cloudy2.

3.1.2 Problem Statement in Detail

Assumptions

In the following the assumptions are described. These are made to assure that the requirements specified in the next section are satisfiable by Stormy.

Each sent event is associated to one stream meaning that the same event must

not be processed by several different streams. An event belonging to a stream is uniquely identifiable by its event number. This is especially important for duplicate elimination. The event numbers of two successive events differ by one whereas the second one has the higher event number (if the first event number is 10, the second event number is 11). To make event processing possible, one machine is always able to handle a single query. To fulfill the requirement of total order, a client sends events to Stormy in order, i.e. the client always waits for the acknowledgement of its last sent events before sending the next ones. Without sticking to this assumption the incoming messages from the client would be subject to race conditions and no order could be guaranteed.

Requirements

The requirements listed in the Cloudy2 chapter for the components (2.1.2) are of course valid for Stormy too. As mentioned in the abstract, an ideal streaming system would automatically adapt the required resources to the load, is fault-tolerant and always consistent. This already points out the first three important requirements for Stormy.

Scalability: As the scalability requirement for Cloudy2 already states, Stormy should scale seamlessly and therefore not require the adding of several servers at once. Scalability should be guaranteed in the number of streams as well as in the number of queries.

Fault Tolerance: Even in case of machine failures the streaming system should still operate normally. Failures are assumed to be the norm rather than the exception.

Consistency: Strong consistency is a requirement. As opposed to a key-value store that still can be used when only weak consistency is guaranteed a streaming system cannot. A query always needs to have the latest state otherwise the stream processing will provide incorrect results. If Stormy once accepts an incoming event (i.e. the receipt of an event is acknowledged) it must not get dropped. As an example imagine two put requests that are sent consecutively to a key-value store. The first one gets dropped while the second one is processed. The final state of the store is consistent. In a streaming system this is not true. Lost events are fatal for a streaming system because the query state depends on each processed event, not only on the last one.

Availability: The availability requirement of Cloudy2 transformed to Stormy means that it is always possible to push an event to the system. Even if a machine fails the events can be pushed to any other server running in the same cluster and are then forwarded by that server to the responsible one. This requirement also states that there is no single point of failure, because otherwise the availability criteria could not be guaranteed.

Total Order: There are two requirements to point out here. The first one declares that there is an order on the messages sent from one client. This requirement is needed to assure consistent query states on all replicas. The second and stronger requirement encapsulates the first one. It states that

all accepted messages must have an order. For queries with multiple input streams this is required because the order of events from several input streams must be fixed to avoid inconsistencies in the query state. Regarding the outputs, monotonicity of the sent output messages is required, i.e. already sent output messages must not be retracted in case of failures.

Latency: Having good response time is a secondary requirement, we do not require latency values as requested for real-time systems (e.g. stock trading). Of course the response times should be optimized; the lower the better. The "the-lower-the-better" principle matches the cost factor too, e.g. regarding the number of Amazon EC2 instances needed to run Stormy, but again this is only a secondary requirement. There are primary requirements that are fixed and have higher priorities, i.e. strong consistency, availability and scalability.

n:m Relationship: The key-value store implemented as a default configuration in Cloudy2 maps one key to one value (1:1 relationship) and a range to a set of values (1:m relationship) respectively. In Stormy the same mapping is used; a stream maps to one or several queries, outputs or shared stores (stream consumers). Compared to Cloudy2, Stormy additionally needs to support a n:1 relationship; a query subscribing to several streams is a mapping of n streams to one query. As a result, Stormy must provide an efficient solution to support the n:m relationship from streams to stream consumers.

Interface

Stormy should provide the following interface for pushing events and for the (de)registration of input streams, queries, outputs and shared stores:

- **StreamID registerStream(description):** To register a new input stream, that can be used to feed events to the system, this method is called. The stream ID of the newly created input stream is returned.
- **StreamID registerQuery(name of query, input stream IDs, description of internal stores, IDs of shared stores, query):** To register a new query at least three parameters are required: the name of the query, the query itself and one or more input stream IDs to whose events the query subscribes. More advanced queries can use internal stores, for which the description has to be provided, or shared stores, for which the store ID has to be provided (a shared store itself needs to be registered by an appropriate call to the registerStore method). A successful response of this method invocation is the stream ID of the query output stream. This is the stream to which the query sends its result events.
- **StoreID registerSharedStore(description of shared store):** The description of the shared store contains the name of the store, the input stream ID and additional store related parameters like block size and the number of seconds to store the data. The method registers the shared store and returns the store ID.
- **deregisterStream(StreamID / StoreID):** To deregister an input stream, a query or a shared store, this method needs to be called. Deregistering is

only possible if there are no stream consumers that are subscribed to the stream being deleted (for input and query output streams).

- **registerOutput(StreamID, target):** An output for the given stream ID is registered by calling this method. Incoming events on the stream will be sent to the specified target (IP address and port).
- **deregisterOutput(StreamID, target):** This method is called to deregister a previously registered output.
- **pushEvents(StreamID, events):** To feed events to Stormy this method is called with two parameters, the stream ID and the events to push.

By implementing the described interface a streaming system provides the flexibility to realize any stream scenario supported by the stream model described in Stormy's introduction section.

3.2 System Design and Architecture

3.2.1 Design Goals

Stormy benefits from Cloudy2 as much as possible. This is especially true for the infrastructure (message, gossiper and others) and the different components involved in load balancing (load balancer, load calculator, repartitioner, cloudburster and cloudburst executor) that are already provided by Cloudy2. This way Stormy does not need to re-implement those components and just needs to make sure that it behaves as expected.

The design goals for the streaming part are described in this paragraph. Queries are one type of a stream consumer, the other two are shared stores and outputs. Most statements made for queries henceforth are valid for the other stream consumers too even if not explicitly stated.
In a Stormy cluster one server is supposed to handle one or many queries. To assure the availability of queries in case of failures each query is replicated, therefore one query is served by several servers. Among the instances of the query there is one master, the other instances are replicas. Each query has an associated output stream. Only the master disseminates the query output results to its associated stream. The replicas are processing the input events but only cache the query output and do not disseminate them (as long as there are no failures, see 3.5 Failures Scenarios).
The subscription of several queries to the same stream (1:m relationship) is handled in Cloudy2 already by using range requests. A query having several input streams (n:1 relationship) is supported by the alias mechanism implemented directly in the Cloudy2 architecture. This mechanism is described in section 3.2.2.
The router component determines where to find the stream consumers subscribed to a given stream, i.e. where to forward the events to. The routing itself is executed by the protocol component. In addition, the protocol is in charge of assuring the strong consistency and of taking care of failure handling. Incoming events are ordered by the protocol by using version numbers (see 2.2.4). The store component is responsible for the execution of the procedure associated with the stream consumers (e.g. query execution in case of queries, event sending in case

of outputs and storing in case of shared stores).

Table 3.1 shows the components used in Stormy. The bootstrapper, the seed preparator and the cloudburst executor each lists two implementations. The implementation is chosen depending on whether Stormy runs in a local network (default / local) or as an Amazon EC2 instance (EC2). The two components marked by a star (⋆) are new components especially designed for Stormy.

Component	Implementation
Messaging System	Cassandra
Gossiper	Cassandra
Leader Election	ZooKeeper
Router	Skip List Router
Protocol	Master Protocol⋆
Store	Stormy Store⋆
Bootstrapper	Default / EC2
Seed Preparator	Default / EC2
Load Balancer	Maha
Load Calculator	Combined
Repartitioner	Token
Cloudburster	Hotspot
Cloudburst Executor	Local / EC2

Table 3.1: Stormy's Components

3.2.2 Basic Design

Two new components have been developed for Stormy, the master protocol and the Stormy store. A new external interface allows to access Stormy by the interface described above (3.1.2). The typical workflow of a request from a client passes first the external interface, then the protocol and is processed by the store in a last step. This works well in case of Cloudy2 because the external interface, the protocol and the store are all providing the same API, i.e. get, put and delete. Stormy's external interface however provides a different API. The calls to the Stormy API first have to be transformed to the three types of requests supported by the protocol and the store. For that reason the Stormy processor was created that works as a mediator between Stormy's external interface and the protocol.

Incoming requests from external clients (e.g. push events) are accepted by the Stormy external interface and just forwarded to the Stormy processor. The external interface does not provide any additional handling; none is needed because the Stormy external interface and the Stormy processor both have the same interface. The Stormy processor translates incoming Stormy requests to one or more Cloudy2 requests (get, put or delete) and sends them to the master protocol. The protocol in the end forwards the Cloudy2 conformant request to the Stormy store which is then executing the appropriate action(s). The store is the component that really executes the queries, processes incoming events or sends events to an output. The Stormy workflow is illustrated in figure 3.2. The reason why the

functionality of the Stormy processor is not directly included in the Stormy external interface is that from inside the system (e.g. store, protocol) no access to an external interface is possible. This is due to the modular architecture of Cloudy2. In case of Stormy for example the query instance produces output events and these events have to be pushed again to the system. This operation is performed by invoking the appropriate method in the Stormy Processor.

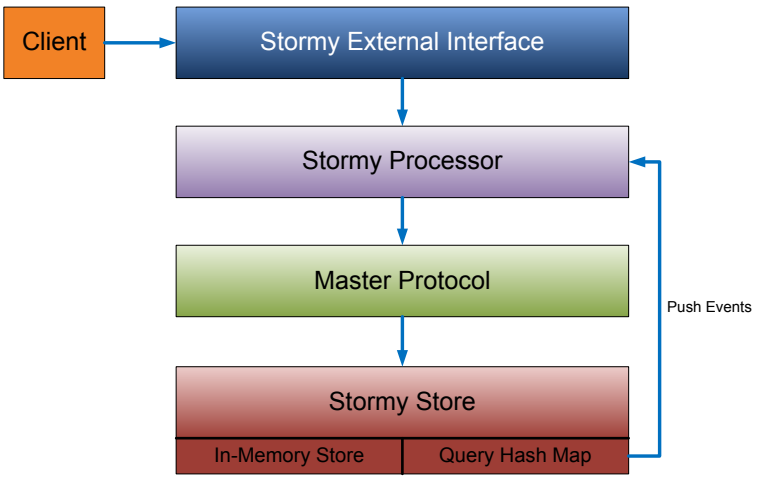

Figure 3.2: Stormy Basic Workflow

The Stormy Store is mainly divided into two parts. On the one hand it can be accessed as any other store and used for normal get, put and delete requests. For this the Stormy store includes the in-memory store described as one of the existing store components (see 2.2.3). On the other hand it takes care of query execution involving the registration and start of queries and the handling of push event requests (i.e. make sure that the events are fed to the queries or the registered outputs).

In the following sections the workflows and processes for each request supported by the interface are presented. For a review of the interface see the problem statement in section 3.1.2 above. These processes mainly involve the Stormy processor and the Stormy store, even though each request first traverses the master protocol before reaching the store. But for the descriptions below the reader can assume that each request put into the protocol is directly forwarded to the store because the protocol does no query related handling. A brief overview of the master protocol is given at the end of this section. The detailed control flow inside the protocol is explained in sections 3.3 to 3.5. Prior to the summary of the protocol the alias mechanism of Cloudy2 is analyzed. Aliases solve the problem of n:1 relationship from streams to a query.

Hinted DPI

Before starting with the detailed function explanation of the interface, some specific information is given about the DPI and how it is represented in this chapter.

The DPI has a field called hint that provides the possibility to transport additional information with each DPI. Stormy uses this field to identify certain kinds of DPIs, usually to advice the store to carry out special operations on the arrival of such a hinted DPI. These hints are set by the Stormy processor. The Stormy store internally uses the in-memory store as already mentioned above. Non-hinted DPIs (i.e. those having set no hint) are just forwarded to that internal store without additional processing. This enables the usage of Stormy as a simple key-value store with strong consistency.

As compared to the Cloudy2 chapter, where hints had an inferior position, they are important in Stormy. The syntax used to describe a DPI with key *key*, value *value* and hint *hint* is `DPI<key, value, hint>`. The type field is never used in Stormy and thus not shown.

Register a new Input Stream

The registration of a new input stream is invoked with the description of the input stream and provides the stream ID of the newly registered stream. The Stormy processor generates a random stream ID. This stream ID is a sequence of characters with length 12 (default, is variable). The sequence only contains letters (A-Z, a-z) and numbers (0-9) for readability reasons. The Stormy processor needs to reserve this stream ID to assure uniqueness. The reservation process also checks if this stream ID has not been reserved before (to assure uniqueness of stream IDs).

The reservation is performed by sending a put request with `DPI<[generated stream ID], [stream description], ID-input-stream>` to the protocol. The parameters given in square brackets are replaced by the corresponding values (e.g. `DPI<GD4JV3SG36DU, my first stream, ID-input-stream>`). The DPI uses the hint ID-input-stream. This hint declares that the DPI references a registered input stream. The hint is necessary to determine what is going to be deleted when deleting a certain stream ID. The description of method to deregister a stream provides more information. When receiving this put request, the Stormy store just forwards it to the internal in-memory store. No other actions are executed by the Stormy store. The internal store, however, might reject the request if this stream ID was already reserved. The next paragraph describes how this works.

Version Flag: The version control mechanism of Cloudy2 provides the possibility to set a special version flag (called the obsolete version flag). If this flag is set, the comparison of the DPI versions in the store always states that the one in the store is newer, independent of the real versions of the DPIs. Due to this an obsolete version exception will always be thrown when the store already contains a DPI with the same key. If no DPI was stored before with the same key, no versions can be compared and a put request is executed successfully. A put request with the obsolete version flag can therefore be called a “put if absent” request.

When registering a new input stream the issued put request uses this flag. If the stream ID was reserved before (i.e. the store already contains a DPI with the same generated stream ID), an obsolete version exception is thrown and the process is repeated by generating a new stream ID and sending the new put request.

Register a new Shared Store

A shared store has to be uniquely identifiable. For that reason each shared store has an associated store ID after the store was registered. Even if a shared store does not provide an output stream, the store ID is treated the same way as the stream ID: as a unique identifier. Therefore we use the term stream ID from now on, referring to both stream ID as well as store ID.

The registration of a shared store is similar to the registration of a new input stream. The only difference is the DPI used for the put request. The one used for the shared store is `DPI<[generated stream ID], [shared store], ID-shared-store>`. The value ([shared store]) contains the description of the store with all information needed by the query to use the store (input stream ID, number of seconds to store events, block size and other attributes). As before, the obsolete version flag is set for the DPI version. The Stormy store only forwards the incoming put request to its internal store. The hint *ID-shared-store* declares that the DPI references a registered shared store and marks the associated stream ID (store ID) as reserved. Obsolete version exceptions are caught and handled as described for the input stream registration.

Shared stores are not shared in the sense that all queries using the store are accessing the same instance of the shared store. Instead, each query instance has its own local version of the shared stores. These instances are equal because all of them are processing the same incoming events, but they are not synchronized. That is once the local shared store instances are created by the query these instances are not in contact with the other instances of the same shared store.

Register a new Query

A new query needs a new output stream ID. The stream ID is reserved as in the two descriptions above. The DPI for the put request is `DPI<[generated stream ID], [various IDs], ID-query>`. This requests reserves the stream ID for the query output stream. The store handles the request as in the other types of two reservations. The value of the DPI contains the input stream IDs of the query, the IDs of the shared stores and their input streams IDs (if the query uses any shared stores). The query deregistration process requires this information to locate the query and for the removal of the aliases used by the query. The deregistration of streams is described afterwards.

As opposed to the two other requests for input stream and shared store, the query itself has to be registered and started. Input streams and shared stores themselves do not have an active part that is processing events; queries do. Before setting up this query instance the Stormy processor needs to retrieve the description of the shared store that was registered earlier on. To retrieve it, a get request with `DPI<[store ID]>` for each of them is executed. The Stormy store forwards this non-hinted request to the internal store and returns the DPI that has been requested. The request is a typical get request as it would be the case for any other system configuration than Stormy. The returned DPI is `DPI<[store ID], [shared store], ID-shared-store>` and the value provides the required information for the query.
A query description including the retrieved shared stores is created and `DPI<[`

`query key], [query description], OP-query>` is prepared for a put request. The DPI key ([query key]) is the concatenation of the first input stream ID, a delimiter (#) and the output stream ID generated and reserved in the first request. Example: the key of a query with the first input stream ID AAA and the output stream ID BBB is `AAA#BBB`. The key format `[stream ID to subscribe to]#[ own identifier]` is the way how stream consumers (queries, outputs and shared stores via aliases) subscribe to a stream. The identifier must uniquely identify the stream consumer, therefore the query is using the output stream ID here. Before this put request is send, the aliases are configured. The appropriate section further below provides an explanation of the alias functionality and explains how queries are using them by an example[1].

After the aliases are configured the put request above with the hint OP-query is sent to the protocol. The Stormy store receiving this request first forwards the request to its internal store (as it was the case with all previous requests). Furthermore, the query description provided in the DPI value is used to create a query instance. The threads responsible for input and output event processing are started and a reference to the query instance is stored in the internal query hash map. This query hash map contains all running query instances; an instance can be retrieved by the query output stream ID. The references to the query instances are required to forward incoming requests to the appropriate queries (see push event method).

Deregister a Stream

This method is used to deregister one of the three elements above. For an input stream the input stream ID has to be provided, for a shared store the store ID and for the query the query output stream ID, i.e. those IDs that uniquely identify the elements.

Before the deregistration takes place the Stormy Processor verifies that no other stream consumers (query, shared store or output) have subscribed to this stream. For this verification two steps are required. The first one checks the alias source entries if they start with the given stream ID. If so, the stream is still in use and cannot be removed. For the second check a get request with `DPI<|[stream ID ]|>` is performed. Because it is a non-hinted DPI no special processing occurs at the storage level, i.e. it is a normal prefix range get request returning all DPIs in the store that have keys starting with [stream ID]. This get request will return a set of DPIs. This set should contain exactly one DPI, the one that identifies the stream ID reservation: `DPI<[stream ID], *, ID-*>` (the * is a wildcard). If more than one DPI is returned, e.g. `DPI<[stream ID]#*, *, OP-*>`, a query (or an output) still uses this stream and it cannot be deregistered yet.

The DPI retrieved from the get request determines the type of stream to deregister. For input streams and shared store no additional processing is necessary besides the deletion of the reservation DPI. For queries the query instance has to

[1] Brief summary: Events sent to the second input stream of a query (or to the input stream of a shared store used by that query) have to reach the query through its first input stream (because it cannot directly subscribe to several inputs streams). For that reason an alias from the source `[second input stream]#[output stream ID]` to the target `[first input stream]#[output stream ID]` is created that will take care of the appropriate forwarding.

be removed as follows:

Query Instance Removal: If the hint of the received DPI indicates a query (ID-query) the value of this DPI contains the input stream IDs of the query and the IDs of the shared stores and their input stream IDs. The shared stores with their input stream IDs and the additional input stream IDs of the query are used to remove the registered aliases that were created during query registration. The query instance is located at `[first input stream]#[stream ID]`. To order the deletion of this instance a delete request with `DPI<[query location],, OP-query>` is issued. The Stormy store forwards this delete request to the internal store to remove the DPI there. Additionally, the query instance is retrieved from the query hash map (by using the output stream ID provided with the DPI key), the query threads are stopped and the entry in the hash map is removed.

The last step to complete the deregistration process is to delete the reservation DPI by sending the non-hinted delete request `DPI<[stream ID]>`.

Register an Output

An output is responsible for forwarding incoming events it subscribed to the target address (hostname / IP address and port). No stream ID has to be reserved for an output because an output is not referenced again within Stormy (it is a terminal). It just consumes incoming events on the stream it subscribes to, but does not produce any new ones within the system. To register an output, the Stormy processor sends a put request with `DPI<[stream ID]#[target identifier], [target specification], OP-output>`. Note the format for stream subscription. The hint identifies the DPI as an output. The target identifier is a unique identifier for a given target. For simplicity (and because it is unique for one input stream) the representation of IP address and port are used. Example: the target 192.168.4.12 wants to receive events on port 4344 that are on the stream AAA. The corresponding DPI is `DPI<AAA#192.168.4.12:4344, [target specification], OP-output>`. The DPI value again contains the IP address and the port but in a more compact representation. No special handling is required by the Stormy store, the put request is just forwarded to the internal store.

Deregister an Output

Deregistering an output is performed by sending a delete request with `DPI<[input stream id]#[target identifier]>`. The Stormy store, when receiving such a request, forwards the request to its internal in-memory store to delete the DPI there. Other processing is not required.

Push Events

As opposed to the different requests explained above, the push event request is performance critical. The above requests are executed comparatively rarely and they do not have to be processed as fast as possible. They need to prepare the system in a way that incoming events can be processed effectively. The Stormy processor transforms the push event request for the stream ID [stream ID] to a put request with `DPI<(|[stream ID]#|, [events], OP-push-events>`. The

events are contained in the DPI value. The hint OP-push-events is an identifier for the Stormy store to execute the appropriate procedure. The range specified for the DPI is a prefix range. The reason why to use a prefix range becomes obvious when considering how the stream consumers subscribe to streams. Recall the format `[stream ID]#[identifier]` for a subscription. The prefix range contains all DPIs whose key starts with the range identifier (`[stream ID]#`).

The single put request sent by the Stormy processor is received by the Stormy store and not forwarded to the internal store. All other requests have always been forwarded to the store. This one is not, because there is no reason to store incoming events, they only have to be processed by the stream consumers. The first step for the store is to execute a get request to its internal store with the received DPI. This request will return all DPIs belonging to the stream consumers which subscribed to this stream.

The DPIs in the result set of the get request either have the hint *OP-query* or *OP-output* and thus are representing a query or an output writer. For each DPI the following action is executed depending on their hint:

Output: `DPI<[stream ID]#[target identifier], [target specification ], OP-output>` is the DPI that is in the set for an output. A connection to the target is opened; the DPI value provides the required target specification (IP address and port). After the connection has been opened the events are emitted and the connection is being closed again.

Query: The according DPI for a query is `DPI<[first input stream ID]#[ query output stream ID], [query description], OP-query>`. The DPI key is split up at the delimiter (#). The second part provides the query output stream ID that is used to retrieve the query instance from the internal query hash map. The events are then forwarded to the query instance. The initial put request provides the events (in the DPI value) and the real input stream ID[2]. A query instance provides an event iterator for each input stream (e.g. a query with one shared store and two input streams provides 3 event iterators). An event iterator queues events until they are consumed by the query engine or the shared store. When the query instance receives events it forwards them to the appropriate event iterator based on the real input stream ID.
One of the query threads fetches new events from the query and pushes them to the query output stream. This is performed by invoking the push events method in the Stormy processor.

Request Examples

This example should help to provide a better understanding of the internals of the Stormy store. For readability the stream IDs used in the examples have a

[2]The real input stream ID is the one that was used for the initial push request. In case of a non-aliased request this stream ID is the same as the one specified in the prefix range (e.g. for the prefix range `[AAA#]` the real input stream ID is AAA). If the request was aliased, the input stream ID is retrieved from the hidden DPI field used by the alias mechanism. Aliases are explained in detail after the request examples.

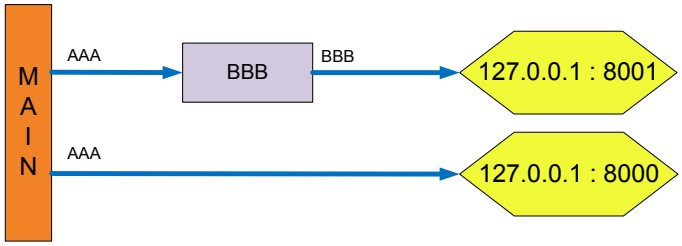

Figure 3.3: Example Stream Graph

length of only 3. Assume there is only one node in the cluster (for simplicity) and the following registrations take place:

1. Input stream with the description "main": stream ID AAA is returned
2. Query that subscribes to AAA: query output stream ID BBB is returned
3. Output (127.0.0.1:8000) subscribing to stream AAA
4. Output (127.0.0.1:8001) subscribing to stream BBB

Figure 3.3 illustrates the corresponding stream graph. The internal in-memory store contains these DPIs:

- `DPI<AAA, main, ID-input-stream>`
- `DPI<BBB, AAA, ID-query>`
- `DPI<AAA#BBB, [query description], OP-query>`
- `DPI<AAA#127.0.0.1:8000, [target specification], OP-output>`
- `DPI<BBB#127.0.0.1:8001, [target specification], OP-output>`

The internal hash map for the queries contains one mapping from BBB to the query instance. An incoming push event request is transformed by the Stormy processor to a put request with `DPI<|AAA#|, [events], OP-push-events>`. The Stormy store executes a get request with the passed DPI, i.e. for the prefix range `|AAA#|`. This get request returns the following two DPIs from the internal store (third and fourth DPI from the list above):

- `DPI<AAA#BBB, [query description], OP-query>`
- `DPI<AAA#127.0.0.1:8000, [target specification], OP-output>`

The first DPI represents a query (OP-query). The query output stream ID is extracted from the DPI key resulting in BBB. This key is used to retrieve the query instance from the query hash map. The events are forwarded to the query instance. The query has two event iterators, one for the input stream ID AAA and one for BBB. The events are therefore processed by the second event iterator (BBB).
An output is represented by the second DPI (OP-output). The DPI value contains the target specification and points to the IP address 127.0.0.1 and port 8000. A connection is opened to the target and the events are emitted.

Aliases

Aliases in Stormy are used to support n:1 relationships from several streams to one query. The mechanism of aliases is an inherent part of Cloudy2 already, but there are no use cases in a key-value store. That is why aliases are explained here. The purpose of an alias is that a request for a DPI key A is routed additionally to the key B. To achieve that, an alias from the source key A is registered for the target key B. The initial target information (in this example A) will not get lost since the DPI sent to B contains the initial key A in a hidden DPI field. The store must take care of this hidden fields itself.

Protocol Handler: Each request to the protocol component first passes the protocol handler. The protocol handler checks if there are registered aliases that match the given request key. If this is the case, the protocol handler invokes the protocol component once with the original DPI and a second time with the modified DPI whose key was changed to the new target. If multiple aliases are registered for the same key (but for a different target), the protocol is called more than only two times. The protocol invocations occur simultaneously and the protocol handler awaits the return of all invocations. Put and delete requests do not return a result, only the confirmation of their execution. Results from get requests are merged.

Metadata: The mapping from aliases to targets is maintained in the metadata. If an endpoint registers a new alias the metadata handler (2.2.4) takes care of storing the new metadata version. Other endpoints will receive the new metadata version number on message transfers and update their stale version immediately. This assures that the metadata is always up-to-date and available on all endpoints.

Usage in Stormy: A query with output stream ID AAA subscribes to the streams BBB and CCC (left side of figure 3.4). Without aliases, the query could only be stored and reached at one of the keys `BBB#AAA` or `CCC#AAA`. With aliases, the query is stored under the first input stream (`BBB#AAA`) and an alias is registered from the source `CCC#AAA` to the target key `BBB#AAA`. This way incoming events for stream CCC are additionally forward to the query AAA located at `BBB#AAA`. A similar approach is used for the shared stores. Each query using a shared store has a local instance of this shared store, because remote access to a shared store is not supported by MXQuery (and it would not be efficient at all). Such a query registers an alias from the store key to the query key. Example: a shared store DDD subscribing to a stream EEE is used by a query FFF (subscribed to an input stream GGG). This example is illustrated on the right side of figure 3.4. An alias from the source `EEE#DDD` to the target `GGG#FFF` (query key) is registered assuring that all events to the shared store are forwarded to the query too.
Stormy uses prefix ranges for push event requests. A request for the range `|EEE#|` matches the registered alias `EEE#DDD` and causes the execution of the initial request (that will push the events to all stream consumers subscribed to the stream EEE) and additionally the execution of the request to `GGG#FFF` that will only push the events to the registered query FFF.

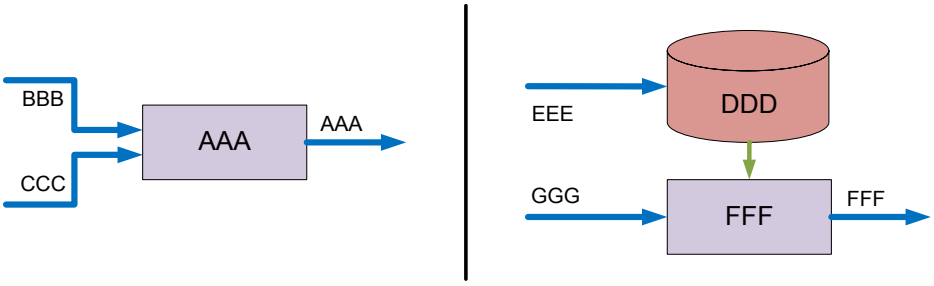

Figure 3.4: Two Alias Examples

Master Protocol

The following description just gives a brief overview of the master protocol. The detailed workflows in the master protocol are explained in the different state scenarios following in the next three sections. The master protocol assures strong consistency with a variable replication factor. The default replication factor is 3. The master protocol, as the name suggests, has one endpoint that is the master. This master endpoint is not unique for a whole cluster. Instead, the master is specified for each DPI, meaning that `DPI<Jane>` may have another master than `DPI<Doe>`. The key of a DPI, or the string representation of the range if one is specified, is the identifier used to determine the master. This is the identifier that is used when asking the leader election component to confirm or deny the ownership of a write lock.

The leader election component guarantees that only one endpoint holds the lock for a specific DPI while the master protocol determines who this is. A preference list returned by the router does contain 3 endpoints (if the replication is 3 as defined above). The first endpoint in the list is called the endpoint in charge, because this endpoint is the one that should mainly handle the request, i.e. is responsible. The other two endpoints are the replicas. What the master protocol tries to assure is that the endpoint in charge and the master for the corresponding DPI are the same. In an ideal world where every endpoint has fresh routing information right after a change, the election of the master would not be necessary. Due to failures and message transfer delays that occur in real world examples, some endpoints might have stale routing information and one of these endpoints would come to the conclusion that endpoint B instead of endpoint A is the endpoint in charge. Such a situation would violate the consistency requirements. This is why and where the leader election is used. Because only one of the two endpoints A and B can hold the lock, the mismatch is detected and handled accordingly.

The handling of requests in the master protocol does not distinguish between get, put or delete requests. What differs of course is the operation invoked at the storage layer and the result propagated back from one endpoint to the one that called the master protocol (in the following referenced to as the caller). The result of a get request is a set of DPIs whereas put and delete requests do not provide a result object. What they respond is the status of the operation invocation, i.e if the operation failed or was executed successfully. Therefore they either send

an acknowledgement or in case of unexpected behavior the caller of the protocol method is informed thereof so to react accordingly. For example the query thread responsible for sending the output events of the query resends the events if it does not receive an acknowledgement for the last request (e.g. because a timeout happened).

The next three sections describe the detailed workflow for normal states, failures and scenarios where the load balancer is active and moves the position of endpoints or adds new nodes to the cluster. The whole master protocol is explained in the next section with a typical data flow in a normal state. The other two sections provide specific information and workflows for their scenarios.

3.3 Normal State Scenarios

3.3.1 State Description

For normal states we assume a scenario where the cluster size is fixed, the position of the endpoints is not changed and only one node is responsible for a prefix range. Furthermore, no messages sent to other endpoints cause a timeout. In a stable and faultless system the endpoint in charge of a DPI always holds the lock, i.e. is the master. All endpoints are assumed to have the same (valid) routing information.

The Stormy push event request is the one occurring most often. It is represented by a put request with a prefix range DPI (e.g. `|AAA#|`) and contains the events in the value field of the DPI. The explanations and figures assume such a put request, but they are valid for the other types of requests too. A request is determined by the request type get, put or delete (also referenced to as *request method*) and the DPI passed along. The terms request and DPI are sometimes used as synonyms even though they are not. But as described at the end of the last section the processing does not distinguish between the different request types and therefore the "request" does not provide more useful information than the "DPI".

The next section explains all the parts of the master protocol to give the reader an overview. The influence of some protocol parts and why they are designed that way may not be explained in detail in this section but in the corresponding sections for the other two scenarios.

3.3.2 Master Protocol Overview

The design of the master protocol requires that all requests are processed by the corresponding endpoint in charge (which ideally is the master). The router determines the endpoint in charge and the associated replicas for one request by its preference list. To follow the design requirement all requests are forwarded to the responsible node for further processing. An endpoint not in charge of a DPI forwards the request to the one in charge. The master protocol achieves that the master always has the newest up-to-date state for its responsibility range. This implicates that when the master changes (for example because of load balancer activity) the new master makes sure that it fetches the newest state from the current master (see section 3.4.4).

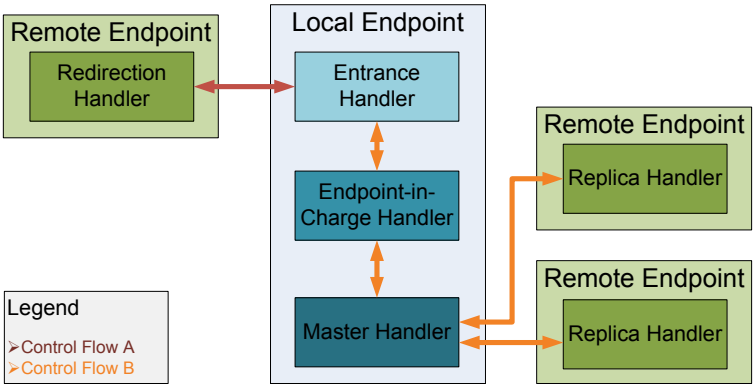

Figure 3.5: Master Protocol Overview

A typical request passes the following three local handlers in the master protocol: entrance handler, endpoint-in-charge handler and master-handler. Two remote handlers respond to messages sent by the local handlers (as shown in figure 3.5).

Entrance Handler

The entrance handler (figure 3.6) serves as the main access point to the protocol from other components (e.g. external interface or query output process). The task of this handler is to decide whether to execute the request locally or to redirect the request to another endpoint because the local endpoint is not the endpoint in charge of the DPI.

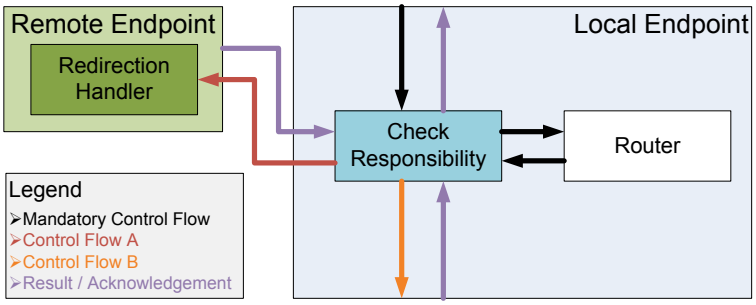

Figure 3.6: Entrance Handler

As a first step the entrance handler asks the router for the preference map. The result of the router invocation was assumed to be a preference list before, but the router always returned a preference map. This map contains DPIs that are associated with their preference lists. In all cases handled up to now only one DPI (the request DPI) was contained in the map. This is why the use of the term preference list is justifiable in these cases. In the load balancing scenario the returned map may contain several DPIs because more than one endpoint is responsible for the given prefix range (e.g. the router assigned a position within

that range to an endpoint). Detailed explanations follow in the corresponding section.
The next step is comparing the local endpoint with the first endpoint in the preference list (in normal state scenarios there is only one mapping DPI to preference list). If these endpoints are the same, i.e. the local endpoint is the endpoint in charge, the request is passed to the endpoint-in-charge handler. Otherwise, if they differ, the request is redirected to the other (remote) endpoint.
Such a redirect request can time out when the endpoint in charge fails (master failure scenario). If this happens the entrance handler repeats the process with the second endpoint in the preference list, meaning that the second endpoint is compared with the local one and the request is then either passed to the next handler or redirected. This is repeated as long as no successful redirection was reported (in case of a replication factor 3 at most 3 times). A failure in a redirection is therefore not considered as a failure of the whole request, unlike a failure in the endpoint-in-charge handler that is equivalent to a request failure. In case of 3 (= replication factor) redirect failures or one endpoint-in-charge handler failure the request is considered as failed and the caller of the protocol is informed thereof. The redirection handler on the remote side takes care of processing the redirected request. This handler is described after the three local handlers.

Endpoint-in-Charge Handler

A request is passed to this handler if the endpoint is responsible for the DPI. In case of failures (redirect timeout) the local endpoint might be the second or third choice for the endpoint in charge, but it is considered as in charge for this request now.

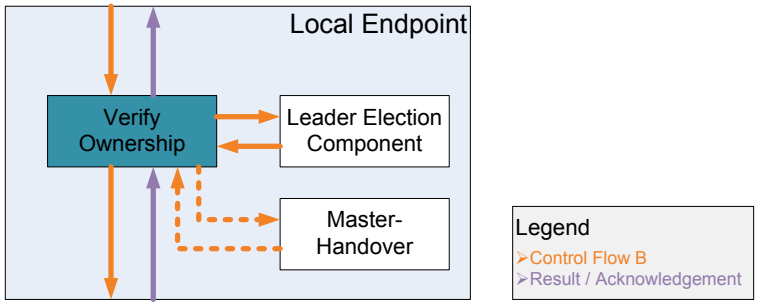

Figure 3.7: Endpoint-in-Charge Handler

The only task of the endpoint-in-charge handler is to guarantee that the local endpoint is the master for the DPI. Being master is assured by acquiring the exclusive write lock (using the leader election component). If the ownership of the lock is not confirmed, the master-handover procedure (3.4.4) is applied to transfer the ownership to the local endpoint. In this scenario the local endpoint is always the master. As soon as the local endpoint holds the lock, the master handler is invoked. The workflow within this handler is illustrated in figure 3.7.

Master Handler

The master handler is reached only when the leader election component confirmed the leadership for the DPI. Its task is to execute the requested operation and to make sure that the replicas do that as well. With a replication factor of 3 the preference list contains three endpoint entries, the local endpoint and the two replicas. The replicas are usually at position 2 and 3 in the preference list (at least in the normal state scenario).

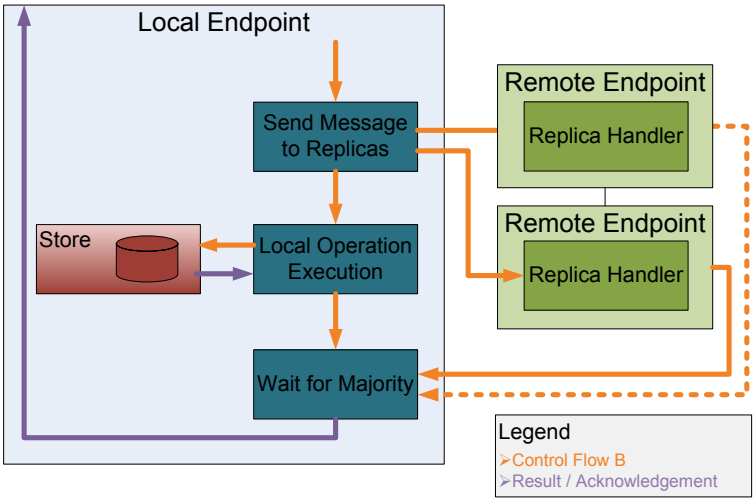

Figure 3.8: Master Handler

Figure 3.8 gives an overview of the process in the master handler. The first step of the handler is to set a version number for the DPI. This version number is incrementing and unique. It is used to assure the preservation of the event order in case of queries with several input streams. Afterwards the master handler creates and sends a message to the replicas that specifies the request type, the DPI and additionally a hash of the master's state. The transferred hash assures that an outdated replica will notice that its state is stale and then invoke the datastreamer (2.2.4) to fetch the fresh state directly from the master before it executes the operation. The hash used is described in the failure scenario section (3.5.1). On a remote Stormy instance the replica handler accepts the sent message. This handler is explained in the next section.
After sending the messages to the replicas the master handler executes the request locally by invoking the corresponding method of the store. For a push event request the handler calls the put method with `DPI<[stream ID]#, [events], OP-push-events>`.
The last step of the master handler is to wait until it receives the responses from the majority. The majority for a replication factor of 3 is 2. Because the handler always has its own response it only needs to wait for one additional response from a replica. As soon as the replica response(s) arrive(s), the request handling is considered as successful. If no or not enough responses are received (e.g. due to a replica failure) the request is considered as failed because of a timeout and the handler informs the caller thereof.

Replica Handler

On receiving a request message this handler verifies that its own state is up-to-date by using the hash provided in the message. If its local state is stale, the replica fist updates itself before continuing.
A replica tries to acquire the exclusive write lock already acquired by the master too. Even so it will never hold the lock, the procedure allows to guarantee that a newly elected master (e.g. when the old one failed) can assure that it has the newest state. Details are explained in the corresponding sections of the failure scenario states.
The last step is the execution of the request, i.e. by invoking the appropriate method of the store. Subsequently an acknowledgement is sent to the master endpoint. Even in case of a get request no result containing the set of DPIs is sent back to the master. This is unnecessary because the master always has the newest state and therefore no benefit emerges from sending back the results of a get request.
The master protocol can of course be used without the Stormy store. If the Stormy store is activated, the replica handler executes additional processing to remove temporarily buffered events in the query output thread (see 3.5.1).

Redirection Handler

The redirection handler accepts messages from the entrance handler of another endpoint. The other endpoint redirected the request because this endpoint is responsible according to its routing information. Before the handler starts processing a redirected request, it checks if it is responsible for the request at all.

The responsibility of the local endpoint is verified as in the entrance handler by first invoking the router to retrieve the preference map (assuming a map with one entry, therefore a preference list). The typical and expected case is that the first endpoint in the preference list is the local endpoint. Under the assumptions of a normal state this will always happen. It means that the other endpoint's routing information was correct and the local endpoint is responsible for the DPI. The local endpoint then passes the request to its endpoint-in-charge handler for further processing. After the call an acknowledgement or a failure notification is sent back to the sender of the redirect message.
There are two more cases briefly mentioned here but explained in detail in the corresponding scenarios. They are caused by a stale routing table (due to a preceding load balancing step) or by a master failure. The first case applies when the local endpoint is not in the preference list at all. In the second case the local endpoint is in the preference list but not at the first position.

3.4 Performance Scenarios

Scalability is one requirement of Stormy. Balancing the load between the existing endpoints in the cluster does not provide scalability but is an essential operation to support it. Cloud bursting is what really makes a system scalable. Without load balancing a cluster does not benefit from cloud bursting because a newly joined member would not take over existing load from other nodes. The load balancing additionally lowers the cost by reducing (not increasing) the number of endpoints

in the cluster: if two endpoints out of a dozen in the cluster are heavily loaded, there is no need to do cloud bursting as long as the load can be balanced among the existing endpoints. The section about the performance scenarios explains the behavior of the master protocol in cases of load balancing, cloud bursting and cloud collapsing (the removal of endpoints due to an underloaded system).

Most descriptions and examples below are assuming that the skip list router is used who is distributing the endpoints on a ring structure. In fact any other router could be used instead, but a scenario explained by a concrete example is easier to understand than one that is generalized.

3.4.1 Range Split Ups

In a new cluster without any previous load balancing steps the probability that queries subscribing to the same input stream (e.g. LLL) are located on the same endpoint (i.e. one endpoint is responsible for all the queries) is very high. If the load induced by these queries is to high for a single endpoint, the load balancer tries to hand over some of the load to another one. In case of a DHT as the underlying routing information (skip list router implementation) there are two ways this is handled: either the position of the heavily loaded node is moved counterclockwise in the ring to a position within the range (`|LLL#|`), which causes the successor of the overloaded node to take over some of the load. The second possibility is that the overloaded node tells its predecessor to move clockwise to a position within the range such that the predecessor takes over some of the load. Flavio Pfaffhauser's thesis describes these scenarios in detail [25]. Once the token of an endpoint is within the range of a request, the responsibility for the range is split between the two endpoints. If the range is split up, the router will return a preference map containing at least two entries.

Examples

Scenario: range request `|LLL#|`, endpoint A has the token J, endpoint D the token M and endpoint B the token `LLL#K` that was assigned to it during a previous load balancing step. The example and the next one are illustrated in figure 3.9. The router will return a preference map containing two entries (assume replication factor is 1 for simplicity):

- `DPI<|LLL#|LLL#K]>` → endpoint B
- `DPI<|LLL#|(LLL#K>` → endpoint D

`|LLL#|LLL#K]` and `|LLL#|(LLL#K` are limiting prefix ranges. The first specified range contains all keys with the prefix `LLL#` but only up to `LLL#K` (including). This range could also be expressed by `[LLL#,LLL#K]`. The second range contains all keys starting with `LLL#K` (exclusive) that have the prefix `LLL#`. There is no other way to express such a range. If there would be an endpoint C in the example above with the token `LLL#S` (i.e. between endpoint B and D within the range `|LLL#|`), the preference map would be:

- `DPI<|LLL#|LLL#K]>` → endpoint B
- `DPI<(LLL#K,LLL#S]>` → endpoint C

- `DPI<|LLL#|(LLL#S>` → endpoint D

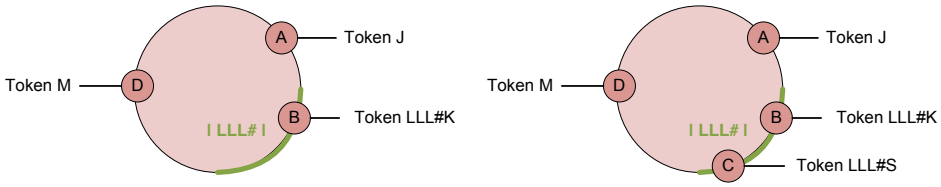

Figure 3.9: Example Range Split Up Situations

Master Protocol Handling

Range split ups are detected by the entrance handler of the master protocol. If the router returns one mapping there is only one control flow. In case of a range split up the router returns several DPIs (and their associated preference lists). For each of the returned DPIs the entrance handler sends a redirect message to the endpoints in charge and waits for the results of all endpoints. If the request has the type get, the result sets are merged to one large set of DPIs. Messages are sent even if the local endpoint is one of these endpoints in charge because this facilitates the concurrency issues tremendously. A message sent only locally does not add much overhead to the overall processing because the messaging system is optimized to not serialize and deserialize local messages. The processing described for range split ups is quite similar to the one described for handling aliases.

3.4.2 Load Balancing

Load balancing steps influence the master protocol due to the changes in the routing table that cause new responsibility assignments for certain DPIs. Due to load balancing also new replicas might be assigned. This is not discussed further because the datastreamer updates the replicas and additional processing is not necessary.

Stale Routing Table

A change in the routing table is distributed by the gossiper, which may, depending on the cluster size, take some time until all endpoints are updated. Stale routing information does not disturb the systems but the performance might decrease during the update phase due to the additional redirections that are necessary.

As an example assume that the routing table of endpoint A was not updated yet and it sends a redirect request to endpoint D. Before its load balancing step endpoint D was responsible for the DPI sent with the request, but it moved counterclockwise and is not responsible anymore. Endpoint D receives the redirected request and verifies the responsibility. The preference list returned from the router of endpoint D contains the endpoints E, F and G. Thus endpoint D is not only not responsible as an endpoint in charge but it is not responsible at all for this DPI (even not as a replica). For that reasons it forwards the redirect request to the endpoint in charge E. Endpoint E receives the request and will, after the

successful execution, send the response directly to endpoint A. Endpoint E does not even know that the message was forwarded by endpoint D.

The load balancer guarantees that the endpoints directly affected by a load balancing step are informed thereof. This assures that endpoint D in the above example knows that endpoint E is the new endpoint in charge. Otherwise loops in the redirection flow might be possible.

Replica as new Endpoint in Charge

Assume that endpoint A is responsible for `DPI<|LLL#|>` and endpoints B and C are replicas. Assume further that the load balancer assigns the new token K to endpoint A. The new endpoint in charge of the DPI is therefore endpoint B with the replicas C and D. The load balancer invokes the repartitioner that will send the necessary states to endpoint D that did not have to care about the range `LLL#` yet (and therefore does not have any DPIs in this range).

A new incoming push event request will at some point end up in the endpoint-in-charge handler on endpoint B. The leader election component will not grant access to the write lock for the range because endpoint A still holds it. This is where the master-handover procedure comes into play (see 3.4.4). It assures that the current master releases its lock and updates the state of the new master if that one is outdated.

Fresh Endpoint as new Endpoint in Charge

If the load balancer moves a node (named T) clockwise on the ring this moved node is the new endpoint in charge for all DPIs in the range that it "skipped". The old endpoint in charge still is the leader. Therefore a incoming request will end up in the endpoint-in-charge handler of node T as that happened in the scenario described before.

3.4.3 Cloud Bursting

This section includes the scenarios for cloud bursting meaning the adding of a new server to the existing cluster as well as the scenario when a server actively involved in the cluster is terminated due to a low overall load. Cloud bursting takes place if the overall load in the cluster is to high to be handled by the endpoints itself. A new server will be started either at a random position or at a heavily loaded one. The master protocol does not provide any additional handling for these situations because they are already caught with the load balancing situations or in the ones concerned with failures (as described in the next section).

Adding an Endpoint

The bootstrapper component is in charge of joining a new node to an existing cluster. Before finally announcing that there is a new endpoint, the bootstrapper assures that the datastreamer fetches the DPIs within the range assigned to the new endpoint. This enables a new node to directly participate in the cluster without the need to update parts of its state on each incoming request. The joining of a new node does not impose more work on the master protocol than a

load balancing step does because as soon as the new node joined, it is not anymore distinguishable from an existing one.

Removing an Endpoint

The removal of an existing node from a cluster is performed by terminating a Stormy instance. The current implementation of the cloudburster does not inform other endpoints about the upcoming removal of the node. Thus the master protocol has no way to distinguish between a permanent replica or master failure and the cloud collapsing executed by the cloudburster. For that reason no extra handling is provided by the protocol and the reader is referred to the corresponding sections in the failure scenarios (3.5.4).

3.4.4 Master-Handover

The master-handover procedure is invoked when an endpoint in charge wants to execute an operation locally but is not in possession of the lock for the DPI. There are two reasons for such a situation. Either the routing table was updated so that the responsibilities have changed or a master failure occurred. In the latter case the master-handover procedure assumes that the endpoint currently holding the lock is able to response to the prepare and finish request or it looses the lock ownership (probably with some delay). Even if the current master is overwhelmed with requests for get, put and delete operations, the two messages sent here are handled by different threads which are assumed to work independently of any others. The cassandra implementation of the messaging system used in Stormy provides the possibility to specify which and how many threads are in duty of handling specific messages. In case of network connection loss of the current master the leader election component will sooner or later release the lock and the failure handling mechanisms of the master protocol take care of guaranteeing an up-to-date master state.

It is important to note that the master-handover procedure is only executed once at the time for one specific identifier (DPI). All requests concerning the same DPI are blocked until the procedure has finished. The first step is to ask the leader election component for the current lock owner. The returned endpoint is a different one than the local endpoint, otherwise the master-handover procedure would not have been called. The handover of the leadership is performed in two phases.

Preparation Phase

This phase makes use of the Datastreamer. The datastreamer provides the possibility to invoke the streaming process manually. The new endpoint in charge advises the datastreamer to prepare itself for receiving data (i.e. by opening a new socket). After advising the datastreamer, the endpoint sends a prepare handover message to the current master. This message specifies the involved identifier and contains a hash of the current state of the endpoint (see 3.5.1).

The master, when receiving the prepare handover message, first verifies that it is really the leader for the given DPI. If this is true, the master protocol makes sure that no requests for this DPI are processed until the master handover has

finished. This prevents that the state of the master gets modified after the prepare phase but before the finishing phase (which would break the assumption that the master always have the newest state). The hash provided with the message is used to check if the endpoint in charge is outdated. If so the current master starts streaming its state to the other endpoint. If the endpoint in charge is already up-to-date, no streaming occurs (but the datastreamer is invoked anyway to signal an acknowledgement).

The datastreamer instance on the endpoint in charge informs the master-handover procedure of the successful execution and the second phase starts.

Finishing Phase

The current master and the almost-master have the same state now and both states cannot be changed because request processing is blocked on both of them for the DPIs involved. The almost-master sends a finish handover message to the current master with the advice to release the lock now. The master follows this advice, releases the lock for the identifier and removes the processing blockade. As soon as the endpoint in charge receives the acknowledgment from the old master the master-handover procedure is finished.

Before continuing now with request processing the endpoint needs to verify that it is the master now. Apache ZooKeeper assigns the lock in order of request arrival. In the situation discussed in the first load balancing scenario (replica as a new endpoint in charge) it might be possible that in the lock order of ZooKeeper the second replica is before the first replica. The first replica (that is the new endpoint in charge) starts the master-handover with the old endpoint in charge. After having completed this procedure the new owner of the lock is the second replica. So another master-handover is necessary with the second replica to finally acquire the lock.

3.5 Failure Scenarios

One of Stormy's requirements states that failures are the norm rather than the exception. A failure will almost for sure have an influence on the performance, but they do not prevent Stormy from fulfilling its purpose of a streaming system. Failures might be occurring infrequently in an environment where the hardware was specially designed to be robust to failures. But Stormy is assumed to be running on off-the-shelf hardware where failures occur comparatively often. The next sections define the kinds of failure that can take place. In a balanced and stable state the master is always equal to the endpoint in charge. For that reason the master failures are put on the same level as endpoint in charge failures. Because in the moment when the endpoint in charge fails it was the owner of the associated lock and therefore the master.

3.5.1 Introduction

Endpoint State

As described in the master protocol workflows in the normal state scenarios the master handler, when sending messages to the replicas, includes a hash of its own state. This hash consists of the event number of last successfully processed events for the same stream. A push event request for the stream AAA is represented by the put request with `DPI<|AAA#|, [events], OP-push-events>`. The master handler, when processing such a request, sends the event number of the last processed events for the stream AAA. For example: last put request for this stream contained the events numbered from 230 to 300. The new put request therefore contains the events starting with the even number 301. The master handler therefore sends the event number 300 as the last successfully processed one to the replicas. A replica having a different (lower) event number locally notices that its state is stale and fetches the up-to-date state from the master. Fetching the new state from the master is done by invoking the datastreamer service that directly transfers the query state (or other DPIs) via a separate connection from store to store. After the stale replica updated its state the events from the request are processed.

Replica Event Buffering

All requests the protocol forwards to the Stormy store contain a flag (in the version) indicating if the request should be processed as a master or as a replica. The replica handler always sets this flag to false, the master handler sets this flag to true for the local store operation. The Stormy store ignores this flag except in the case of a put request with the hint *OP-push-events* (i.e. a push event request). All other requests are processed equally on replicas and on the master (query registration or non-hinted key-value requests).

Before the Stormy store forwards the events to the query, the current flag status (replica or master) is communicated to the thread responsible for the query results (i.e. the one that is pushing the output events of the query to the queries output stream). When this thread fetches new events from the query engine the processing depends on the flag set.

Replica: A replica does not send out the events that were produced by the query. Instead it puts the events to a event history. The event history is responsible for buffering the events until either the replication status changes (i.e. from replica to master) or until the query output thread receives an order to delete the history up to a specific event number. The first situation is explained in the master paragraph. The order to delete some parts of the event history is given by the master handler. When sending a message to the replica containing the request and the hash of the master state, the master handler additionally sends the event number of the last successfully pushed event. It gets this event number from its local query output thread. The replica query, when receiving this order, deletes the events up to the last confirmed event.

Master: As mentioned in the paragraph before, the output thread of the query makes the event number of the last successfully pushed event available so that the

master handler of the protocol can send it with its next message to the replicas. The master query thread always sends out the events as soon as the query provides new ones. If a master query output thread receives for a replica flag (instead as the master flag as before), it switches to the replica mode by just buffering the events instead of pushing them. Such a situation might happen when the load balancer was active and the positions of the endpoints have changed, so that a new master was elected for the affected query. The opposite case, where a replica query is switching to the master mode, causes that all events in the event history are sent out. By sending out all the yet unconfirmed events, the system assures that no events get lost. This is especially important in the case where the master failed and thus was not able to send out the latest result event from the query. Because these events are unconfirmed, the newly selected master query have not deleted them yet and will send them out.

3.5.2 Lost Event

Events can get lost when the message send from one endpoint to another is dropped. This might either occur because of an instable network connection or the messaging system threads concerned with the handling of put requests are overloaded and the processing of the message takes longer than the counterpart is willing to wait for. Depending on whether the events are sent to the endpoint in charge or to a replica, two situation occur.

If the events are sent to a replica by the master handler this handler does not detected the lost event almost for sure, because it only waits for the majority of the responses and not for all of them. Because the events get lost, one of the replicas is stale. With the next arriving message the replica will note that its state is outdated (hash comparison) and is going to fetch the newest state from the master. This situation is the same as the one that is going to be described for transient replica failures.
In the unlikely case that the events of the majority of the replicas get lost, the master handler remarks that and will inform the caller of the master protocol thereof. Usually the caller will in this case (i.e. timeout exception) just resend the events.

In the other situation the events to the endpoint in charge get dropped. Events are only sent remotely to an endpoint in charge by the entrance handler when it redirects the request. This situation is the same as the one of a transient master failure and is therefore described in the appropriate section.

3.5.3 Transient Failure

Transient failures usually arise due to network loss (e.g. network cable is pulled out by accident) or due to overwhelmed messaging system threads that are not able to handle the whole load immediately. The duration of a transient failures is between a few seconds up to half a minute. An endpoint sending a request to an endpoint that failed (transient) notices that by a timeout (e.g. the endpoint does not receive a response). As opposed to permanent failures transient failure are not (yet) detected by the gossiper respectively by its failure detector. Therefore the router still assumes that the endpoint is alive and is returning that one in

preference lists too.

Master

A transient failure for an endpoint in charge (referenced as endpoint F) manifests itself when another endpoint (A) sends a redirect message to the transiently failed endpoint (therefore F). This redirect message is sent in the entrance handler. The waiting for the response causes a timeout and the entrance handler checks the second endpoint (B) in the preference list. Depending on whether the second endpoint is the local one or not, two workflows are possible:

Local Endpoint: The second endpoint (B) is the local endpoint A (A = B). The control is passed to the endpoint-in-charge handler. This handler verifies the leadership. If the failure on the remote endpoint F was caused due to a network connection loss the lock originally held by the endpoint F is released. In this case either the local endpoint holds the lock already or the second replica (C) got the lock. In the latter situation master-handover procedure is invoked to transfer the ownership from endpoint C to the local endpoint.
If endpoint F still has network connection but is not able to respond to get, put or delete requests due to an overloaded thread pool, the remote endpoint is still the leader. The master-handover procedure will be invoked in this situation too and is assumed to complete because of the higher priority of the associated threads.

Remote Endpoint: The endpoint B is not the local endpoint. The entrance handler sends a new redirect request to this endpoint. Assuming that this endpoint has not failed (otherwise the entrance handler would continue with the third endpoint in the list) the redirect message is accepted by the redirection handler on endpoint B. This handler compares the preference list received from the router with its local endpoint. The local endpoint B is in the second position. The handler first tries to send a redirect message to the real endpoint-in-charge F.
If this redirection is successful, the response is sent back to the initial endpoint having sent the request. Such a situation can happen if the failure on endpoint F is already repaired or due to a routing problem or a failed switch endpoint A cannot directly reach endpoint F but endpoint B can.
Usually the redirection will fail (because endpoint F still has problems) and endpoint B will therefore execute the local endpoint-in-charge handler. The operations required in the endpoint-in-charge handler are the same as described in the paragraph above (where A = B).

Replica

Replicas are only contacted in the master handler. A transient failure of a single replica is not directly noticed because the master handler, after sending the messages to the replicas, only waits for the majority of the responses, therefore (with replication factor 3) for one replica response. Only in the case that two replicas are failing at the same time the master handler would throw a timeout exception of which the caller of the protocol is informed. The caller then usually resends the events. In the more likely case of a single replica failure it is the replica itself noticing that it is outdated. The replica detects that the last processed event number received from the master is newer than the local one. Before processing

the new incoming events the replica invokes the datastreamer to fetch the newest state from the master endpoint. Most likely the state fetched from the master already includes the incoming events but that does not matter because already processed events are just ignored by the event iterators.

3.5.4 Permanent Failure

Permanent failures are those that were already detected by the gossiper. Endpoints with permanent failures are therefore no longer returned in a preference list. Because failure detection needs some time to declare a permanently failed endpoint as dead, each permanent failure is first a transient failure. Only in case a failed endpoint is not involved in any requests before the gossipers announces its death, there is no transient failure phase. Permanent failures do not cause message timeouts because no messages are sent to them. Therefore the different handlers in the master protocol cannot observe a permanent failure directly, only the endpoint newly in a preference list due to the removal of the failed one notices that its state is not up-to-date. The same handling as in the transient replica failure scenario occurs.

A permanent failure can be caused by several issues. The Stormy instance had a fatal error, the machine running the Stormy instance might have crashed completely (e.g. due to power outage) or a switch has a malfunction and cuts off the network connection. In case of permanent failures any locks held by the failed instance are released by the leader election component.

Master

As mentioned above a permanently failed master has released all its locks. A failed master was the first in the preference list for some DPIs. If this one fails, the first replica will be the new endpoint in charge (assuming the skip list router implementation). One of the two replicas holds the lock now, the other one is trying to acquiring it (because replicas always try to acquire the lock too but never get it as long as the master is in place). The new endpoint in charge must be able to guarantee that it has the newest state. For that reason additional steps are necessary because it might have been a stale replica before that have missed the last sent events. Therefore the new endpoint in charge, before serving any requests, contacts all nodes that are waiting for the lock. The waiters for a lock can be retrieved by querying the leader election component. If, after the death of the master, the second replica got the lock (instead of the first one), the master-handover procedure is used to handover the ownership of the lock (as described in the performance scenarios).

Replica

Imagine a preference list where endpoint A is the endpoint in charge and endpoint B and C are the first and second replica respectively. Endpoint C fails permanently. The new preference list would therefore contain endpoint A and B as before and a new endpoint D that does not have any DPIs previously served by endpoint C. When endpoint D receives a request message from endpoint A the first time, it will update its state. Endpoint D detects its staleness because the request message sent contains a higher last processed event numbers then it has

locally. Endpoint D in fact does not have any last processed event numbers for the involved stream ID.

3.6 Lessons learned

We encountered several problems during our work with Stormy. Three of them are mentioned here where I think they are worth to be explained and can save a lot of time when considering them early enough.

3.6.1 Stormy on Top of Cloudy2

Cloudy2 provided an excellent architecture to build a system serving other workloads on top of it. Stormy is one proof for this, Raman Mittal's thesis [22] will give the second one. This does not mean that building Stormy was an easy task. But thanks to the architecture provided we could focus on the important aspects like the handling of streams and the assurance of strong consistency and did not have to argue with basic cloud storage issues like providing a stable and efficient messaging system, discovering endpoint membership or the implementation of sophisticated load balancing algorithms.

Despite the modularity of Cloudy2 components must have the knowledge how certain tasks are handled by other components. For example the protocol needs to be aware of which endpoints are directly informed by the load balancer after a load balancing step. The announcement of the new routing table is of course done by the gossiper when endpoints states are exchanged. This process still takes too long in a heavily loaded cluster (up to some seconds) which can cause that requests are wrongly redirected several times because of stale routing information. That is why the load balancer directly informs the endpoints in the vicinity of its changes. If the protocol does not know which endpoints are already aware of the new situation it is quite difficult to handle an incoming requests when some of the contacted endpoints provided misleading information. Therefore it is, as stated earlier already, very important to clearly document each component. This is what makes the developers life more comfortable.

3.6.2 Query Engine

Stormy uses MXQuery as its query engine. On one hand this was great because we did not have to care about developing a query engine. On the other hand MXQuery was not designed to be used in a distributed scenario where the state of a continuous query has to be serialized, send to another endpoint, deserialized and started again. Such a procedure is necessary to assure a given replication level if one node goes down. MXQuery already supports query serialization, but only for new queries that were not started yet. The required modifications made to the engine were mainly concerned with synchronization issues, i.e. that the state of the query during the serialization is and stays consistent until the process has finished. Several concurrently ongoing writing operations were feeding and fetching events to and from the query what interfered with the serialization.

The summary of this paragraph is that it is important to study the specifications of existing software before using them. If a software behaves as expected in simple

test scenarios (e.g. single-threaded), this does not provide a guarantee that it does the same in others.

3.6.3 Java & Memory Leaks

Java has the advantage of a garbage collector taking care of deleting no longer used objects. Therefore memory leaks do not occur as for example in C++ by a forgotten delete statement that should have freed allocated memory. But this does not prevent a developer from creating memory leaks by himself, especially when working with unbounded sets or lists provided by the Java util packages. In case of Stormy such a collection was used to buffer outgoing query events on a replica. This event history was cleared up to a certain event when a successful execution of a push request with the corresponding events was confirmed. Unfortunately the key that was used to define what to delete was of another runtime type so that the buffer was never cleared. The memory exhaustion caused wired behavior in the cluster that was very hard to trace back to the real problem because the symptoms were distributed over different source code parts. Just consider that when the next time you are encountering strange and unexpected behavior for parts that worked perfect so far.

Chapter 4

Performance Experiments and Results

The architecture of Stormy was explained in the last chapter. Besides knowing how the system is built, it is evenly important to see how well the built system performs in different scenarios.
The first section describes the linear road benchmark, a benchmark used to test stream data management systems. Section 4.2 shows three scenarios using the linear road benchmark to test Stormy under different assumptions. In the last section the results are presented and discussed.

4.1 The Linear Road Benchmark

The linear road benchmark, in the following referenced to as the LR benchmark, is a benchmark designed to measure how well a system can meet real-time query response requirements in processing high-volume streaming and historical data. This section provides a summary based on the paper "Linear road: a stream data management benchmark" [5] and some further explanations.

4.1.1 Overview

A benchmark for a streaming system is based on other assumptions than a benchmark for a data storage system. The differences in the assumptions are a consequence of the way how data and queries are interpreted by the respective systems. A data storage system, like a simple key-value store, is populated with data (e.g. a value assigned to a key). Queries are issued to retrieve the stored data. In a streaming application the system is populated with queries and fed with data to invoke the associated queries so that they output results. Queries in a data storage system are queries executed only once. Queries for streaming systems are long-running, so-called continuous queries.

For data storage systems a simple benchmark measures the completion time of a query; the time from when the query is issued to when the query has finished and outputs the requested data. Continuous queries do never finish, they run as long as the system is running. That is why the performance metrics of the LR bench-

mark is based on response time rather than completion time. The response time is the difference between the time when an input arrives to a streaming system and the time when the system outputs a computed response.

The LR benchmark provides different queries and input data and makes demands on the response time. The final output of the benchmark is a value representing the supported query load (benchmark score). A stream system supports a query load factor X if it can process that amount of load (given by the benchmark score X) while still meeting specified response time and correctness constraints (see section 4.1.4 for more details).

Simulation is used to generate input data that is not purely random but semantically valid. Semantically valid input data should be used to allow the queries to process useful, realistic data. Feeding only random data to a query would result in incomparable results. The LR benchmark simulates a toll system for the motor vehicle expressways of a larger metropolitan area. The simulated city, the notifications and the input data used are explained in the next section.

4.1.2 Benchmark Architecture

The LR benchmark implemented in Stormy only supports a limited part of the full LR benchmark as described in the cited paper [5]. The complete benchmark contains more sophisticated queries and provides the input data that is required to answer them. These queries are sent interleaved with the position reports (see Stream Data below). They are asking for the currently charged amount (account balance) and for the expenditure on a specific expressway and on a specific day during the previous 10 weeks (daily expenditure). The third and most complex query is asking for an estimation of the travel time and tolls for a journey on a given expressway on a given day and time. Stormy's benchmark implementation does not support these types of queries. What Stormy supports is described below. In the following the term LR benchmark references the limited implementation and not the full benchmark with the queries mentioned.

Simulated City

The metropolitan area simulated is Linear City, a 100 miles wide and 100 miles long area containing 10 parallel expressways in horizontal direction, each of them 10 miles apart. The expressways are numbered from 0 to 9. Every expressway has four lanes in west and four lanes in east direction. These are split up in three travel lanes (lane #1-3) and one lane dedicated to entrance (lane #0) and exit ramps (lane #4). Each expressway has 100 entrance and 100 exit ramps in each direction, dividing it into 100 one-mile long segments (see figure 4.1 for an illustration of one segment).

Stream Data

The simulator provides input data in terms of position reports. Each position report contains the vehicle ID that uniquely identifies a vehicle in Linear City and the exact position of the vehicle (on which expressway, on which lane, in which segment and at which position within this segment). Roughly every 30 seconds a

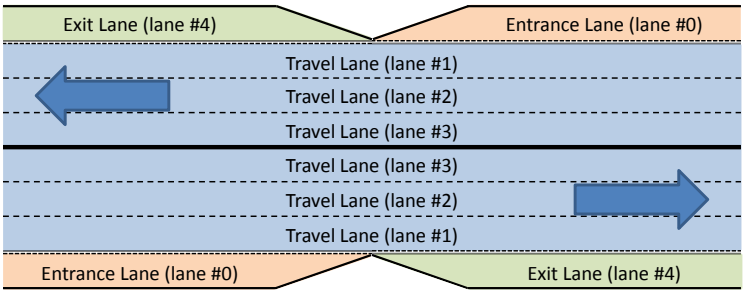

Figure 4.1: One Expressway Segment

vehicle emits such a position report to the stream system. Based on these input events the stream system informs the vehicles about the tolls being charged and notifies them if they are approaching an accident location.

Notifications

Toll and accident notifications are the two types of notifications sent as an output of Stormy. The arrival times of these outputs are compared with the emission time of the position report that caused these notifications. These times have to satisfy the response time constraints.

Toll: The stream system, when receiving a position report from a vehicle, determines if this vehicle has entered a new segment on the expressway. If so, the vehicle is informed about the tolls that are going to be charged to its account. A vehicle is charged the toll of the last segment as soon as it enters a new segment (segment crossing). If a vehicle leaves a segment by using the exit lane, no toll is charged for the current segment. Tolls are calculated based on the number of vehicles present in the current segment and the average speed. Tolls are not charged if an accident was detected 0-4 segments downstream.

Accident: An accident is detected whenever two vehicles emit the same position within at least 4 consecutive position reports. If so, all vehicles entering in one of the 4 segments upstream to the accident location are informed about the accident and therefore being left the choice to exit the expressway.

4.1.3 Implementation

ETH Zurich in cooperation with Oracle extended MXQuery, the XQuery engine used in Stormy, with window functions. To measure the performance of their extension they implemented the LR benchmark in XQuery [6]. We took their XQuery LR benchmark implementation for our tests and made some modifications. Although it is possible to implement the whole LR benchmark in one XQuery expression, several queries are used to provide a base for better optimizations (the performance gain of the optimization of a few simple queries is larger than the one of a single complex query). Figure 4.2 shows the manual query partitioning used in our implementation. A total of 8 queries (lilac rectangles), one

internal store (pink cylinder), two shared stores (violet cylinders) and two outputs (yellow hexagons) implement the LR benchmark. One input stream feeds the events containing the position reports. The blue arrows represent the event flow, the green arrows indicate store usage (internal & shared).

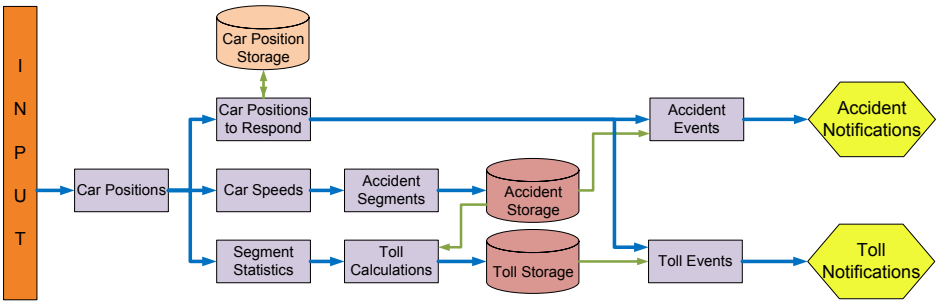

Figure 4.2: LR Benchmark Implementation for Stormy

4.1.4 Measures

The benchmark score as explained in the overview section (4.1.1) is the L-rating. The L-rating represents the number of expressways simulated. With a higher number of simulated expressways, the number of vehicles in Linear City is higher. The number of position reports is linearly dependent with the number of vehicles. Hence with a higher L-rating the overall load to the streaming system is higher. The lowest possible L-rating is 0.5, i.e. half an expressway, that means that only one direction of the expressway (east or west) is simulated. There is no real upper bound except for the fact that the area was defined to contain 10 expressways (L=10), but that is not a technical limit.

A system supports a given query load (i.e. L-rating) if it still meets the specified response time and correctness constraints. The correctness constraints can be verified by the provided validator [30]. The required response times for toll and accident notifications are 5 seconds. Therefore the time difference between the emission of a position report and the receipt of the corresponding toll or accident notification is allowed to be at most 5 seconds.

4.2 Experiment Scenarios and Measurements

The LR benchmark is used as one component inside the test scenario but it does not serve as the test scenario itself. This is because the benchmark score (the L-rating) tells us about the performance of the query engine on a fixed set of machines. The goal is to show how scalable Stormy is, therefore a variable number of machines are involved. Tests on a single machine have shown that a L-rating of 2.5 is supported. Distributed tests on three machines with a replication factor of 3 still support a query load of 2.0. These results were measured when running a three hour simulation on one of the ETH internal clusters. These machines are equipped with 24 GB of main memory, the maximum heap size was set to 1 GB and each of them was running Linux on 16 Intel® Xeon® CPUs with 2.27 GHz.

In the three test scenarios, the L-rating was fixed to the lowest possible one, 0.5 (one expressway with four lanes in only one direction).

The tests measure how well Stormy scales and behaves under load and in case of failures. For that, we run multiple LR benchmarks concurrently on our cluster. For a fixed query load of 0.5 we are interested in the number of machines that are needed to run these multiple LR benchmarks while still meeting the response time constraints. The load of a single LR benchmark is increasing with time, thus starting with a small load and develop the full load at the end of the benchmark (i.e. after 180 minutes). Therefore the number of concurrently running benchmarks is not a good indicator for the real load in the system. Instead, the number of outgoing notifications is used to determine the load. During the run of each test scenario described below we recorded the number of running machines on a regular base (i.e. every minute) as well as each received notification (toll and accident) with their emission and report time. The report time of a notification is the time when the notification was send out by the output. The two recorded times for each notification are used to calculate the response time later on and indicate how many notifications did not meet the response time requirements.

4.2.1 Scenario 1: Scale the Number of Benchmarks

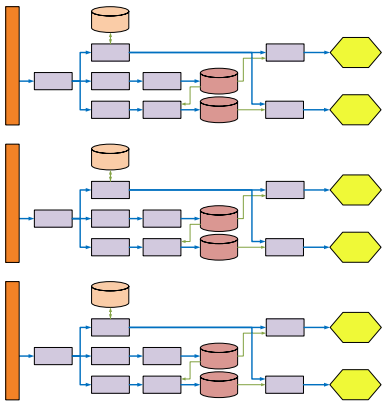

Figure 4.3: Benchmark Configuration of Scenario 1

In this test scenario we vary the number of independent instances of the LR benchmark. Independent means that each LR benchmark has its own input generator feeding events into the system. Therefore the different LR benchmark setups do not interfere with each other because all of them are using different input stream IDs. Figure 4.3 shows an example running three independent benchmarks. The simulation time of all benchmarks used in this scenario is 180 minutes. The test starts with a single LR benchmark (at minute 0). A test coordinator registers and starts a new LR benchmark every 10 minutes (i.e. at minute 10, 20, 30, ...). Starting the benchmark means to instruct a dedicated machine (one that is not running an instance of Stormy itself) to begin feeding the position reports for the newly registered benchmark. The last LR benchmark is started at minute 170. This results in 18 LR benchmarks running at the same time between minute 170 and 180. Each of the benchmarks is simulating half an expressway, therefore

the whole cluster is simulating 9 expressways in total (L=9.0, even though this score should not be directly compared to a single LR benchmark with an L-rating of 9.0). The total running time is 5 hours and 50 minutes.

The purpose of this test scenario is to show the co-work of the load balancing components with the streaming part. Stormy should always be able to handle the load at least by increasing the numbers of machines (i.e. do cloud bursting) if not by the execution of simple load balancing steps.

4.2.2 Scenario 2: Scale the Number of Queries

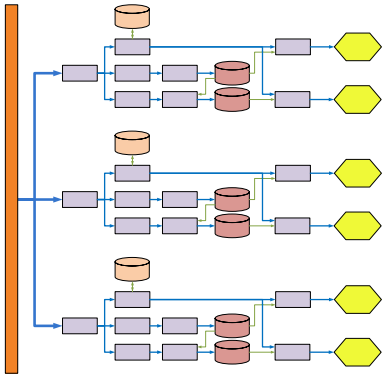

Figure 4.4: Benchmark Configuration of Scenario 2

Scenario 2 tests how well Stormy can handle high load to mainly one single input stream. To focus the load, only one input stream is registered and one dedicated machine feeds events into this input stream (during a period of 180 minutes). Every 10 minutes (starting at minute 0 up to minute 170) LR benchmarks are registered for this single input stream. Therefore all the benchmark instances depend on this input stream (see figure 4.4). The total benchmark running time is 180 minutes.

4.2.3 Scenario 3: Benchmark in the Presence of Failures

One requirement of Stormy is its robustness to failures. The purpose of this test scenario is to confront Stormy with the different failures that can occur and observe the behavior in these cases. For a review of the possible failures study section 3.5. At minute 0 a fixed number (five) of independent LR benchmarks are registered and started. After an adaptation and load balancing phase of 20 minutes the test coordinator starts simulating failures at a regular interval of 10 minutes. The failures are simulated by advising a physical machine running a Stormy instance in the cluster to shutdown, therefore simulating a permanent master or replica failure. The simulation of the transient failures is not yet supported. The repair mechanism that would be called in case of lost events is the same that is called in case of a transient replica failure (stale replica).

4.3 Results

This section should present the results for the three test scenarios listed above. But instead it explains why we were not able to run these scenarios and describes two alternative test scenarios with their results.

4.3.1 Problems

The current implementation of Stormy, more precisely the implementations of the master protocol and the Stormy store, has bugs that could not be fixed before the thesis deadline. These bugs are concerned with the query serialization and deserialization which are often performed operations in heavy loaded clusters.
If the load balancer decides to move the position of an endpoint to balance the load, the responsibilities for the DPIs change and thus the queries have to be moved to new endpoints. A query is moved by serializing it on one endpoint, transfer it to the new endpoint and deserialize it there. Serializing a query includes the serialization of the query state, i.e. the query iterator tree with its currently cached attributes (e.g. an XQuery windowing expression with the elements being processed).
Load balancing is not the only occasion where queries are serialized. When a new node joins the cluster (e.g. due to a previous cloud bursting request), this node takes over some of the load of existing endpoints.
The third situation where queries have to be serialized is when failures occur. For example due to a transient replica failure that causes the replica to be stale, the master protocol will make sure that the replica is updated as soon as its staleness is detected.

To fetch output events from the MXQuery engine an own thread invokes a blocking *next* method of the query engine. This method is blocking until there are result events. In the LR benchmark some queries have a high output rate of less than a millisecond per event (e.g. *car positions*, *car positions to respond* and *toll events* shown in figure 4.2). Others have a very low output rate of one event each 30 or 60 seconds (e.g. *accident segments* and *toll calculation*) once the system is running for a while. When a new LR benchmark registers the queries, it may take up to 4 minutes until all queries have at least sent out one output event. These rates are mentioned because a query cannot be serialized during a blocked *next* call. In such a call the query state is inconsistent until the call has finished. Serializing a query in this state itself works, but after the deserialization the state of the query is corrupt and the query cannot produce correct output results anymore.

For that reason a load balancing step or the updating of stale replicas can require up to 4 minutes. The master protocol considers this delay and accepts it (because with the current implementation of MXQuery no other solution exists). When running Stormy with activated load balancing the system gets in a state where a lot of serialization requests are waiting for their execution (i.e. they are waiting for the *next* call to finish). The problem is that some of the *next* calls never finish because the queries do not receive input events anymore and thus cannot produce new output events. We were not able yet to figure out the real reasons why the cluster gets caught in such a state.

4.3.2 Adapted Test Scenarios

Due to the unsolved problems discussed above the scenarios as described in section 4.2 could not be tested. To present results for Stormy nevertheless, two new test scenarios were designed and are described next. These test scenarios do not test the scalability of Stormy because of the reasons mentioned above. Thus the load balancing and cloud bursting was deactivated for the scenarios.

Adapted Scenario 1: Scale the Number of Benchmarks

This scenario is based on the original test scenario 1. What differs is described in the following.
Ten independent LR benchmarks are started on ten machines from on of the ETH internal clusters. They have the same configuration as the ones described for the single LR benchmark test (see the first paragraph of section 4.2) except that the maximum heap memory was increased to 16 GB. All ten machines are started at the same time. A form of manual load balancing is done by replacing the generation of the random stream IDs (in the Stormy processor) by a sophisticated algorithm assuring the responsibility assignment of one LR benchmark to one Stormy instance. Each LR benchmark needs eleven stream IDs (including the store IDs). The algorithm creates the first eleven stream IDs so that endpoint A is the endpoint in charge, the second eleven stream IDs are created so that endpoint B is responsible and so on. In the end all endpoints have 3 times 8 query instances running (replication factor 3).

Every 18 minutes a new LR benchmark registers and starts sending events, beginning at minute 0. Thus the last benchmark is registered at minute 162. The total running time of the benchmark is 342 minutes. Due to the manual load balancing the queries are evenly distributed across the nodes in the cluster.

Adapted Scenario 2: Scale the Number of Queries

Instead of independent LR benchmarks, this test scenario uses dependent LR benchmarks. This means that there is only one benchmark instance that feeds events to the cluster. The same setup as in the previous scenario is used. Manual load balancing guarantees that all ten *car position* query instances (see figure 4.2) are in the responsibility range of one endpoint. The sub-queries of each LR benchmark are evenly distributed across all the nodes in the cluster.

Every 18 minutes a new LR benchmark registers to the main input stream. The test scenario finishes after 180 minutes, e.g. after all events are fed to the main input stream. Therefore the LR benchmarks registering as the second (third, ...) one does not run for the full duration of 3 hours.

4.3.3 LR Benchmark Load Distribution

Figure 4.5 shows the load distribution within a single LR benchmark run. The load is expressed by the number of incoming notifications. Toll and accident notifications are shown separately. The load is increasing during the first 50 minutes of the LR benchmark. The accident notifications are responsible for a load peak at the minutes 47 to 55. Afterwards the load slightly decreases. During the last

hour of the benchmark the accident notifications are again causing a raise of the load with a peak at the minutes 128 to 140.

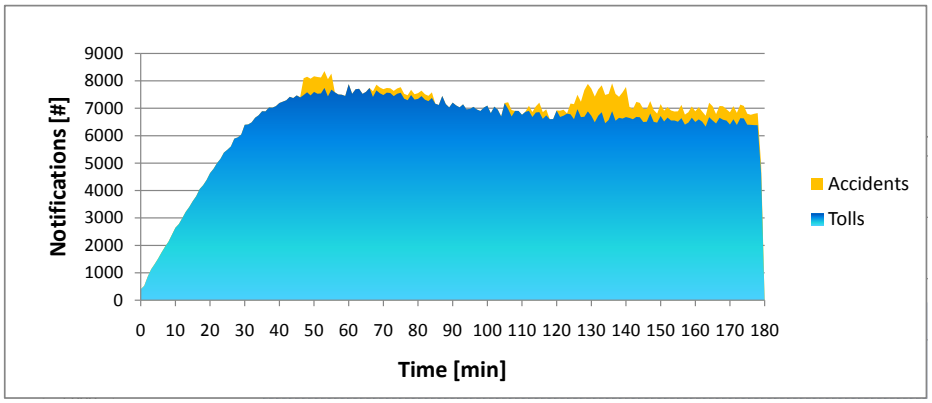

Figure 4.5: Load Distribution of a Single LR benchmark

These load values were calculated based on the measurements of four independent LR benchmarks. The average load is around 6'600 notifications per minute. 8'357 events per minute was the maximum load value measured at minute 53. A cluster running ten LR benchmarks thus has to handle ten times this load.

4.3.4 Adapted Scenario 1: Results

Two test run results are described here, one that failed one that was more successful.

First Test Run

A first test run with adapted scenario 1 has encountered several difficulties because the connection to Apache ZooKeeper (the leader election services used by Stormy) was lost and no lock acquiring was possible. That caused the failure of different handlers in the master protocol (i.e. replica handler and endpoint-in-charge handler) what in turn caused the failure of several benchmark instances feeding events into the system (because they did not catch this failure before). The results of this first test run are shown in figure 4.6. They are shown because the impact of a failed event feeding instance is directly viewable in the corresponding break of the load curve.
All incoming notifications fulfilled the response time constraint of 5 seconds. At the beginning of each LR benchmark there are about 20 to 40 events that do not meet the constraint. The corresponding input events (position reports) that triggered these notifications are consumed and delayed during the preparation phase of the query and thus are not considered as failed notifications. These events are characterized by their appearance order. All incoming events after the start of the LR benchmark that have failed (i.e. do not meet the response time constraint) before any non-failed notification arrives are considered as valid (i.e. non-failed). This type of events appears especially in the second test scenario in a much higher quantity.

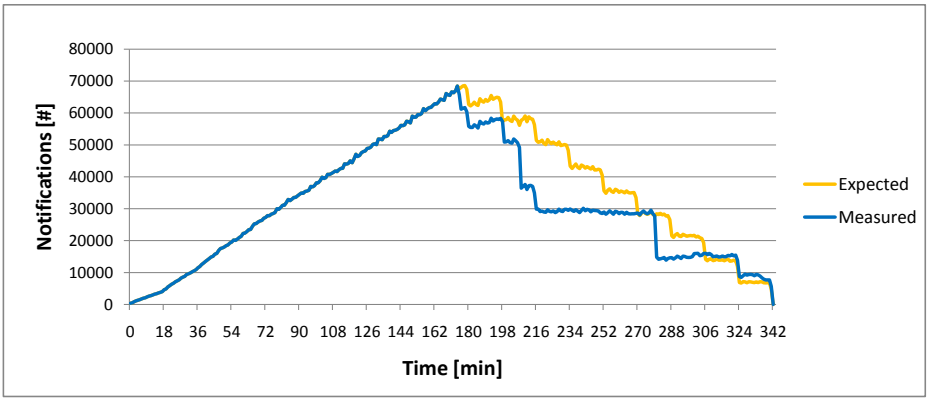

Figure 4.6: Results of Scenario 1 (first failed test run)

In figure 4.6 the blue line shows the measured notifications while the orange line shows the expected load behavior. The first LR benchmark instance starts at minute 0 and finishes after 3 hours (notice the break at minute 180). Before that, one LR benchmark instance has stopped feeding events due to a failure. This is the position where the measured and the expected numbers of notifications start to differ. The second benchmark stops at minute 198. Between this stop and the next at minute 216 two benchmarks have failed (most likely those that would have finished at minute 234, 252 or 270 otherwise). The third and fourth last benchmarks have failed too before they could finish. The last two benchmarks were executed successfully.

Second Test Run

Figure 4.7 presents the results for the second test run that finished successfully. The measured number of notifications conforms to the expected ones shown in figure 4.6 (failed test run).

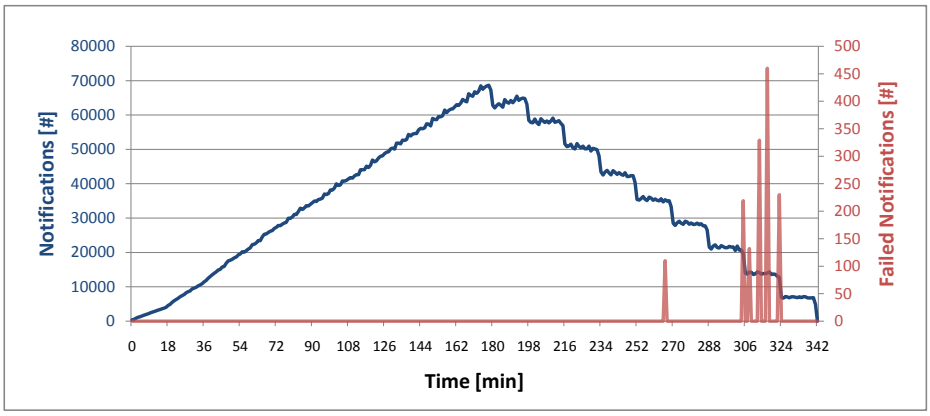

Figure 4.7: Results of Scenario 1

The load, i.e. the number of notifications, is shown in blue with the scale on the

left side. In the first half of the benchmark run the load is linearly increasing, followed by a stepwise decreasing of the load in the second half. After a benchmark instance has finished, the number of notifications drops by about 4'700 to 6'000 notifications.

The last three benchmark instances had some incoming notifications that did not meet the response time constraint (shown in red with the scale on the right side). This is a rather strange behavior because the high load is in the middle of the benchmark and not towards the end. The response times of the "failed" notifications are either 6 or 7 seconds, thus not really that far away from the constraint. What might be an explanation for the occurrence of the failed notification towards the end is that the cleanup of successfully finished LR benchmarks is not done properly. Or it might even be that some other components like the messaging system or the leader election component do not free its unused resources. This behavior must be investigated in more detail.

4.3.5 Adapted Scenario 2: Results

About 2 hours and 10 minutes after starting the test run, Apache ZooKeeper began to occasionally lose the connection to the endpoints. The problems increased heavily in the last 20 minutes, but this time the system was able to handle it and the test did end successfully. The results are shown in figure 4.8.

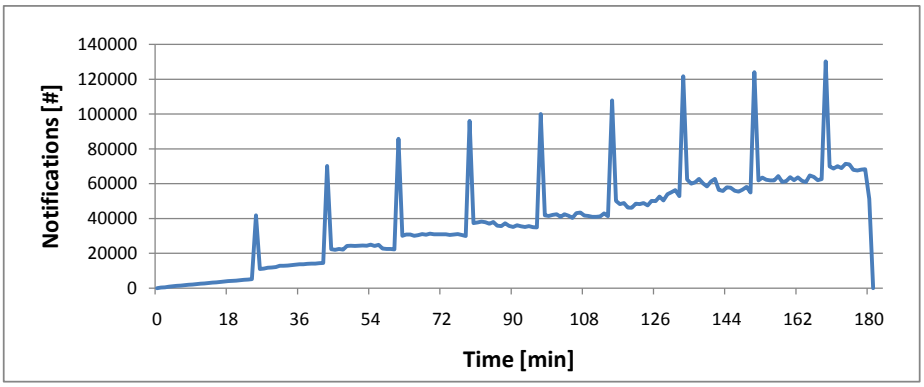

Figure 4.8: Results of Scenario 2 (raw data)

As opposed to the first scenario the queries registering in this one subscribe to an existing input stream and have missed the events sent prior to the query registration. This is of course not true for the first LR benchmark that registers its queries and sends the position reports to the system. When missing the start events, an LR benchmark requires about 4 to 6 minutes until it produces the first notifications. The first data packages sent out to the output listener contain a huge amount of "failed" notifications (up to 58'000 events in one minute). These notifications are the ones that were jammed until the first notification was sent. Almost all of them do not meet the response time constraints but are considered as valid (non-failed) notifications due to the reason already described in scenario 1. These jammed notifications are represents by the 9 peaks in figure 4.4. To have a better view on the step-wise increasing of the load due to the registering of a

new LR benchmark, these non-failed notifications are subtracted from the number of incoming events. Figure 4.9 shows the cleaned load. Notice that another scale is used in this chart so that the different load levels are better distinguishable.

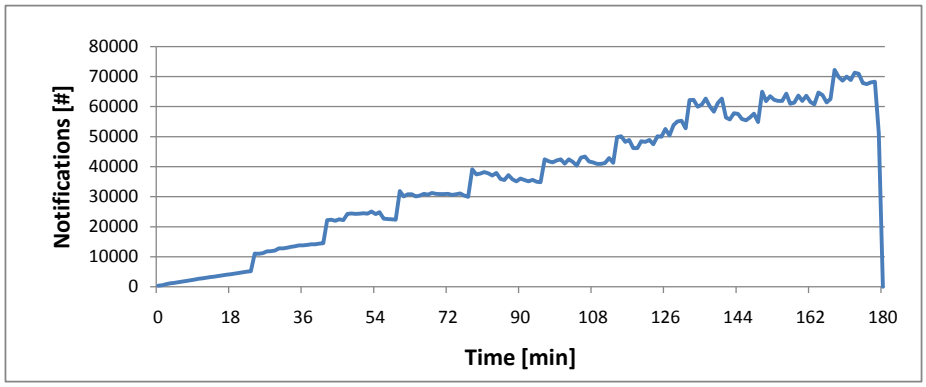

Figure 4.9: Results of Scenario 2 (cleaned)

With a delay of 4 to 6 minutes the load increases rapidly after a new LR benchmark was registered. Even without the registration of a new benchmark the load is slightly increasing during the first 50 minutes (see section 4.3.3). Around the minutes 132 to 140 there is a little peak probably caused by jammed notifications from the seventh registered benchmark. The load after this peak is the expected average load when running 8 LR benchmarks.

4.3.6 Discussion

The results presented in the previous sections are not the ones we wanted to show when we designed Stormy and the benchmark scenarios. Nevertheless these results give a glimpse into the possible performance of Stormy. Of course the results are measured under very optimistic assumptions and we know that at least some, when not a considerable part of the notifications will not meet the response time constraints once the load balancing is activated or endpoints start failing. The master protocol is designed in a way to handle failures as the norm rather than the exception. Even in our simple scenarios we encountered failures, not permanent ones, but at least transient master and replica failures that caused a timeout. These failures probably interrupted the typical workflow in the protocol but were handled by the failure mechanisms and therefore did prevent an inconsistent state.

We noticed that the current implementation of the leader election service does not behave very well in a highly loaded system, We might need to rethink the usage of Apache ZooKeeper for the acquisition of exclusive write lock or redesign the master protocol to prevent that much access to the leader election component.

All in all the execution of the adapted test scenarios can be considered as successful. The next step is to eliminate the bugs in the query serialization and deserialization and then to measure the performance of Stormy in a more real world example by using the three test scenarios described first.

Chapter 5

Conclusion

5.1 Summary

This thesis presented two contributions: Cloudy2 and Stormy. With Cloudy2 we have built a very flexible architecture that can easily be extended or reconfigured to serve a different workload. Only a few implementations for the different components are available yet. Flavio Pfaffhauser's thesis [25] presents the benchmarks for the highly-scalable key-value store. These benchmark results give a first impression on the load our system is able to handle.

Our goal for Stormy was to build a highly-scalable, fault-tolerant streaming service. Unfortunately we have not reached this goal yet. As discussed in the result section of the last chapter, implementation difficulties did prevent us running the real benchmark scenarios. Stormy with its current implementation works in a static context where the number of machines is fixed prior to the start of the benchmark. This is not what we targeted at. The development of Stormy does not end with this thesis but instead is going to continue so that Stormy will meet its goal once.

Our vision for the future of Cloudy2 and its co-projects is that Cloudy2 itself chooses the right implementations to serve a specific workload. This might involve the selection of a different protocol that supports strong instead of weak consistency or the exchange of the storage layer to store data persistently instead of just keeping it in the memory. These decisions are based on the specified requirements of the workloads that might even evolve at runtime. There is still a long way to go to realize our vision but the first steps are taken towards this goal.

5.2 Future Work

The current implementation of the Stormy components and of the other contributions suffer from some limitations. This section provides an outlook of some topics for future work.

5.2.1 Multi-Query Optimizations

The LR benchmark and other examples from the real world usually work with several queries to fulfill their task. Although implementing these queries in a single query is feasible, the query engine is not able to optimize such large queries. A future work for Stormy is to optimize multi-query scenarios like the LR benchmark. One of the primary targets to fulfill this task is to make sure that queries invoked by one input stream are located on the same node. This would include queries that subscribed to the output of other queries (and so forth). So in the ideal case one single LR benchmark is execute on one node (and replicated on two others). Another LR benchmark, independent of the first one, is located on another node.

5.2.2 Simulation of Transient Failures

Benchmark scenario 3 is testing Stormy in case of failures. Permanent failures are simulated by shutting down a Stormy instance, thus removing it from the cluster. The simulation of transient failures is missing. These failures can be simulated by disconnecting a node from the network for a short period of time. To realize that, probably the best part to start with would be the messaging system. An existing messaging system could be modified to allow that all incoming or outgoing messages are discarded. This could include the simulation of lost events where single message get dropped. Such a simulation framework would allow a more accurate performance measurement that is closer to a real environment.

5.2.3 Full LR Benchmark

The current implementation of the LR benchmark in XQuery only supports the two types of notifications. To make Stormy comparable with other stream data management systems the full LR benchmark must be used to measure the performance. Our implementation misses the handling of queries that are fed to the system interleaved with the position reports. Besides that, an interface is required allowing to bulk load the historical data to Stormy that is used for answering the queries. Furthermore, the implementation of these queries, especially the travel time estimation query that is by far the most complex and difficult one, would provide an excellent proof for the expressiveness of XQuery.

Bibliography

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *In VLDB*, 2002.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[3] M. Abdallah and H. C. Le. Scalable Range Query Processing for Large-Scale Distributed Database Applications. In *IASTED PDCS*, 2005.

[4] Amazon.com, Inc. Amazon Simple Storage Service (Amazon S3). Available from `http://aws.amazon.com/s3/`, Feb. 2010.

[5] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: a Stream Data Management Benchmark. In *VLDB*, 2004.

[6] I. Carabus, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending XQuery with Window Functions. *In VLDB*, 2007.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertan World. In *CIDR*, 2003.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *In VLDB*, 2008.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *In SIGOPS*, 2007.

[10] Dropbox. Dropbox - Online backup, file sync and sharing made easy. Available from `https://www.dropbox.com/`, Feb. 2010.

[11] ej-technologies. Java Profiler - JProfiler. Available from `http://www.ej-technologies.com/products/jprofiler/overview.html`, Feb. 2010.

[12] P. Fischer. MXQuery - A lightweight, full-featured XQuery Engine. Available from `http://mxquery.org/`, Feb. 2010.

[13] Free Software Foundation, Inc. GNU General Public License. Available from `http://www.gnu.org/copyleft/gpl.html`, Feb. 2010.

[14] A. Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.

[15] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *SOSP*, 2003.

[16] M. Hirabayashi. Tokyo Tyrant: network interface of Tokyo Cabinet. Available from `http://1978th.net/tokyotyrant/`, Feb. 2010.

[17] T. Inoue. Kai - a distributed Key-value Datastore inspired by Amazon's Dynamo. Available from `http://sourceforge.net/projects/kai/`, Feb. 2010.

[18] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD*, 2006.

[19] R. Jones. Anti-RDBMS: A list of distributed key-value stores. Available from `http://www.metabrew.com/article/anti-rdbms-a-list-of-distributed-key-value-stores/`, Feb. 2010.

[20] A. Lakshman. Cassandra – A structured storage system on a P2P Network. Available from `http://www.facebook.com/note.php?note_id=24413138919`, Feb. 2010.

[21] C. Lenzen, T. Locher, and R. Wattenhofer. Clock Synchronization with Bounded Global and Local Skew. In *FOCS*, 2008.

[22] R. Mittal. Query Processing in the Cloud. Master's thesis, ETH Zuerich, Mar. 2010.

[23] Open Source Initiative. The BSD License. Available from `http://www.opensource.org/licenses/bsd-license.php`, Feb. 2010.

[24] Oracle Corporation. MySQL - The world's most popular open source database. Available from `http://www.mysql.com/`, Feb. 2010.

[25] F. Pfaffhauser. Scaling a Cloud Storage System Autonomously. Master's thesis, ETH Zuerich, Mar. 2010.

[26] Project Voldemort. Project Voldemort - A distributed database. Available from `http://project-voldemort.com/`, Feb. 2010.

[27] Scalaris. Scalaris - Distributed Transactional Key-Value Store. Available from `http://code.google.com/p/scalaris/`, Feb. 2010.

[28] The Apache Software Foundation. Apache ZooKeeper! Available from `http://hadoop.apache.org/zookeeper/`, Feb. 2010.

[29] The Apache Software Foundation. The Apache Cassandra Project. Available from `http://incubator.apache.org/cassandra/`, Feb. 2010.

[30] N. Tran. Linear Road - A Stream Data Management Benchmark. Available from `http://www.cs.brandeis.edu/~linearroad/`, Feb. 2010.

[31] W3C. Resource Description Framework. Available from `http://www.w3.org/RDF/`, Feb. 2010.

[32] W3C. XML Fragment Interchange. Available from `http://www.w3.org/TR/xml-fragment.html#terminology`, Feb. 2010.