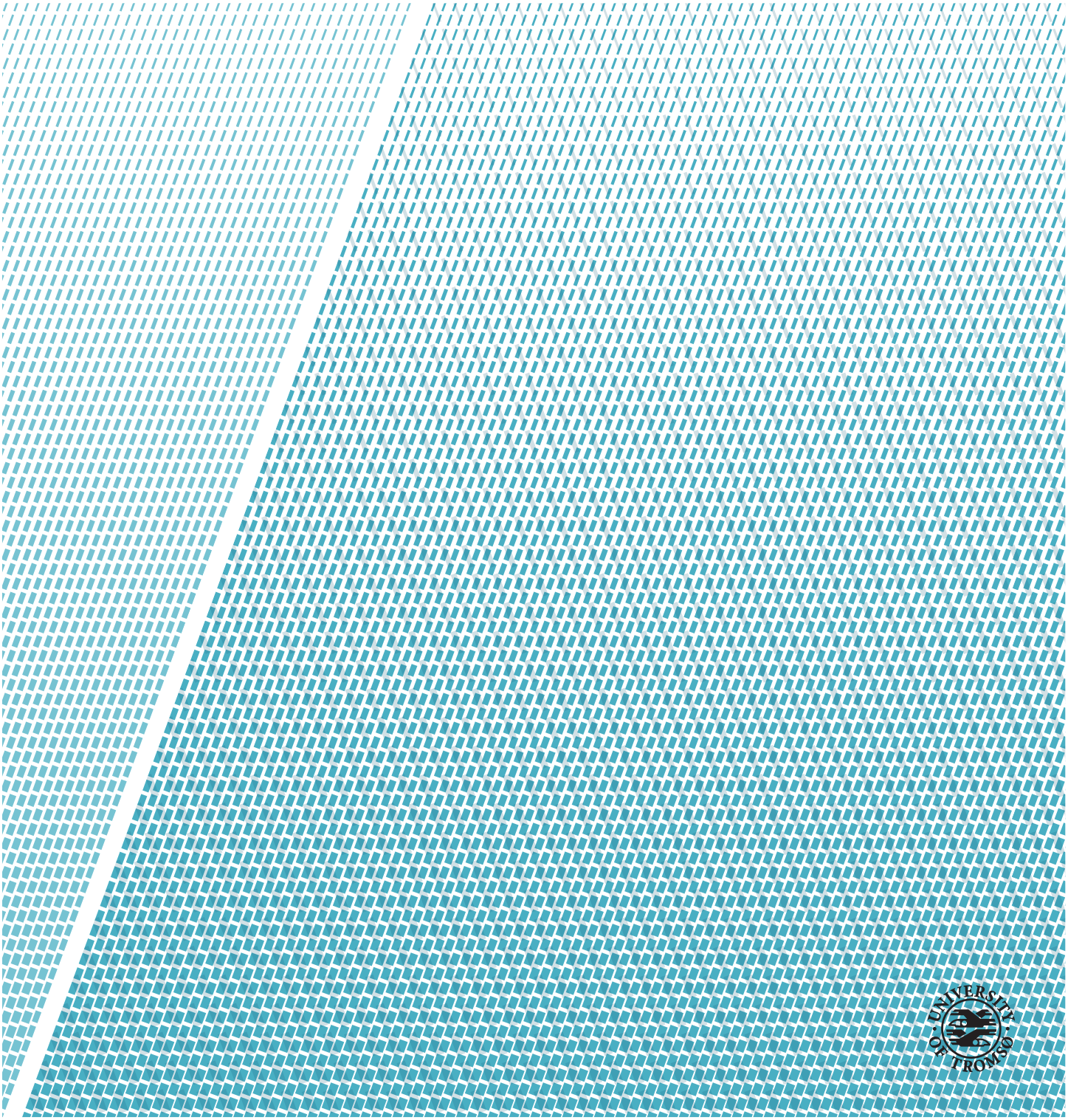
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology

Eatnu: A storage system for evaluating and persisting sensor data

Magnus Stenhaug

INF-3981 Master's thesis in Computer Science, May 2014



Abstract

The amount of information generated exceeds the current available storage. Big Data, the Internet of Things and the increasing popularity of self-tracking gadgets call for new storage solutions to manage and analyze the data.

To handle the constant flow of information, we have implemented Eatnu. Eatnu is a storage system designed to handle large data streams, where programmers can specify what parts of the stream to persist to disk.

Acknowledgements

I would like to thank my advisor, Prof. Dag Johansen for his ideas, input and motivation throughout these last three years at the university.

Thanks to the people of the iAD lab in Tromsø for providing an excellent working environment. Thanks to Erlend Helland Graff for reviewing my implementation and for the countless discussions on the C programming language. Thanks to Bjørn Fjukstad, Einar Holsbø, Jan-Ove 'Kuken' Karlberg, Ida Jaklin Johansen and the rest of my classmates.

Finally, I would like to thank my girlfriend, friends and family for supporting me throughout my life as a student.

Contents

List of Figures

List of Tables

List of Abbreviations

API Application programming interface

DBMS Database Management System

GFS Google file system

GPS Global Positioning System

HDFS Hadoop Distributed File System

IOT Internet of Things

NAS Network-attached storage

NFC Near field communication

PHR Personal health records

RFID Radio-frequency identification

RPE Borg Rating of Perceived Exertion

SLA service level agreement

TIL Tromsø Idrettslag

WAS Windows Azure Storage

ZAB ZooKeeper Atomic Broadcast

İAD Information Access Disruption

1

Introduction

The amount of information generated exceeded the available storage capacity in 2007[27], and IBM estimates that by 2020, we have created 40 Zettabytes[1] of data [2]. With prices for Internet going down, high-speed speed Internet is becoming a household item. We use the Internet to access social networks, share and view multimedia content such as video and music, play online games and messaging with other users. When we access these services we generate large volumes of data. As an example, the Facebook Data Warehouse receives 600 Terabytes of data every day[8]. Companies like Google and Facebook are mining the vast amount of information at their disposal, such as search logs, messaging and images, to enhance the quality of their services. These services generate revenue by offering an advertisement platform that can target the individual interests of the user. These datasets are rich, complex, and can bring value to companies that are able to efficiently process and analyze the data. The term *Big data* is used to describe these new collections of data, and can be described using the four V's of big data:[3] *Volume*(scale of data), *velocity*(analysis of streaming data), *variety*(different forms of data) and *veracity*(uncertainty of data). The overload of information driven by big data bring a whole set of challenges for the research community:[4]

1. 1 Zettabyte = 1 billion terabytes
2. http://www.ibm.com/software/data/bigdata/
3. http://www.ibmbigdatahub.com/infographic/four-vs-big-data
4. http://research.microsoft.com/en-us/projects/bigdataanalytics/

1. How do we process information that is generated with such a high velocity that it needs to be processed as it arrives?

2. How do we handle volumes of information that exceed the capacity of a single machine and needs to be spread accross multiple machines?

3. How can we efficiently extract knowledge from the information available?

The use of smartphones and tablets is becoming increasingly prevalent, with mobile Internet usage accounting for 25% of page views in 2014.[5] Another emerging trend is the use of wearable body sensors and self-tracking gadgets. These devices capture and store information on our day-to-day activities. The Fitbit Flex[6] and the Jawbone UP[7] are examples of wearable pedometers that can capture steps and sleep quality. The Google Glass[8] can capture images, sound, video and movement, and connect to the Internet with a range of applications including search, social and maps. Smartphones with built in accelerometers and Global Positioning System (GPS) can be used as an alternative to dedicated devices. RunKeeper[9] and Strava[10] are smarphone applications that let users track fitness activities with the built in GPS. The applications upload the positional data to a centralized site, users can view and share detailed information of their activities. The quantified self movement[11] aims at capturing every aspect of a persons' daily life, such as activities, diet, mood and sleep. This rich dataset can be beneficial to individual users as well as the general public.

The Internet of Things (IoT) denotes a new set of devices, or things, that are connected to the Internet. These are everyday objects that we rely on and interact with on a daily basis, ranging from sensors, such as temperature gauges, to cars and houses. These devices interact and share knowledge with us and each other, and Gartner estimates that IoT devices will grow to 26 billion units by 2020[4]. With IoT we can envision being able to monitor the position of our recently ordered package in real-time, and having sensors monitoring our physical well-being.

Clouds offer a wide range of services for companies that want to store and process large amounts of data. Cloud providers typically host these services in large data centers with thousands of machines. Microsoft, Amazon, IBM, Google and Oracle Cloud are examples of vendors that currently offer cloud

5. http://www.kpcb.com/internet-trends
6. http://www.fitbit.com/uk/flex
7. https://jawbone.com/up
8. http://www.google.com/glass/start/
9. http://runkeeper.com/
10. http://www.strava.com/
11. http://antephase.com/quantifiedself

services in one or more data centers spread geographically. The cloud provider is responsible for maintenance, power, cooling and networking. The services provided span the entire software stack, ranging from virtual machines to specialized software. Users can rent these services on a metered basis. A service level agreement (SLA) is a contract between a cloud provider and the user that describe the service provided by the vendor, typically in the form of measurable factors such as uptime, latency and throughput. The primary costs of a running a datacenter are servers, infrastructure, power and network with their respective estimated amortized cost of 45%, 25%, 15% and 15%[31]. With high costs, an important goal for cloud providers is achieving high utilization of resources in a datacenter. Higher utilization increases the profit margin, but may come at the cost of violating SLAs.

Using a cloud can help companies build and deploy services with a modest upfront investment compared to hosting everything locally. Storing and processing large amounts of data can be costly, and application developers need to consider the trade off between application needs and cost. For a large scale service running in the cloud, minor implementation details such as verbose logging to disk can amount to a substantial cost.

This thesis present Eatnu,[12] build to handle large streams of sensor data.

1.1 Problem Definition

"This thesis shall study the problems of creating a non-intrusive, privacy-preserving life logging system capturing, storing, and partially analysing performance indicators in the sports domain. The concrete prototypes developed will be in cooperation with our partner Tromsø IL and their soccer A-team. Main focus will be on building and evaluating an end-to-end system that captures the digital footprints of such athletes."

1.2 Interpretation

The challenges of building a non-intrusive, privacy-preserving life-logging system are multifaceted and complex. The properties of the data and the client applications have to be considered before deciding on an architecture and design. To reason with design choices, we need to fully understand the application domain. The implementation of storage systems can be complex and

12. Eatnu is Sámi for "stream" or "big river""

requires a deep understanding of the entire software stack to achieve optimal performance. As an example, the Apache Hadoop[13] project currently consists of over 1 million lines of Java code.

Building a lifelogging system that can scale to thousands of users relies on a storage system that can handle the amount of information generated. This thesis primarily focuses on the storage component of such a system, and investigates models that can facilitate storing large volumes of life-logging data.

1.3 Methodology

The final report [23] of the ACM task Force on the Core of Computer Science divides the discipline of computing into three major paradigms:

Theory The mathematical foundation of the computing discipline.

Abstraction The experimental foundation of the computing discipline.

Design The engineering foundation of the computing discipline.

A theory is developed by first identifying the of objects the study(definition). Next, hypotheses are built to describe the relationships among objects(theorem). Finally, proofs are constructed and the hypotheses are evaluated by interpreting the results. The theory paradigm is the foundation of computing as a discipline.

The approach investigates the viability of an hypothesis by constructing a model and making predictions. Finally, the model is evaluated and the the experimental results are interpreted to validate the predictions. As such, the abstraction paradigm is more experimental, but relies on an understanding of the underlying processes and components.

By following a set of requirements, a system is designed, implemented and tested to solve a given problem. The design paradigm focuses on building complete systems, rather than trying to understand the underlying theory.

The approach used in practice draws from all three paradigms. While not providing a new theoretical models, this this builds on a foundation of existing theory. Abstraction is used in system design, where experimental results are used to evaluate the impact of high-level design and architectural elements.

13. http://hadoop.apache.org/

This thesis is closely related to the design paradigm, but rely on the other two paradigms to complete the specifications of the design.

This thesis is written as a part of the Information Access Disruption (iAD) center. The iAD center targets core research for next generation precision, analytics and scale in the information access domain. Partially funded by the Research Council of Norway as a Centre for Research-based Innovation (SFI), iAD is directed by Microsoft Development Center (Norway) in collaboration with Accenture, Cornell University, University College Dublin, Dublin City University, BI Norwegian School of Management and the universities in Tromsø (UiT), Trondheim (NTNU) and Oslo (UiO).

1.4 Outline

The remainder of the thesis is structured as follows:

Chapter 2 presents the current trends and applications related to big data and the IoT.

Chapter 3 describes the ongoing research collaboration with Tromsø IL. We describe the applications currently in use and their properties.

Chapter 4 gives a formal description of the requirements, both functional and nonfunctional.

Chapter 5 describes the design of Eatnu. We describe the individual components and how they interact with each other.

Chapter 6 gives a brief introduction to the implementation the the client interface.

Chapter 7 evaluates the performance of the implemented design.

Chapter 8 concludes and outlines future work.

2 Background

This chapter outline some of the ongoing research,trends and applications related to the IoT and big data.

2.1 The Internet of Things

From the early days of computing and up to today, we have seen technology becoming an increasingly larger part of everyday life. The early computers were few in number and rather large, but as technology progressed, computers became household items. An estimated three billion people are connected to the internet by the end of 2014[9], and an increasing number of people are using handheld devices such as cellular phones and tablets. These devices are replacing the more traditional home computer. We primarily use our devices to connect the Internet and interact with other people through social networkings, read and send emails, play games, stream music, watch movies and messaging. It is estimated that the number of smartphone users will total 1.75 billion worldwide[13].

Advances in technology enables us to create smaller and more powerful integrated circuits, and connect all sorts of devices to the internet. The idea of an IoT was first proposed by Kevin Ashton[15] and is defined by Cisco as: "*The Internet of Things (IoT) is the network of physical objects accessed through the Internet, as defined by technology analysts and visionaries. These objects contain*

embedded technology to interact with internal states or the external environment In other words, when objects can sense and communicate, it changes how and where decisions are made, and who makes them."[5] This definition is broad in the sense that it includes any object that is connected to the Internet. But the primary focus lies on connecting everyday objects. For example, if your refrigerator could monitor that items are currently stocked, it would be able to alert you when you've run out of milk, or you could track the current position of a package you have ordered online using GPS. Atzori et. al. [17] grouped the potential IoT applications into four different domains:

1. Transportation and logistics domain.
2. Healthcare domain.
3. Smart environment (home, office, plant) domain.
4. Personal and social domain.

2.1.1 Transportation and logistics

Radio-frequency identification (RFID) and Near field communication (NFC) technology can be used to monitor the individual chains of the logistics chain, providing the detailed information of an item from its conception, production, transportation and its usage once it reaches the consumer.

The modern car is equipped with technology to improve safety and enhance the driving experience. An autonomous self-driving car is being developed at Google [12]. The car uses video cameras, radar sensors and a laser range finder combined with GPS and map data to navigate the highways alongside other motorists. Human error accounts for 90 percent of all road accidents[1], and a self-driving vehicle has the potential to minimize accidents by removing the human component. Within the IoT domain we can envision traffic lights that are connected to the Internet, that in turn can alert the driver (human or computer).

2.1.2 Healthcare domain

Tracking people and objects such as medicine and equipment has the potential to improve the workflow in a hospital by eliminating the need for forms and

1. http://www.alertdriving.com/home/fleet-alert-magazine/international/human-error-accounts-90-road-accidents

by maintaining a detailed history of events. Sensors can provide real-time information on a patient's well-being both inside and outside the hospital. For example, patients suffering from Alzheimer can be equipped with GPS bracelets to track their current position.

2.1.3 Smart environment

Sensors and actuators placed in our homes or workplaces can monitor electrical systems and environment to make decisions such as changing the room lighting, heating or setting off alarms if something is wrong.

2.1.4 Personal and social domain

Social networks is a convenient way to expose everyday life to friends and family. By automatically uploading events such as visits to public places or meeting other people, people can share their daily activities without any effort. This type of tracking can also be used by the individual to build a history of activities throughout the day.

2.2 Big Data

With more devices comes more information, and we are generating more data than we are currently able to efficiently store and process. The exponential growth and heterogeneity of this type of data lead to coining *Big Data*. Big Data is a term that describes this type of data and is can be characterized by the three[42], four[2] or five V's:

1. **Volume** Scale of data. As an example, Youtube users upload 100 hours of video every minute[10]

2. **Velocity** Speed of data. When the amount of information produces exceeds the storage capacity, we need to be able to analyze the data as it is being generated.

3. **Variety** Different types and sources of data. The structure of the data can be complex and unstructured, ranging from multimedia content (images and video) to sensor data.

2. http://www.ibmbigdatahub.com/infographic/four-vs-big-data

4. **Veracity** Trustworthiness of data.

5. **Value** The usefulness of data.

2.2.1 Machine Learning

The set of algorithms needed to process these types of datasets typically fall outside the application domain of traditional Database systems. Machine learning is the theory of making programs that automatically improves with experience. Machine learning algorithms can be particularly useful when the structure of the data is unknown. Data is typically represented as objects. These objects are represented by feature vectors, that is an n-dimensional vector of values derived from the object.

Algorithms for classifying objects into different classes can mainly be split into two classes: Supervised learning algorithms and unsupervised learning algorithm. The distinction between the two is made by the use of training data to build a classifier. A supervised learning algorithm uses a set of labelled objects, or training data, that is fed into the algorithm that in turn will be able to classify new objects. Unsupervised learning operate without labeled training data, and tries to discover patterns in the data. Clustering algorithms can group together similar objects into one or more clusters.

Applications in the machine learning domain include computer vision, language processing, search engines, stock market analysis and sentiment analysis.

2.3 E-health

With the amount of information currently available and it's rapid growth, we cannot expect that our physicians to have complete knowledge of every journal, tomography, lab tests and input from other sources such as sensors and explicit annotations. From a physicians perspective the amount of information can exceed what is feasible to interpret and might lie beyond their knowledge domain.

At the intersection between Computer Science and Medicine, there is push towards aiding physicians in diagnosing by analyzing the available information using machine learning models. From the description of IBM's Watson: "Physicians can use Watson to assist in diagnosing and treating patients by having it analyze large amounts of unstructured text and develop hypotheses based on

that analysis." The WebMD Symptom Checker[3] and the Mayoclinic Symptom Checker[4] offers an interface where users can input their symptoms and returns a list of possible diseases that might cause the symptoms. These rely on having a large database containing the diseases and their related symptoms, that in turn is used to decide which diseases most likely causes the symptoms.

Ginsberg et. al. [30] showed that it was possible to detect influenza epidemics by analyzing the Google search engine query data. The number of queries per minute is in the order of millions, that provides a unique insight into what is currently happening in the world. Current trending topics can give clues as to what is currently occupying the public, ranging from queries on current events to queries related to diseases. A similar method for detecting influenza epidemics by analyzing Twitter messages has also been presented[24]. The drawback of this sort of analysis is that it does not really benefit the individual as it is used to forecast and detect larger epidemics.

The increasing number of low-cost sensors available to consumers means that we can create applications that actively monitor individuals. The current trend is sensors that can monitor movement, GSR (galvanic skin response), temperature and heart rate [51]. Wearable body sensor devices, such as Fibtit Flex and the Nike+FuelBand, have become increasingly popular among the hobbyist. These are both good examples of technology that can be beneficial for the users health. The success of these can be attributed to being simple to use and being non-invasive[5].

Personal health records (PHR) contain personal information on users and is managed by the user themselves. Combining these different sources of information is a difficult task, since each device typically connects and stores the data in a service hosted by the provider. Microsoft's HealtVault[6] gatherers PHR from multiple sources, including Fitbit devices, to store and manage these at single place. Similarly, Open mHealth[26] proposes an open architecture where users can benefit from sharing the information gathered from multiple sources.

The Quantified Self is an international collaboration of users and makers of self-tracking tools[6]. The primary users are people who are interested at keeping a detailed log of their day-to-day activities, and keep these for personal use in the future.

3. http://symptoms.webmd.com/
4. http://www.mayoclinic.org/symptom-checker/select-symptom/itt-20009075
5. i.e the sensor is aesthetically pleasing and comfortable to wear.
6. http://quantifiedself.com/about/

2.4 Technology in Sport

The sports domain is another example of how this new type of wearable and analytical technology can have a positive impact. Top athletes relies on small margins to have the competitive edge over their competitors. This includes consistently performing at a high level. Preventing injuries is benificial as it can greatly increase the time that an atleet is active. In the soccer domain an injury can put players out for several matches, that in turn can be a large economical penalty for the team. Implementing new technology into the day-to-day activities of an athlete calls for solutions that is non-intrusive and provide useful insight with as little effort as possible. As an example, the Seattle Sounders are using sleep analytics to optimize player performance. The soccer Seattle based soccer team is monitoring the players sleep quality by using the Readibands[7] from Fatique Science.

Soccer has a long history of broadcasting popular matches to the public. With technologies such as cable, satellite television and lately streaming, more and more people are able to see their favourite teams playing. The use of broadcasting video is not only limited to that of entertainment, but can also be used for analysis and preparation. Companies such as Prozone [7] and ZXY[1] aims at providing solutions for soccer teams, with detailed event analysis and statistics. These systems typically relies on low-level features such as positional data and high-level features such as manual annotations. Another drawback is that they can be very expensive and requires experienced operators.

2.5 Security and Privacy

The growing market of self monitoring devices has the potential to improve our quality of life. The data is being stored at the service providers, that may own the rights to your personal data. The service may resell the information to a third party. One such example is the Strava mobile-fitness app for tracking the GPS coordinates of cyclists and runner, who are selling the data to governments who use the data in urban planning. Facebook provides data to advertising partners and customers, but states that this does not include any personal identifiable information[3].

7. http://fatiguescience.com/solutions/readiband/

2.6 Summary

This chapter has given an overview of the IoT paradigm and it's application in the medical and sports domain. Additionally, we have presented an ongoing case study with Tromsø Idrettslag (TIL).

3

Tromsø IL: A soccer case study

At their home stadium Alfheim above the arctic circle in the city of Tromsø, the local soccer team TIL are participating in a research collaboration with the iAD group located in Tromsø and Oslo. The primary goal of this collaboration is to discover new ways to incorporate technology into the everyday life of the elite athlete, in ways that enrich the training sessions of the team and the individual player. The systems range from self-tracking cellular applications to high-performance video processing engines.

3.1 Muithu

Manually browsing through a large collection of video can be a time consuming process. To illustrate this with an example, consider a surveillance scenario where one or more cameras are set up around a store. More often than not, the surveillance tapes are not viewed at all and only consulted on specific occasions. The amount of information can be reduced to the sequences where there is movement in the frame using computer vision techniques such as background subtraction. Classifying an action as “stealing” is difficult, especially with the sort of equipment a modest convenience store can afford.

We have built Muithu[36, 50], a system designed to store short sequences from a continious video stream. A video sequence is only stored if it captures an event that is considered to be important enough. The decision is made by a human operator with a cellphone application, who makes an annotation if an event meets this requirement.

When deploying Muithu in TIL, we placed the main expert in control of operating the system. The expert in our case is the head coach, as this person is in charge of the team and responsible for the tactical decisions. By capturing only a subset of the events out of a sequence, we reduce the number of sequences to those considered important enough. To capture these events, we have implemented Bagadus[32]. Bagadus is currently installed at Alfheim, and provides a high quality panoramic view of the stadium.

The external trigger that decides whether a part should be persisted or not does not necessarily need to be a human. It could be based on real-time analytics similar to the approaches used in [35, 16].

3.2 Ohmage

Ohmage [47] is a an open source participatory sensing platform for conducting surveys, where the data is collected from explicit input from the user as well as sensory data from the users mobile phone. Ohmage is open source and consists of a server application and a mobile phone application for Iphone and Android. The application gathers self-reports, accelerometer data, GPS position, WiFi and cell tower radio connections and acoustic traces.

3.2.1 RPE and Wellness reports

The self-reporting functionality provided by the Ohmage platform is currently operational with two surveys running in the TIL Cohort: A RPE[18] and a Wellness survey. The players report their perceived rating of exertion on a zero to ten scale after each practice, that in return is uploaded to the storage backend of Ohmage hosted locally by our group. Figure 3.1 shows the interface as presented to the players on their cellular phones.

The data collected can be used to track the well being of the players over time, and the medical support staff has access through the Ohmage portal. A sample application is shown in Figure 3.2, that shows the collected mean RPE over a

1. http://www.highcharts.com/

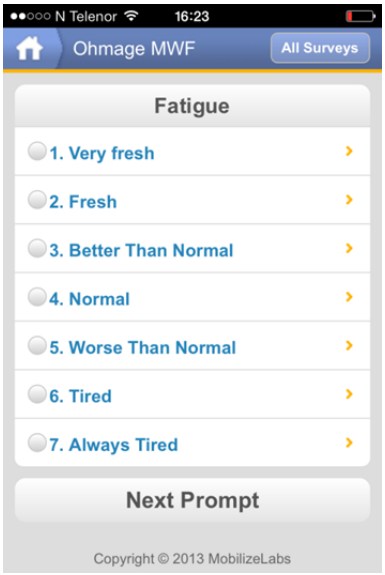


Figure 3.1: A question from the wellness survey as presented to the players

time period.

3.3 Fitbit

Wearable sensors are also being tested at TIL. Each player have received a personal Fitbit Flex bracelet, that is used to track activity and sleep [11]. The aim is to investigate whether the use of such devices raises the awareness level over the users, and in the long term be able to process and analyze this type of data in correlation with the other sources of information.

Fitbit offers a limited Application programming interface (API) to the public, with some additional features available to selected partners[2].

3.4 Application properties

The workloads presented by the three applications have different characteristics. Table 3.1 shows a comparison between the application.

2. https://wiki.fitbit.com/display/API/Fitbit+Partner+API

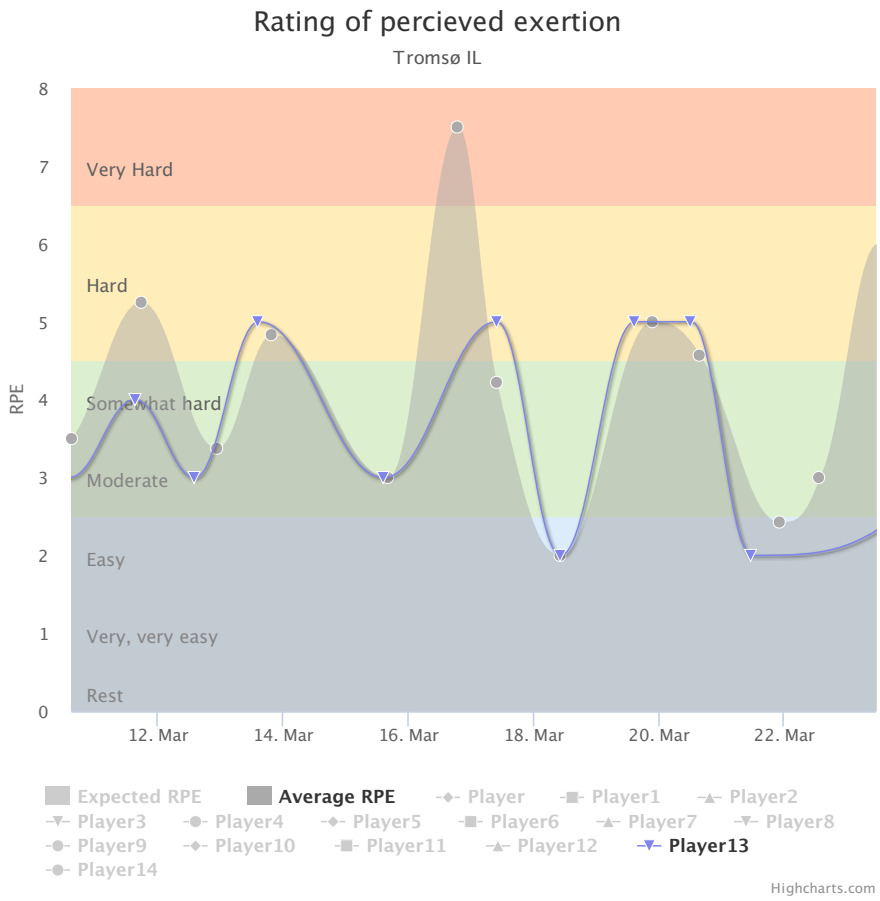


Figure 3.2: Chart showing the mean RPE for the entire team and the RPE values of single player. The chart is implemented using Highcharts[1]

Application	Type	Volume	Velocity
Muithu/Bagadus	Video	Gigabytes	High %
Ohmage	Schema	Kiloytes	Low %
Fitbit	Sensor	Megabytes	High %

Table 3.1: Application requirements

3.5 Summary

In this chapter, we have presented an ongoing case study with TIL. By presenting some of the applications and devices currently available, we can better understand the storage requirements of this type of data. Muithu, Bagadus, Ohmage and Fibit each represent different source of information. Muithu bring the notion of only keeping a small percentage of the data that is deemed relevant, and is one of the primary inspirations behind this system.

4

Requirement specification

This chapter descibes the system model and outlines a set of functional and non-functional requirements describes the needs of the systems.

4.1 System model

Before deciding on an architecture, we need to fully understand the properties of the applications in the domain. One key observation we made from working with this type of data was that the data itself was rarely changed, and new data is appended to the old data. Another observation we made was that portions of the data is often more important than others, and that we could safely discard the unnecessary data. Finally, the decision of whether or not the data is important cannot be made at the moment the data is stored, but rather, once a certain state was reached. A conventional storage system might store the incomming data on the same storage that will eventually persist the data. The applications interact with the storage system, and deletes records once they become obsolete.

Figure 4.1 shows the proposed system model for Eatnu. Eatnu acts as intermediate storage for the data, and applications evaluate the data before persisting it to stable storage.

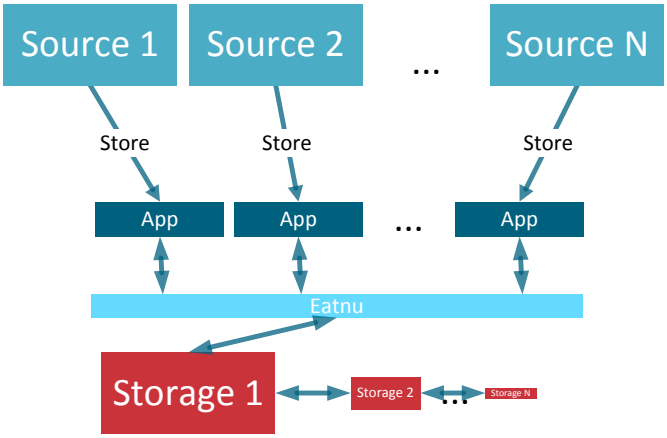


Figure 4.1: Proposed system model

4.2 Functional requirements

A functional requirement describes a functionality of the system and is specified using inputs, the behaviour and outputs. The Eatnu needs to support the following functional requirements:

Write to stream A client must be able to write data that is in turn stored by the system. If an error occurs during a write the client needs to be alerted.

Read from stream Once the data has been written to a stream, a client needs to be able to read the data.

Policy creation A policy is a small piece of code that is set to run once the stream reaches a specific state. The user should be able to add policies to run on specific stream

Policy execution Once the execution requirement of a policy is met, the system needs to run the code specified by the policy.

Persist data The main task of a policy is to decide on what portions of the stream needs to be persisted (kept) to storage.

4.3 Non-functional requirements

Non-functional requirements specifies the requirements for the operation of the system, and is typically judged by a set of criterias. The Eatnu needs to support the following non-functional requirements:

Scalability The system should be able to scale beyond the capacity provided by a single machine.

Throughput The system needs to support a high write throughput to deal with data that is generated at a high velocity.

Fault tolerance If a server fails, the data should not be lost. Additionally, the system needs to detect and correctly recover from failures.

Availability The system should remain available even if servers fail or new servers are added to the system.

Extensibility Future versions of the system might need additional functionality to support integration with other system such as batch and stream processing engines.

Usability The final non-functional requirement is usability, the ease of that clients can learn and use the system. This requirement is hard to quantify, but is typically realized by hiding complexity and exposing well-defined interfaces to the user.

5 Design

This chapter describes the design of Eatnu, covering the overall architecture and design elements. We describe the individual components that interact to provide a single service. The design goal is to build a distributed storage system with fault-tolerance, high throughput, availability and scalability. The design choices are motivated by their impact on these properties.

This chapter first outlines the stream data model and stream policies. Next, each architectural element is presented and the different roles of each node. Then describe how the different operations are performed. Finally, we look at how the design incorporates fault tolerance and how the system can recover from failures.

5.1 Data model

Eatnu offers a data model similar to that of a file-system. The system supports has a similar interface to that of a file-system functions, supporting to *open*, *read*, *write*(append) and *delete* files.

Sensor data is stored in *streams*, that in turn is made up of one or more *blocks*. The stream consists of the continuous stream of data flowing from the application, and is split into blocks of data replicated across several nodes. The streams are accessed by a unique *streamname*, that is arranged in an hierarchical names-

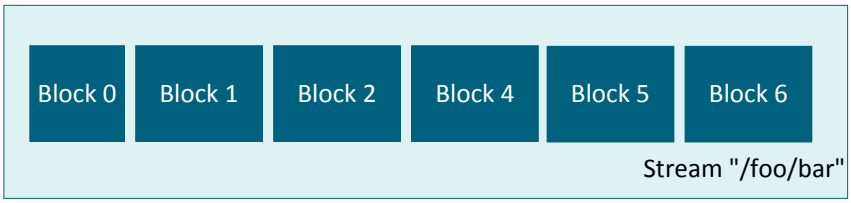


Figure 5.1: A stream with six blocks

pace. A stream can be read from any position, but data can only be appended to the end. Figure 5.1 shows an example stream with the name "/foo/bar" that consists of six blocks of data.

Each block is assigned a set of N *replicas* to tolerate the failure of $N - 1$ replicas. These replicas are assigned at random, but can be spread across multiple fault domains for fault tolerance, that in turn will impact the performance. A write is appended at the end of each block and replicated to the secondary replicas.

When a block reaches a pre-determined size, or an error occurs, the block is *closed*. A closed block is immutable and no more appends will be accepted by the block replicas. When a new block is allocated, N replicas are chosen and informed that they have been assigned a block.

5.2 Stream triggers

A *stream trigger* is an abstraction offered by Eatnu. For a stream s, the trigger $p_s()$ is a task that is executed for s when the condition $c_i(p, s, e)$ is true. The execution state e is updated on a successful execution of $p_s()$. The condition c_p and execution state e_s can differ from trigger to trigger . For Eatnu we have implemented two trigger conditions: c_1 and c_2. The first condition $c_1(p, s, e)$ is true when the current size of the stream s_{size} is p_{size} larger than the size of the stream at the previous execution $e_{\text{prev\_size}}$:

$$c_1(p, s, e) = \begin{cases} \text{true} & \text{if } s_{\text{size}} - e_{\text{prev\_size}} >= p_{\text{size}} \\ \text{false} & \text{else} \end{cases} \tag{5.1}$$

The second trigger condition checks if the time since a trigger was last executed exceeds a pre-determined interval p_{seconds}. The current time is here denoted

by t.

$$c_2(p,s,e) = \begin{cases} \text{true} & \text{if } t - e_t >= p_{\text{seconds}} \\ \text{false} & \text{else} \end{cases} \tag{5.2}$$

The stream policies are similar to that of a *database trigger*. A database trigger is a procedure that is executed on specific changes in a Database Management System (DBMS)[46], and a database with associated triggers is called an *active database*. A trigger is described using three parts:

Event: The internal change to the database that in turn causes an activation of the trigger mechanism. This can be an insert, update or delete.

Condition: The trigger test that determines if the trigger action will be activated.

Action: The procedure accosiated with the trigger. The action can be executed before, after or instead of the trigger event.

Eatnu shares some of the semantics of database triggers. An important distinction is that Eatnu does not follow a strict before, after or instead ordering of the execution of the trigger in relation with the event. The trigger invokes an asynchronous *task*, that is scheduled for execution. The task is stored as a small BASH[1] script, that in turn is may schedule other programs.

Coupling data with code with code has been done in other systems. The term *meta-code* is used [34] and shows an abstraction where code is coupled with data.

5.3 Architecture

Before going into the specifics on the architecture, we distinguish between three different types of processes:

Stream master A single process that is responsible for maintaining the stream namespace, allocating nodes, closing blocks, executing policies and orchestrating the error recovery if a node fails.

1. https://www.gnu.org/software/bash/bash.html

Stream servers Responsible for storing the blocks and serving client requests to read and write data.

Stream client The the client side API that is responsible for communicating with the stream nodes and master node to read and write the stream data.

Trigger monitor The process responsible for monitoring streams and executing the associated procedure once a the trigger condition is met.

Stream servers store the block data in main memory, and is able to respond to request without scheduling disk access. They listen to incoming read, write and close request. The block data is stored in a key-value store, with the key being the unique block name and the value pointing to a memory buffer.

5.4 Stream namespace

Each stream is identified by a unique path, e.g. "foo/bar". A list of *pointers* to block servers is kept for each unique path. A client needs to read this list of pointers before accessing the stream. The stream *namespace* is the collection of these pathnames and their associated block replicas. Strong consistency of the stream data can be provided by having a namespace with strong consistency. When a stream is updated, the stream namespace atomically writes the new state such that any subsequent read is the same.

A consistent view of the namespace can be maintained by only allowing a single server to update the namespace. This approach was adopted by Google file system (GFS)[28, 43]. GFS was designed and implemented by Google to meet the demands of their applications. By accepting component failure as the norm, they built a distributed file system that could store and serve files in the terabyte and petabyte scale even when components are failing. A GFS master is responsible for allocating new *chunkservers*, and client only communicate with the master to discover the location of these servers. The clients communicate directly with chunk servers to do read an write operations. The master also updates a set of replicas to recover from errors without having to rebuild the entire namespace. When a master fails, a new master replaces the faulty node.

State machine replication was first suggested by Leslie Lamport[39], and later described by Fred Schneider[48]. Paxos[40] is a state machine replication algorithm for reaching consensus between multiple replicas even in the event of failures. The core algorithm has been generalized to reduce en-to-end message

delays[41] and to eliminate the need for a single distinguished leader[44]. Viewstamped replication[45] is another replication protocol based on *primary-backup*[14, 20]. A single primary copies each action to a set of replicas. If the primary fails, one of the replicas becomes the new primary.

ZooKeeper[33] is a highly reliable centralized service for maintaining system configuration, naming and can be used to implement synchronization primitives. ZooKeeper provides FIFO execution of client requests and linearizability of all client requests. This is combined with a highly reliable service spread across several machines to tolerate failures. ZooKeeper is similar to Chubby [21], that provides a locking service for coarse-grained locking and a reliable low-volume storage. Chubby maintains a set of replicas by using the Paxos algorithm to reach consensus across multiple machines. ZooKeeper servers are replicated using ZooKeeper Atomic Broadcast (ZAB)[37]. ZAB is a primary-backup protocol where the primary executes client operations and then propagates the incremental updates to the backup processes. ZAB is optimized to handle multiple outstanding operation without violating FIFO ordering.

ZooKeeper stores stores data as *node* with a similar interface as a filesystem. A node is has a unique path, may have one or more children nodes. Each node may hold data, and the all data is read or written as a single operation. Two additional options can be specified when creating a node: *empheral* and *sequence*. An ephemeral node only exists as long as the creating process maintains a session with ZooKeeper. When the sequence option is specified, ZooKeeper will append a monotonically increasing counter at the end of the path of the new node. Sequence nodes can be used to implement locking functionality by creating an sequence node with the sequence flag set. A process that wishes to acquire the lock create a new child of the lock node ands sets sequence and ephemeral flag. The owner of the child with the smallest sequence number holds the lock. A lock is released by deleting the locknode. ZooKeeper will delete the node if the current owner lock fails, thus the lock is released when processes fails. A client process can set a one-time *watch* to keep track of changing nodes, and ZooKeeper will notify the client when the watch condition is met.

5.4.1 The Eatnu namespace

The Eatnu stream namespace is kept in a single ZooKeeper instance. The blocks that make up a stream is stored in a node corresponding to the stream name. ZooKeeper nodes containing the stream metadata are stored with a "_stream_" prefix. The metadata is a list of pointers to blocks and their replicas. Figure 5.2 shows an example of this hierarchical namespace. Ephemeral nodes are used to register available servers. ZooKeeper will delete the node, and notify any

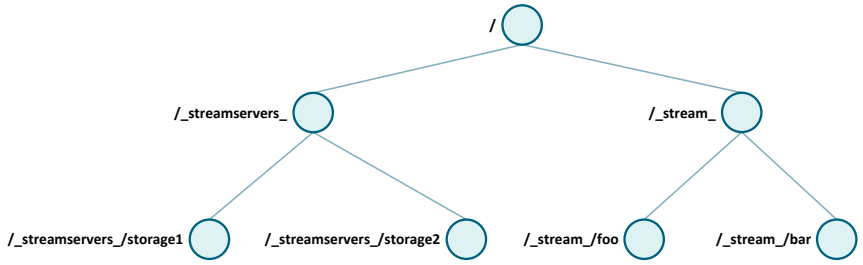


Figure 5.2: ZooKeeper nodes with two registered servers and two streams

process that have set a watch to monitor the nodes.

The master process is the only process permitted to update the namespace stored in ZooKeeper. This ensures that the close and alloc operations are performed atomically since ZooKeeper does not execute client code.

5.5 Client operations

The client side API provides an interface for programs to access the storage system. Multiple clients can read from a single stream at the same time, but only one client may write to the stream. To facilitate multiple readers, single writer, the client API will need to grab an exclusive lock before writing data to a stream. As the system is append-only, we only need to grab a write lock.

5.5.1 Open

When a client opens a file, it reads the content of from the ZooKeeper node containing the block pointers. If the file does not exist, a new empty stream is created by requesting that the stream master allocate the first block. The master will then assign the required number of replicas to the new block. The stream servers are notified that they have been assigned a new block. Before returning that the operation succeeded, the master updates the ZooKeeper node containing the newly allocated block. Figure 5.3 shows the steps that is required to open an empty stream.

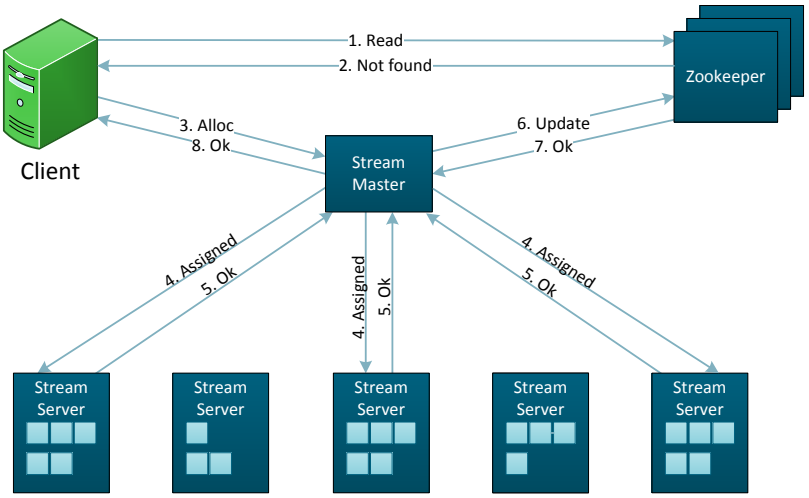


Figure 5.3: Steps of opening a new stream

5.5.2 Append

The replicas are arranged in a chain from $replica_1$ to $replica_N$ as seen in Figure 5.4. $replica_1$ acting is the primary and the next replica in the chain after $replica_i$ is $replica_{i+1}$. A write request is sent to the primary replica and forwarded along the chain, and once all replicas have successfully stored the data the primary returns success to the client.

In the event that the client is unable to append to the stream block, the client will have the stream master close the block. This operation prevents further any further appends to the same block to complete. Since one or more stream servers may have commited to storing the data locally, retrying the operation may result in duplicate records.

Arranging replicas in a chain i often used to acheive a high consistency. Chain Replication[52] is a technique used to coordinate clusters of fail-stop storage servers. Queries (write requests) are sent to the first node or emphhead of the chain, and a successful write is sent from the last node or *tail* of the chain.

The CAP theorem[29] presented by Eric Brewer stated that we can at most have two of three following properties in a network shared-data system: *Consistency*(C), *availability*(A) and tolerance to network *partitions*(P). Consistency is a guarantee that all nodes accessing the data see the same data. Availability guarantees that we are able to access the data items. Network partitioning

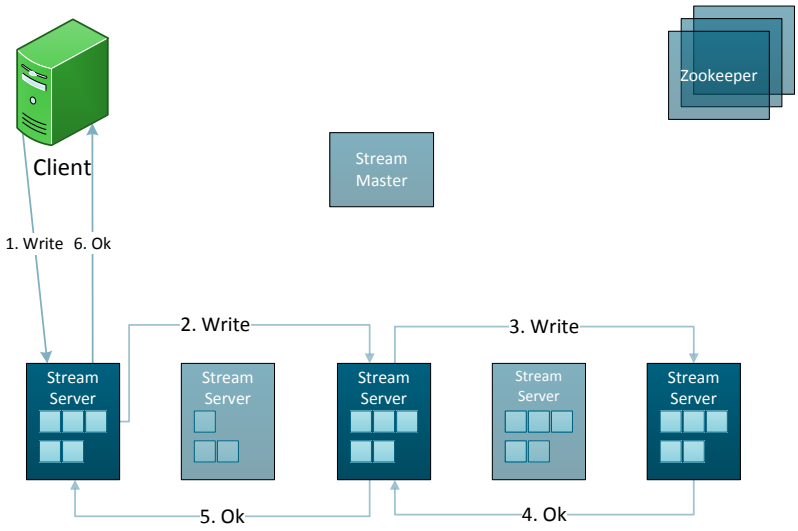


Figure 5.4: The replication chain and request steps with three replicas

occurs when nodes experience arbitrary loss of messages, or is unable to reach one or more other nodes.

The "choose two out of three" formulation of the CAP theorem is stated as misleading by Eric Brewer[19], and in reality the properties are more intertwined and their relations complex. Windows Azure Storage (WAS)[22] by Microsoft is a storage service that offers strong consistency and availability in the face of most types of network partitions. WAS is built on top of an extension of Bing's storage system Cosmos[2]. The storage system replicates the data across multiple nodes with a similar technique as chain replication but only support writes in the form of appends. WAS builds higher level abstractions such as *blobs*(files), *tables*(structured storage) and *queues*(message delivery). Some systems[38, 25] use an optimistic approach that reduces the consistency requirements, allowing clients to proceeds in case of failure, thus increasing the availability.

5.5.3 Close block

When the stream master is requested to close a stream block, the master contacts the stream servers and asks for their current length and to stop serving request for that block. If all servers return the same value, the block is closed at the current length. Otherwise, an append error has occured and the master

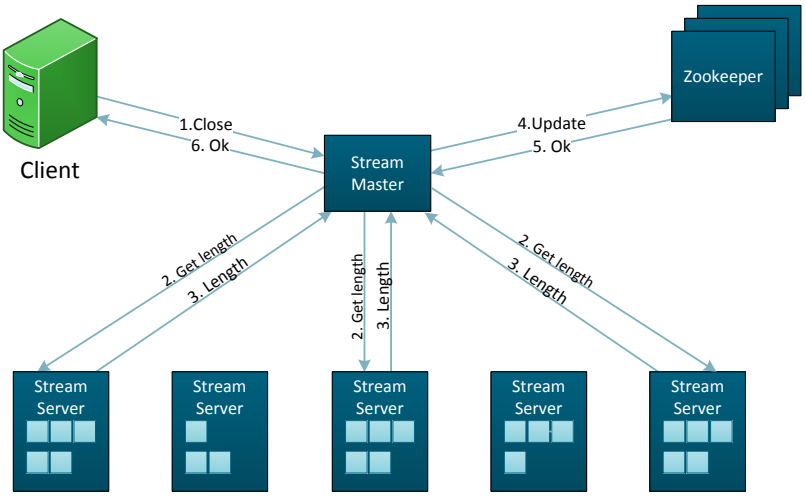


Figure 5.5: Closing a block

selects the lowest value as the block length. Since the client will close the block if an append error occurs, the latest append might be written successfully to all replicas or some of the replicas. As the client maintains the position of the last successful append, the client can check if the block was written to all replicas and retry the write operation for the new block. Figure 5.5 shows the steps of a successful close operation.

5.5.4 Read

A client read may span several blocks, and as such it is the responsibility of the client to determine that streams servers to contact. A read request is sent to the stream server with an offset and a desired length. This offset is calculated at the client and is relative to the starting offset of the block and not the stream. Stream servers does not keep a notion of streams, only blocks, and only servers requests at a block level.

The client can read data from any replica, and the read operation ensures that only data written to all replicas can be read. The writer is responsible for periodically updating the stream master of the current size of the replicas. The master will in turn update the stream metadata kept in ZooKeeper. The read operation sacrifices read freshness in favour of high throughput, consistency and availability. The tradeoff is acceptable as the processing workloads are typically batch oriented rather than stream oriented.

5.5.5 Commit

One of the key functionalities provided by Eatnu, is the ability to transparently flush the contents stored in memory to a specified storage. The storage can be a Network-attached storage (NAS), or a distributed file-system. A commit operation flushes the content of a requested range to storage. The operation invokes a storage handler responsible for communicating with the destination storage system. The client determines which stream blocks that needs to be persisted, and instructs to stream servers storing the block to write to the target destination.

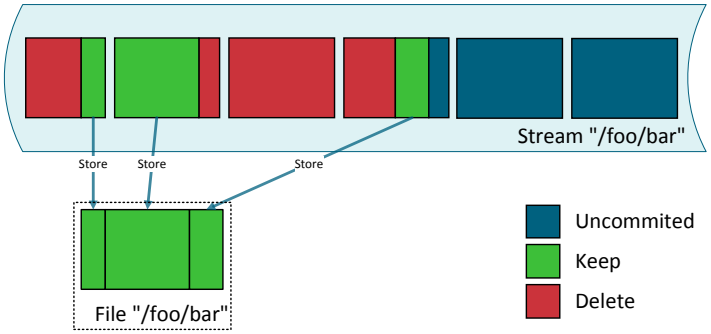


Figure 5.6: Commiting to either storing or deleting stream data

5.6 Master server

The leader election process selects a single master process that is responsible for serving the requests of the clients. Any process that wants to participate in the leader election creates a child node of the leader node in ZooKeeper. With both ephemeral and sequence options set, the process that owns the node with the lowest sequence number is considered to be the leader. If the current leader fails, ZooKeeper will delete the node and the owner of the node with the currently lowest sequence number will be the new leader. The other master processes competing to be the leader watches the node of the process next in line, as this only triggers a single watch when a node fails.

Internally, the master node maintains an event queue of tasks that have been assigned to the master. The different types of tasks have different priorities and completion time constraints. Some tasks such as the closing of a block needs to be performed as soon as possible, and other tasks such as garbage collecting deleted blocks have more relaxed time constraints. A static number of worker threads concurrently selects tasks and performs the necessary steps to complete

Task description	Task Constraints	Task Priority
Close block	Synchronous	High
Allocate block	Synchronous	High
Restore block	Asynchronous	Medium
Execute policy	Asynchronous	Medium
Check stream	Asynchronous	Low
Delete block	Asynchronous	Low

Table 5.1: The different types of tasks performed by the master.

it.

Table 5.1 shows a list of the types of tasks the master performs along the synchronization constrains and task priority. The worker threads will select tasks based on their priorities. A synchronous task will block the caller the task is completed. This is done when a remote caller blocks until a response, as is the case when a block is allocated or closed.

5.7 Summary

Eatnu implements a storage service for capturing and evaluating stream data. The data model stores each stream as a sequence of blocks and maintains a consistent namespace in ZooKeeper. Each block is replicated across multiple block server, where data is first appended to the primary and forwarded along a chain of replicas before the primary responds to the calling client. A read operation reads the stream definition from ZooKeeper, selects one of the replica block servers, and issues a read request to the server.

A key design element of Eatnu is the commit operation. When a client issues a commit for a specified range within the stream, the corresponding data is moved to stable storage. Uncommitted data can safely be discarded if the range precedes a committed range. A stream can be monitored by assigning a stream policies to streams. Each policy stores a small piece of code, a condition that triggers the execution and the target path. Once the policy condition is met, a server executes the piece of code associated with the policy.

6 Implementation

This chapter describes the API and implementation of Eatnu. The system is implemented in approximately 6000 lines of C, divided into storage components and a client-side API. The API shares some of the functionality and semantics as a traditional file-system, with the exception of the *commit* function.

6.1 API

Eatnu exposes the following interface to the programmer:

Connect() Connects to the Zookeeper instance and initializes all the local data structures.

Open() Opens a an existing stream with the given path, or creates a new empty stream if the path does not exist. The call returns a descriptor handle.

Close() Closes a file descriptor and releases all local data structures.

Append() Appends data at the end of the stream. Either all data is successfully written, or none at all.

Read() Reads data from at the current position. A successful read moves the

read pointer to the end of the last byte read.

Seek() Sets the position of the read pointer.

Commit() Commits to either storing the data at a given offset. All bytes up to the last committed range can safely be discarded.

6.2 Zookeeper

Zookeeper simplifies the implementation of a distributed system by implementing a coordination service with strict consistency. The small set of primitives can be used to build higher level constructs such as synchronization primitives, membership, naming and configuration managing.

7 Evaluation

In this chapter we evaluate the non-functional requirements of the Eatnu. We start by outlining the experimental benchmark and setup. Next we evaluate our system and compares it with a state of the art distributed file system before finally discussing the experimental results.

7.1 Experimental setup

The Hadoop Distributed File System (HDFS)[49] is a distributed file system for storing and streaming large datasets for MapReduce applications. Hadoop and HDFS is an Apache project[1], and is available with an open source licence. HDFS and GFS share many of the same design elements. Both systems use a master/slave architecture master to maintain the namespace, accept failure as the norm and replicate the data across multiple replicas for fault tolerance. HDFS was chosen to since it shares many of the design elements of Eatnu. HDFS is built from the Hadoop version 2.4.0 source code using the default configuration.

All experiments are run a cluster of HP ProLiant BL460 server blades, running Ubuntu 13.10. Each server is equipped with two Quad-Core Intel Xeon X5355 processors running at 2.66 GHz, connected to eight 2048MB DDR2 memory

1. http://hadoop.apache.org/

modules running at 667 MHz, for a total of 16GB of ram. The disk is a single 2.5 inch Fujitsu 160gb spinning at 5.4K RPM. The servers are interconnected with 1Gbit ethernet.

ZooKeeper version 3.4.6 is used for all Eatnu experiments, with master servers and Zookeeper servers co-located on the same physical machine unless otherwise stated.

7.2 Benchmarks

To evaluate the performance of our system, we have designed two simple benchmarks that evaluate the non-functional properties of the system.

7.2.1 Throughput

Throughput is the rate that a network application successfully delivers over a communication channel. To evaluate the read/write throughput, we store and read random bits to isolate the I/O bound network component at the client side.

The first benchmark stores and evaluates a continuous flow of data from numerous sensors. This simulates a workload where we are only interested in storing a portion of the data, and only when something is considered important enough to store. The evaluation uses outlier detection algorithms, and only persist the data when and outlier is detected. The motivation behind this type of benchmark is to understand how the system behaves when data is being appended at a constant rate.

For this experiment, we evaluate the performance of the commit operation on a stream. We measure the average throughput per block server. For the commit workloads, we use set that 0%, 25%, 50%, 75% and 100% of the stream is persisted. The result is shown in Figure 7.3, Figure 7.1 and Figure 7.2. The results show that Eatnu is able to achieve read and write throughputs comparable to HDFS, and in some case achieves a higher throughput.

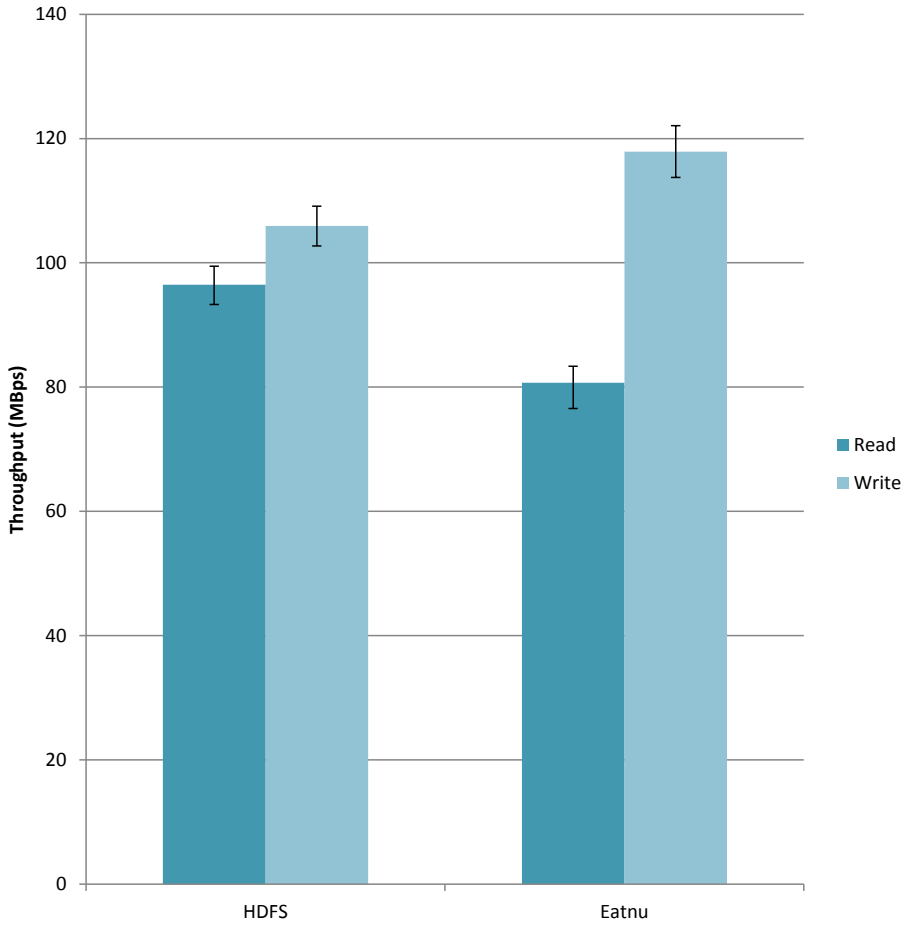


Figure 7.1: Mean throughput with one replica per block

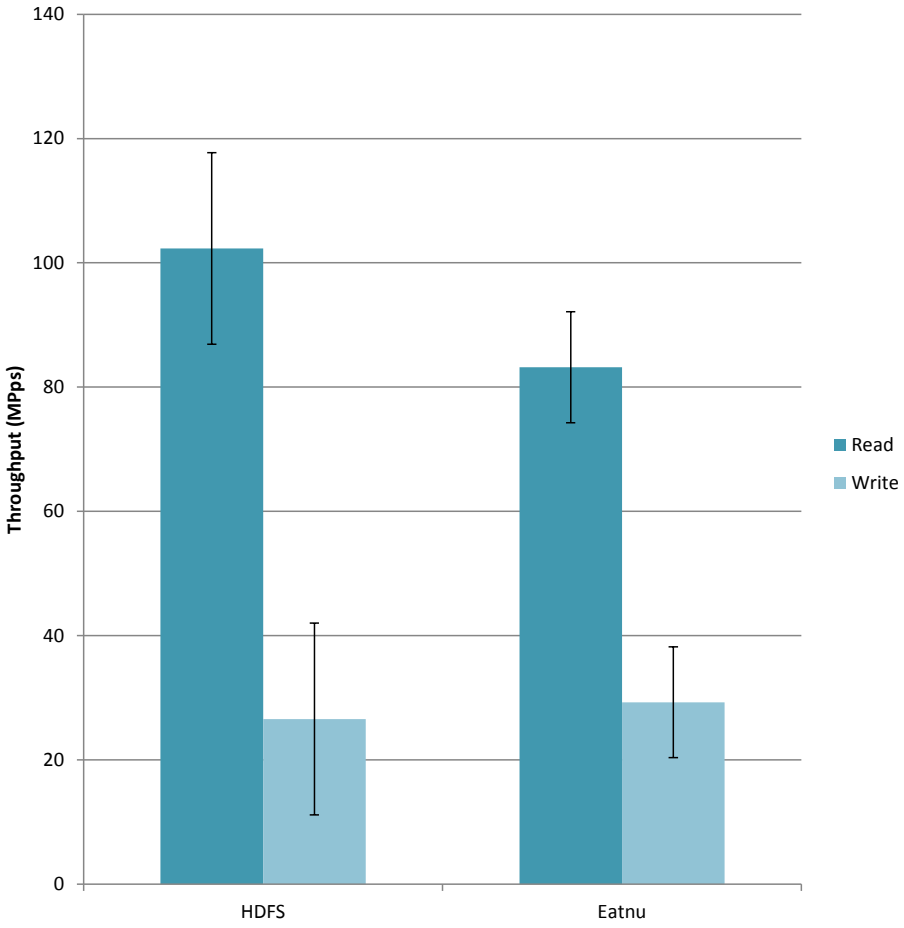


Figure 7.2: Mean throughput with three replica per block

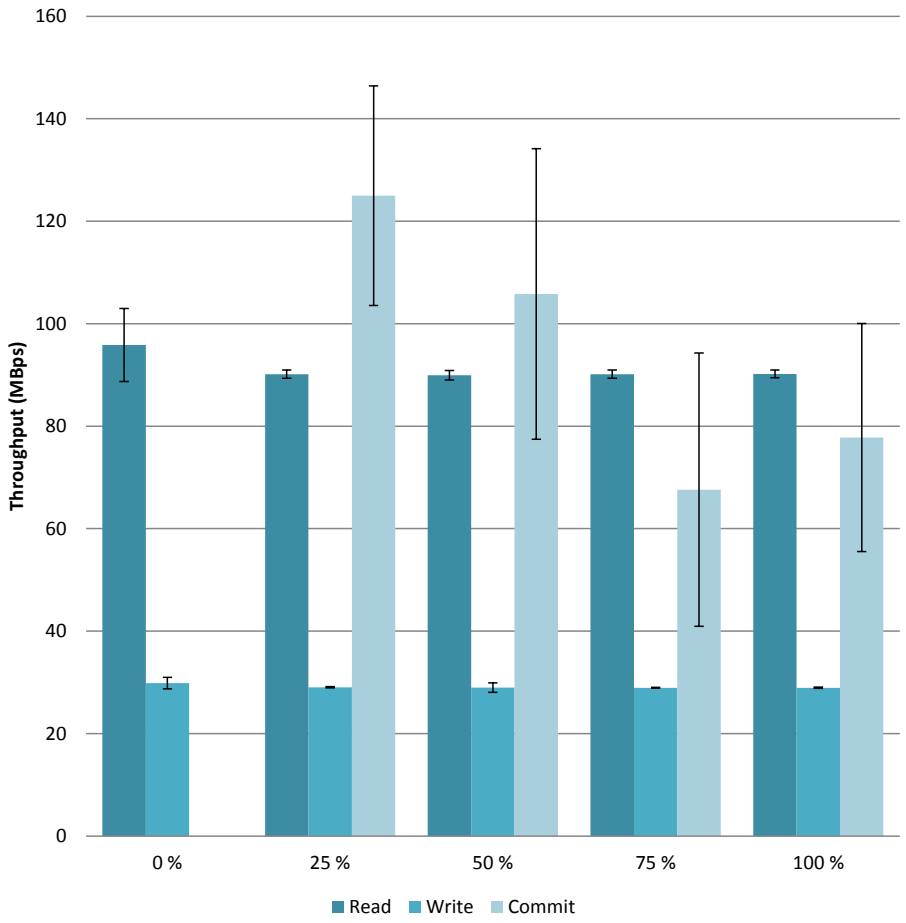


Figure 7.3: Read, write and commit throughput with three replicas per block

8

Concluding Remarks

This thesis presents Eatnu, a storage systems designed to capture, evaluate and persist data arriving at a high speed. Eatnu is designed to scale to thousands of users and distribute the data to multiple machines. The design incorporates fault tolerance, where component failure in most scenarios will not case the data to be unavailable.

A key observation that led to the creation of Eatnu, was the characteristic that sensordata often remains unchanged (immutable) and that only a portion of the data has real value. As such, we have designed and implemented a model where the developer can write to a append only storage system with similar semantics as a file-system, stored in main-memory and replicated across multiple server. The key feature of Eatnu is the addition of a *commit* operation. This operation selects a portion of the stream and persists it on stable storage. Any prior data that has not been committed can safely be discarded, and is garbage collected.

To facilitate writing applications that monitors and reads from these streams, we have also implemented a stream policy abstraction. A policy consists of a piece of code and a condition. The condition is a test that triggers the execution of code, that in turn will start an analysis on the stream that triggered the policy.

8.1 Contributions

The main contribution of this thesis can be summed up as follows:

1. We have identified the key properties of the data in the application domain related to large scale storage of personal sensor data.

2. We have outlined operations that allows programs to make decisions on which portions of data to persist.

3. We have designed and built a working prototype incorporating the these operations.

Bibliography

[1] http://www.zxy.no/zxy_about.html. Online; accessed 2014-05-21.

[2] Cosmos big data and big challenges. http://research.microsoft.com/en-us/events/fs2011/helland_cosmos_big_data_and_big_challenges.pdf. Online; accessed 27-May-2014].

[3] Data use policy. http://www.facebook.com/fulldata_use_policy. Online; accessed 2014-05-19.

[4] Gartner says the internet of things installed base will grow to 26 billion units by 2020. http://www.gartner.com/newsroom/id/2636073. Online; accessed 2014-05-30.

[5] Internet of things. http://www.cisco.com/web/solutions/trends/iot/overview.html. Online; accessed 2014-05-19.

[6] Microsoft healthvault. https://www.healthvault.com/no/en/Howitworks. Online; accessed 2014-05-19.

[7] Prozone. http://www.prozonesports.com/about/. Online; accessed 2014-05-21.

[8] Scaling the facebook data warehouse to 300 pb. https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/. Online; accessed 2014-05-20.

[9] Smartphone users worldwide will total 1.75 billion in 2014. http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536. Online; accessed 2014-05-20.

[10] Statistics. http://www.youtube.com/yt/press/statistics.html. Online; accessed 2014-05-20.

[11] Til-spillerne flges opp 24 timer i dgnet. http://www.nrk.no/nordnytt/na-

kan-gutan-kontrollere-egen-sovn-1.11505750. Online; accessed 2014-05-19.

[12] What we're driving at. http://googleblog.blogspot.no/2010/10/what-were-driving-at.html. Online; accessed 2014-05-19.

[13] The world in 2014. http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf. Online; accessed 2014-05-20.

[14] Peter A Alsberg and John D Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering*, pages 562–570. IEEE Computer Society Press, 1976.

[15] Kevin Ashton. That 'internet of things' thing. *RFiD Journal*, 22:97–114, 2009.

[16] Jürgen Assfalg, Marco Bertini, Carlo Colombo, Alberto Del Bimbo, and Walter Nunziati. Semantic annotation of soccer videos: automatic highlights identification. *Computer Vision and Image Understanding*, 92(2):285–305, 2003.

[17] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[18] Gunnar Borg. *Borg's perceived exertion and pain scales*. Human kinetics, 1998.

[19] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[20] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[21] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[22] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[23] Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[24] Aron Culotta. Towards detecting influenza epidemics by analyzing twitter messages. In *Proceedings of the first workshop on social media analytics*, pages 115–122. ACM, 2010.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[26] Deborah Estrin and Ida Sim. Open mhealth architecture: an engine for health care innovation. *Science(Washington)*, 330(6005):759–760, 2010.

[27] John F Gantz and David Reinsel. The expanding digital universe: A forecast of worldwide information growth through 2010. IDC, 2007.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[29] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[30] Jeremy Ginsberg, Matthew H Mohebbi, Rajan S Patel, Lynnette Brammer, Mark S Smolinski, and Larry Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, 457(7232):1012–1014, 2009.

[31] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM Computer Communication Review*, 39(1):68–73, 2008.

[32] Pål Halvorsen, Simen Sægrov, Asgeir Mortensen, David KC Kristensen, Alexander Eichhorn, Magnus Stenhaug, Stian Dahl, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, et al. Bagadus: an integrated system for arena sports analytics: a soccer case study. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 48–59. ACM, 2013.

[33] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed.

Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[34] Dag Johansen and Joseph Hurley. Overlay cloud networking through meta-code. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 273–278. IEEE, 2011.

[35] Dag Johansen, Håvard Johansen, Tjalve Aarflot, Joseph Hurley, Åge Kvalnes, Cathal Gurrin, Sorin Zav, Bjørn Olstad, Erik Aaberg, Tore Endestad, et al. Davvi: A prototype for the next generation multimedia entertainment platform. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 989–990. ACM, 2009.

[36] Dag Johansen, Magnus Stenhaug, Roger Bruun Asp Hansen, Agnar Christensen, and P-M Hogmo. Muithu: Smaller footprint, potentially larger imprint. In *Digital Information Management (ICDIM), 2012 Seventh International Conference on*, pages 205–214. IEEE, 2012.

[37] Flavio Paiva Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256. IEEE, 2011.

[38] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[39] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[40] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[41] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[42] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6, 2001.

[43] Marshall K McKusick and Sean Quinlan. Gfs: Evolution on fast-forward. *ACM Queue*, 7(7):10, 2009.

[44] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM,

2013.

[45] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

[46] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[47] Nithya Ramanathan, Faisal Alquaddoomi, Hossein Falaki, Dony George, C Hsieh, John Jenkins, Cameron Ketcham, Brent Longstaff, Jeroen Ooms, Joshua Selsky, et al. Ohmage: an open mobile system for activity and experience sampling. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2012 6th International Conference on*, pages 203–204. IEEE, 2012.

[48] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[49] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[50] Magnus Stenhaug, Yang Yang, Cathal Gurrin, and Dag Johansen. Muithu: A touch-based annotation interface for activity logging in the norwegian premier league. In *MultiMedia Modeling*, pages 365–368. Springer, 2014.

[51] Melanie Swan. Sensor mania! the internet of things, wearable computing, objective metrics, and the quantified self 2.0. *Journal of Sensor and Actuator Networks*, 1(3):217–253, 2012.

[52] Robbert van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.