CHALMERS

Implementation of a transport system with stateful consumers

Master of Science Thesis in the Master Degree Programmes
Intelligent Systems Design and Networks and Distributed Systems

MATHIAS SÖDERBERG
GUSTAF HALLBERG

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2013
Report No. 2012:083
ISSN: 1651-4769

Abstract

When handling large amounts of data, parallel processing is key. Distributed computation makes Big Data work, but it presents new challenges for scaling and reliability. These challenges are made even more interesting when computations are stateful, and done in near real-time as in stream processing.

To make a stateful distributed system scalable and resilient to failures, a framework for message passing and state persistence is needed. As of today no such system is available but the components are all there to build one.

This thesis gives an outline of how such a framework could be constructed using Apache ZooKeeper and Apache Kafka, describing algorithms and components which could be used for such a system.

Also provided are some experimental performance benchmarks of Kafka running in Amazon's EC2 cloud, to use as a measuring stick for its viability as a transport system for messages.

Sammanfattning

Att hantera stora mängder data i realtid eller nära realtid kräver distribuerade beräkningssystem. Denna typ av distribuerad beräkning är nödvändig men skapar också nya typer av problem när det gäller motståndskraft mot krascher och skalning för att möta efterfrågan. Dessa problem ökar exponentiellt när datan som behandlas behöver aggregeras över tid.

För att lösa dessa problem behövs ett ramverk för att hantera meddelandedistribution och tillståndshantering. I dagsläget finns inget sådant ramverk men alla komponenter för att bygga ett är tillgängliga.

Denna rapport beskriver hur ett sådant ramverk kan utformas med hjälp av Apache ZooKeeper och Apache Kafka. Rapporten innehåller algoritmer och idéer om hur ett ramverk kan se ut samt diskussion av en test-implementation av ett sådant ramverk byggt i JRuby.

Rapporten innefattar också ett antal tester av prestandan hos Apache Kafka när det används i Amazons EC2-moln.

Acknowledgements

We would like to thank our supervisors Claes Strannegård at Chalmers and David Dahl at Burt, as well as Theo and Mathias at Burt for the support and feedback during the project.

Also thanks to our fellow master's students with whom we discussed all kinds of topics pertaining to this thesis.

Finally thanks to our friends and families, for the support and love.

Without you this project would not have been possible.

Contents

Chapter 1

Introduction

This report describes the results of a master's thesis project at the programmes *Intelligent Systems Design* and *Networks and Distributed Systems* at Chalmers University of Technology, Gothenburg, Sweden. The project was performed in collaboration with Burt AB, a start-up company that has created a Business Intelligence and analytics platform for digital ads, with focus mainly on publishers.

The aim of the thesis was to implement and evaluate a distributed solution for a modified version of the classical producer-consumer scenario, which adds requirements on how messages are routed between producers and consumers, and how node failures should be handled in a graceful and efficient manner.

1.1 Purpose

The purpose of this master's thesis was:

- To explore the aforementioned use case in detail and formulate concrete definitions of the problems associated with the use case.
- To investigate possible existing libraries and tools that could be used to solve the problems defined by the previous item.
- To design, implement and evaluate a proof of concept system that solves the identified problems.

1.2 Goals

The goals of the project were divided into two groups, algorithm goals and implementation goals.

- Algorithm goals
  - When a consumer is added or removed the workload should be redistributed as evenly as possible.
  - When the consumers redistribute the work, it should be able to resume data consumption without data loss.
  - When the consumers redistribute the work, it should duplicate as little work as possible, while minimizing loss and duplication of data.
- Implementation goals
  - Redistributing work should be reasonably fast, to prevent stalls in data processing.

1.3 Scope

For the project to finish within the given time frame the following limitations were decided upon:

- The implementation is assumed to run in a secure environment, authentication and hardening should be handled outside of the application if deemed necessary.
- Close to real-time performance is ideal, but not a requirement.
- Convergence time for redistribution of load after addition or removal of consumers is not considered a major issue. Scaling and frequent consumer failures are not commonplace and a delay on the order of a minute will not affect system performance overall.
- It is assumed that messages being passed through the system can be partitioned based on a key attribute.
- Consumers are assumed to run on equally powerful hardware.

Chapter 2

Background

Websites often want to analyze usage of the site, information which is often referred to as *activity data*. This typically consists of things such as page views, purchases, searches and even what the user is currently looking at [1]. By analyzing this type of data the owner can gain insight into user behaviour and see the effect of different parts of the site, as well as presenting recommendations or other customized content to users.

Batch processing

The most common approach to handling activity data has previously been to log the activity and periodically process the data and subsequently using the aggregated data for one's desired purpose. The approach of aggregating data in large batches has seen quite a lot of success so far, especially with MapReduce [2] and Hadoop [3]. The obvious downside of using this approach is that one will not get any particularly useful data in real-time, or even near real-time[1], and depending on ones needs this can be perfectly fine, but for other use cases real-time access is desirable.

Stream processing

An alternative approach is stream processing which is becoming increasingly popular with frameworks like Storm [4] and S4 [5], making it easier to build such computing systems. Actor-based [6] frameworks such as Akka [7] and Celluloid [8] can also be leveraged to create stream processing systems. Compared to batch processing, stream processing systems are inherently more complex as nodes are constantly processing incoming data and it might be more difficult to handle failures and restarting at various stages of the system, due to other parts of the system relying on continous processing in each layer.

[1] Assuming a large data set.

2.1 Problem description

Burt records user behaviour on webpages by tracking user sessions over time. A JavaScript *agent* tracks the user's actions on each page and periodically sends relevant information to a backend platform. Each message from the agent is known as a *fragment*, and contains information about which session it belongs to.

On the backend platform, the fragments need to be reassembled into sessions. This is done by gathering all fragments for each session until either a specific timeout is reached or a certain END-fragment is received (such a fragment is sent by the agent when a session is terminated by the user, e.g. the user closed the browser window). When the session is determined to be over, the fragments are passed on as a finished session object to be further analyzed.

The continous stream of fragments is too large for a single machine to handle, which makes it necessary to distribute the work across several nodes. This creates a new problem, since all fragments belonging to the same session must be sent to the same node if the session is to be assembled correctly. There also needs to be a mechanism to handle failing nodes, or scaling of the node cluster. This presents a second problem, it must be possible to transfer responsibility for sessions from one node to another, and to resume processing from a stored state without duplicating work or data.

System load, in the form of messages to parse, varies over time. This means that spikes in traffic could possible overload consumers if message load increases too quickly for scaling measures to take effect.

To overcome this situation, messaging or transport systems are usually introduced into the infrastructure to be used as buffers or queues between the nodes producing data ("producers" or "producer nodes") and the nodes crunching numbers ("consumers" or "consumer nodes"), so that data can be put on "hold" when the rate of incoming data greatly exceeds the rate of processing data. To date there exists a wide variety of systems for messaging and transport of data, such as ActiveMQ [9], RabbitMQ [10], ZeroMQ [11], Cloudera's Flume [12], LinkedIn's Kafka [13], Facebook's Scribe [14] and Twitter's Kestrel [15] to name a few.

2.1.1 Requirements

From the scenario described above, the following requirements can be formulated:

Routing of data
: Messages sent from producers to consumers are routed dependent on a key attribute contained within each message, and thus all messages with the same key attribute should be consumed by the same consumer.

Construction of state
: Consumers use the messages to build some form of *transient* state which in turn is forwarded to another entity at a given interval (can be time-based, dependent on the number of messages consumed, etc.) and is too large to effectively store in a shared datastore.

Addition and removal of nodes
: Consumers can be added and removed from the system at any given time, thus the system should automatically redistribute the workload among the consumers without any loss of information and with as little duplicated work as possible.

Resuming processing at other nodes
: For consumers to be able to *resume* processing from other consumers in case of addition or removal of consumers, a representation of what a consumer has processed needs to be accessible and available to other consumers at all times. This can be achieved either through common off-node storage and/or by *hinted handover*[2] between consumers.

Managing fast load fluctuations
: While scaling can take care of load increases when load is scaled manually (such as when enabling the agent on a new site) the system also needs to handle temporary spikes, to do this a buffering layer needs to be present between consumers and producers to provide reliability in times of high system load.

[2]Hinted handover means that peers will transfer necessary information before shutting down, if given time to do so.

2.2 Tools and third party libraries

During development of the proof of concept implementation several tools and external libraries were used to ease the development process and for providing system coordination. This section aims to give a brief overview and description of these tools and libraries.

2.2.1 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud, or EC2, is a web service provided by Amazon and is designed to make web-scale computing easier for developers by providing resizable compute capacity in a manageable way, and giving administrators complete control of their resources [16].

EC2 provides a selection of environments such as Ubuntu, Fedora and other Linux distributions as well as Microsoft Windows servers. In addition Amazon provides storage, load balancing and a multitude of other services to go along with the computing power of EC2.

By providing a simple and quick interface for renting virtual servers it allows clients to scale at will while avoiding making up front investments which may become a liability if operations must be scaled back or if architecture changes at a later point.

2.2.2 Ruby and JRuby

Ruby is an interpreted, object-oriented, dynamically typed language from a family of so-called *scripting languages* [17], which includes programming languages such as Python, PHP and Perl. Ruby was created by Yukihiro 'matz' Matsumoto sometime around 1993 [17] and was first published in 1995. It blends different parts of Perl, Smalltalk, Eiffel, Ada and Lisp [18] together to form a new language with focus on simplicity and productivity.

As Ruby's focus has been on productivity and making programming more enjoyable for developers, it is not hugely efficient in terms of execution speed. One of the issues caused by this is that Ruby uses *green threads* and a Global Interpreter Lock (GIL) meaning that it is not possible for several CPU cores to execute the same Ruby code simultaneously, which is a serious issue when working with multiple threads in a data crunching system.

JRuby is a Java implementation of the Ruby language and runs on the Java Virtual Machine. As a consequence of this it brings true concurrency to the Ruby world using Java's native threads.

JRuby also provides access to countless Java libraries, extending the already large eco-system of Ruby to provide most everything a developer might need in terms of third party tools.

2.2.3 Apache ZooKeeper

Apache ZooKeeper ("ZooKeeper") is a service providing distributed synchronization and group services for distributed applications. Its aim is to remove the need for re-implementation of these primitives for new distributed projects.

Since such synchronization solutions are hard to build, it is favorable to have a well-tested and verified product to use instead of attempting to implement a new one. This avoids many of the race condition pitfalls and synchronization bugs that inevitably plague this kind of service in implementation.

ZooKeeper is run decentralized on a cluster of machines and provides redundance and versioning of data, ordering on creation and modification of data, as well as a notification system for clients that needs to be alerted when certain events happens.

Information in ZooKeeper is stored in a file-system-like tree, where each named node can contain a small amount of data (hard limit of 1 megabyte) and have a number of children. Nodes can also be specified as *ephemeral*, meaning that they will disappear if the client who created them is disconnected from ZooKeeper. Clients can also set *watches* on nodes to be notified whenever they or their children are modified in some way.

These properties along with guarantees of synchronization and redundancy allows for implementation of many distributed synchronization structures in a reliable and relatively simple way.

ZooKeeper is used by several well-known companies and projects such as Yahoo!, Zynga and Akka [19].

2.2.4 Apache Kafka

Apache Kafka ("Kafka") was originally created by LinkedIn and open sourced in 2011 [20], and is a message queue system designed to handle large amounts of data, both for stream and batch processing.

Compared to traditional messaging system such as RabbitMQ and ActiveMQ, Kafka is primarily designed to send messages from producers to consumers, not to facilitate communication back and forth amongst peers.

Kafka can be instructed to distribute received messages onto *topics*. Each topic is labelled with a name and consumers can read from specific topics. Topics can be thought of as queues and is the main way for a user of Kafka to manually split a stream of messages into smaller parts.

Furthermore, the user can also configure Kafka to divide each topic into a number of *partitions*, and can provide a method for partitioning messages of a topic onto these partitions. This allows several consumers to consume from the same topic while ensuring that each message is only provided to one consumer. By utilizing the partitioning method one can also make sure that messages of a certain type are all on the same partition.

Partitions are used to distribute Kafka's workload across brokers[3], while topics are used by users to label messages for consumers.

[3] A broker is a node running Kafka.

Kafka uses ZooKeeper for synchronization and coordination of brokers in a cluster. Producers and consumers use ZooKeeper to gain knowledge about which brokers are available, as well as for load-balancing and coordination.

Kafka also persistently stores messages, which allows the user to index backwards in a topic while still providing high throughput [21]. Kafka is used in production at LinkedIn, Tumblr [22] and several other big data companies [23]. For a more in-depth description of the design of Kafka see [24].

2.3 Previous work

While researching the subject before and at the start of the project it was quite difficult to find any previous work done that related to the specific use case, and no complete existing solution was found. Various parts of the use case has been solved for other scenarios before, such as consistently routing data given a set of nodes.

However, at a late stage of the project a framework was discovered that solves part of the use case for this thesis. Since it was discovered after the design proposed in this thesis was completed it was not taken into consideration while developing the solution. It will however be discussed and compared to the proposed design in detail in Section 5, and a brief introduction is given below.

Ordasity is an open source framework created by Boundary [25] and is designed for building reliable, stateful, clustered services. It is written in Scala, thus it runs on the JVM, and uses ZooKeeper for communcation and coordination among nodes in a cluster.

Ordasity solves some common issues for clustered services, such as cluster membership, distribution and load balancing of work among nodes as well as graceful "handoff" of work between nodes.

Chapter 3

Method

The project consisted of the four overlapping phases which are described in the following chapter.

Firstly, an information gathering phase took place where the "Big Data" and "Cloud Computing" domains were explored and different distributed designs and algorithms were researched to get an overview and feel for the project and challenges ahead. Secondly, after gaining more knowledge of the subject domain and available tools, algorithms for coordination and synchronization of nodes were designed. Thirdly, the algorithms were implemented and a *proof of concept* system built around the algorithms. Lastly, verification and benchmarking of the proof of concept system were performed to measure performance and stability of the system.

3.1 Research and gathering of information

"Big Data" and *"Cloud Computing"* have been hot topics for quite some time when it comes to distributed systems, with large companies such as Google, Twitter, Facebook and Amazon on the forefront regarding products and development. Contained within these broad terms are more complex subjects such as distributed synchronization, fault-tolerance and resilience, load-balancing, transport systems, distributed storage solutions and algorithms for distributed environments.

To gain better insight into this world and current development and progress being made, the first phase of the project revolved around information gathering.

For the gathering of information regarding current progress and development related to Big Data and Cloud Computing mainly white papers, articles and company tech blogs were used as sources of information.

Documentation, code and tests for various frameworks were studied in depth to gain knowledge in how they work under the hood and how people generally develop distributed systems.

3.2 Design of algorithms

When a general overview of the problem at hand had been acquired through the aforementioned information gathering phase, the first step in implementation was to determine a reasonable algorithm design for message distribution and state retention, as well as for managing cluster membership of nodes.

An algorithm inspired by *consistent hashing* [26, 27] was designed to facilitate consistent routing of messages in case of addition and removal of consumers. To manage membership of nodes in the system, a distributed algorithm was constructed which utilizes ZooKeeper and more specifically its ephemeral nodes and watches to notify each consumer when a peer joins or leaves the cluster. A strategy for handling the transient state of consumers in case of failure was based on representing each *stream* in the system as a topic with one partition in Kafka, and one node in ZooKeeper, and furthermore utilized the powerful indexing feature of Kafka. More details and explanations of the solutions are presented in Chapter 4.

After discussion and theorizing with coordinator and developers at Burt it was deemed ready for a trial implementation.

3.3 System implementation

When the algorithms were deemed feasible and reasonable in conjunction with personnel at Burt, development was started on a *proof of concept* implementation to test performance and reliability of the ideas.

Initial plan

At first a complete framework for transporting messages was designed and implementation started. After a first meeting with coordinator and chief architect at Burt this plan was deemed to be overly ambitious given the constraints of the project and the complexity of the algorithm. Plans were reset and building a framework was postponed until the algorithm and third party tools had been proven to work in the expected way.

Proof of concept implementation

After the initial plan was terminated, a new iteration started with a new plan which focused more on getting a system up and running as fast as possible. The goal with this kind of plan was to verify that the selected algorithm designs were viable and actually worked in practice.

3.4 Agile process

A basic agile development process was adopted to facilitate early results in testing the algorithm design. Changes, implementation choices and problems were discussed and decided upon and then implemented in an iterative fashion.

Test-driven development

To simplify the implementation and to allow for more frequent refactoring a test-driven approach was taken, all classes and interfaces were tested to make sure they conformed to the specified behaviour in the design.

Tests were created using the RSpec library [28], a testing framework for Ruby which provides an expressive DSL[1] for specifying behaviour of objects.

Prototyping

Several prototype implementations were made and discarded throughout the development process, especially on the more complicated consumer side. This was done to try out different architectures. Each new prototype was built using the lessons learned from its predecessor such that good parts were kept and various issues resolved through redesign of components.

Pair programming

While for the most part of the project responsibilities were divided so that producer and consumer were developed separately, there was frequent use of pair programming in solving problems and to get input on ideas and support for design decisions.

[1] Domain Specific Language

3.5 Verification

To verify that the developed proof of concept implementation worked and performed correctly as a whole, three main features of the system were considered:

(a) *routing of data between producer and consumer nodes*

(b) *balancing of streams*[2] *among consumer nodes*

(c) *resuming state in case of consumer node failure.*

3.5.1 Routing and flow of data

Firstly, the implementation of the consistent hash algorithm was unit tested using RSpec [28] to ensure that it would return the correct stream for a given key.

Secondly, to verify that data was being routed correctly a test system was set up consisting of one Kafka broker and a number of producers and consumers. The producers read lines from text files, performed some basic parsing of each line to find the key property that would be used for routing the parsed data to a stream and then subsequently sent the parsed data to the stream. The producers also kept track of how many lines were read from the text files, how many of the lines that were parsed correctly (correctness in this case is equal to the parsed data having a valid key property to use for routing) and finally how many messages that were sent.

On the other end, the consumers would read data from each claimed stream and count the number of messages seen and then proceed to *suspend*[3] each stream when there were no more messages to read from them. When all streams were suspended, the number of messages seen were summarized across all the streams. The consumers stored their persistent state, which in this case was simply the number of messages consumed, periodically in ZooKeeper after having consumed a number of messages.

Lastly, Kafka exposes a number of attributes via JMX[4], among others available were the *total number of messages in*, *total bytes written* and *total bytes read*. These attributes were fetched after the producers and consumers had been shut down.

The number of messages sent (counted by the producers) was compared to the number of messages in (from Kafka) and then compared to the number of messages seen at the consumers. As a final touch the remaining attributes from Kafka, being total bytes written and total bytes read, were compared to ensure that all data that was sent by the producers was also read by the consumers.

[2] See Section 4.1 for description and more details.

[3] Meaning that they would not attempt to read any more messages from the stream.

[4] Java Management Extensions, is technology that lets one implement management interfaces for applications running on the Java Virtual Machine [29]

3.5.2 Balancing of streams among consumer nodes

The balancing of streams was tested in two different ways with different granularity. The implementation itself was unit tested to ensure that each consumer node claimed a sufficient number of streams among the available streams.

Futhermore the implementation was tested by starting a number of consumer nodes and waiting until they all had reached a stable state. When the system reached a steady state, the number of claimed streams was compared to the number of available. Note however that the verification did indeed not take in consideration to verify that each consumer had an (approximately) correct number of streams, as this was verified with unit testing.

3.5.3 Resuming state in case of consumer node failure

Verifying that no messages were left *unconsumed* in the case one of several consumer node failures was done in a similar fashion as is described in Section 3.5.1, with some slight modifications.

After a random period of time one (or several) of the consumers was abruptly shutdown by issuing a kill signal to the process. The killed consumer's workload was redistributed among the surviving consumers, who fetched the stored state from ZooKeeper and effectively continued approximately from the point where the terminated consumer was interrupted.

Further, instead of comparing that the number of messages in were equal to the number of messages out, the comparison verified that the number of messages out were equal or greater than the number of messages in. The same applied to the total number of bytes read and written.

3.6 Experimental benchmarks

Performance of the proof of concept implementation was not a prioritized subject for this thesis project. However, performance of running the transport system, Kafka (see Section 2.2.4), in Amazon EC2 (see Section 2.2.1) was considered interesting and some time was devoted to running a number of benchmark experiments.

The main motivation as to why this would be interesting is the fact that there exists close to none benchmarks of running Kafka in a virtualized environment like Amazon EC2[5]. Accompanying this fact is running Kafka with a large number of topics. Though there has been some mentioning of a situation like this on one of the mailing lists for Kafka by Taylor Gautier at Tagged [30], although there is no mentioning of what kind of hardware or cluster setup they were using. T. Gautier briefly mentions that Tagged were running out of file descriptors, when using Kafka 0.7, because the "cleaner" that removes topics still left directories and files lying around and thus Kafka still kept the file handles open. They proceeded to implement a custom cleaner to get around

[5]That could be found at the time of writing.

this. This is hopefully something that will be resolved in newer versions of Kafka.

As the design proposed in this thesis benefits from having a large number of topics (see Section 4.1) it would definitely be interesting to see how Kafka performs with a large number of topics. Both of the aforementioned reasons most likely stems from the fact that Kafka is a fairly new creation, being open-sourced by LinkedIn in January of 2011 [20].

3.6.1 Metrics

When benchmarking systems that are meant to be used for transportation of data there are in general three different metrics that are obviously interesting:

(a) *send performance.*

(b) *receive performance.*

(c) *throughput performance.*

With regards to Kafka, send performance is equivalent to write performance to disk and receive performance is equivalent to read performance from disk. As a consequence of this the terms *write performance* and *read performance* will be used when presenting the results from the benchmark experiments in Section 4.5.

3.6.2 Setup of system

As mentioned in Section 3.6, the setup used Amazon EC2 and the associated instance types in the following way:

- 3 x *M1 Small* ("`m1.small`") instances for ZooKeeper nodes.
- 3 x *M1 Large* ("`m1.large`") instances for Kafka nodes.
- 3 x *M1 Extra Large* ("`m1.xlarge`") instances for Kafka nodes.
- 8 x *M1 Large* instances used for nodes producing and consuming data.

Various combinations of the last eight instances were used for producing and consuming data to and from the currently running Kafka cluster. Each producer node in the cluster parsed around 1.2 gigabytes of data, which contained 1 250 000 messages, meaning that each message was roughly 1 kilobyte.

The system used Kafka version 0.7.1 and ZooKeeper version 3.3.4, which came bundled with Kafka. Ephemeral storage was used on the Kafka nodes for storing data, without any kind of RAID configuration or the like. Also, all benchmarks were run without any type of replication as this feature of Kafka was not available at the time (scheduled for the 0.8 release of Kafka).

3.6.3 How were the benchmarks performed?

In general the benchmarks were performed in two separate rounds, the first round were running the Kafka cluster on `m1.large` instances and the second round used `m1.xlarge` instances.

As the benchmarks were of a very experimental nature, a slightly different set of benchmarks were run on the Kafka cluster using `m1.xlarge` instances compared to what was run on the `m1.large` instances. The main motivation for using two different Kafka cluster was, to some extent, to evaluate the difference between the two instance types.

As noted by the Kafka team in [21], producer and consumer benchmarks should be run separately since simultaneous production and consumption of data tends to improve performance as the page cache of the operating system is "hot". Thus, the benchmarks for producers and consumers were run separately, but to also test throughput there were some benchmarks were producers and consumers were run simultaneously.

Chapter 4

Result

The project resulted in an algoritm design for distributing messages and managing consumer membership, as well as a strategy for maintaining state and throughput in the event of consumer changes. These were implemented into a proof of concept framework, using Kafka (see section 2.2.4) and ZooKeeper (see section 2.2.3). Provided in the result is a description of the components involved in the message transportation and routing, distribution of work and persistence of state in case of node failure.

Also included are the results of benchmarking experiments that were performed to verify the throughput of Kafka when running in Amazon EC2.

4.1 Routing

To facilitate transfer of responsibility of a smaller part of the total data stream, the data stream needs to be partitioned into smaller subsets, these smaller subsets will be referred to as *streams*. These streams should also be approximately equal in terms of the workload they represent. Partitioning can be done in a number of ways, as long as all messages that belong together (for example all events that are part of the same session) are guaranteed to be in the same stream.

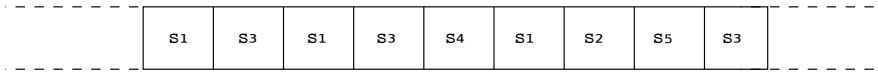

Figure 4.1: All sessions in a single stream

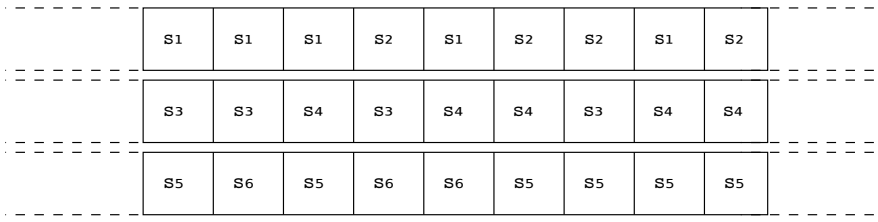

Figure 4.2: Sessions distributed over three streams

The number of streams should be larger than the number of consumers by a reasonably large factor since this allows for a higher granularity when distributing the streams across the consumers. If there is a large number of small streams, the impact of imbalanced distribution (a necessary evil) will be less than if the streams are of a larger size.

The need for related messages going to the same consumer, and therefore the same stream, sets an upper bound on the number of streams, depending on the cardinality of the attribute used to partition the stream. Given a large enough key space and the large amount of events and objects in most big data use cases this should not be a limitation.

The routing is relatively simple, messages are sent to one of a predefined number of streams based on simple hashing of a user-defined key included with the message. The algorithm used to partition the stream should be chosen so that the given keys are distributed evenly across the streams to make each stream contain approximately the same amount of work. It is then the consumers' responsibility to divide the streams amongst themselves in a fair way.

By handing the responsibility for distribution over to the consumers, producers become simple to scale up and down, and allowing the consumers to handle their own scaling without having to communicate with producers, which lowers the complexity of the system overall.

4.2 Distribution

The consumers are responsible for distributing responsibility of the streams, where producers put messages, amongst themselves. Intra-node communication is necessary to prevent conflicts where multiple consumers use the same stream at the same time.

4.2.1 Balancing of workload

Whenever the number of consumers changes, each consumer independently computes how many streams it should be responsible for based on the number of available streams and currently active consumers.

If the consumer determines that it has too few streams, as will be the case if a peer left the group and thus released its streams, the consumer then attempts to *claim* that many random streams by communicating to its peers that they are taken. If a claim is unsuccessful (because a peer claimed that stream after the consumer acquired a list of streams but before it was able to claim it) the consumer will try again until it is satisfied with its share of streams or there are no streams available.

While the claiming of a stream can fail, there is no chance of starvation, since no consumer will claim so many nodes as to starve another consumer. Deadlock is prevented since even if multiple consumers try to claim a stream, one of them will also acquire the stream, and thus the system as a whole will make progress.

In the opposite case where the consumer has too many streams (because a new peer joined the group) it will release streams until it reaches the target number of streams, allowing the new peer to claim the released streams.

After balancing streams the consumer will ensure that the number of consumers is still the same as before, if it has changed it will rerun the balancing algorithm if necessary.

Pseudocode for the balancing algorithm can be found in Appendix A.

4.2.2 Start up

Starting the system from fresh is a special case of regular balancing, each consumer will be notified whenever another consumer is started, so there will be a period of continous rebalancing at the start where not much work is done. This will however not be an especially long process, and during testing it has never taken more than a couple of seconds to reach a balanced state after starting a system with several consumers, though this naturally depends on the number of consumers and streams that are present in the system.

4.2.3 Overflow streams

When the number of streams is not evenly divisible by the number of consumers, there will be an *overflow* of streams. To ensure that all streams are claimed, but that the distribution of streams is still as even as possible, each consumer is only allowed to claim one overflow stream on a "first come first served" basis.

Before claiming or releasing streams, a consumer will calculate the number of overflow streams, which is the remainder when performing integer division of the number of streams by the number of consumers. It will try to claim each overflow stream in turn, until it has successfully claimed one or there are none

left. If it got an overflow stream, it increases its target number of streams by one before balancing.

Whenever rebalancing starts all notions of overflow streams are forgotten, and new overflow is computed as necessary.

4.3 State persistence

To make it possible to resume processing of a stream in case of the stream owner changing, it is necessary for the new owner to be able to receive a representation of the state the previous owner had when releasing the stream.

If the stream transfer is the result of a controlled increase or decrease in the number of consumers it is possible to use some form of controlled or hinted handover where the new stream owner contacts the previous one and requests a serialized representation of the current transient state. If however the previous owner crashed or has already shut down, it is necessary to be able to retrieve an up-to-date representation of the state from an off-node storage.

The state of a consumer node will in many use cases, including the one at Burt, be too large to frequently write to the off-node storage. Because of this it is necessary to represent the state in a more compact way.

4.3.1 Representing state

Using a transport system which allows indexing of the fragment streams, so that previously consumed data can be retrieved again allows a consumer to resume from the place in the stream where the previous owner left off. This means that instead of storing the complete active state of a consumer, one can store only the offset in the stream where consumption should be restarted.

In the proof of concept implementation there is a secondary piece of state which is stored along with the offset. It represents a list of keys of sessions which had already been processed and passed forward. By ignoring any fragments tagged with these keys, the consumer can reduce both duplicated work and duplication of data in the following processing. By combining the offset with this type of ignore list or some other user-defined data it is possible to represent a much larger state in a smaller amount of data, at the cost of some duplicated processing for the new stream owner.

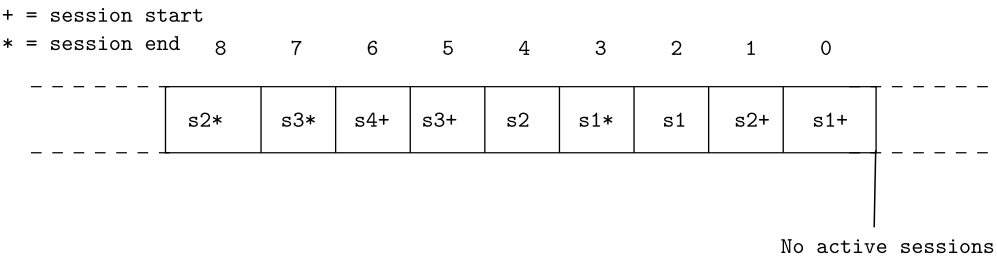

Figure 4.3: Sessions interleaved in a stream.

In Figure 4.3 above, we can see fragments from several session (`s1` through `s4`) interleaved in a stream. Consumption starts from the right at fragment 0, and there are no active sessions at that time.

Processing the first fragment of a session (marked with `+`) creates an object for that session, which is continually updated whenever a fragment for that session is encountered. When the final fragment (marked with a `*`, as denoted either by a timeout or by information in the fragment) is received, the object is finalized and passed on to the next stage of processing.

Whenever an object is finalized, the safe starting offset is updated in the off-node state storage, and the session is added to the list of sessions to ignore on resumption of the stream.

For example, after fragment 3 has been processed, session `s1` will be finished, which means that a consumer taking over the stream should start at fragment 1 (since `s2` is not yet finished) and ignore all fragments belonging to `s1`. When `s3` finishes after fragment 7, the safe offset cannot be moved forward, since `s2` is still active, however `s3` should be added to the list of sessions to ignore.

When `s2` finally finishes after fragment 8 is processed, the safe offset can finally be moved ahead to the start of the only active session, `s4`, and `s1` can be dropped from the sessions to be ignored, since it has no fragments after the safe offset, 6.

4.3.2 Resuming processing

When a consumer receives a new stream, and determines that it is not possible to retrieve state through a handover, it reads from the shared state store, and starts from the offset found there when consuming data. It also acquires whatever other information was stored in the state, so that it can filter out fragments that do not need to be consumed again.

The consumer is responsible for writing this data to the store at appropriate times based on the computations being performed on the data so that any consumer taking over responsibility for a stream can start at the latest possible offset when resuming. This data should probably be updated whenever any data is handed over to the next layer in the processing pipeline, changing the index to start from and/or the representation of fragments to ignore when resuming.

4.4 Coordination

Coordination between the consumers requires a reliable synchronization system to provide a mechanism for communicating claims on streams and notifying consumers of new peers joining or leaving the group.

Implementing a synchronization and coordination algorithm or system from scratch is by far not an easy feat and is assessed in more detail in Section 5.3.3. For the purpose of this thesis project, Apache ZooKeeper (see Section 2.2.3) was chosen for providing reliable synchronization and coordination among nodes in the system, and the following section aims to cover how ZooKeeper has been utilized.

4.4.1 ZNode structure

When the system is started, ZooKeeper nodes ("znodes") are created for all streams under the `active_streams` parent znode, to be used as a listing of which streams are generally available in the system. When a consumer joins the group, it creates an ephemeral znode under the `active_consumers` parent znode. This signifies that it is part of the group of consumers and should be considered when calculating how many streams each consumer should claim.

When a consumer wants to claim a stream it attempts to create an ephemeral znode under the `claimed_streams` parent znode with the name of the stream. If such a znode already exists the consumer will be notified and it can then retry by claiming another stream instead. When releasing a stream the corresponding znode under `active_streams` is simply deleted, signifying that it is now available for other consumers to claim.

4.4.2 Communication through ZooKeeper

By watching the `active_consumers` znode for changes to its child znodes, consumers are notified whenever a consumer joins or leaves the group, allowing them to start the balancing procedure previously mentioned.

Since both the consumers' znodes and the claimed streams' znodes are ephemeral, a consumer which is unexpectedly shutdown will leave its previously claimed streams available for other nodes, and balancing will be triggered since the consumer's own znode under `active_consumers` will disappear at the same time.

This structure can also be used when implementing handovers, to provide information about the previous owner of a stream, an address can be saved on the persistent znode for the stream (under `active_streams`) so that it is available even after the claim has been released.

4.5 Experimental benchmark results

This section presents the results from the benchmarks performed and is divided into three different parts reflecting the three metrics identified in Section 3.6.1.

4.5.1 Write performance

Figure 4.4 below shows send, or rather *write* performance, for a single Kafka node using four nodes with two threads each sending data. The graph also displays the difference in performance when using different settings for batching. As expected performance increases when sending data in larger batches.

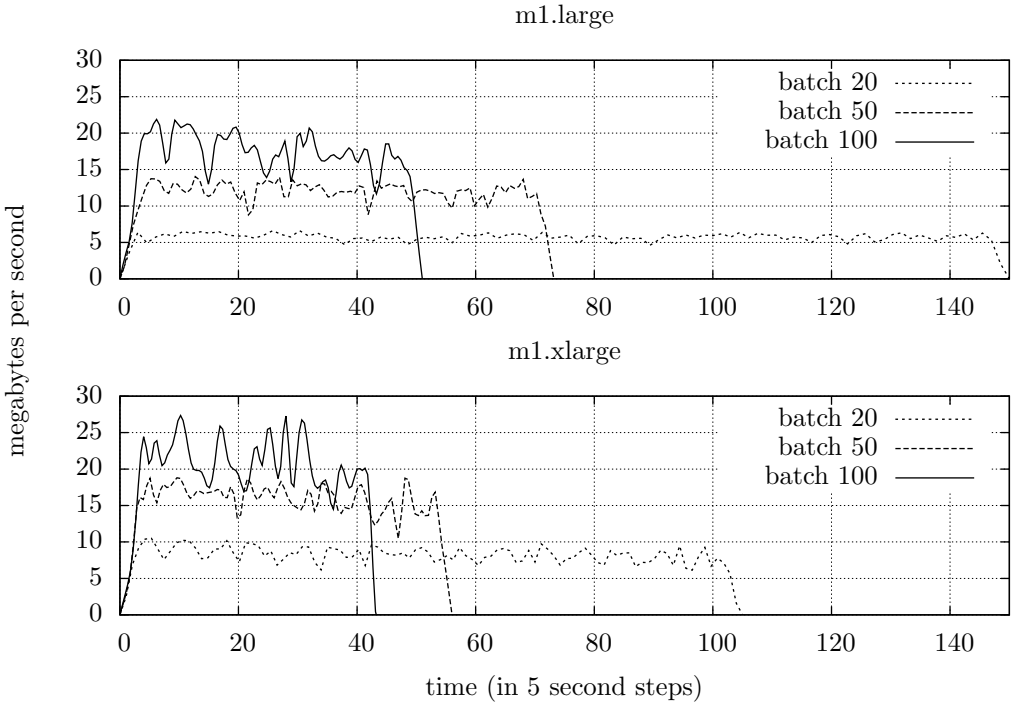

Figure 4.4: Send performance for one Kafka node using m1.large and m1.xlarge instance types, and four producers and 288 topics.

Evidently the `m1.xlarge` instances benefits from having twice the number of cores and memory compared to the `m1.large` instances [31], with regards to handling requests from clients.

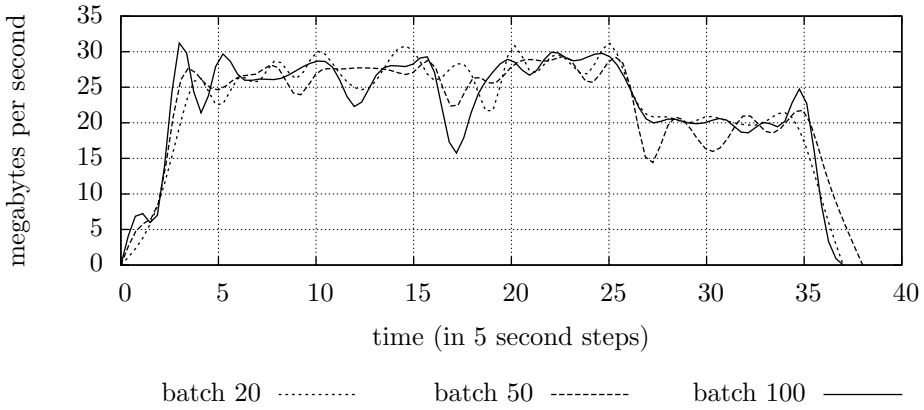

Figure 4.5: Send performance for Kafka cluster with 288 topics, using 4 producers.

Figure 4.5 above shows performance when running Kafka on a cluster consisting of three `m1.xlarge` nodes, with 288 topics distributed over the cluster. Using different levels of batching does not seem to affect performance to any great extent unlike what was seen in Figure 4.4. This is most likely explained by the fact that each Kafka node is only responsible for 72 topics instead of 288. Surprisingly however, performance is not that much greater than running a single Kafka node and using a batch level of 100 messages per batch. One possible reason for this behaviour could be that the producers cannot send data with a higher rate because of the preprocessing of data before being able to send it.

Flush rates

Two settings of Kafka that are configurable and that can influence performance are `log.flush.interval`, which controls the number of messages that are accumulated in each topic (and partition) before data is flushed to disk, and `log.default.flush.interval.ms`, which controls the maximum time that a message in any topic is kept in memory before flushed to disk [32].

Figure 4.6 shows performance when running a cluster of xlarge instances with two different `log.flush.interal` settings, 600 and 10 000 respectively. For this benchmark eight nodes using two threads each for sending data were used, and 24 topics were distributed over the cluster.

An interesting observation is that the performance is more than double compared to the other benchmark that used four nodes for producing data, and 288 topics. The largest impact is most likely the added number of producers, but the lower number of topics obviously affects as well.

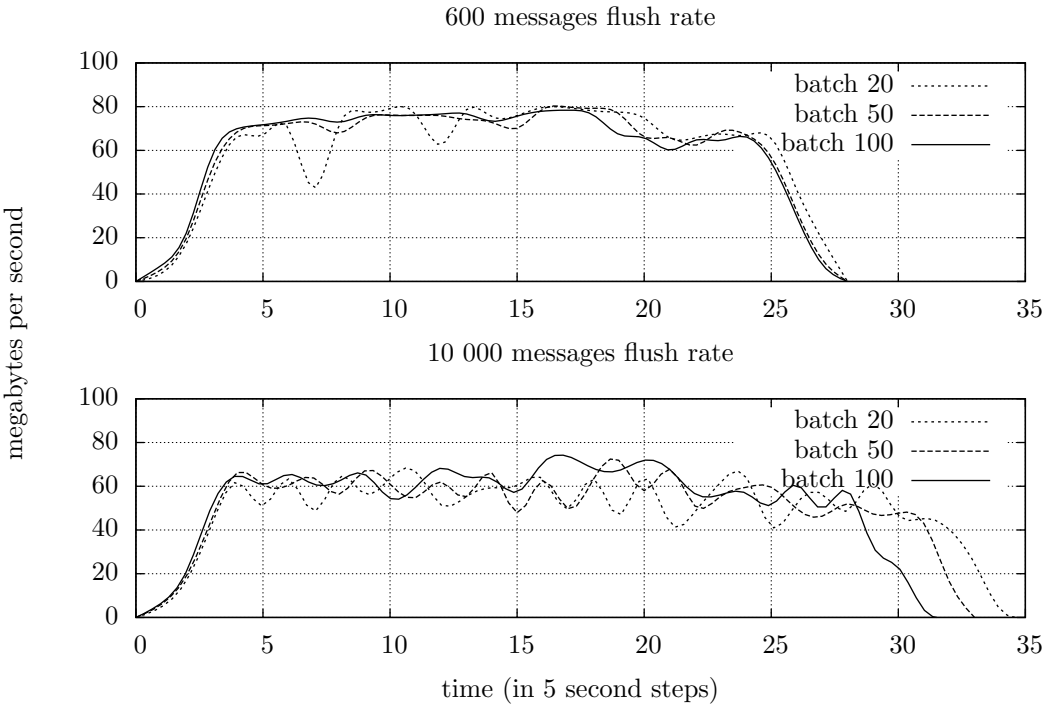

Figure 4.6: Send performance for Kafka cluster with 24 topics and different flush rates, using 8 producers.

One obvious difference between the `log.flush.interval` settings is that when set to 10 000 the performance fluctuates more than when it is set to 600. A possible reason for this might be that Kafka writes less often to disk, but when it eventually performs the writes they are larger chunks of data, which could possibly cause the behaviour exhibited in Figure 4.6.

4.5.2 Read performance

This section covers receive performance, or rather *read* performance, of running Kafka on a single node as well as on a cluster of nodes.

Figure 4.7 shows read performance when running one Kafka node on `m1.large` and `m1.xlarge` instances, with 288 topics. Results indicate that performance between `m1.large` and `m1.xlarge` instances are approximately equal, in contrast to the results from write performance.

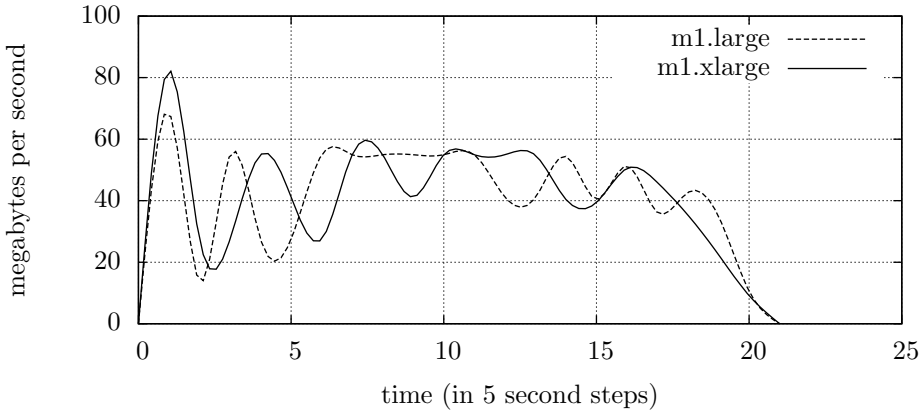

Figure 4.7: Receive performance for one Kafka node, using 4 consumers.

As a sidenote it should be mentioned that the peak followed by an steep decrease during the first 20 seconds of the benchmarks most likely comes from nodes receiving a lot of data and it takes some time to process the data before they can request more data.

Running the same benchmark on a cluster yielded similar performance as for a single node, though with a more consistent rate, which is to be expected since there are now three nodes sharing the load. However, using more nodes for consumption would most likely increase performance. In fact, some experiments were run with eight nodes for consumption rather than four, and during the experiments the read performance had peaks in the 80-100 megabytes per second range.

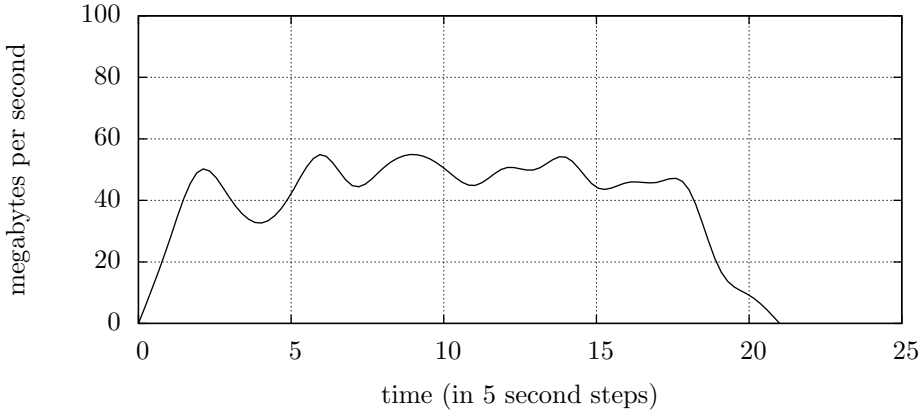

Figure 4.8: Receive performance for Kafka cluster, using 4 consumers.

4.5.3 Throughput performance

Figure 4.9 shows throughput performance when running four nodes for producing data and four nodes consuming data against a single Kafka node with 288 topics. Performance is quite a lot more consistent for the `m1.xlarge` instance compared to the `m1.large` instance. As mentioned in Section 4.5.1 the cause for this is most likely because of the increased number of cores and memory in the `m1.xlarge` instance, which allows it to process requests faster and more consistently.

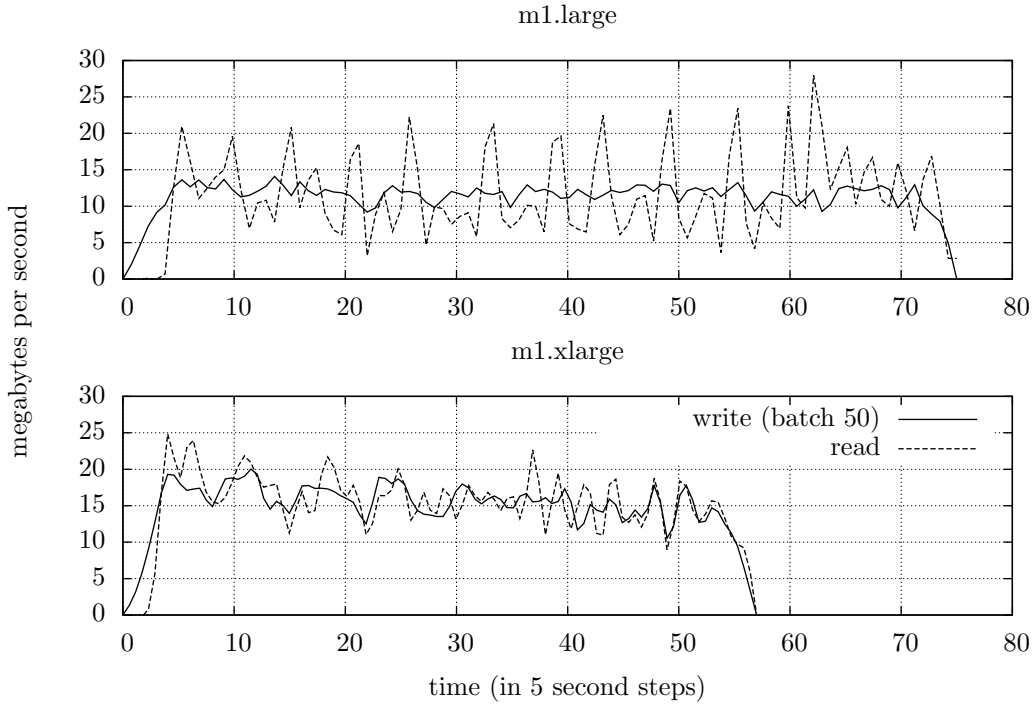

Figure 4.9: Throughput difference between large and xlarge instance types.

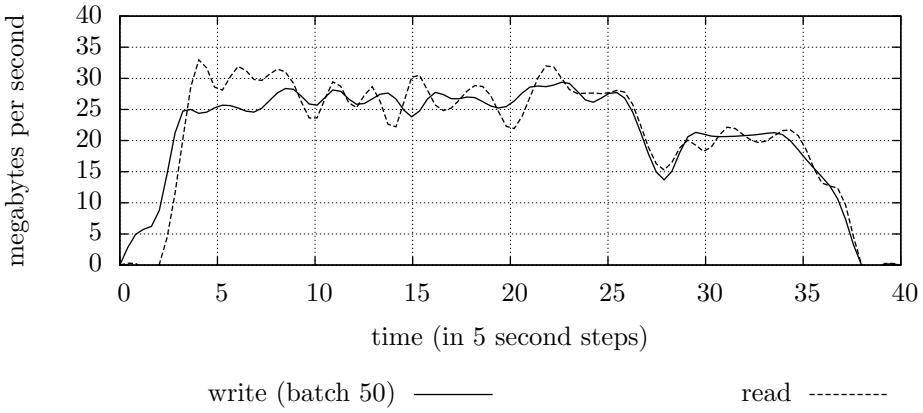

Figure 4.10: Throughput difference between large and xlarge instance types.

Finally, running the same benchmark using a cluster of Kafka nodes on `m1.xlarge` instances yields some pretty good results with consistent write and read performance. By increasing the number of producers and consumers, performance would probably be even better (i.e. even higher throughput). As could be seen in Figure 4.6 the write rate increased when using more producers, and increasing the batch size could furthermore increase performance. The drop in performance towards the end is because one of the prodcer nodes finished earlier than the other ones.

Chapter 5

Discussion

In this chapter, various topics that require more consideration or that are not addressed in other parts of the report will be discussed in greater depth.

- Firstly, the Ordasity framework created by Boundary will be discussed and compared to the proposed design outlined in this report.
- Secondly, issues and limitations with the proposed design will be presented and discussed.
- Thirdly, design choices regarding programming languages, synchronization and coordination of nodes and message delivery semantics of the proposed design and proof of concept implementation will be assessed.
- Fourthly, a brief rundown of the tools used during development of the proof of concept implementation and the experience from using them will be described.
- Finally, a section about future work and possible solutions to some of the issues outlined earlier will be described.

5.1 Ordasity comparison

As mentioned in Section 2.3, Ordasity is an open source framework developed by Boundary and was released publicly in October of 2011. The framework solves a subset of the issues outlined in this thesis, which most likely comes from the fact that the scope of Ordasity is more general and the subject of this thesis has a more specific use case. This section will present similarities and differences between Ordasity and the proposed design, and discuss how the proposed design could have been built on top of Ordasity if it would have been discovered at an earlier stage of the project.

5.1.1 Similarities

Most of the similarities between Ordasity and the design outlined in this report revolves around coordination and to some extent distribution of available workload.

Coordination and synchronization

Both solutions are using ZooKeeper for managing membership of the cluster, in very similar ways. Two of the key features of ZooKeeper are the ephemeral znodes and watches. Combining these together yields a robust and simple way to manage group membership in a cluster. So it is not surprising that there exist similarities related to cluster management.

Units of work

Ordasity has similar semantics when it comes to "work units" as they create znodes with the name of a work unit as child nodes of a designed path in ZooKeeper, which is again very similar to how streams are handled in the proposed design (see Section 4.4.1). The key difference is that all streams are created on system startup and the number of streams remain consistent until the system is shutdown (refer to Section 5.5 for further discussion on this subject).

Distribution of work load

When distributing work units, or streams, among the currently available nodes there are both similarities and differences between Ordasity and the design proposed in this thesis. Ordasity offers two different approaches to distributing load, one that is count-based and one that is load-based. The count-based strategy is similar to how streams are distributed among nodes in the proposed design, with the exception of how "overflow" streams are handled when they cannot be distributed equally among the nodes in the cluster. The proposed design does not currently offer any load-based distribution of work, but it is however something that could be added and will be discussed further in Section 5.5.

The reasoning behind the choice of overflow handling in the proposed design can be seen from the following example:

In a system with 5 consumer nodes and 13 work units (streams), Ordasity will allow all consumers to claim `round(13/5)` = three work units, this means that the first four consumers will claim three work units each, leaving only one for the slowest consumer, resulting in a very uneven work load. Using overflow streams as defined in Section 4.2.3, only the first three consumers will be able to claim overflow streams, resulting in three consumers with three streas and two with two streams. This problem will only continue to grow when the number of consumers and streams increases.

5.1.2 Differences

There are a number of differences between Ordasity and the proposed design, and to re-iterate, some are because of the different scope of Ordasity and some have been discussed as future features or extensions of the proposed design. It should be noted that all features that are explained in this section are configurable and are not enabled by default in Ordasity.

Automatic rebalancing

Ordasity offers automatic rebalancing of work, and even manual rebalancing if desired, however they recommend using the automatic feature instead of the manual [33]. The main reason for having an automatic rebalance option is to ensure that load is distributed as evenly as possible. However, for the proposed design it is assumed that the data is evenly distributed (see Section 1.3), thus there is no particular need for having an automatic (or rather scheduled) rebalancing option.

Graceful exiting of nodes

A quite interesting and nice feature of Ordasity is the *draining* feature. The purpose of this is to avoid increasing the load of an already-loaded cluster at once [33]. When a node is about to release work units, instead of releasing all work units at once and then shutting down it simply releases the work units over a defined time period (such as one minute).

For the use case of the proposed design it is assumed that whenever a node is about to leave the cluster it is to adapt to a decreased or decreasing total load for the cluster as a whole. It would however be benefical to have some sort of "handover duration" when transferring state between two nodes when a node is about to perform a graceful shutdown (see Section 5.5 for further discussion about handover).

Handoff of work between nodes

When a node has decided that it is going to release one or several work units, there is an option for nodes to handoff the work unit(s) directly to another node and thus eliminating the (small) gap that occurs when a node releases a work

unit and another node picks it up after being notified that there is a new work unit available.

Implementing a handoff, or rather handover, feature has been discussed during the project and is a possible extension of the proposed design and thus it is further assessed in Section 5.5. In short the handover feature would be useful for transferring the transient state, that is associated with a stream, between nodes in the cluster. This would lower the time when nodes are releasing streams because the node that is taking over a stream can start from the exact same position in the stream and does not have to retrieve the persistent state and perform the duplicated work of processing data that has already been processed.

Explicitly assigning work to nodes

When creating work units in ZooKeeper one has the option of assigning a specific work unit to a certain node. This is done by including the id of the node in the data saved at the znode for the work unit.

The proposed design has not seen any particular usage for this kind of feature but it would not be difficult to implement if the need was requested.

State management

Finally, Ordasity leaves management of state up to the user while the proposed design attempts to partly solve this issue. This is probably a quite conscious design choice by Boundary as Ordasity has a more general scope compared to the proposed design, where the main purpose is to provide a solution for a producer-consumer scenario where the focus is on transport and resumption of state.

5.1.3 Possible integration

If Ordasity would have been discovered earlier in the project, it would have been possible to build the proof of concept implementation on top of or integrated with Ordasity. Since the similarities are in fact really similar it would not have changed the overall design to any great extent, the main changes would have been implementation specific.

As the streams of the proposed design are stored in an identical way to how Ordasity manages work units, the routing and and *work claiming* of consumers nodes could work in the same way as in the proof of concept implementation.

5.2 Issues and limitations

Apart from the previously imposed limitations in Section 1.3, there are details that can be improved and/or which need to be taken into consideration when using the framework.

5.2.1 Number of streams

As mentioned in Section 4.1, a large number of streams are needed to make balancing as fair as possible. If the number of streams is not divisible by the number of consumers, the amount of extra work which needs to be performed by a consumer with an overflow stream is directly dependent on the number of streams in the system. The more streams the smaller the difference in work, and the more effective use of computing resources.

Changing the number of streams at runtime is another issue, while Kafka provides facilities for this it would require a lot of extraneous work to expand the algorithm for this use case. For most use cases it is possible to set the number of streams sufficiently high to make this a non-issue, or to restart the system with an increased amount of streams at the point where it actually becomes a problem. Some rudimentary ideas for solving this is presented in Section 5.5 below.

5.2.2 Throughput vs. scalability

When using Kafka, there is a tradeoff to be had between number of topics (corresponding to streams) and the throughput of the transport system. The reason for this is Kafka's reliance on file handles and writing to disk. For each topic, one file handle is needed, and when the number of topics grows large, this can reduce the throughput of the system. The reason why the increase in topics will affect the throughput of the system is because Kafka then has to write and read data from several different locations on disk. This causes some latency issues with mechanical disks as the *drive head* will have to move (or rather *seek*) between all locations. However, this will cause less impact when using solid state disks for example, though this will of course lead to higher hardware costs.

For the algorithm this translates into a tradeoff between a higher throughput and smoother scalability when using a larger amounts of stream, as mentioned above.

5.2.3 Reliance on third parties

While Kafka and ZooKeeper are useful, they are also third-party components, the evolution of which one cannot control. As with all projects reliance on third-parties is a liability. If development is stopped or goes in a diferent direction than

expected it can cause problems for the future operability and maintainability of the system.

Given the time and man power restrictions inherent in a master's thesis, it was not feasible to remove these dependencies, and both Kafka and ZooKeeper were deemed to be reliable and stable enough that it was not considered an issue. It would however be possible to remove the dependencies and replace them with systems built specifically for the purpose, but there are no guarantees that replacements will be better than or even as good as the originals.

5.3 Design choices

During the information gathering phase and initial system design, several choices were made regarding which technologies and strategies to use in implementing the proposed design. Most choices were made based on the practicality of the systems, maximizing the utility to be had from each third party component to reduce the strain of implementation and allow for focus to be put on a powerful and correct algorithm.

5.3.1 Development specifics

Development was performed in a Linux environment since the project team has been using Linux as main operating system for the last couple of years and the target platform for a possible finished product was servers running UNIX/Linux. Several of the third party libraries are also most easily usable on UNIX/Linux.

Ruby (and JRuby) is the language of choice at Burt and it is well-suited for the agile development methods used in the project. JRuby specifically runs on the Java Virtual Machine which allows for easy access to powerful threading and good garbage collection without incurring the extraneous integration and communication costs of developing and running Java applications.

5.3.2 Transport system

The Kafka transport system provides a robust mechanism for searching backwards in a stream of messages, which is very useful for the proposed use case since it removes the need to implement such a system on top of any other transport or messaging solution.

Kafka is also industry tested and has good throughput, both important traits when handling large amounts of data.

Topics and partitions

Kafka provides two levels of partitioning for the data sent to it, topics and partitions. Topics are named subsections served by the producers. The topics are then further divided into a user defined number of partitions, to distribute the load on the Kafka brokers.

For the development of the proof of concept implementation, streams were represented by topics having one partition each. While it would have been possible to represent the streams through partitions, it would not have given any advantages since partitions are not named and cannot be added or removed at runtime whereas topics can.

5.3.3 Synchronization and coordination of nodes

For coordination of nodes there were two obvious choices:

Decentralized A fully decentralized solution where nodes communicate directly to each other using raw sockets, or some form of further abstraction such as ZeroMQ or Akka.

Centralized A centralized solution where nodes talk to a centralized, external service to coordinate their actions.

For the decentralized variant mainly two different solutions were considered, one using a consensus algorithm such as the Paxos algorithm [34] ("Paxos") and one using a gossip-style protocol for reaching consensus among nodes. The main motivation for using a decentralized solution is the reduced hardware cost compared to running a centralized solution on separate matchines. Another benefit is that the complete system will contain less external dependencies when using a decentralized solution and thus less machines that can fail.

Decentralized solution using Paxos

In 2007 Google presented a paper [35] detailing their experience in building a fault-tolerant database using Paxos. It turned out to be a non-trival feat to implement a production system despite the extensive literature available on the subject. The Paxos algorithm is about one page of pseudo-code but their complete implementation consisted of several thousand lines of C++ code, which comes from the fact that they had to implement a number of features and optimizations to make it into a practical and production-ready system. The main usage of Paxos in [35] was to build a fault-tolerant log of actions taken for the database and group membership of nodes in the system. A similar strategy of using a fault-tolerant log to record actions of the system could have been used for this thesis project and Paxos would have been used to coordinate mappings between consumers and streams. In short one consumer would act as a "proposer" and either propose each mapping separately or more likely create all mappings, propose the mappings as a single value and have all other nodes agree on this value. However, as it took the team at Google considerable time and effort to create a production-ready system with all extra features and optimizations to have the system perform in an efficient manner it was not deemed feasible to implement a solution using Paxos for coordinating nodes within the limited time for this project.

Decentralized solution using a gossip protocol

There exists a quite large body of literature when it comes to gossip protocols, and a Google search gives a quick overview of services built using this style of protocols [36, 37], and their correctness [38] and robustness [39].

Two strategies were considered for a gossip solution, either one consumer would act as a leader and, similar to the solution using Paxos, "gossip" mappings of consumers and streams, and eventually all consumers would know about the mappings and thus which streams they would be responsible for. Another option would be to have each consumer gossip the streams that it was trying to claim. This would however likely lead to a lot of clashes and an algorithm for solving clashes (perhaps using vector clocks) would have been necessary.

No generally applicable framework for building systems using gossip-style protocols was found for neither Java nor Ruby and thus such a framework would have to be created from scratch. To absorb all necessary knowledge, design and implement a framework from scratch would have been as intricate as developing a solution using Paxos, and thus it was not deemed feasible to implement a solution using a gossip-style protocol within the given time frame for this project.

Centralized solution

In contrast to the decentralized solutions, a solution using a centralized service for coordination and synchronization costs more in processing power and therefore is a more expensive solution. However, it is easier to handle synchronization when a single entity is responsible for managing it, though it should be mentioned that implementing such as service is still not an easy task. Fortunately there exists third party solutions that can be applied to the problem at hand. Kafka, among a lot of other applications and organizations, utilizes ZooKeeper for synchronization and coordination, which tends to indicate stability and quality of an implementation.

ZooKeeper supported the primitives that were needed by the proposed design and is as mentioned already required for running Kafka and choosing it was not a hard decision.

5.3.4 Message delivery semantics

As with all distributed systems there is a choice to be made between *at-least-once*, *at-most-once* and *exactly-once* semantics for delivery of messages. Exactly-once semantics are difficult to achieve and requires a mechanism for consensus among the nodes in a system. Examples of such mechanisms include the two- or three-phase commit protocols [40] and variants of the Paxos algorithm. They are however quite complex and have their own drawbacks, such as multiple round trips (which obviously leads to increased latency in the system) and sometimes poor guarantees of liveness, which leaves at-least-once and at-most-once semantics.

In the proposed design the differentiation between at-least-once and at-most-once semantics is the ordering between delivering data to the next stage of processing and storing of the persistable state. If results are passed forward first, at-least-once will be achieved while delivering the persistable state to off-node storage first will result in at-most-once behaviour.

For Burt, at-least-once was deemed most appropriate since data loss is less manageable than data duplication, but in a final system this will be configurable by the user as they are in charge of when to deliver data to the next stage and when to save persistable state.

5.4 Tools

As described in Section 2.2, a selection of third-party tools and supporting systems were used in the implementation of the project. This section contains a rundown and discussion of the experience had while using them.

5.4.1 Apache Kafka

Kafka has shown great performance numbers and offsets into topics have been vital for the state persistence in the proof of concept implementation and works as expected. As a data transport layer it has a large number of advantages over something like RabbitMQ, Kafkas design focus on transporting data instead of passing messages shows clearly, and it is absolutely a good choice for streaming data.

Even given the above it is still not in a stable state and lacks a few features. Most sorely needed is probably replication, to be truly reliable, data needs to be replicated across the brokers. This feature is coming along with several other improvements to the general usage of Kafka. The fact that all these changes are ongoing is another sign of Kafka not being as mature as other messaging solutions.

In conclusion Kafka feels like a great step in the right direction for data transport, it will soon be a great choice for any users in the domain.

5.4.2 Apache ZooKeeper

There has been no issues with ZooKeeper during the project, it worked flawlessly in regard to performance as well as stability. Actual operating of ZooKeeper has also worked well, all libraries have worked well and the documentation has been detailed and well-structured.

The node-tree approach of communication and the watch system along with ephemeral nodes has provided a very powerful and easily adapted synchronization system for the design. Without ZooKeeper the implementation would not have gotten as far as it did.

5.5 Future work

As mentioned above in Section 5.2 there are some issues that one may want to handle in a framework built around the ideas presented in this report. This section contains a description and discussion of some points which could be implemented in such a framework.

5.5.1 Handover of transient state

Controlled or hinted handover was mentioned previously in Section 4.4, but is an important concept since it would reduce the duplication of work by a large amount when streams are redistributed.

A suggested implementation would create a transferable representation of a stream's state when that stream was released. An address to the previous owner of a stream would be stored on the persistent znode for the stream, so that when the stream was claimed by another consumer, the address would be read from ZooKeeper and a request sent to the previous owner to transfer the state of the stream. This transfer would preferably be performed via a lightweight messaging system, such as ZeroMQ [11].

5.5.2 Changing number of topics at runtime

Increasing the amount of streams at runtime would necessarily involve dividing the existing streams into multiple smaller streams. This introduces several problems which will be presented below.

Splitting responsibility is not a great problem, whichever node controlled the original stream now controls both the "child" streams. However there is a need for communicating that the split has happened and that the claims should be updated some how.

Determining switchover point is more involved since it will need to be done in such a way that each consumer knows which objects belong to which stream. This can then be used to determine when it is safe to transfer a stream to another consumer (i.e. when all objects started before the split are finished and sent forward). At this point a regular handover of a stream can be done.

Splitting state is an alternative method which can remove the need for waiting until all previously started objects are finished. By somehow determining which objects belong to which streams a state for each stream can be constructed and transfer of streams can start immediately.

This is not a comprehensive listing of issues, and solutions above have not been proven as correct, but they give an outline of how splitting streams could be done and which problems are likely to be encountered.

5.5.3 Reduce reliance on dependencies

The proof of concept implementation relies heavily on both Kafka and ZooKeeper. While Kafka provides many benefits and is a requirement for the state representation used in the proof of concept implementation, it is as of yet in a quite volatile state in its development. While work is progressing at a fair pace, one might want to use a more stable transport system as a base for a future framework.

While ZooKeeper provides a very nice and easy model for synchronization and has not posed any great problems during the time of the project it does require adding a number of additional machines to the cluster running the transport layer. It also adds another service that can fail, and this may or may not be a problem, but it will always be a cost and it might be worth the effort of designing a gossip protocol or other cluster-internal method of synchronization.

As with ZooKeeper, storing persistent state on a separate cluster from the actual consumer nodes is beneficial for reducing consumer code complexity. However, it does add hardware requirements and again, yet another service that can fail. Since the performance demands of the storage will be relatively low it might benefit the user to keep this storage in a replicated storage on the consumers themselves, using a distributed database such as Riak [41] or MongoDB [42], utilizing the database systems' replication mechanisms to provide redundancy in case of consumer crashes.

5.5.4 Advanced options for work distribution

While the count-based balancing solution presented in Section 4.2.1 is viable for the case defined by the limitations, where workload is evenly distributed over the key space, it would be beneficial to have other options for balancing metrics. As mentioned above in Section 5.1, Ordasity provides a balancing system based on the reported workload of the system. Something like this could be very useful and has been discussed at several points during the project.

To alleviate the issues of heterogeneous hardware configuration it is possible to run multiple consumer nodes on the same physical (or virtual) machine, to increase the workload on a larger machine. However, balancing support for this would of course be an improvement.

Chapter 6

Conclusion

The aim of the thesis was to develop a solution to the problem of routing data to stateful consumers in such a way that no data is lost in case of a consumer crash, and so that work is distributed evenly across consumers.

By partitioning the data stream into multiple smaller streams, and letting the consumers distribute these amongst themselves, routing and work distribution was achieved. By representing the state of each stream as an index in its own Kafka topic, it became possible to store the state often enough that it could be resumed whenever a crash occurred.

The proof of concept implementation showed that the throughput and resilience of the proposed solution were viable and that it is a theory worth pursuing.

In addition to the proposed solution for the consumer problem, the benchmarking experiments provided insights into Kafka's performance on EC2 which can be useful to a wide range of users.

Appendix A

Balance pseudocode

```
def balance(claimed_streams)
  drop_claimed_overflow_streams

  num_consumers = count_consumers
  num_streams = count_all_streams
  num_current_streams = claimed_streams.count

  num_desired_streams = num_streams % num_consumers
  num_overflow_streams = num_streams / num_consumers

  for i in range 1..num_overflow_streams
    if try_claim_overflow_stream(i)
      # if claim sucessful, we should have an extra stream
      num_desired_streams += 1
      break
    end
  end

  claimed_streams = current_streams

  if num_current_streams < num_desired_streams
    while claimed_streams.count < num_desired_streams
      streams_missing = num_desired_streams - num_current_streams
      claimed_streams += try_claim_streams(streams_missing)
      # Someone else might claim streams while we aren't looking
      # if so, we need more.
  else if num_current_streams > num_desired_streams
    drop_streams(num_current_streams - num_desired_streams)
  end
end
```

References

[1] *What is Activity Data?* Feb. 11, 2011. URL: http://www.activitydata.org/What_is_Activity_Data.html#What_is_Activity_Data (visited on 12/03/2012).

[2] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[3] *Apache Hadoop*. URL: http://hadoop.apache.org/ (visited on 08/25/2012).

[4] *Storm*. URL: https://github.com/nathanmarz/storm (visited on 08/25/2012).

[5] *S4: Distributed Stream Computing Platform*. URL: http://incubator.apache.org/s4/ (visited on 08/25/2012).

[6] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular ACTOR formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

[7] *Akka*. URL: http://akka.io/ (visited on 08/25/2012).

[8] *Celluloid*. URL: http://celluloid.io/ (visited on 08/25/2012).

[9] *Apache ActiveMQ*. URL: http://activemq.apache.org/ (visited on 08/22/2012).

[10] *RabbitMQ - Messaging that just works*. URL: http://www.rabbitmq.com/ (visited on 08/22/2012).

[11] *ZeroMQ*. URL: http://www.zeromq.org/ (visited on 08/22/2012).

[12] *Apache Flume*. URL: http://flume.apache.org/ (visited on 08/22/2012).

[13] Jay Kreps, Neha Narkhede, and Jun Rao. *Kafka: a Distributed Messaging System for Log Processing*. June 2011.

[14] *Facebook's Scribe*. URL: https://github.com/facebook/scribe (visited on 08/22/2012).

[15] *Kestrel*. URL: https://github.com/robey/kestrel (visited on 08/22/2012).

[16] *Amazon Elastic Compute Cloud (Amazon EC2)*. URL: http://aws.amazon.com/ec2/ (visited on 11/25/2012).

[17] Bruce A. Tate. *Seven Languages in Seven Weeks*. Pragmatic Programmers, LLC, 2011.

[18] *Ruby Programming Language*. URL: http://www.ruby-lang.org/en/ (visited on 11/25/2012).

[19] *Projects powered by ZooKeeper*. URL: https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy (visited on 08/21/2012).

[20] Jun Rao. *Open-sourcing Kafka, LinkedIn's distributed message queue*. URL: http://blog.linkedin.com/2011/01/11/open-source-linkedin-kafka/ (visited on 08/21/2012).

[21] *Performance Results*. URL: http://incubator.apache.org/kafka/performance.html (visited on 08/21/2012).

[22] Todd Hoff. *Tumblr Architecture - 15 Billion Page Views a Month and Harder to Scale than Twitter*. URL: http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html (visited on 08/21/2012).

[23] *Applications and organizations using Apache Kafka*. URL: https://cwiki.apache.org/confluence/display/KAFKA/Powered+By (visited on 08/21/2012).

[24] *Apache Kafka*. URL: http://kafka.apache.org/design.html.

[25] Scott Andreas. *Ordasity: Building Stateful Clustered Services on the JVM*. Oct. 20, 2011. URL: http://boundary.com/blog/2011/10/20/ordasity-building-stateful-clustered-services-on-the-jvm/ (visited on 12/08/2012).

[26] David Karger et al. "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. STOC '97. El Paso, Texas, United States: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[27] David Karger et al. "Web caching with consistent hashing". In: *Computer Networks* 31.11–16 (1999), pp. 1203 –1213. ISSN: 1389-1286. DOI: 10.1016/S1389-1286(99)00055-9. URL: http://www.sciencedirect.com/science/article/pii/S1389128699000559.

[28] *RSpec*. URL: http://rspec.info/ (visited on 11/28/2012).

[29] Daniel Fuchs. *What is JMX?* URL: https://blogs.oracle.com/jmxetc/entry/what_is_jmx (visited on 11/26/2012).

[30] Taylor Gautier. "Re: Thousands of topics". July 31, 2012. URL: http://mail-archives.apache.org/mod_mbox/incubator-kafka-users/201207.mbox/%3CCAFy60Pc8i_7V5qTFUNoYRmytEzEGz5Bf8QO=gMRAaODFtuE3NQ@mail.gmail.com%3E (visited on 08/27/2012).

[31] *Amazon EC2 Instance Types*. URL: http://aws.amazon.com/ec2/instance-types/ (visited on 11/25/2012).

[32] *Kafka configuration*. URL: http://incubator.apache.org/kafka/configuration.html (visited on 08/21/2012).

[33] *boundary/ordasity - GitHub*. URL: https://github.com/boundary/ordasity (visited on 12/08/2012).

[34] Leslie Lamport. "Paxos Made Simple". In: *ACM SIGACT News* 32.4 (2001), 18–25. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.3093&rep=rep1&type=pdf.

[35] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. PODC '07. Portland, Oregon, USA: ACM, 2007, pp. 398–407. ISBN: 978-1-59593-616-5. DOI: 10.1145/1281100.1281103. URL: http://doi.acm.org/10.1145/1281100.1281103.

[36] Rajagopal Subramaniyan et al. "GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems". In: *Cluster Computing* 9.1 (Jan. 2006), pp. 101–120. ISSN: 1386-7857. DOI: 10.1007/s10586-006-4900-5. URL: http://dx.doi.org/10.1007/s10586-006-4900-5.

[37] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. *A Gossip-Style Failure Detection Service*. Tech. rep. Ithaca, NY, USA, 1998.

[38] André Allavena, Alan Demers, and John E. Hopcroft. "Correctness of a gossip based membership protocol". In: *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. PODC '05. Las Vegas, NV, USA: ACM, 2005, pp. 292–301. ISBN: 1-58113-994-2. DOI: 10.1145/1073814.1073871. URL: http://doi.acm.org/10.1145/1073814.1073871.

[39] Lorenzo Alvisi et al. "How robust are gossip-based communication protocols?" In: *SIGOPS Oper. Syst. Rev.* 41.5 (Oct. 2007), pp. 14–18. ISSN: 0163-5980. DOI: 10.1145/1317379.1317383. URL: http://doi.acm.org/10.1145/1317379.1317383.

[40] Henry Robinson. *Consensus Protocols: Three-phase Commit*. Nov. 29, 2008. URL: http://the-paper-trail.org/blog/consensus-protocols-three-phase-commit/ (visited on 08/26/2012).

[41] *Riak — Basho*. URL: http://basho.com/riak/ (visited on 11/29/2012).

[42] *MongoDB*. URL: http://www.mongodb.org/ (visited on 11/29/2012).