High Availability on a Distributed Real Time Processing System

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

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-135
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-135.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.

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.

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

Ashkon Marco Soroudi

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

High Availability on a Distributed Real-Time Processing System

Ashkon Soroudi

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" 2014), 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" 2014) 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" 2014). 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" 2014). 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

This project involves implementing a version of k-safety in Apache Storm, an open-source distributed stream processing system ("Storm" 2014). We are working with Cisco Systems, who suggested this project. Distributed streaming systems are built to process large amounts of real-time data, or data that has just been created, in a distributed parallel environment. Many such systems have been developed and used recently in the past such as Aurora, Borealis, Storm, S4, and Truviso. (Abadi et al. 2003, 2005; Neumeyer et al. 2010; Kamburugamuve et al. 2013; Krishnamurthy et al. 2010). In these processing systems, data safety, availability and distribution are crucial factors in ensuring an effective system (Stonebraker et al. 2005). By duplicating data during processing, we can allow the processing to continue unhindered in the presence of failures. The idea of keeping a number of copies of data is called k-safety, in which we have k + 1 instances of the data. While duplicating data will increase network traffic, it will ensure that data does not need to be replayed if there is a copy of it being processed. By eliminating the need for data replay that Storm uses for fault tolerance, we can have lower latency during processing when a failure occurs.

In order to implement k-safety in Storm, there are a number of parts that we have to build and combine. We need a way to duplicate the data, and figure out where to duplicate it. After data is duplicated, we then need a way to ultimately remove duplicates. This is necessary in order to maintain the correct results. For example, if we are counting words, we do not want to incorrectly over-count. We need to make sure that the duplicate records actually provide us with

fault tolerance, which is the guarantee correctness in the event of failures. This means, among other things, that we need to make sure that the duplicates are created on different machines. Lastly, we need a way to handle the process of recovery after a failure has occurred.

In order to meet all of these requirements, we split the tasks up and assigned them to different people. Enrico Tanuwidjaja handled duplication and removing duplicates. I created the scheduler to ensure that duplicated data was on different machines and handled load balancing among the machines. Jianneng Li handled the implementation of recovery. The scheduler is an integral part of the system. It integrates with the duplication mechanism to ensure that duplicated data resides on different machines, and helps reassign tasks during machine recovery.

In addition to these tasks, significant time was spent on deployment and setting up Storm. All of us had to get up to speed on Storm's implementation and install and provision Storm set up on our personal machines. We could not begin brainstorming ideas until we all had a deep understanding of Storm's design. Another very time consuming task was setting up Storm on a distributed cluster. Finally, we created several topologies and a benchmarking method to test and benchmark our system.

In this paper, I discuss our implementation of the scheduler, including an overview of how Storm works, as this is crucial to understanding the reasoning behind our scheduler implementation. The primary goal of the scheduler is to ensure that duplicated data resides on different machines. I discuss related works that deal with scheduling as well as load balancing, which is an important part of any scheduler. Finally I reflect upon the results of our project as a whole.

2. Related Work

Scheduling and load balancing are topics that apply to a wide range of different systems, including operating systems, database systems, and distributed processing systems. There has been much research in these fields as well. As we are dealing with a distributed stream processing system, we are more interested in previous work in this area. Distributed processing systems will distribute data to different processes on machines to process data in parallel, and it is necessary to distribute data and tasks in a balanced and efficient manner. For example, S4 is another distributed streaming platform that uses entities called processing elements that do some processing on data and send output to other processing elements in a distributed environment (Neumeyer et al. 2010). S4 handles its load balancing and scheduling with the use of Apache Zookeeper, a service used for coordinating processes at scale in a distributed environment (Hunt et al. 2010).

There has been work by Rychly et al. on an offline scheduling approach that schedules according to resource requirements (Rychly et al. 2014). For example, there could be some resources or machines that require more memory compared to others, and some that require more CPU power compared to others (Rychly et al. 2014). In order to efficiently schedule processing for a particular application, they would first run the application on that cluster and run benchmark performance tests on each resource, or machine type. Using that information, they are able to make appropriate scheduling choices for each resource, such as storing data on memory optimized machines, or processing more data on computation optimized machines. This can be a good strategy for targeted applications, but requires tuning to work for a specific application.

Fischer et al. investigate improving the throughput of a linked data stream processing system, similar to Storm, which takes in RDF triples and outputs them into a data stream (Fischer et al. 2013). RDF triples are way of modeling entity-relationship information. The system is made out of a series of nodes in the shape of a graph, where each node takes in and outputs data. This type of system can be used to implement SPARQL algebra to perform RDF queries, where each node uses some algebra operator (Fischer et al. 2013). In their research, they propose that graph partitioning algorithms can be used to optimize traffic sent between machines, or across networks. Ghaderi et al. also explore the idea of using balanced graph partitioning to balance a scheduler for stream processing (Ghaderi et al. 2015). Each node's data processing is partitioned into several tasks and computed in parallel, similar to tasks in Storm.

There has also been research on creating an adaptive online scheduler for Storm. Aniello et al. developed two different schedulers for Storm with the goal of reducing overall latency through reduced transfer time (Aniello et al. 2013). The first scheduler is an offline scheduler that schedules executors based on the topology structures, where a topology is a set of components that send data to each other. For two components c_i and c_j, they define a $c_i < c_j$ if c_i emits a tuple that is received by c_j (Aniello et al. 2013). The dependency relationships among the components form a directed graph. The scheduler iterates through the components in order based on the topology of this graph. For any component c_j, the scheduler attempts to schedule each executor of this component on the same supervisor as executors of components c_i that this component c_j receives tuples from. When scheduled this way, tuples that are sent from one task to another will not have to travel across machines, lowering overall processing latency.

The second scheduler Aniello et al. created is an adaptive online scheduler, which schedules based on the traffic of a Storm topology at runtime. This scheduler takes into account the computational power of each node and also seeks to minimize transfer time as in the first scheduler. This is accomplished by measuring CPU utilization on each node, and also measuring the rate at which tuples are sent at from one executor to another. Their results show that the offline scheduler produces lower latency than Storm's typical scheduler, while the online scheduler produced the lowest latency (Aniello et al. 2013). In the following section, we discuss the possibility of using these ideas in our scheduler.

3. Methods and Materials

In order to describe how we implement a scheduler that guarantees that duplicated data is sent to different machines, and to discuss the reasoning behind the implementation, we first need to describe the design of Storm. A Storm topology consists of components called spouts and bolts. Data in the form of tuples originates from spouts, which gets passed to bolts, which do some form of processing on the data. After processing, the bolts potentially send the data to other bolts for more processing, and the tuples eventually arrive at an end bolt where the data is output. Internally, each spout and bolt can contain multiple tasks, which all do some processing. When data is sent from one bolt to the other, data will be sent from tasks on the sending bolt to tasks on the receiving bolt. When a component, a spout or a bolt, is sending data to another bolt, the decision of what task to send to on that bolt is decided by one of many different grouping strategies that Storm provides.

One of these grouping strategies is called FieldsGrouping, which decides the task to send to based on a defined key from the tuple being sent (Storm 2014). As an example, if bolt1 is sending words to bolt2, which counts words, we want to ensure that all instances of a given word are always sent to the same task to have one complete count for that word. Storm's fields grouping can ensure this by using the word as a key, and deciding which task of bolt2 to send this word to based on the hash value of the word. Our duplication strategy involves using a k-safety FieldsGrouping, which is similar to the normal FieldsGrouping, but sends the tuple to multiple tasks, with the number depending on the value of k. While this strategy provides a simple and easy method of duplicating the data, it is possible that the duplicated data does not provide fault tolerance depending on what machines these tasks are running on.

In a Storm cluster, each machine used will run either a Nimbus process or a Supervisor process. The goal of the Nimbus process is to coordinate the Supervisors and schedule Storm tasks. The Supervisors handle data processing and have multiple worker slots. Executors can be assigned to these slots, where each executor runs one or more tasks from Storm components. When creating a spout or bolt, the user can decide how many executors and tasks to create. Typically, the number of tasks and executors will be equal, as this is the most efficient way to process data. This is because executors run tasks serially, so having more than one task on a single executor is less efficient than running only one task per executor(Storm 2014). For our implementation, we require that the user has one task per executor, so we can treat tasks and executors as the same thing for the purposes of the scheduler. This is a fair requirement, because each executor is essentially a thread, so having more than one task scheduled to a single executor would be less efficient as those tasks would be run serially (Storm 2014).

The primary goal of the scheduler is to enable the use of k-safety. Given that data is duplicated on multiple tasks in a bolt, we need to make sure that these tasks are running on different machines to achieve k-safety. If these tasks with duplicated data were running on the same machine, then one machine failure would cause the tuples to be lost without any other copies being processed. There are several possible strategies we could take here. One possible strategy, without actually needing a scheduler, would be to decide which tasks to duplicate to based on the machines that are running these tasks. However, this approach would not work very well, because we would be no guarantee that there are enough tasks of the destination bolt that are running on different machines.

A better possible strategy involving a scheduler is to keep track of which tasks have duplicated data, and make sure that these tasks are always scheduled to run on different machines. This strategy would require constant maintenance and updating based on the Storm topology. However, because of our duplication strategy, each task will have duplicates stored on up to k+2 other tasks, where k is the number of duplicates. There would be a fair amount of overhead, and the end result would often be that every task on a bolt is required to be scheduled on a different machine. Given this fact, we can take a much simpler approach.

The strategy we decided to use was to schedule the tasks of every bolt such that all tasks in any one bolt are running on different machines. The benefit of this strategy is that it is a very simple and efficient approach that also simplifies the duplication process, as we don't have to keep track of which specific bolts have duplicated data. By ensuring this condition, duplicated data can simply be duplicated to k tasks without having to worry about which machines are running these tasks. We also would not have to reschedule while the topology is running.

However, this strategy does require that the number of tasks on any single spout or bolt are greater than the number of available machines or supervisors. This is a fair requirement, as Storm typical usage of Storm will already follow this requirement.

We used Storm's custom scheduler interface to implement our scheduler. The scheduler code is run by Nimbus every 10 seconds to check for any scheduling that needs to be done. Storm's Cluster class provides a function that returns a mapping from component (spout or bolt) to a list of executors of that component, one per task, that need to be scheduled. Using this mapping, we created a method that returns a new mapping from component to a list of supervisors that this component has tasks already scheduled on. With this mapping, the algorithm works as follows. It iterates through each component, finding the tasks that need to be scheduled. Then, it finds the list of supervisors that already have tasks of this component scheduled on them. Based on this list, it creates a list of supervisors that it can schedule to and schedules each remaining task one by one. This method ensures the correctness of k-safety, but we also need to consider load balancing so that our scheduler does not have a negative effect on overall latency.

The simple way to load balance is to spread out the tasks evenly on machines. This is how Storm's default scheduler works. If we have multiple choices for where to schedule a task, we schedule the task on the supervisor with the most worker slots available. This is a simple and easy approach to ensure each machine is roughly running an equal amount of tasks. The downside of this approach is that it does not account for the structure of the topology being run and does not account for the volume of data being processed and transferred at run time.

We chose to implement the simple strategy of evenly distributing tasks on each machine to handle load balancing. We chose this strategy is because of our k-safety and time constraints. We need to ensure that all tasks from a specific bolt are running on different machines, which limits our options. The strategy used in the offline scheduler by Aniello et al to reduce network traffic requires assigning tasks on the same machine if their bolts communicate with each other (Aniello et al. 2014). However, this strategy will not work for us, as we cannot schedule more than one task from the same bolt on one machine. If we violate this constraint, we would not be able to guarantee that copies of data reside on different machines, and thus would not have k-safety. Fortunately scheduling each task on a separate machine does, on average, also lower inter-network traffic when compared with the typical storm scheduler, similar to the first scheduler created by Aniello et al (2014). This is because tasks in a single bolt will never send tuples directly to each other, so enforcing this constraint allows more opportunity for other tasks to be scheduled. Thus, the simple even distribution strategy, while not optimal, is still quite effective, and also allows us to satisfy the constraints to make k-safety work.

4. Results

In order to test the performance of our system, we created several Storm topologies to test both stateless and stateful processing. We ran Storm on a cluster of 8 virtual machines, each having 4 cores, 8GB of memory, and running CentOS 6. We used one Nimbus instance on one machine and 6 supervisors with 4 worker slots each on 6 other machines. We also created a data server application to communicate with the topologies. The data server sends timestamps to the spout, and each timestamp is attached to a tuple. When the tuples reach the final bolt, the

timestamps are sent back to the data server, which then computes the time difference to get the latency of the tuple. The data server controls the tuple emission rate by changing the rate at which it sends timestamps.

Our first set of topologies test stateless processing. These topologies simply have one spout and two bolts, where the spout outputs 256 byte strings along with a timestamp per tuple. Each tuple is then passed on to a middle bolt and then a final bolt. Two of the topologies use normal storm with the default scheduler, and one of them uses storm's acking system, while the other doesn't. By using Storm's acking system, tuples will be replayed if they fail to be acked for any reason after a timeout. The third topology uses our k-safety implementation to duplicate the data sent to the second bolt, and de-duplicate the data sent to the final bolt. Testing for correctness, we found that our k-safety version with at least k=1 would produce the correct results after a machine failure, while the normal storm version without acking would be missing tuples. The topology design is shown below, where nodes represent spout or bolt tasks.

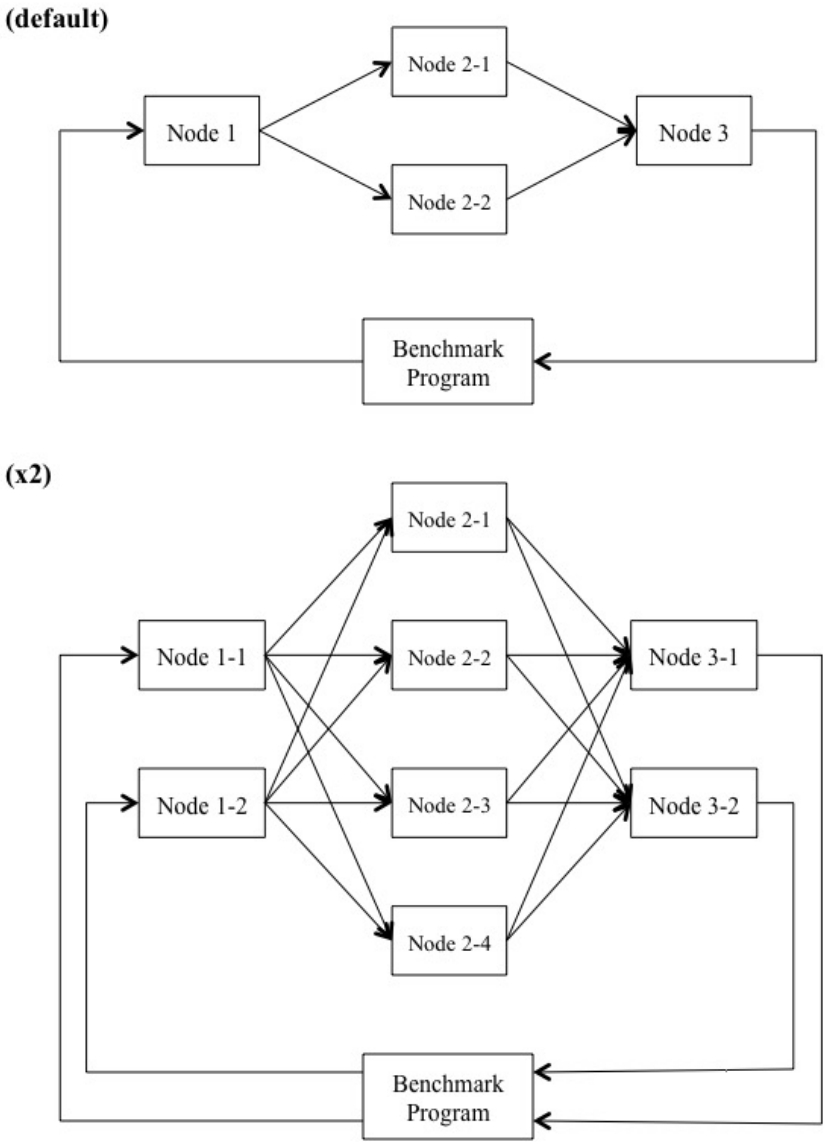
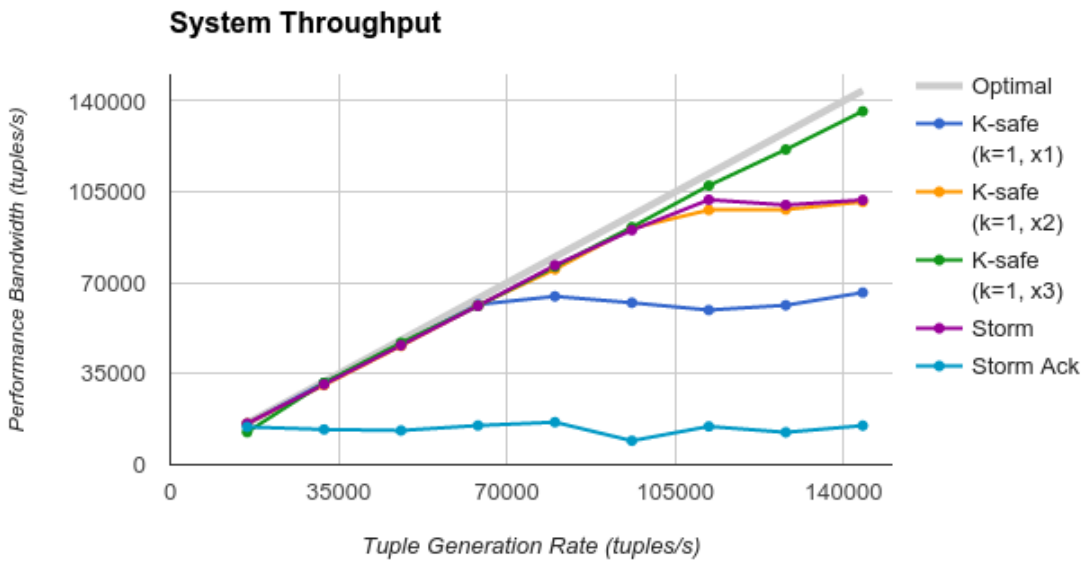
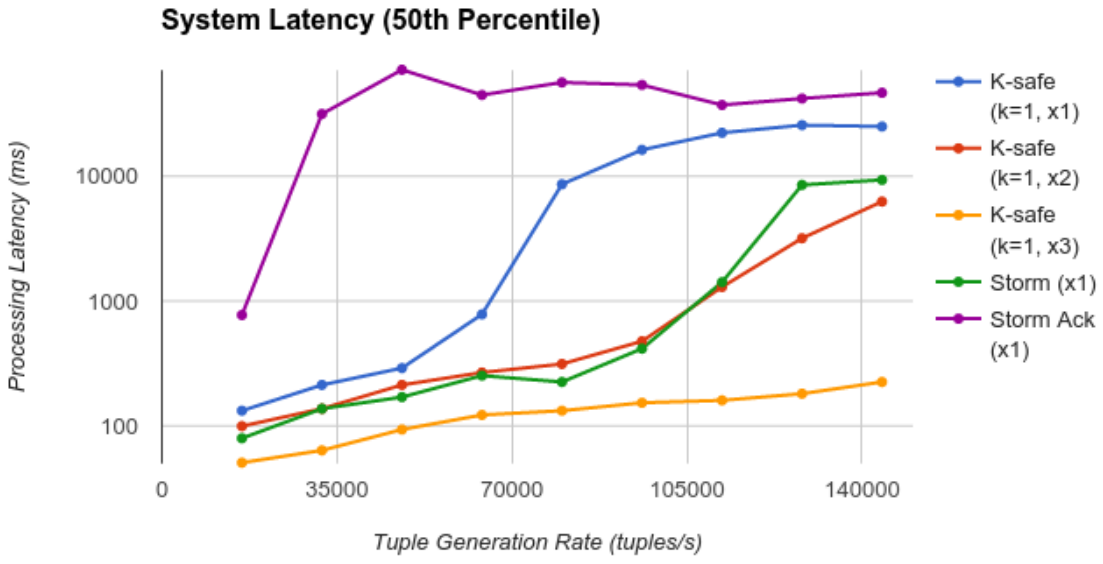


Figure 1: Benchmark Topologies

Figure 1 shows the topology design. The base topology x1 uses one spout task, 2 middle bolt tasks, and one final bolt tasks. The x2 and x3 have 2 times and 3 times the amounts of tasks for each component respectively. We measured both throughput and latency with 50th and 95th percent quantiles, using a method for computing biased quantiles for efficiency (Cormode et al. 2005).


System Throughput
Performance Bandwidth (tuples/s)
140000
105000
70000
35000
0
0
35000
70000
105000
140000
Tuple Generation Rate (tuples/s)
Optimal
K-safe (k=1, x1)
K-safe (k=1, x2)
K-safe (k=1, x3)
Storm
Storm Ack


System Latency (50th Percentile)
Processing Latency (ms)
10000
1000
100
0
35000
70000
105000
140000
Tuple Generation Rate (tuples/s)
K-safe (k=1, x1)
K-safe (k=1, x2)
K-safe (k=1, x3)
Storm (x1)
Storm Ack (x1)

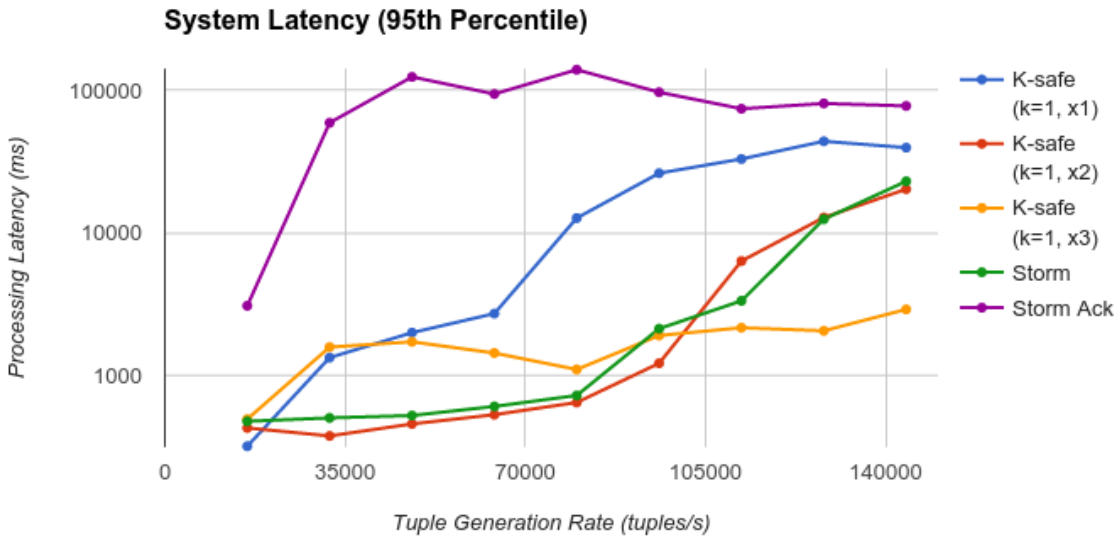
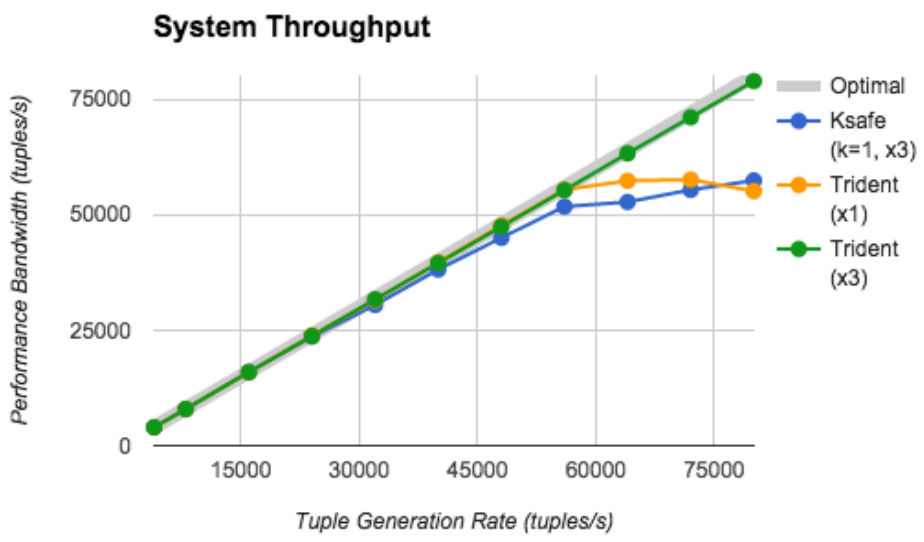


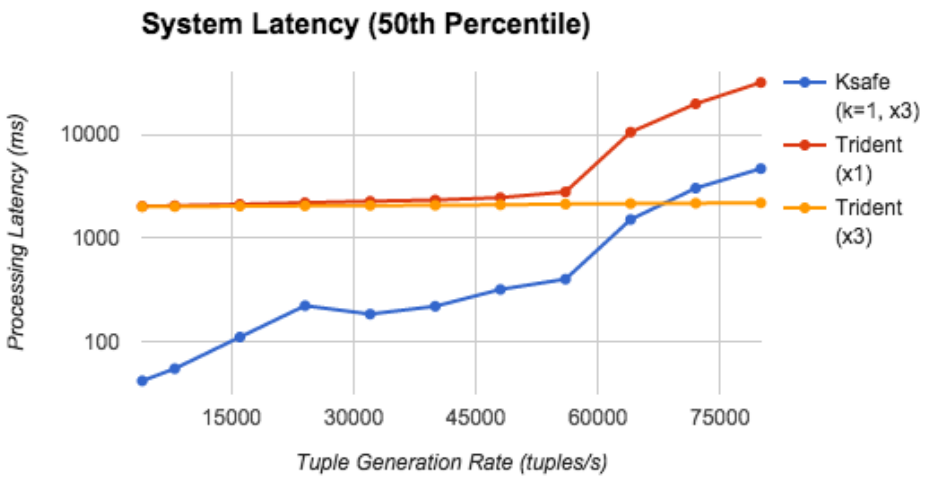
Figure 2: Benchmark results for stateless topologies

Figure 2 shows the benchmarking results for the stateless topologies. We can see that basic storm without acking can sustain around 100,000 tuples per second with one spout, while using acking severely reduces the throughput down to around 15,000 tuples per second. For our k-safe version with k=1, we can sustain about 65,000 with the same amount of tasks, 100,000 with twice the amount, and 135,000 with three times the amount. Our k-safe implementation scales well with more tasks being used, and requires roughly 2 times the tasks to perform equally with normal storm without acking. However, when taking fault tolerance into account, our implementation has a clear advantage, as Storm's implementation of fault tolerance, the use of acking, drastically lowers its throughput. In terms of latency, we observed that all versions of storm would have slightly increased latency as throughput is increased, then a large increase when it hits its bandwidth limit. We would expect the latency to remain constant until the topology hits its bandwidth limit. It is likely that the issue lies with the Java Virtual Machine, mostly with garbage collection. The 50th percentile and 95th percentile have similar shapes,

with 95th percentile latencies being higher due to the fact that some tuples, especially those running right when the topology starts, have higher latency than others.

Our second set of topologies test stateful processing through a word count. Our k-safe implementation uses k=1 and has a spout that outputs words as tuples, a subsequent bolt that keeps the word counts, and a final bolt for emission that de-duplicates as well. The second topology is created using Trident, an abstraction built on top of storm which processes tuples in batches and allows for high-throughput stateful processing and distributed querying (Trident 2014). The trident topology outputs words to an Apache Cassandra database to store the counts persistently (Cassandra 2015). The words being processed are 256 bytes long, with 26 different words being counted. Here are the results for these topologies.





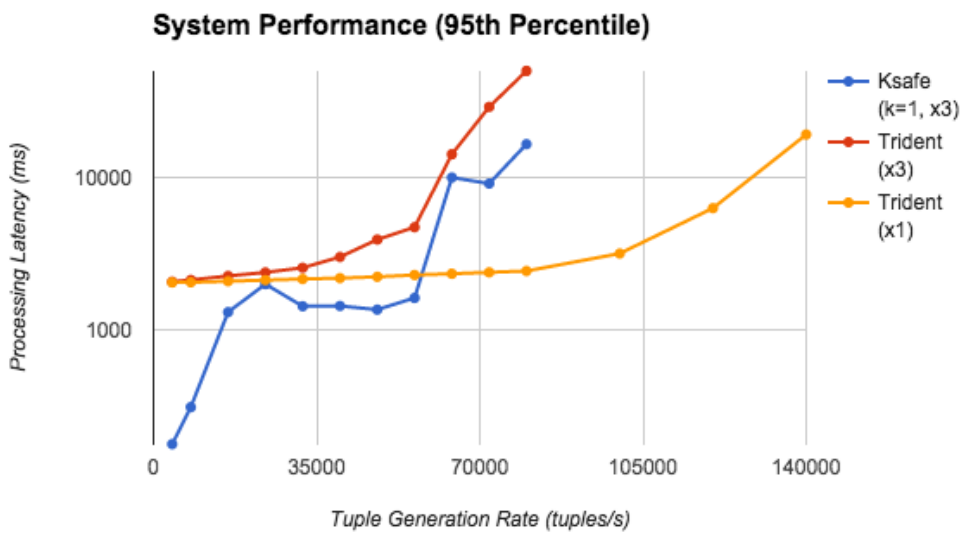


Figure 3: Benchmark results for stateful topologies

Figure 3 shows our benchmarking results for stateful topologies. Our stateful topology's throughput limit with k-safety is 55,000 tuples per second with 3 spouts, which is a little less than half than that of our stateless topology with an equivalent number of tasks. This is due to the overhead required for our window recovery strategy, as well as the constant state updates after each tuple. Trident with one spout is roughly equivalent to our k-safe system with three

spouts, while Trident with 3 spouts can handle 120,000 tuples per second. In the latency graphs, Trident's latency is measured per batch. Similar to the stateless topologies, the latency rises slowly with increased throughput, with a large increase when hitting its bandwidth limit. Additionally, the 95th percentile latencies are much higher than that of the 50th percentiles, with fairly large increase even before hitting the throughput limit. This can be attributed to the increased delay between the starting of the topology and the outputting of tuples caused by our initial word construction. While Trident performs better in terms of throughput, our implementation has better per-tuple latency until hitting its throughput limit. This makes our version a better choice when per-tuple latency is important.

Our system produced the correct results in all situations. In all cases tested, after a supervisor went down, the scheduler successfully reassigned the task to another supervisor as long as there were still available supervisors unused by the component. Otherwise, the scheduler would wait until the supervisor came back up, then reschedule back onto the supervisor. When we tested our stateless topology, we found that without k-safety, part of the count output would be missing some data if a supervisor went down, as expected. When we had data duplicated at least once, the process would output the correct results even with a machine down, as each count was present on 2 different machines. When we tested the topology that stores state at the counting bolt, the behavior was identical to the stateless topology, even after a machine failure, due to our window recovery strategy.

To test recovery, we took down a machine and observed the effect on latency. We made sure to only take down a machine that was running one of the middle bolt tasks, as these are the machines that have duplicated data. When testing any of our topologies with k-safety, the

latency would not increase and would actually slightly decrease the latency by about 5%. The decrease is likely due to the decrease in work that the de-duplication bolts need to do when receiving less tuples. Normal Storm and Trident have large delays for some tuples or batches when machines fail, due to the fact that these tuples or batches have to be replayed after a timeout.

5. Conclusion

In this section, we discussed our implementation of k-safety into Storm, focusing on the task scheduling aspect of the system. We covered related work dealing with scheduling in distributed systems, and analyzed whether we could potentially use these ideas. While the requirements and constraints of our k-safety implementation make scheduler optimizations difficult, the evenly balanced scheduling distribution still allows us to achieve k-safety without a large loss in processing speed. Our system is capable of running stateless and stateful topologies with k-safety. The performance of our system is roughly equivalent to a normal Storm topology without acking when our system uses 2 times the amount of tasks than that of the Storm topology. When comparing to Trident, our system requires about 3 times the amount of tasks to have equivalent, but we have lower per tuple latency as well. When taking fault tolerance into account, our implementation has a clear advantage, as our k-safe version does not stop processing and has no latency increase when a machine fails. As such, our version of Storm with k-safety is well suited for applications that need to maintain low latency without gaps when machines fail.

Concluding Reflections

We implemented a version of k-safety into Apache Storm, a distributed stream processing system that processing real-time data. The addition of k-safety allows the system to have fault-tolerance without the presence of delays or latency drops when machines fail. While this also increases the network traffic, we show that increasing the number of machines in proportion to the level of duplication lets us match the current version of Storm in terms of performance. Our system is particularly suited for applications that prioritize low latency and no delays.

There is still work that can be done to further improve our system. We can explore alternative de-duplication mechanisms to reduce the overhead required for removing duplicates. Additionally, we can explore alternative recovery schemes, such as transferring state between tasks after failures. There are many optimizations we can try on to add to our scheduler implementation as well. While our main goal with the scheduler was to guarantee fault tolerance and an even distribution of work across machines, there are other scheduling techniques, such as those discussed earlier, that we could potentially use to further reduce latency. Online schedulers, or schedulers that make scheduling decisions based on the topology while it is running, may prove particularly useful in this regard.

As the number of online services and users increase, there will also be demand for new and efficient methods of processing data. Additionally, with the growth of the data processing industry, more companies will be looking for different systems to satisfy their needs. By making our system open source, we can allow a wide range of people to benefit from it, while also

opening possibilities for improvement from the community. More and more services require processing real-time data, and our system a great solution for processing this data with low latency and no delays.

Works Cited

Abadi, Daniel J., et al. "Aurora: a new model and architecture for data stream management." *The VLDB Journal—The International Journal on Very Large Data Bases* 12.2 (2003): 120-139.

Abadi, Daniel J., et al. "The Design of the Borealis Stream Processing Engine."*CIDR*. Vol. 5. 2005.

Aniello, Leonardo, Roberto Baldoni, and Leonardo Querzoni. "Adaptive online scheduling in storm." *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013.

"Apache Spark - Lightning-Fast Cluster Computing." *Apache Spark - Lightning-Fast Cluster Computing*. Apache Software Foundation. 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.

Borthakur, Dhruba. *Looking at the code behind our three uses of Apache Hadoop*. Facebook. 2010. <https://www.facebook.com/notes/facebook-engineering/looking-at-the-code-behind-our-three-uses-of-apache-hadoop/468211193919> accessed February 16, 2015.

"Cassandra." The Apache Project., n.d. Web. 27 Apr. 2015. <http://cassandra.apache.org/>.

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

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

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. <http://clients1.ibisworld.com/reports/us/industry/ataglance.aspx?entid=1281>, accessed February 13, 2015.

Fischer, Lorenz, Thomas Scharrenbach, and Abraham Bernstein. "Network-Aware Workload Scheduling for Scalable Linked Data Stream Processing."*International Semantic Web Conference (Posters & Demos)*. 2013.

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

"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.

Ghaderi, Javad, Sanjay Shakkottai, and R. Srikant. "Scheduling Storms and Streams in the Cloud." *arXiv preprint arXiv:1502.05968* (2015).

Harland, Bryant. *Social Networking - June 2014 - US*. Mintel. <http://academic.mintel.com/

homepages/sector_overview/6/> accessed February 16, 2015.

Hulkower, Billy. Consumer Cloud Computing - US - December 2012. <http://academic.mintel.com/display/624303/>, accessed February 13, 2015.

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

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

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

Kahn, Sarah. *Social Networking Sites in the US.* Rep. no. 0D4574. IBISWorld, Feb. 2015.

Kamburugamuve, Supun, et al. *Survey of Distributed Stream Processing for Large Stream Sources*. Technical report. 2013. Available at http://grids. ucs. indiana. edu/ptliupages/publications/survey_stream_proc essing. pdf.

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.

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

Lerner, Josh and Jean Tirole. "The Economics Of Technology Sharing: Open Source and Beyond," Journal of Economic Perspectives, 2005, v19(2,Spring), 99-120

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>.

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

Neumeyer, Leonardo, et al. "S4: Distributed stream computing platform." *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010.

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

R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a High-Performance, Distributed Main Memory Transaction Processing System," Proc. VLDB Endow., vol. 1, iss. 2, pp. 1496-1499, 2008

Rychly, M.; Koda, P.; Smrz, P., "Scheduling Decisions in Stream Processing on Heterogeneous Clusters," *Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on* , vol., no., pp.614,619, 2-4 July 2014

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

Stonebraker, Michael, Uğur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing." *ACM SIGMOD Record* 34.4 (2005): 42-47.

"Storm, Distributed and Fault-tolerant Realtime Computation." *Storm, Distributed and Fault-tolerant Realtime Computation*. Apache Software Foundation, n.d. Web. 28 Nov. 2014. <https://storm.apache.org>.

Toshniwal, Ankit, et al. "Storm @Twitter." *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014): 147-156.

"Trident Tutorial" *Trident Tutorial*. Apache Software Foundation, n.d. Web. 10 April. 2015. <https://storm.apache.org/documentation/Trident-tutorial.html>.

"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. February 22, 2015.