High Availability on a Distributed Real Time Processing System

Jian Neng Li
Ashkon Soroudi
Enrico Tanuwidjaja
Michael Franklin, Ed.
John D. Kubiatowicz, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-137
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-137.html

May 15, 2015

Acknowledgement

We would like to thank Professor Michael Franklin for his advice and support throughout the year, and Dr. Zhitao Shen as well as the rest of the Cisco team for making this project possible. We would also like to thank Professor John Kubiatowicz for agreeing to be our second reader.

University of California, Berkeley College of Engineering

MASTER OF ENGINEERING - SPRING 2015

Electrical Engineering And Computer Sciences

Data Science & Systems

High Availability on a Distributed Real Time Processing System

Jian Neng Li

This **Masters Project Paper** fulfills the Master of Engineering degree requirement.

Approved by:

1. Capstone Project Advisor:

Signature: ____________________________ Date _____________

Print Name/Department: Michael Franklin/EECS

2. Faculty Committee Member #2:

Signature: ____________________________ Date _____________

Print Name/Department: John Kubiatowicz/EECS

Abstract

Our Capstone project involves working with an open source distributed real time processing system called Apache Storm, in collaboration with Cisco Systems, Inc. The term “real time processing” in this context means that the system is able to respond within seconds or sub-second to requests, while “distributed” means that it is running on multiple computers. The goal of the project is to add a feature called “k-safety” to Storm. With k-safety, Storm will be able to tolerate up to k machine failures without losing data or reducing its response time, making the system highly available. Cisco plans to integrate our modified version of Storm into their data processing pipeline and use it to support internal and customer-facing products.

Table of Contents

Problem Statement

Today, many online applications require enormous amounts of computing power to operate. Consequently, these applications are run across many machines. Companies such as Google, Microsoft, and Amazon use close to if not more than one million servers to support different online services such as YouTube, Xbox Live, and online shopping (Anthony 2015). But, as the number of machines increases, the time it takes before one of them fails also decreases statistically. This theory in fact holds in practice: according to Google, for an application running on 2,000 machines, the average number of machine failures per day is greater than 10 (Twitter University 2015). Failures can be temporary, such as when a network problem causes servers unable to communicate with each other, or permanent, such as when a hard drive fails to read or write data.

Fortunately, online applications are often run on top of systems that are fault tolerant, which is a property that provides guarantees on the correctness of results in spite of failures. However, a fault tolerant system does not make any claims on response time, which means after a failure, the system can be unresponsive for some time before being able to output results again. The lack of guarantees on latency, the time it takes for one piece of data to be fully processed, is undesirable, especially when the application is expected to deliver real-time information, or new information that has just been received. In some situations, such as the last minute of an online auction, delays can be detrimental.

Our Capstone project aims to address the problem of providing results with constant latency even under failures. We achieve this goal by processing the same request on multiple machines, and making sure that only one copy of the results is sent back to the user. This way, as long as not all machines processing a particular request fail simultaneously, at least one machine will finish processing as expected, and output with the expected latency. We implement this idea on top of an open source stream processing system called Apache Storm (“Storm” 2015), so that applications running on top of the system can stay responsive throughout its lifetime.

Strategy

1. Introduction

Our Capstone project involves working with an open source distributed real-time processing system called Apache Storm (“Storm”, 2015) in collaboration with Cisco Systems, Inc. The term “real-time processing” in this context means that the system is able to respond within seconds or sub-second to requests, while “distributed” means that it is running on multiple computers. The goal of the project is to add a feature called “k-safety” to Storm. With k-safety, Storm will be able to tolerate up to k machine failures without losing data or reducing its response time. It will make the system “highly available”, which means the system is available to serve requests most of the time. Cisco plans to integrate our modified version of Storm into their data processing system and use it to support internal and customer-facing products.

Since Apache Storm is open source, the modifications we make will be open to public, allowing other interested parties to also use it for their needs. However, in order to understand our project’s impact on the community, we need to analyze the stakeholders, industry, market, competition, as well as various trends that are taking place. We will explore these topics in this section, providing insight to the current competitive landscape, and discussing the strategies we are adopting to give us the best competitive edge.

2. Industry

Given that we are designing a stream processing system, we are in the Data Processing and Hosting Services industry. This industry encompasses a wide variety of activities such as application and web hosting, as well as data processing and management. According to the IBISWorld industry report by Diment (2015), this industry is expected to produce a total revenue of $118.1 billion in 2015. Additionally, it is expected to have an annual growth of 4.8% for the next 5 years to $149.2 billion by 2020.

Diment also mentioned that the complexity of data processing systems increases rapidly, and managing such complexity requires a high level of engineering talent. At the same time, the demand for better technologies in this industry is continuously rising. Thus, this industry rapidly attracts new firms, and many of them employ strategies such as outsourced labor to aggressively innovate while keeping the costs low. Due to such competition, it is important for us to hold to a high standard and deliver the most efficient and reliable data processing system in order to attract users.

While there are many existing data processing systems, most of them are specialized for certain tasks. We can differentiate ourselves from our competitors by having a system specialized for certain needs, such as the needs of Cisco.

3. Market

Our market consists of entities that continuously process a massive amount of data as part of their regular operations. The main market is social networking companies, which include well-known companies such as Facebook, Twitter, and LinkedIn. Social networking companies have a total annual revenue of more than $11 billion and growing (Kahn 2015). As their user base grows, their needs of data processing systems increase. For example, with 86% user penetration, Facebook needs a system capable of processing massive data traffic (Harland 2014). Currently, it uses many open source systems combined, including Hadoop, a popular processing system for bulk instead of real-time data (Borthakur 2010). As the demand for real-time information goes up, it may need to adopt our highly available real-time processing system to better tailor their real-time features. Our market also includes large firms such as Apple and Cisco since they need a system to process a massive amount of logs for analytic purposes. In general, companies processing a massive amount of data are within our target market as they need a distributed data processing system to operate efficiently.

Our stakeholders are mainly large hardware or software firms, social media providers, and end users of social media such as ourselves. Large hardware or software firms benefit from our system when processing logs, as mentioned previously. Social networking companies can use our system to process a massive amount of data in real-time, delivering news and posts to the users as soon as they arrive. At the same time, the users start to demand more ways to filter and customize the content that they see (Hulkower 2013). Social media providers and users form an ecosystem with a massive amount of data traffic, and a distributed data processing system such as ours has become an integral part of this ecosystem.

4. Marketing Strategy

To holistically address marketing, in this section we describe our strategies for the 4P's: product, price, promotion, and place (Kotler and Armstrong 2011).

4.1 Product

Our product is a highly available distributed real-time processing system, which is mainly useful to meet the demands of large firms that need to continuously process a massive amount of data. What makes our product unique is its k-safety feature, which allows the data processing to continue seamlessly without any pause in the event of machine failures. In contrast, other systems might stagger for some amount of time in the event of machine failures, ranging from milliseconds to more than 30 seconds (Hwang et al. 2003). Therefore, our product offers a distributed real-time processing system that has a consistently low latency despite machine failures.

4.2 Price

We plan to make our product open source so that the price is free. It is less probable to commercialize our system, since it is more of a platform where firms can build upon rather than a complete solution. Moreover, many of our competitors are also open source. If we were to commercialize this system, we would adopt a variable-based pricing with no upfront costs, since zero upfront cost is one of the most important attributes that makes data processing products appealing (Armbrust et al. 2010). In other words, users would pay based on the amount of time using our system or the amount of data processed. In the current situation, however, we believe that making our product free is the most appropriate pricing strategy.

Although our product will be free, we can generate revenue by selling a proprietary product that takes advantage of our system or by offering technical consulting regarding our system. For example, Cisco may use our system to run one of their proprietary products and generate revenue this way. Also, we may offer technical consulting at a premium as we are the experts of the system.

4.3 Promotion

We plan to promote our system to software architects through engineering press releases or conferences. The decision of whether or not a firm uses our system most likely lies in the hands of the firm's software architects. The best channel to reach them is presumably through engineering press releases and engineering conferences. Such promotion often requires minimal to no cost, but rather a working proof of the system's success. Therefore, we have to show a working prototype in the press in order to attract software architects to use our system.

4.4 Place

Our product will be distributed through the Internet. It will be published and can be downloaded on the Apache Storm website ("Storm" 2015). As an open source project, the source code will also be published online in a public repository such as GitHub ("GitHub" 2015). Thus, we do not need a physical medium, as everything will be distributed through the Internet.

5. Porter Five Forces Analysis

In this section, we conduct a Porter Five Forces analysis (Porter 2008) on our product's industry: Data Processing and Hosting Services.

5.1 Threat of New Entrants

The threat of new entrants for our industry is moderate. The primary reason that increases the threat is that software products are in general cheap to build. In particular, for distributed systems, there are decades of research that produced simple and proven solutions to various problems in the field. For example, Apache Storm, the streaming system that we are working with, was initially solely developed by its creator, Nathan Marz, over the course of 5 months (Marz 2014). The low starting cost means that any person or research group in the short-term future can publish a new system with better performance, and potentially take away users of our system.

However, in the open source software community, the goal generally is to collaboratively build systems or packages that can be easily used by anyone free of charge. Although there are companies that offer consulting services for different open source software, these software are not usually developed to make money. Moreover, open source software often issue “copyleft” licenses, which seek to keep intellectual property free and available rather than private, by requiring modified versions of the original work to be equally or less restrictive (Lerner and Tirole 2005). These characteristics makes open source less appealing to people hoping to gain profit from software, reducing the threat of new entrants.

5.2 Threat of Substitutes

Traditionally, users would collect data, and then run queries through them to receive answers in batch. While this method is effective, the collection and querying take time, producing results in a slow pace. Over time, stream processing systems were developed in response to the increasing demanding in receiving results in shorter windows. Today, especially in the fields of real-time analytics and monitoring, systems such as Storm are crucial to the success of various companies. For instance, one of the biggest users of Storm is Twitter, which runs it on hundreds of servers across multiple datacenters, performing a range of tasks from filtering and counting to carrying out machine learning algorithms (Toshniwal et al. 2014).

Storm is crucial, because Twitter needs to produce content in real-time for their users. Therefore, the threat of substitution in this case is low. Rather, streaming systems are substitutes of traditional data processing systems.

5.3 Bargaining Power of Suppliers

One of the main reasons that software development has such low starting cost can be attributed to the low bargaining power of suppliers. In this context, a supplier is any entity that provides raw materials for us to produce our product. However, we do not require any raw material. To make software, a programmer only needs a computer and a code editing software that is free or can be purchased at low price. There is no need for a constant supply of goods or services, which means that the development of Storm will likely never be hindered by problem with suppliers.

5.4 Bargaining Power of Buyers

While customers are very important the open source community, they do not necessarily have much bargaining power. The reason is that most open source software communities have their own philosophies in terms of design and development, which cannot be easily changed without agreement among the majority of the project’s key maintainers. As a result, a minority of the users will not be able to change a certain aspect of the software if the change violates the philosophy, because the software exists to serve the majority.

In another aspect, like with other software, software migrations are costly. In fact, there exists software for performing such tasks (Fleurey et al. 2007). As a user continues to use Storm, it may be increasingly more difficult to switch to another product. This phenomenon actually motivates the users to contribute back to Storm, improving the software. Although, while users using Storm will not switch to another product unless they are extremely discontent, it can also be difficult to make users of other software to adopt Storm.

5.5 Industry and Rivalry

As real-time analytics become more important to business needs, many companies such as Microsoft (Chandramouli et al. 2014) start to develop their own stream processing systems. However, many of these solutions are tailored to specific requirements, and are not suitable for general use. Moreover, in order to remain competitive, many of them do not open their source code to the public. Nevertheless, there are many competing data processing systems in the open source community, many of which offer streaming capabilities. Some examples are Apache Hadoop, Apache Spark, and Apache Samza. The relative strengths and weaknesses of these systems will be discussed in the next section.

To combat intense rivalry, we are focusing on high availability. While most other stream processors can tolerate machine failures, many of them have to recompute the non-persistent data stored on those machines when failures occur, which can increase the system latency. Our approach reduces this latency, by minimizing the impact of machine failures on the timeliness of results. Due to our focus on a niche market, we rate our threat of rivalry to be moderately high instead of high.

6. Competitive Analysis

In such an attractive market, we have a lot of competition. In this section, we will focus on our three main competitors: Apache Hadoop, Apache Spark, and Apache Samza.

6.1 Apache Hadoop (Hadoop)

Hadoop (“Welcome to Hadoop” 2014) is an open-source framework that includes many tools for distributed data processing and storage. The data processing component of Hadoop is called Hadoop MapReduce, an implementation of the MapReduce programming model. The MapReduce programming model was originally introduced by Google (Dean & Ghemawat 2004), and it enables users to easily write applications processing vast amounts of data in a parallel and fault-tolerant manner.

However, Hadoop’s processing model is by nature very different from Storm. As previously mentioned, Storm uses stream processing, which means input is processed one by one as soon as it arrives. In contrast, Hadoop uses batch processing, which processes a large batch of data together at the same time. While Storm processes a potentially infinite stream of data as it comes in, Hadoop processes a finite stored data usually from a database.

To illustrate the difference between batch and stream processing, we will walk through a few examples. Imagine that we need to compute the statistics (e.g. mean, median, and standard deviation) of students’ exam scores. It is easier to compute such statistics if we can access the entire data (all students’ exam scores) altogether, rather than accessing one score at a time. Therefore, batch processing is more suitable for this task, since we need to process a finite amount of data at once. In contrast, imagine that we need to count the number of visits to a website and display it in real time. Since we need to process an infinite stream of data (people can keep coming to the website forever) and update our count as soon as someone visits the page, stream processing is definitively more suitable for this task. As can be seen from these examples, both batch and stream processing have their strengths and weaknesses due to their different natures. This difference is the key differentiator between our project and Hadoop MapReduce.

Furthermore, the expected result of our project should perform much better than Hadoop in terms of fault tolerance. When a machine fails, a Hadoop job will fail and needs to be restarted. If that job has computed a lot of data before the machine crashes, the system will unfortunately have to recompute everything again. As we can imagine, this property will cost time. Also, there are overheads associated with relaunching the failed job. In contrast, with our implementation of k-safety, our system will not waste any time when some machines fail. That is, if k (or less than k) machines fail, our system will continue to process data normally as if nothing has happened. This is a major advantage that we have over Hadoop. In the world of distributed computing, machine failures are inevitable such that an efficient fault-tolerance mechanism is absolutely necessary. Our system guarantees a smooth performance when at most k machines fail, which Hadoop does not offer.

6.2 Apache Spark (Spark)

Another competitor of our project is Apache Spark (“Apache Spark” 2014). A project that came out of the UC Berkeley AMPLab, and open-sourced in 2010, Spark has gained much attention in the recent few years. The distinguishing feature of Spark is that it uses a novel distributed programming abstraction called Resilient Redundant Datasets (RDDs). With this abstraction, the application can operate on a large set of data as if they are located at the same place, while underneath the cover, the data can be split across multiple machines.

Compared to the way Apache Hadoop processes data, RDDs offer many benefits. First, while Hadoop MapReduce reads data from disk and writes data back to disk for each operation on data, RDDs allow result to be cached in the machine's random-access memory (RAM). This characteristic offers great performance boost, because accessing data in memory is many orders of magnitudes faster than accessing data on disk. As a result, Spark can be used to implement iterative (accessing the same data multiple times) or interactive (requiring response time in seconds instead of minutes or hours) applications previously unsuitable for Hadoop. Second, RDDs offer many more high-level operations on data than Hadoop MapReduce. As a result, applications written using Spark tend to be much shorter than those written using Hadoop. Finally, an RDD keeps a lineage of operations performed on it to obtain the current state, so in the event that a machine fails, the lost data can be reconstructed from the original input. In contrast, Hadoop MapReduce requires storing the intermediate results at every step if the application wishes to recover from failures.

Another interesting aspect of Spark is Spark Streaming, which is an extension of Spark that provides stream processing. The difference between Spark Streaming and traditional stream processing systems is that instead of processing data one record at a time, it groups data into small batches and process them together. Since each processing incurs some computation overhead, processing more data at a time means higher throughput. However, the benefit comes at the cost of longer latencies, usually in seconds rather than milliseconds for individual data points.

While Spark is strong in many aspects, the winning factor of our project is its consistent low latency. As mentioned above, since every RDD contains information on its lineage, it does not need to be replicated in case of failures. However, the extra latency required to recompute data after a failure is unacceptable for some applications, while we can return results at a much more consistent rate. To compete against Spark's other advantages, our system is linearly scalable, which means that in order to obtain higher throughput, we can simply add more machines. As for the programming interface, we target applications that do not contain very complex logic.

6.3 Apache Samza (Samza)

The third competition that we face is Apache Samza ("Samza" 2014). Samza is stream processing system developed by LinkedIn, and was open-sourced in 2013. Even though the system is still gaining traction on the market, it has proven use cases inside LinkedIn, powering many of its website features such as news feed. Philosophically, Samza is very similar to Storm. It processes data a record at a time, and achieves sub-second latency with reasonable throughput. On the design level, Samza offers features such as the ability to process input in the order they are received, and keep metadata or intermediate data about the processed input. However, these features are less relevant to our target applications.

Similar to other systems mentioned previously, Samza again doesn't provide k-safety in our manner. Its fault tolerance model is to redo the computations previously done by the failed machine. To not lose data, it takes a similar approach to Hadoop, and writes intermediate results to disk. This way, Samza's authors argue, less recomputation is required after a failure. However, in the context of latency, writing every input to disk takes time, so the overall response time in Samza is slower than Storm. In the context of tail tolerance, our solution of duplicating computation is also superior, since a live copy of the result can be quickly retrieved from the backup machines. Again, while it is true that running every computation multiple times in our system decreases the throughput in general, our goal is to be linearly scalable. Our stakeholders are willing to add more machines in order to have the assurance of both consistent low latency and high throughput even with failures.

7. Trends

Aside from understanding the industry, market, and competitors, it is also important to observe relevant trends that can help predict the future landscape. In our case, there are several economic and technological trends that affect our Capstone project. One major trend that affects our project is the movement of data and services to the cloud. According to a MINTEL report on consumer cloud computing, more than a third of internet users are now using some form of cloud-based service (Hulkower 2012). As more data is moved to the cloud, and more people are using these cloud-based services, there will be a greater need for ways to process this data efficiently. The popularity of Internet of Things is also growing, as more devices are connected to the internet and more information is generated. According to Gartner Inc., an information technology research firm, information is being to reinvent and digitalize business processes and products (Laney 2015). Businesses will need curate, manage, and leverage big data in order to compete in the changing digital economy. Efficient data processing systems will be even more important in the future based on these trends.

8. Conclusion

We have described the business strategy for our Capstone project, a highly available distributed real-time processing system using Apache Storm in collaboration with Cisco Systems, Inc. From the IBIS report (Diment 2015), it is clear that the Data Processing and Hosting Services industry is rapidly growing. As more components in the world are connected through the web, the demand for real-time information will also rise accordingly. Hedging on this trend, we are confident about our product, despite entering a competitive industry filled with similar data processing systems.

Another contributing factor to our positive outlook is the ability of our system to deliver results with minimal delays, despite various failures that can occur in a distributed environment. By focusing on this niche use case, our product will likely be noticed by various high-tech, social media firms that we are targeting, especially if they have previous experience working with Apache Storm.

Finally, it is important to note that while we may potentially face many uncertainties, our end goal is to make the open source project more accessible to our target customers. By this measure, the probability of failure for the project is therefore very low. If we can even satisfy a small fraction of total customers in the industry, we have served our purpose.

Intellectual Property

1. Introduction

Our Capstone project, which we develop in collaboration with Cisco, aims to implement a high availability feature on a distributed real-time processing system named Storm by using a method known as “k-safety”, which involves creating duplicate records of data and processing them simultaneously. However, this method has been public knowledge for a while such that our project is not eligible for a patent, and therefore we will make our project open source instead. This paper is divided into sections as follows. Section 2 further describes our rationale of choosing open source as a strategy. Section 3 explains our analysis regarding open source, which includes its benefits and risks. Section 4 describes a patent that is most closely related to our project and how it affects us.

2. Rationale of Choosing Open Source

Our project does not have any patentable invention. According to the United States Patent and Trademark Office, an invention has to be novel in order to obtain a patent (USPTO 2014). However, highly available algorithms for real-time processing systems have been a research topic for a while. Many researchers have already studied these algorithms and published papers describing them in great detail. Hwang et al. published a paper that describes and compares the performance of these algorithms, including the k-safety algorithm that we are implementing (Hwang et al. 2003). Furthermore, other researchers have explored the usage of k-safety in various fields other than data processing. For example, Kallman et al. published a paper that describes k-safety usage in the database context (Kallman et al. 2008). Given that k-safety for data processing has been public knowledge, our project, which mainly relies on k-safety, fails the novelty requirement to obtain a patent, and therefore it is not patentable.

Since getting a patent is not an option, we turn to open source as it seems to be the natural choice for our project. The reason is because our project relies on an open source system — Storm — as we are building k-safety on top of it. The Storm project already has a considerably large community of programmers. By keeping our project open source and committing our modifications as a contribution to Storm, we can join the existing community and gain their support. Furthermore, there are many benefits of open source, which we will describe in the next section. The opposite strategy that we can adopt is trade secret. However, there is nothing that we can hide as a secret, since the algorithm that we are implementing is already well known to the public. Therefore, we believe that it is best to choose open source as our intellectual property (IP) strategy by contributing our code to Storm.

Additionally, our project will implicitly obtain a trademark and a copyright, though these rights are not an important concern of ours. Storm is copyrighted and trademarked by the Apache Software Foundation (ASF) ("Storm" 2015). As we are contributing to Storm, our project will be a part of Storm, and it will automatically inherit the trademark of Storm. Although not necessary, we can write our copyright notice for each piece of the code that we write. Storm as a whole, however, is copyrighted by ASF. Nonetheless, our names will be recorded when we submit our code to Storm, thus people will know our contribution. In general, we permit anyone to develop, modify, and redistribute our code in the spirit of open source. As such, we do not worry much about trademark and copyright, since Storm has been set up to manage both rights appropriately.

3. Analysis of Open Source

In this section, we describe our analysis regarding the benefits and risks of implementing open source strategy for our project. We begin with mentioning all of the benefits, followed by all of the risks.

First, open source makes development faster. An article published by the Harvard Business Review (HBR) states that open source accelerates the software's rate of improvement, because it can acquire many more developers to work on the project than any payroll could afford (Wilson and Kambil 2008). By contributing to Storm's open source project, we gain the support of Storm's large community of developers without having to pay them, thus allowing development to accelerate faster with less financial burden.

The nature of open source projects usually nurtures innovation. According to the book Innovation Happens Elsewhere: Open Source As Business Strategy (IHE), the unlimited size of the open source community makes it more likely for people to come up with new innovative features that the original developers may not have considered (Goldman and Gabriel 2005). For example, we can add a new k-safety feature to Storm because it is open source. In general, open source tends to encourage new innovative features that will further improve the overall quality of the project.

Open source increases the size of the market. According to HBR, closed companies have smaller market opportunity than open companies (Wilson and Kambil 2008). According to IHE, open source increases market size by building a market for proprietary products or by creating a marketplace for add-ons, support, or other related products and services (Goldman and Gabriel 2005). Cisco owns a proprietary product that uses our project as its foundation. In this case, IHE suggests that by making our project open source, it builds awareness of Cisco’s proprietary product, or it may even persuade some customers to upgrade and try the proprietary product. Therefore, open source can help increase the market size of our product and Cisco’s proprietary product.

Additionally, HBR suggests that companies often pay a reputational price for being closed, as the market tends to advocate open source (Wilson and Kambil 2008). This is especially true in our situation. Making our project proprietary may damage Cisco’s and our reputation significantly given that Storm was originally open source. Thus, being open source avoids the risk of damaging our reputations.

From the individual perspective, contributing to an open source project offers two main benefits: building a better resume and obtaining technical support from experts. Previous studies showed that contributing to an open source project can lead to a higher future earning and helps solve individuals’ programming problems (Lerner and Tirole 2005). In our case, our names will gain more exposure, and we can acquaint ourselves with experts in the data processing industry. Thus, open source benefits not only the project, but also the individuals.

Although open source offers many benefits, there are some risks associated with it. First, according to HBR, open source projects have a greater risk of IP infringements (Wilson and Kambil 2008). The reason is because open source projects are developed by an amorphous community of unknown people. They are less controlled compared with homegrown software engineers. In order to mitigate this risk, every code submission must be reviewed seriously, which incurs additional efforts. However, this is a concern of ASF instead of ours, since ASF (the owner of Storm) controls the code repository. Our responsibility is to make sure that our code does not infringe any IP. Thus, although IP infringement is a general risk in open source projects, we are not in position to worry about it.

Open source projects tend to risk the quality. IHE states that open source projects may have issues in the design level and in the code level (Goldman and Gabriel 2005). The founders may not have the same design mindset with the open source community, and the code quality written by the community has no guarantee, since there is no way to impose a strict management to the community as in a company. IHE suggests that a careful code review is required to mitigate this risk (Goldman and Gabriel 2005). Again, since ASF controls Storm's code repository, this issue is not our concern as we do not have control over code submissions. Fortunately, the current repository technology tracks every code submission such that we can choose the version that we desire. When there is an unsatisfactory update, we can revert to a previous version while submitting a ticket requesting a fix. Therefore, we can mitigate the risk of low quality for our purposes, but ASF may be more concerned about the overall quality of Storm over time.

There are some risks of "free riders". According to the National Bureau of Economic Research, the open source system has the opposite effect from the patent system (Maurer and Scotchmer 2006). That is, while the patent system incentivizes racing to be the first one, the open source system incentivizes waiting to see if others will do the work. Thus, there will be entities who do not contribute anything while taking advantage of the open source project, and they are called "free riders". Unfortunately, this is a drawback of the open source system that currently does not have any solution.

Finally, the obvious risk of open source is losing competitive advantage. Open source allows anyone to see the source code of the project, which allows competitors to see what we are doing or even steal it. In fact, HBR suggests that the risk of losing competitive advantage must be seriously considered before committing to open source (Wilson and Kambil 2008). However, as previously mentioned, we do not have anything to hide since the algorithm that we implement has been known to the public for a while. Cisco uses our system to run their proprietary product, and they can maintain their competitive advantage by simply keeping their product proprietary. Therefore, although the risk of losing competitive advantage is a serious matter, it does not necessarily apply to us or Cisco.

We have described our analysis regarding open source in relation to our project. Open source have some significant benefits and risks, but we believe that the overall benefits overweigh the risks for our particular case.

4. Closely Related Patent

There is an existing patent for a distributed stream processing method and system. This method involves splitting data into real-time and historical data streams. Data that comes in will be split into a system that processes real-time data, a system that processes data created within 2 to 30 days, and a system that processes data exceeding thirty days based on timestamps (Zhang et al. 2013). Each system can process data differently. For example, the historical data processing may require more precision can be done offline. The real-time processing system is further split into different modules depending on the type of data, and these modules are further split into data units for parallelization (Zhang et al. 2013). This is similar to components and tasks in

Apache Storm. Finally the outputs from the historical and real-time systems are integrated for a final result.

Like our project, the system described in this patent also deals with distributed stream processing, but we are not concerned about it. Our project does not specifically deal with splitting data by time values. In some way, our system does overlap with the patent. Apache Storm can be used as the real-time processing component of the patent's system, as it is capable of processing data into modules (bolts) and splitting the data to process it in parallel (task parallelization). Also, Apache storm, and therefore our system, can somewhat imitate the full function of this patent's system by simply creating a topology involving splitting the data into different bolts based on timestamps, although it would not be capable of the more precise and time-consuming calculation that could be done on an offline system. However, this approach is simply a topology that a user would create using Storm, as Storm itself is not designed with that purpose in mind. As such, we would not have to worry about licensing fees regarding this patent, as our system does not directly infringe on the patent.

5. Conclusion

This paper describes our IP strategy. Since our project does not contain any patentable idea, we opted for open source. We described several reasons for choosing open source strategy, and we analyzed its benefits and risks. Overall, the benefits of choosing open source strategy overweigh its risks. Finally, we researched related patents, and we described one that we believe is most related to our project. While there are some similarities, we believe that our project will not infringe any of its IP. With the benefits offered by open source and the lack of prior patents protecting the ideas used in our project, we believe that our project can reach its maximum potential.

Technical Contributions

1. Overview

As discussed in previous components of this paper, our Capstone project involves implementing k-safety — the ability to sustain up to k simultaneous machine failures without outputting incorrect results — on an open source distributed stream processor called Apache Storm (“Storm” 2015). We are working in conjunction with Cisco Systems, Inc. (“Cisco” 2015), whose engineers first proposed the idea. The project is planned to span over the 2014-2015 academic school year. For the majority of the fall semester, we met with Cisco engineers as well as our academic advisor to discuss the technical requirements. Starting in the spring semester, we shifted our focus to implementation and testing.

The rest of the paper is organized as follows: Section 2 discusses the context and motivation for the project. Sections 3 and 4 describe my main contributions, using related work to justify our designs and implementations. Section 5 shows the evaluations of our work, offering comparisons and possible areas of improvements. Finally, Section 6 closes with concluding remarks as well as ideas for future work.

2. Project Context

As mentioned before, the primary inspiration for our Capstone project originated from Cisco. Cisco has a proprietary streaming analytics based on Truviso Continuous Analytics (Krishnamurthy et al. 2010). It provides order-independent processing of data by dividing them into sub-streams, and generating partial results to cope with input that arrive late. However, the system is typically deployed on a single machine, and does not have the notion of fault tolerance. Cisco wishes to extend it to run in a distributed fashion, and is experimenting with Storm to provide data sharding and distribution. By using Storm's fault tolerance guarantees, Truviso can then easily scale out.

Our implementation of k-safety is roughly divided into 4 components: replication, scheduling, de-duplication, and recovery. Replication means we send copies of the same data to multiple machines, and de-duplication means when query aggregates data, duplicate results need to be eliminated. For scheduling, because of Storm's abstractions for data processing, we needed to modify the scheduler to ensure that duplicated data are sent to different machines. Finally, in the event that failures have occurred, we need proper recovery mechanisms to make the system k-safe again.

My main contribution to the Capstone project is researching, comparing, designing, and implementing our methods for recovery. Throughout the project, I reviewed papers, discussed with other engineers and professors, as well as used my own experience to gauge the most suitable approach to our problem. My findings are summarized in the following sections.

Note that the open source Storm project does have an implementation of fault tolerance ("Storm" 2015). However, it does so in a way that introduces delays, and is thus not suitable for use cases such as Cisco's. More details will be discussed in the following sections.

3. Related Work

The notion of fault tolerance in a real-time system has been extensively studied in previous research. In this literature, two main approaches are most well-known: replication and upstream backup. This section looks into these two methods as well as some others, and analyzes the situations for which each approach is best suited.

One recurring theme in these methods is that when a machine fails, it is assumed to be fail-stop. In other words, if a machine crashes, it will simply stop responding to requests. We are not concerned in this case about scenarios where failures are considered byzantine (Castro & Liskov 2002), meaning that at any point in time, any machine can be compromised and tampered with to respond inaccurately or maliciously to requests. This assumption is reasonable, as data processing systems like Storm are normally deployed in private networks that are not directly accessible from the public.

3.1 Fault Tolerance through Replication

Replication, as the name implies, means that data is replicated to multiple places, so that in the case of failure, the backup copy can be used to produce the result. If a system needs to be k-safe, then every piece of data must reside on at least (k+1) different servers. This concept is not only used in real-time systems, but is also applicable in other areas, especially data storage (Ghemawat et al. 2003; Stoica et al. 2001).

In practice, the value of k is usually 2 or 3. For example, the Google File System, an influential distributed file system designed for storage of large volumes of data, has a replication factor of 3, making the system 2-safe (Ghemawat et al. 2003).

3.1.1 Coordination

In a system that relies on replication, it is necessary for correctness to ensure that the various replicas of a given data item are kept synchronized. Numerous techniques have been shown to effectively perform this task, producing correct outputs. The most straightforward approach is to allow the duplicated data to be processed in parallel at separate machines. When the system needs to output data, all results computed using copies of the same data are sent to the same machine, which in turn de-duplicates them and outputs the de-duplicated results. However, the location at which de-duplication occurs needs to be carefully selected, as it can be considered as a single point of failure. In practice, this location can be at an application boundary, where the output is handed off to another system for storage or to be further analyzed. In this case, one possible design is to have the insert to the consumer system be idempotent: as long as two operations have the same ID, then the system guarantees that the operation is executed at most once even with multiple invocations.

Another possible way to coordinate duplicated data is through a primary-backup scheme. One implementation of this approach is Flux (Shah et al. 2004), a fault-tolerant generalization to the Exchange operator (Graefe 1990). In Flux, one copy of the data is dedicated as the primary, and is always sent to the next machine in the workflow when available. The backup copy, when ready, remains on the backup machine until further notice. If the next machine successfully receives the data from the primary copy, then the backup machine is notified, and can subsequently discard its copy. If the primary fails, however, then the backup machine is again notified, and sends its copy before discarding.

The biggest advantage of the second approach over the first one is a reduction in network traffic. Since the backup copy is held until being told to drop or output depending on whether there is a failure, only one copy of the result is sent across network between steps. This contrasts with the first version, where since de-duplication is done in the following step, multiple copies are always sent. The advantage of the first approach is its simplicity, since the coordination logic required for the primary-backup scheme to work is non-trivial.

3.1.2 Recovery

In a k-safe system, a machine failure may cause it to lose the property. To make the system k-safe again, numerous recovery algorithms can be used. The easiest one is simply to wait it out. More specifically, after a failure, ensure that each new piece of data is replicated to (k+1) locations. Generally, queries running on stream processors are concerned about results over specific time periods. For example, if an e-commerce company wants to know how many purchases are made per minute, the time window of concern in their application is one minute. If the system does not have any other failures within the next minute, all relevant data will again reside at (k+1) places. Of course, if fewer than (k+1) machines are running after one or more failures, the system cannot regain the property until its total number of machines increases again to (k+1) or greater.

The obvious disadvantage of the above approach is that if the time period is very long, such as in minutes or days, then the system will not be k-safe again for a prolonged period of time, increasing risk of additional failures. Therefore, in many cases, recovery is done by replicating again to ensure that all data has sufficient redundancy. This replication can be done by halting the system, making it unavailable for new requests until it is back being k-safe (Shah et al. 2004). Alternatively, the system can process new requests and replicate lost data at the same time. If the replicated data is continuously updated by new requests, then the system can copy it up to some threshold, and pause for as short as tens of milliseconds to finish synchronization (Clark et al 2005). The second approach is more difficult to reason about and to implement, but can significantly reduce the system downtime.

3.2 Fault Tolerance through Upstream Backup

With replication, network traffic tends to increase many fold. The primary-backup scheme also incurs additional network traffic, because even though only one copy of each result is sent downstream within the system, the original replication process still needs to duplicate the data across multiple machines. One way to alleviate this problem is by trading output latency with network traffic, with a technique called upstream backup (Hwang 2005).

Often, applications are deployed on a real-time system as a workflow consisting of many operators connected in a direct acyclic graph. Between input and output, data can go through multiple operators, each one executing a single part of the workflow. For example, to count the number of distinct Canadian users visiting a website over one hour, the website might generate an event every time a user logs on and sends the event to the system. Inside, the event might first be cleaned by an operator, removing unneeded metadata, then sent to a second operator, which checks whether the user is a new user from Canada, and outputs to the next operator if the condition is true. These outputs are collected by a third operator, which keeps track of the hour boundary and outputs the final result as needed.

The upstream backup approach utilizes this operator pipeline to provide k-safety. More concretely, to make the system k-safe, for every piece of data passed into the *i*th operator, the data's predecessors are kept at k previous operators. This way, the data is also effectively located at (k+1) machines, and can withstand failure.

3.2.1 Recovery

To recover from a failure in upstream backup, the closest available upstream operator of the failed machine emits the same data, which is then processed again to reach the desired state (Hwang 2003). Since we are simply recomputing lost data, this approach requires less extra logic added to the system when compared to duplicating inputs and de-duplicating results.

3.3 Other Fault Tolerance Approaches

One traditional approach to fault tolerance is to store data physically on disk until it is no longer needed. For example, MapReduce stores intermediate results between jobs into a distributed file system, so that if the next job fails, the application does not have to start over (Dean & Ghemawat 2004). However, due to the high latency of disk writes, this approach is often not suitable for real-time systems. An alternative to storing data on disk is to store it in memory (Ongaro et al. 2001), although due to the relative high cost of RAM, large volumes of reads and writes can make the system more expensive to scale.

Another approach is introduced by a streaming system introduced recently called Spark Stream (Zaharia et al. 2013). Based on Apache Spark, Spark Streaming uses Resilient Distributed Datasets (RDDs) (Zaharia et al. 2010; Zaharia et al. 2012) as its underlying data abstraction. A RDD represents a collection of read-only data distributed across multiple machines, and in addition contains information on how it is built from the original input. Having information on its lineage, an RDD can be built from any machine, rather than requiring specific machines to act as certain operators. This design is also important in the context of recovery, because in order to rebuild lost data quickly, Spark can simply distribute the RDD across more machines, and have all of them work in parallel to perform the necessary computation.

4. Design and Implementation

After researching known methods of implementing recovery, we compared them to decide on the most suitable one for our use cases. This section offers insight into our process of design and implementation.

4.1 K-Safety

In Section 2, we mentioned that while Apache Storm does have an implementation of fault tolerance, it is not suitable for our purposes. The reason it is not appropriate is that Storm uses a variant of upstream backup. It assumes that the source providing information to the system is reliable, and will always be able to provide the same data until Storm allows it to be discarded ("Storm" 2015). If an input is lost in the system, Storm asks the source for the input again, and reprocesses it. This approach, while simple, leads to increased latency of results upon failure. This problem also proportionally becomes more severe as the window specified by the query grows: if a query asks for some statistics computed over a day, then every time there is a failure, Storm has to reprocess one day's worth of data. Moreover, the input source has to keep one day's worth of data, which can be undesirable.

Speaking more generally, any upstream backup-like approach is not suitable for our use case, because we want the latency of results to be as low as possible despite failures. While the parallel recovery mechanism proposed by Spark Stream (Zaharia et al. 2013) alleviates this problem, since Storm is not built on top of Spark, we do not have access to the RDD abstraction.

As a result, we collectively decided to use replication as our method of providing k-safety. In an effort to make a working prototype, we opted for the simplest approach to coordination, which is to send duplicated outputs, and to not de-duplicate them until the very last stage of processing in the system.

4.2 Recovery

With k-safety, the system can continue outputting correct results for as long as no more than k machines fail at the same time. However, to continue benefiting from k-safety, it is crucial to recover the system back to a k-safe state upon a failure. For stateless topologies, recovery is immediate, since the input contains all of the required information for correct output. For stateful topologies, the machine replacing the failed one needs to have the relevant state in order to output correctly.

For simplicity, we decided to implement recovery for stateful topologies by making the replacement machine wait until the next window before outputting results. To implement this method, we needed a way for Storm to tell, at the end of the window, whether it has received and processed all of the data belonging to the time period. For example, if the query has a window length of one hour, and one machine crashes and restarts halfway through the hour, then we do not want the machine to output its results at the end of the hour.

We allow each machine to perform the check through a special marker that denotes the start of a new window, similar to a punctuation (Tucker et al. 2003). When a machine receives a special marker, it checks to see if it has received the marker associated with the window that has just ended, and decides whether to output accordingly.

5. Evaluation

Because each of the team members contributed to each part of this project, this section will discuss all of the results that we obtained.

5.1 Terminology

Storm processes data by running graphs of computation called topologies ("Storm" 2015). A topology is comprised of two types of components: spouts and bolts. Spouts provide ingest of data called tuples into the topology by reading from sources such as sockets or distributed queues, while bolts are operators that perform operations on their inputs and emit the results. By subscribing bolts to outputs of spouts or other bolts, the topology can perform arbitrary computation. To speed up processing, each component can have one or more tasks to partition its input and perform operations in parallel.

A typical deployment of Storm involves 3 components: Zookeeper, Nimbus, and Supervisor ("Storm" 2015). Nimbus is a single daemon acting as a master, and is responsible for distributing code around the cluster, assigning tasks, and monitoring for failures. Tasks are assigned to Supervisors, which are worker daemons running on one or more machines. Each Supervisor can have one or more slots, each one capable of running one or more tasks. To synchronize Nimbus and the Supervisors, interactions between them are done through Zookeeper, a distributed coordination service (Hunt et al. 2010).

Storm offers at-most-once, at-least-once, and exactly-once semantics for processing ("Storm" 2015). For at-most-once processing, a spout emits tuples and forgets about them, and those tuples may or may not be processed by the bolts downstream. For at-least-once processing, the spouts keep track of the tuples they emit, and re-emit them if they are not fully processed by a timeout. The act of marking tuples as processed is called acking. A tuple can be potentially processed multiple times because it is possible that the final output is sent out of the system, but the message marking the tuple as fully processed never reaches the spout due to network partitioning. Exact-once processing is made possible by an abstraction built on top of Storm called Trident. Instead of processing input tuple by tuple, Trident aggregates them into a number of micro-batches per second. Each processed batch is committed to an external database, making the topology transactional.

5.2 Environment Setup

Our testing environment is a cluster of 8 virtual machines, each having 4 cores, 8GB of memory, and running on CentOS 6. For simplicity, we run a single-node deployment of Zookeeper and Nimbus on the same machine. We reasoned that it is acceptable to not set up Zookeeper and Nimbus in a more robust manner, because our focus is on recovering from failures of Supervisors, which are the daemons that are processing the data. Up to 6 of the remaining machines are used for Supervisors with 4 slots each, and the last machine serves as a backup for other testing purposes.

Our benchmarks are conducted on two types of topologies: stateless and stateful, representing whether state, such as count of some metric, is stored on bolts. Regardless of the benchmark type, we use the same technique to measure latency and throughput. Figure 1 depicts our topologies visually, and shows our naming conventions. For each benchmark, we set up a data server on the machine running Nimbus. Each spout task in the topology connects to this data server, and receives tuples. To ensure that the data server is not a bottleneck, the data server only sends timestamps. At each spout, additional payload is added to the tuples before emitting them downstream. Finally, at the end of the topology, the original timestamp of the tuple is sent back to the data server, which uses them to compute statistics on the latency and throughput. To efficiently estimate the median, we used an open source implementation of the CKMD algorithm with an error rate of 5% (Cormode 2005; Wang 2015).

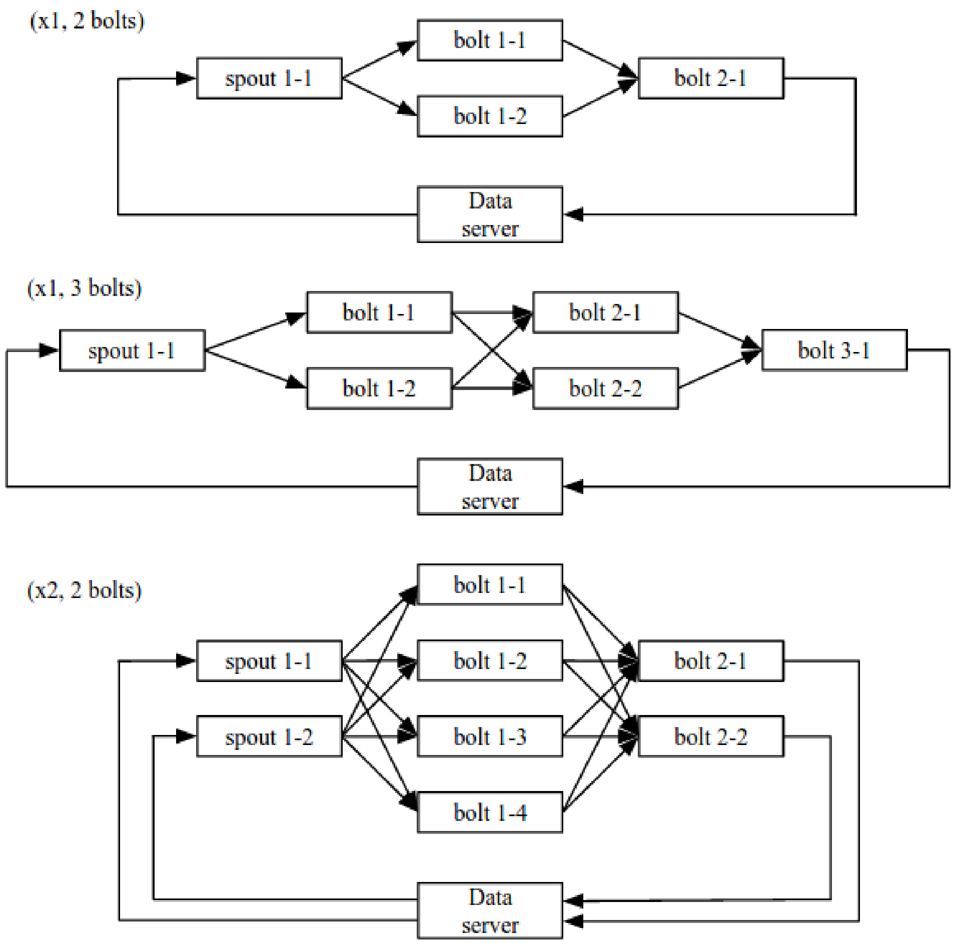
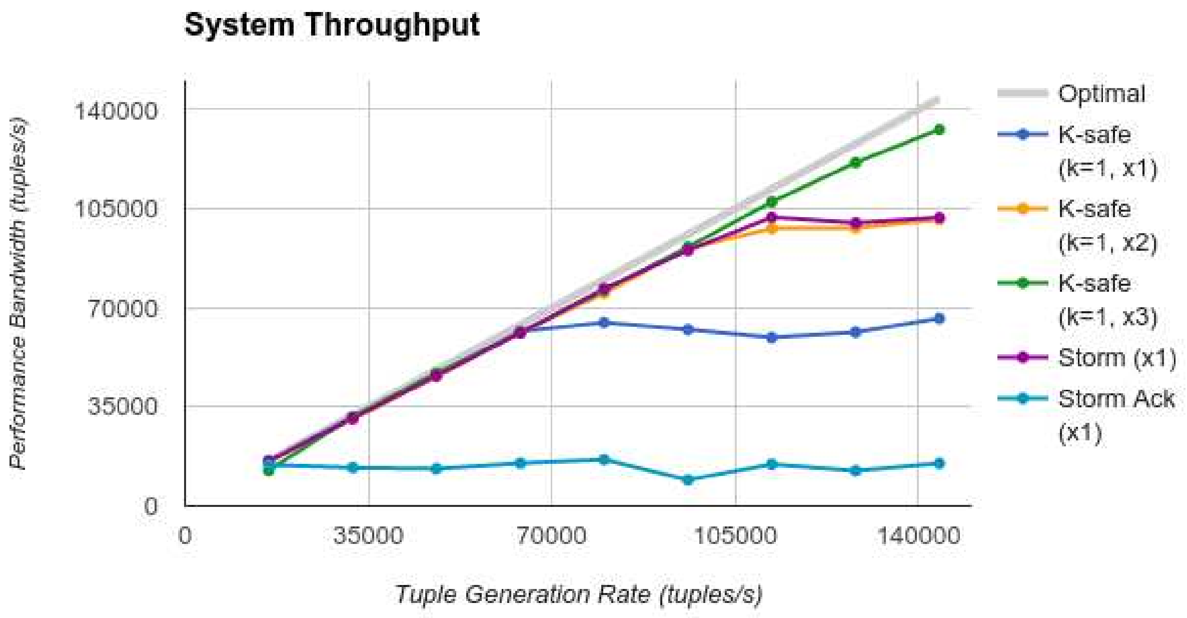


Figure 1: Example benchmark topologies. There is always 1 spout, 1 or more middle bolts that sends tuples to other bolts, and 1 final bolt that output to the data server. Here, "x1" means 1 task for the spout and final bolt, and 2 tasks for each middle bolt; "x2" means 2 tasks for the spout and the final bolt, and 4 tasks for each middle bolt, etc.

5.3 Stateless Topologies

The benchmarks for stateless topologies aim to test topologies that do not store state on any component. After the spout receives a timestamp from the data server, it attaches a 256-byte string, and emits it as a tuple. Each middle bolt forwards the tuple to the next one without performing any other computation. After the final bolt receives a tuple, it extracts the original timestamp, and sends it back to the data server.



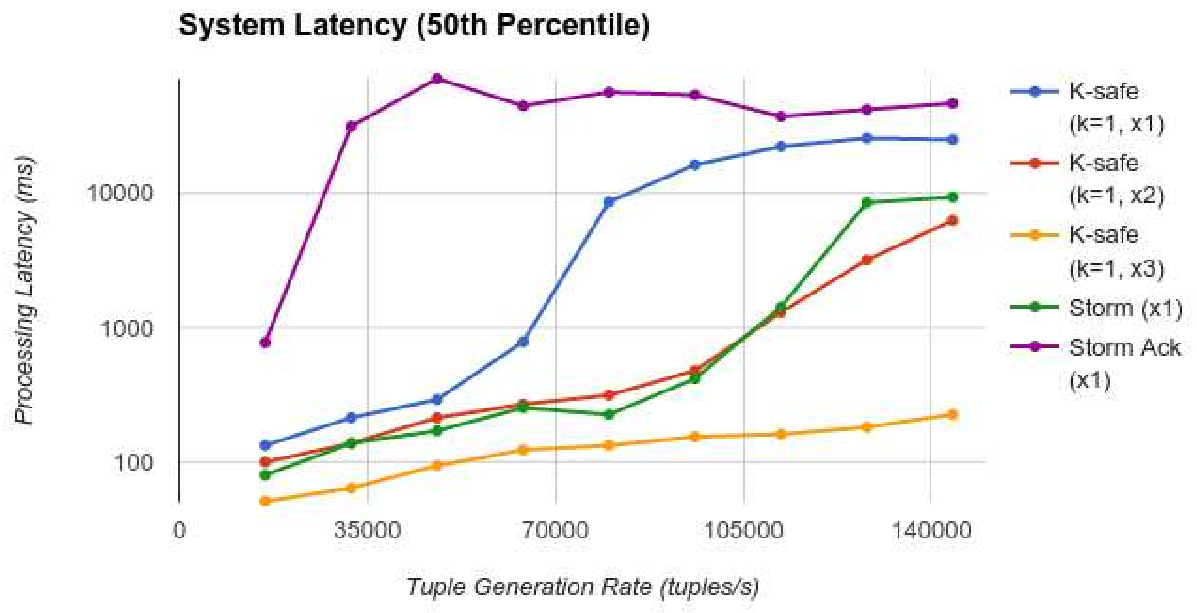
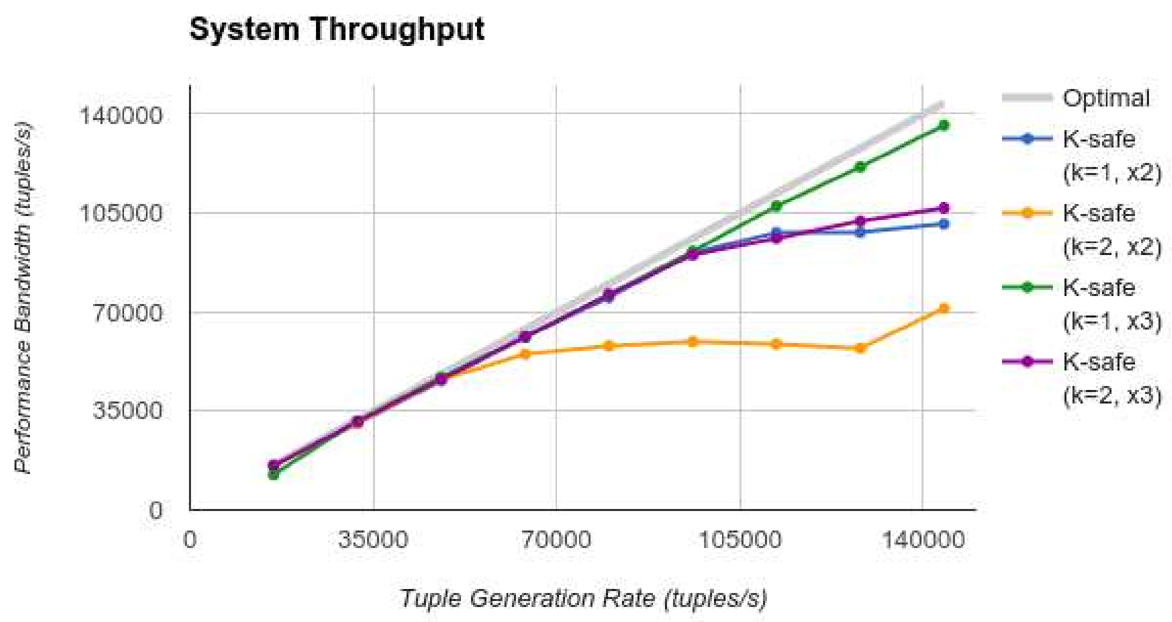


Figure 2: Benchmark results on stateless topologies with different number of spout tasks.

Figure 2 compares the performance of different stateless topologies with 1 spout and 2 bolts. The k-safe topologies benchmarked are x1, x2, and x3, each with k=1. Normal Storm (i.e. Storm without our implementation of k-safety) topologies with and without acking are also tested with x1. Note that our new scheduler schedules tasks belonging to the same spout or bolt on different machines, so the k-safe topologies use more machines accordingly.

From the throughput graph, we can see that for k-safe topologies, x1, x2, and x3 can sustain throughputs of around 65K, 100K, and 135K tuples per second, respectively. The Storm topology can handle around 100K tuples per second without acking, but only around 15K tuples per second with acking. The 50th-percentile latency graph tells a similar story, with latencies jumping over 500ms when the thresholds are reached. Above the threshold, the system becomes unstable, which makes the latency results less meaningful. Overall, our implementation of k-safety scales linearly. The 2x k-safe topology has the same maximum throughput as Storm without acking, which is reasonable, since it has twice the normal traffic due to k equaling 1.



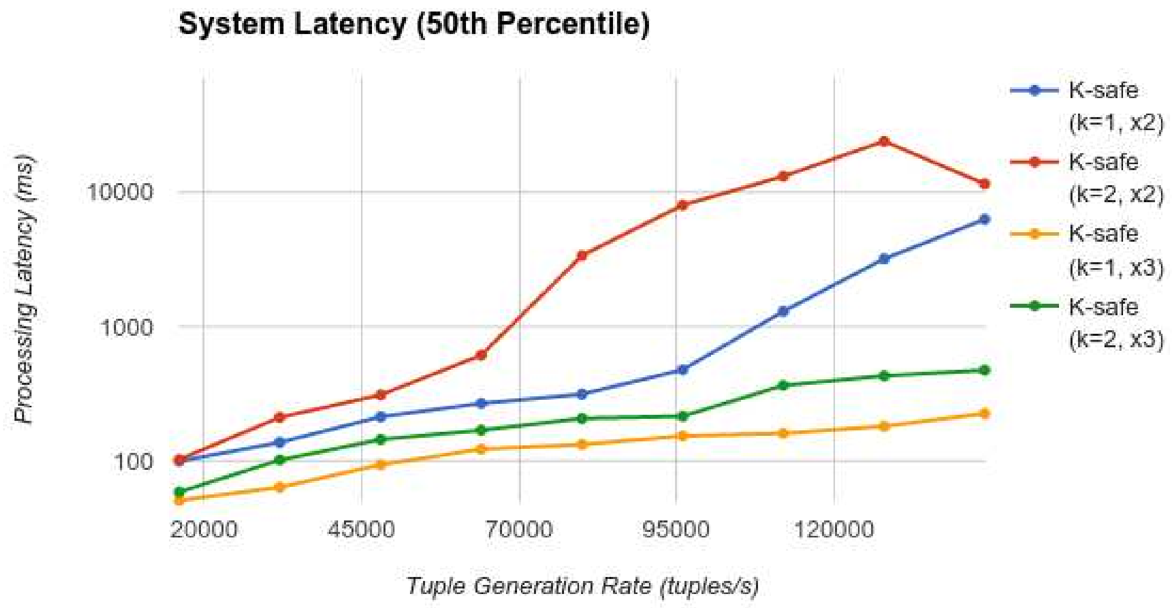
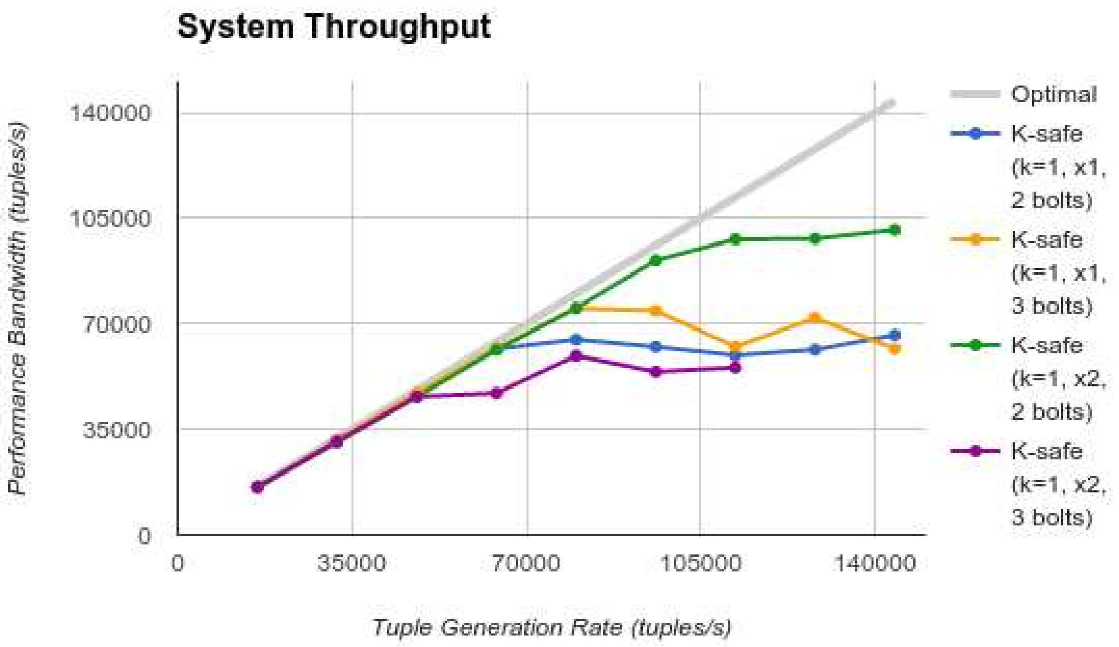


Figure 3: Benchmark results on stateless topologies with different number of spout tasks and values of k.

Figure 3 compare the effect of varying the value of k. The lines with k=1 are the same as the ones in Figure 2. Intuitively, increasing k from 1 to 2 means a 50% increase in network traffic, since each copy of the input data is sent to 3 places. For both x2 and x3, the throughput decreases by approximately 30% when k is increased. Again, this fact is reflected in the 50th-percentile latency graph, judging from the point where the value sharply increases.



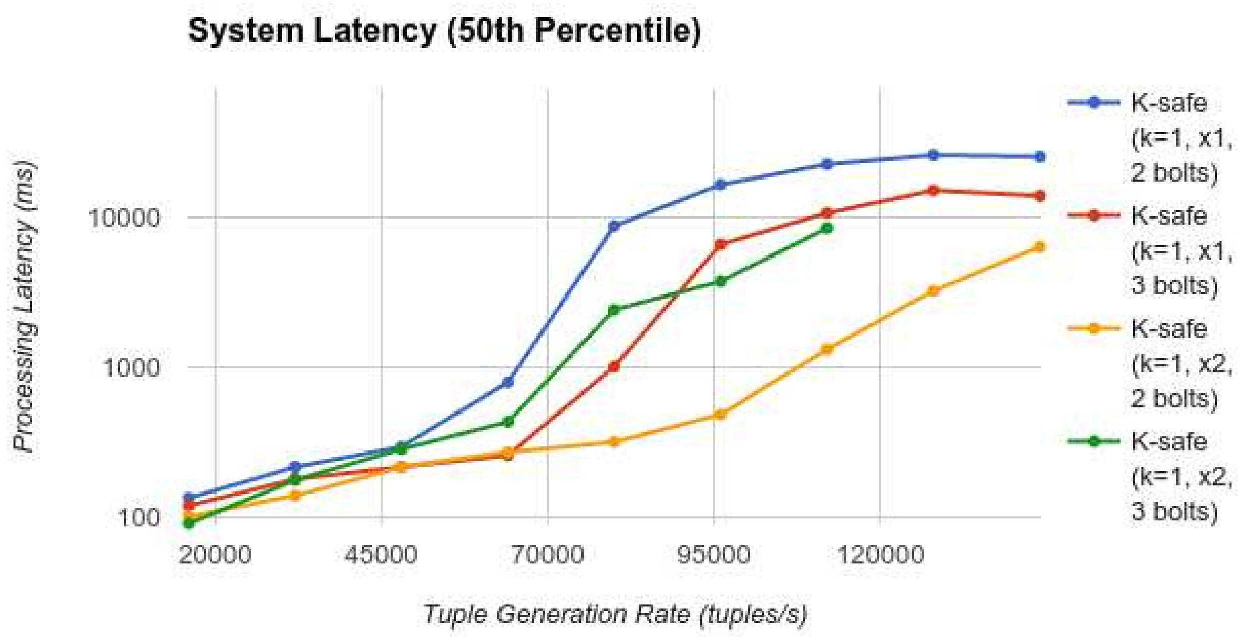


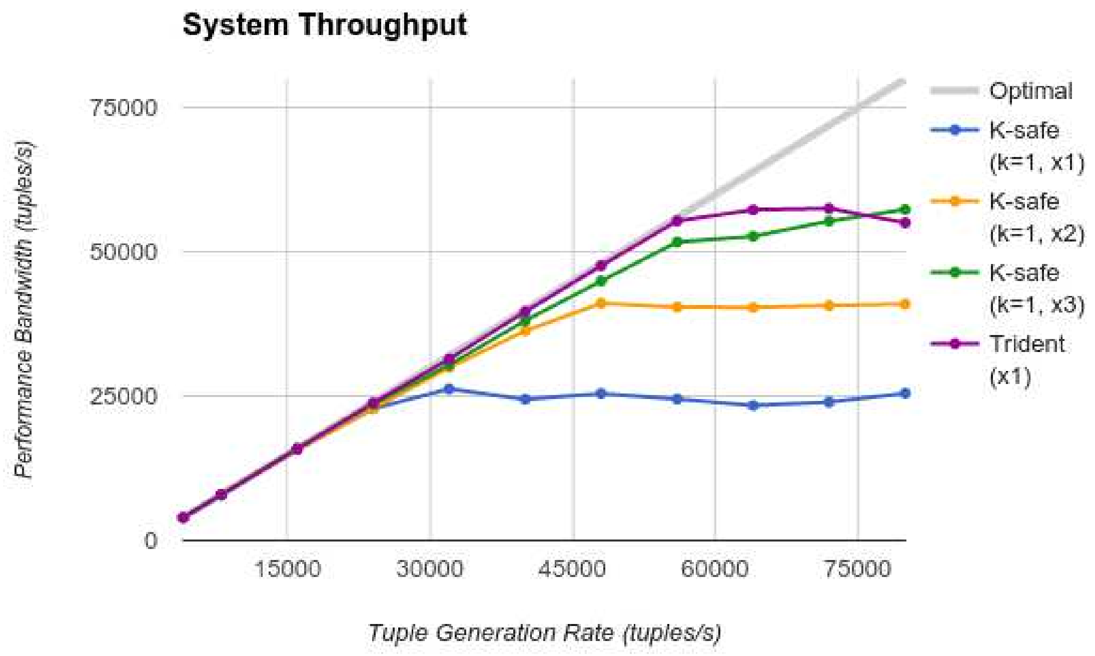
Figure 4: Benchmark results on stateless topologies with different number of bolt tasks.

Figure 4 shows the performance as we vary the number of bolts. The previous topologies used 2 bolts, where the first one passes along the tuple, and the second one extracts the timestamp from the tuple to send back to the data server. In these topologies, we increase the number of bolts to 3, where the first two pass along the tuple, and the third one sends the timestamp (see Figure 1 for reference). From the throughput graph, we can see that unlike topologies with 2 bolts, topologies with 3 bolts do not scale very well. In fact, with 3 bolts, having more spout tasks hurts the performance: the topology can sustain around 70K tuples per second with 1 task, but only around 55K tuples per second with 2 tasks. Moreover, with 3 bolts at over 100K tuples per second, some tuples fail to be processed by the topology. One potential cause for the bottleneck is network. However, the maximum network traffic in these experiments is on par with the maximum network traffic in other experiments described above. As future work, we will look more into diagnosing this issue.

5.4 Stateful Topologies

In these benchmarks, we run topologies that perform word count, with counts stored inside the middle bolts as state. The data server emits timestamps to the spout like before, but instead of attaching an arbitrary 256-byte payload, it attaches one of 26 possible 256-byte strings. The middle bolt treats the string like a word, and keeps track of the counts. The middle bolt still forwards the tuple to the final bolt, which sends the timestamp back to the data server.

To perform correct stateful computation, we need exactly-once processing. Therefore, we compare our performance with Trident, a feature built on top of Storm to provide transactional processing with micro-batches ("Storm" 2015). After a batch is processed, the result is committed into a database through a transaction that occurs atomically. In our case, the state is stored into a distributed database called Apache Cassandra ("The Apache Cassandra Project" 2015), which provides the transactional guarantees. Because Trident is a batching system, a timestamp is emitted and received per batch rather than per tuple.



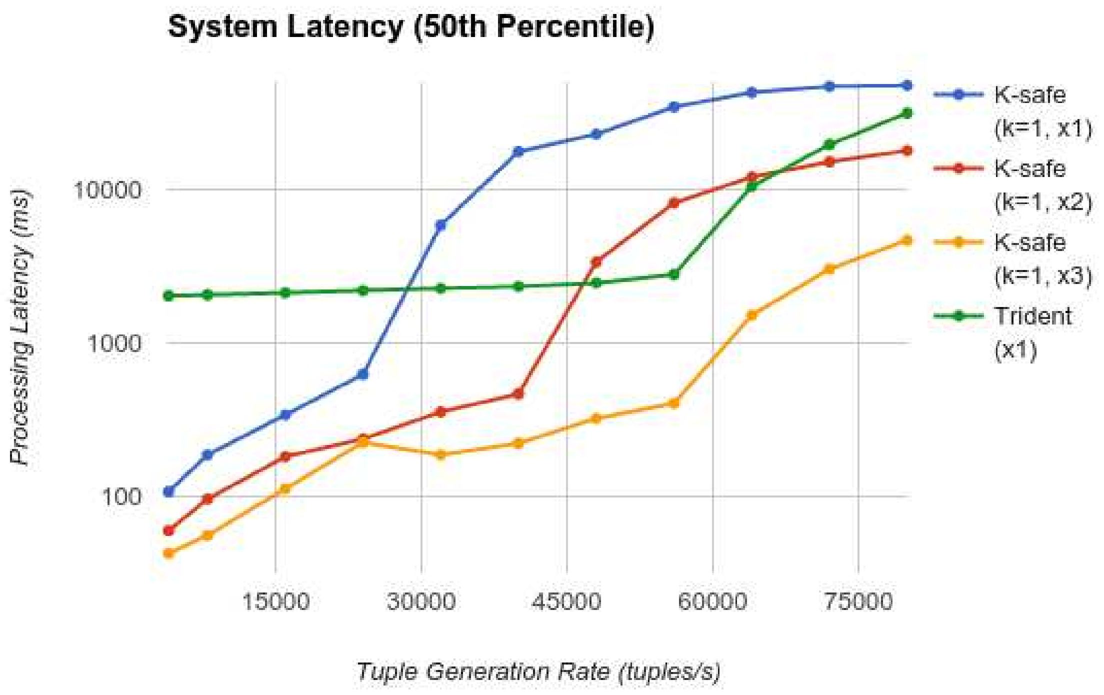


Figure 5: Benchmark results on word count topologies.

Figure 5 illustrates the performance of the various stateful implementations. From the throughput graph, the k-safe implementation with 1x, 2x, and 3x can process approximately 25K, 40K, and 55K tuples per second, respectively. Compared to our throughput of 65K with 1x for when state does not need to be stored, this is a significant reduction in performance. The slowdown is likely caused by the constant update of state, as well as the overhead imposed by our recovery strategy. On the other hand, Trident is able to handle around 55K tuples per second with only 1x. However, since Trident processes tuples in micro-batches, a tuple is not processed until the batch containing it is processed. For this reason, the latency for Trident is relatively high even before it reaches its throughput limit. Also, before the limit is reached, Trident's latency stays nearly constant, implying that for small number of tuples, the overhead of batching dominates the processing time. These benchmarks indicate that if the throughput requirement is not very demanding, and if per-tuple latency is important, then the k-safe topology is a better option. This is a desirable outcome, as our goal is to reduce delays.

5.5 Recovery

To test k-safety and recovery, we randomly kill one or more Supervisors, depending on the value of k, to test for correctness upon failures. Since input is only replicated in the middle bolts, we make sure to not kill a Supervisor that runs the spout or the final bolt. Our observations show that even with failures, our k-safe topologies would deliver results without introducing additional latency. In fact, oftentimes after a failure, the end-to-end latency was seen to decrease. A likely cause for this behavior is that because there are fewer middle bolt tasks, the final bolt receives less traffic, and is therefore able to send timestamps faster back to the data server. If a Supervisor in a stateful topology is failed, each new input originally sent to the now-failed

Supervisor is instead sent to a different one not in the set of other k Supervisors. The new Supervisor starts to output results at the next window, restoring the system to be k-safe.

For Storm, fault tolerance is provided through either acking or Trident. With acking, a tuple will be re-emitted to the topology if it is not processed within a timeout, which is by default 30 seconds (“Storm” 2015). While the timeout can be reduced, this approach will nonetheless cause delays in delivering results. With Trident and Cassandra, after we kill a Supervisor, the topology pauses before continue processing, possibly waiting for the failed Supervisor to perform some action. But unlike the case with acking, we were unable to find documentation for re-emitting a failed batch into the topology. This means that in our Trident topologies, all data associated with a failed Supervisor is lost.

Based on the results of our experiments, we believe that we have achieved our goal of being able to survive failures and restore back to being k-safe again after a set period of time (immediately for stateless topologies, and the length of the window for stateful topologies).

5.6 Discussion

We encountered numerous difficulties while implementing k-safety and experimenting with different topologies. The biggest challenge we faced is the instability of results. For example, between two runs of the same benchmark, we saw that the 50th-percentile latency can vary by a factor of 2 or more. The results we showed in this section were obtained by running each benchmark 3 times, and picking median of the overall results. For each benchmark, we also collected statistics for 90th-, 95th-, and 99th-percentile latencies, but decided to not show them here because the numbers vary significantly across runs.

Another difficulty is that despite Storm's claims on how at-least-once delivery can be provided through acking ("Storm" 2015), we encountered frequent losses of tuples at high throughputs. Out of millions of total tuples for a particular benchmark, the number of tuples sent but not received by the data server can range from hundreds to millions. The data server computes statistics by continuously accepting timestamps from the final bolt until no new packet is received in 60 seconds, so the missing tuples we encountered were either very delayed or complete lost by the system.

Missing documentation, as explained in our Trident experiments, also made our jobs more difficult. Although Storm appears to have an active open source community, we could not find anyone to answer our questions on replaying tuples in Trident. We sent our question to the mailing list as well, but never received a response.

Finally, it is worth noting that throughputs in our benchmarks are limited by the CPU rather than network capacity. To prove our hypothesis, we ran the same benchmarks on a single machine with 16 cores and 32GB of memory. The results show that the single machine performance is always worse than our results above. For example, in a k-safe topology with k=1 and 1x, 20K tuples can be processed per second. This number is lower than the 65K tuples per second throughput achieved by same topology on the cluster.

6. Conclusion

In this Capstone project, we have researched, designed, and implemented k-safety for Apache Storm. Our benchmarks have shown that the performance of our k-safety implementation scales linearly in the simple stateless case, and degrades reasonably in other cases. For future research, it is worth looking into the performance of our k-safety implementation under more complex workloads. For instance, we have shown that the performance degrades with longer topologies, but did not further investigate the scaling properties. Another possible future work is to implement other versions of recovery discussed in this paper, such as by replicating state, and measure the impact on latency as the system recovers to being k-safe.

In the end, we are confident that our results will be meaningful to Cisco as they look into building new real-time analytics products with Truviso. The entire experience has been eye-opening, as we were able to apply the concepts that we have learned in the classroom into practical use. In addition to the design processes outlined in these papers from our team, we will also make sure to write sufficient documentation about our experiments, so that the results are easily reproducible to other people who are interested in learning about our findings.

Concluding Reflections

I would like to thank my teammates, Ashkon Soroudi and Enrico Tanuwidjaja, for working on this Capstone project with me. Thanks to their accommodating personalities, we had very few problems with project management throughout the school year. We became good friends over the course of the Master of Engineering program, and had helped each other out when needed both academically and outside of school.

As for the project, we are content with our results. We had the goal of implementing a distributed, real-time processing system that does not incur delays in spite of potential machine failures, and we have accomplished that goal. And perhaps more importantly, our system can handle more input as more machines are added, an important and desirable property for any distributed system today as we collect more data than ever before.

Our Capstone project has come a long way. Over time, we constantly changed our problem statement. Some of the changes were due to miscommunication, while others were due to time constraints. In retrospect, I believe that if we had decided on our final problem statement earlier, then we could have potentially done much more. Nonetheless, many exciting new directions can be taken for this project, a few of which I had pointed out in my conclusion of the Technical Contributions section. Since Cisco will be using our results for reference as they develop their new products with Truviso (Krishnamurthy et al. 2010), they will likely explore many of those possibilities. It is reassuring to know that while we are finishing our work here, the project lives on.

Works Cited

Anthony, Sebastian. “Microsoft now has one million servers – less than Google, but more than Amazon, says Ballmer.” *ExtremeTech*. ExtremeTech, 19 Jul 2013. Web. 5 May 2015. <http://www.extremetech.com/extreme/161772-microsoft-now-has-one-million-servers-less-than-google-but-more-than-amazon-says-ballmer>.

"Apache Spark - Lightning-Fast Cluster Computing." *Apache Spark - Lightning-Fast Cluster Computing*. Apache Software Foundation, n.d. Web. 28 Nov. 2014. <https://spark.apache.org>.

Armbrust, Michael, et al. *A view of cloud computing*. Communications of the ACM 53.4 (2010): 50-58.

Balazinska, Magdalena, et al. "Fault-tolerance in the Borealis distributed stream processing system." ACM Transactions on Database Systems (TODS) 33.1 (2008): 3.

Borthakur, Dhruba. *Looking at the code behind our three uses of Apache Hadoop*. Facebook, 10 Dec 2010. Web. 16 Feb 2014. <https://www.facebook.com/notes/facebook-engineering/looking-at-the-code-behind-our-three-uses-of-apache-hadoop/468211193919>.

Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance and proactive recovery." *ACM Transactions on Computer Systems (TOCS)* 20.4 2002: 398-461.

Chandramouli, Badrish, et al. "Trill: A High-Performance Incremental Query Processor for Diverse Analytics." *Proceedings of the VLDB Endowment* 8.4 (2014): 401-412.

Cherniack, Mitch, et al. "Scalable Distributed Stream Processing." CIDR. Vol. 3. 2003: 257-268.

"Cisco Systems, Inc." Cisco. N.p., n.d. Web. 16 Mar. 2015. <http://www.cisco.com/>.

Clark, Christopher, et al. "Live migration of virtual machines." Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association, 2005: 273-286.

Cormode, Graham, et al. "Effective computation of biased quantiles over data streams." Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on. IEEE, 2005: 20-31.

Dean, Jeffrey and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6* (OSDI'04), Vol. 6. USENIX Association, Berkeley, CA, USA, 10-10.

Diment, Dmitry. *Data Processing & Hosting Services in the US.* Rep. no. 51821. IBISWorld, Jan 2015. Web. 13 Feb 2015. <http://clients1.ibisworld.com/reports/us/industry/ataglance.aspx?entid=1281>.

Fleurey, Franck, et al. "Model-driven engineering for software migration in a large industrial context." *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg (2007): 482-497.

"General Information Concerning Patents." *General Information Concerning Patents*. United States Patent and Trademark Office, Oct 2014. Web. 28 Feb 2015. <http://www.uspto.gov/patents-getting-started/general-information-concerning-patents>.

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." *ACM SIGOPS operating systems review*. Vol. 37. No. 5. ACM, 2003: 29-43.

"GitHub." *GitHub*. N.p., n.d. Web. 17 Feb. 2015. <https://github.com/>.

Goldman, Ron, and Richard P. Gabriel. *Innovation Happens Elsewhere: Open Source as Business Strategy*. Amsterdam: Morgan Kaufmann, 2005. Print.

Graefe, Goetz. *Encapsulation of parallelism in the Volcano query processing system*. Vol. 19. No. 2. ACM, 1990: 102-111.

Harland, Bryant. *Social Networking - June 2014 - US*. Mintel, Jun 2014. Web. 16 Feb 2015. <http://academic.mintel.com/ homepages/sector_overview/6/>.

Hulkower, Billy. *Consumer Cloud Computing - US - December 2012*. Mintel, Dec 2012. Web. 13 Feb 2015. <http://academic.mintel.com/display/624303/>.

Hulkower, Billy. *Social Networking - June 2013 - US*. Mintel, Jun 2013. Web. 16 Feb 2015. <http://academic.mintel.com/ insight_zones/8/>.

Hunt, Patrick, et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." *USENIX Annual Technical Conference*. Vol. 8. 2010: 9-23.

Hwang, Jeong-Hyon, et al. A comparison of stream-oriented high-availability algorithms. Technical Report TR-03-17, Computer Science Department, Brown University, 2003.

Hwang, Jeong-Hyon, et al. "High-availability algorithms for distributed stream processing." *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2005: 779-790.

Kahn, Sarah. *Social Networking Sites in the US.* Rep. no. OD4574. IBISWorld, Feb 2015. Web. 15 Feb 2015. <http://www.ibisworld.com/industry/social-networking-sites.html>.

Kallman, Robert, et al. "H-store: a high-performance, distributed main memory transaction processing system." Proceedings of the VLDB Endowment 1.2 (2008): 1496-1499.

Kotler, Philip, and Gary Armstrong. *Principles of Marketing*. Boston: Pearson Prentice Hall, 2012. Print.

Krishnamurthy, Sailesh, et al. "Continuous analytics over discontinuous streams." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 1081-1092.

Laney, Doug. *Gartner Predicts Three Big Data Trends for Business Intelligence*. Forbes, 12 Feb 2015. Web. 16 Feb 2015. <http://www.forbes.com/sites/gartnergroup/2015/02/12/gartner-predicts-three-big-data-trends-for-business-intelligence/>.

Lerner, Josh, and Jean Tirole. "The economics of technology sharing: Open source and beyond." *Journal of Economic Perspectives* 19.2 (2005): 99-120.

Maurer, S.M., Scotchmer, S. *Open Source Software: The New Intellectual Property Paradigm*. National Bureau for Economic Research Working Paper W12148 (2006).

Marz, Nathan. "History of Apache Storm and Lessons learned." Thoughts from the Red Planet. N.p., 6 Oct. 2014. Web. 17 Feb. 2015. <http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html>.

Ongaro, Diego, et al. "Fast crash recovery in RAMCloud." Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011: 29-41.

Porter, Michael E. "The Five Competitive Forces That Shape Strategy." *Harvard Business Review* 86.1 (2008): 25-40.

"Samza." *Samza*. Apache Software Foundation, n.d. Web. 28 Nov. 2014. <https://samza.incubator.apache.org>.

Shah, Mehul A., Joseph M. Hellerstein, and Eric Brewer. "Highly available, fault-tolerant, parallel dataflows." *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004: 827-838.

Slee, Mark, Aditya Agarwal, and Marc Kwiatkowski. "Thrift: Scalable cross-language services implementation." *Facebook White Paper* 5.8 (2007).

Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." *ACM SIGCOMM Computer Communication Review* 31.4 2001: 149-160.

"Storm, Distributed and Fault-tolerant Realtime Computation." *Storm, Distributed and Fault-tolerant Realtime Computation*. Apache Software Foundation, n.d. Web. 16 Mar. 2015. <https://storm.apache.org>.

“The Apache Cassandra Project.” *The Apache Cassandra Project.* Apache Software Foundation, nd. Web. 01 May 2015. <https://cassandra.apache.org>.

Toshniwal, Ankit, et al. "Storm@ twitter." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014: 147-156.

Tucker, Peter A., et al. "Exploiting punctuation semantics in continuous data streams." Knowledge and Data Engineering, IEEE Transactions on 15.3 (2003): 555-568.

Twitter University. “Cluster Management at Google.” Online video clip. *YouTube*. YouTube, 3 Sep 2014. Web. 5 May 2015. <https://www.youtube.com/watch?v=VQAAkO5B5Hg>.

Wang, Andrew. *QuantileEstimation*. GitHub, Inc., nd. 18 Apr. 2015. <https://github.com/umbrant/QuantileEstimation/commits/master>.

"Welcome to Apache Hadoop!" *Welcome to Apache Hadoop!*. Apache Software Foundation, n.d. Web. 28 Nov. 2014. <https://hadoop.apache.org>.

Wilson, Scott and Ajit Kambil. *Open Source: Salvation or Suicide?* Harvard Business Review, 2008. Web. 22 Feb 2015.

Zaharia, Matei, et al. "Discretized streams: Fault-tolerant streaming computation at scale." *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013: 423-438.

Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010.

Zhang, Xu, Zhixiong Yang, Jia Xu, and Zhonghua Deng. *Distributed Data Stream Processing Method and System*. Alibaba Group Holding Limited, assignee. Patent US 20130139166 A1. 30 May 2013.