High Availability on a Distributed Real Time Processing System

Enrico Tanuwidjaja
Michael Franklin, Ed.
John D. Kubiatowicz, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-134
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-134.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

Enrico Bern Hardy Tanuwidjaja

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.

High Availability on a Distributed Real Time Processing System

Enrico Tanuwidjaja

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”, 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 a massive amount of 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 massive amounts 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, this software is 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, 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 has its own philosophies in terms of design and development, which can not 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 re-compute 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 and 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 processes data 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, batch processing and stream processing have their own strengths and weaknesses due

to their different natures. This difference is the key differentiator between our project and Hadoop.

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 re-compute everything again. As we can imagine, this property will cost time. Also, there are overheads associated with re-launching 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 fewer 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 re-compute 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, meaning 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 a 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 re-computation 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 will 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 that 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 Cisco or us.

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 output 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 section describes the technical contributions to the capstone project that I have been working in collaboration with Ashkon Soroudi and Jianneng Li. The project is supervised by Zhitao Shen from Cisco Systems, Inc. and Professor Michael Franklin.

The project aims to implement a variation of *k*-safety on an open source distributed real-time stream processing system, namely Apache Storm ("Storm" 2015). A stream processing system allows applications to execute continuous queries over an infinite stream of data (formatted as tuples), and there have been many different implementations of such systems since more than a decade ago (Babcock et al. 2002; Carney et al. 2002; Madden and Franklin 2002; Abadi et al. 2003; Chandrasekaran et al. 2003). We define a stream processing system to be *k*-safe if it is capable of tolerating *k* machine failures at a time, where *k* is a predefined parameter. Storm was designed with fault tolerance as one of its goals; hence it already has *k*-safety (Toshniwal et al. 2014). However, its performance degrades when machines fail. Our goal is to make the system capable of running normally without stopping or having a delay when at most *k* machines fail.

In order to successfully create the system, there are three main components that we need to implement. First, we need to implement tuple duplication, which creates replicas of a tuple and sends them to different machines, followed by deduplication, which eliminates duplicated tuples to ensure that all tuples are processed exactly once. Second, we need a scheduler that

allocates tasks to machines. In order to enforce the *k*-safety that we desire, the scheduler needs to ensure that duplicate tuples are directed to different machines. We must design the scheduler carefully as its decisions impact performance (Rychlý et al. 2014). Third, we need to implement a recovery method to resurrect the correct state for any machine that has just come back from a failure (Koo and Toueg 1987). These three components are equally important for the success of the entire system.

Furthermore, there are some rudimentary tasks that are nonetheless essential and need to be performed. In order to test our system, we need to set up Storm to run correctly on a cluster of machines. We also need to create Storm topologies (applications that use Storm) that concretely utilize *k*-safety for demo as well as testing.

To cover the entire project, we divide the discussion as follows. I discuss the tuple duplication and deduplication aspect of our system and the Storm topologies that we created. Soroudi discusses the scheduler, and Li discusses the recovery solution. This division closely resembles how we divide the task in reality.

The rest of the paper is organized as follows. I begin with analyzing related work in Section 2. I discuss the duplication and deduplication technique that our system uses in Section 3. I describe the Storm topologies that we created in Section 4. Finally, I present the results in Section 5, followed by a summary in Section 6.

2. Related work

Fault tolerance, which is the ability to continue operating in the event of some failures, is commonly implemented in systems. The first fault tolerance technique was introduced in 1951 for memory error detection (Siewiorek 1982). All fault tolerance techniques involve redundancy

in some way (Cristian 1991). The Google File System, for example, uses both checksums and data duplication (Ghemawat et al. 2003). Different fault tolerance methods are tailored for different purposes. Our project focuses on fault tolerance that emphasizes high-availability. In our context, high-availability refers to the ability of a system to operate continuously in almost all of the times. Duplicating the data is generally required to achieve high-availability because it enables a fast recovery time on failures (Hwang et al. 2003). This section focuses on analyzing data duplication and deduplication from prior art in stream processing.

Hwang et al. provided a nice comparison of techniques used to achieve fault tolerance that emphasize high-availability (2003; 2005). These techniques were discussed in the context of tolerating a single machine failure, but they can be extended to tolerate k machine failures. Figure 1 provides a high-level summary of these techniques. Diagram **(A)**, **(B)**, and **(C)** depict previous techniques called passive standby, upstream backup, and active standby respectively, while diagram **(D)** depicts our technique.

Figure 1 illustrates a simple k-safe topology with $k = 1$ that uses four nodes. For the purpose of this discussion, assume that Node 1 and Node 3 are always available (a common assumption in many related literatures, as these nodes can be made k-safe using the same techniques described in this paper). Also, assume that the connections between nodes are reliable (such as TCP connection) so there will be no packet loss unless the machine fails (Postel 1981). Node 1 takes input from the outside world, such as the Internet or sensors. Node 1 sends tuples to Node 2-1 and in some cases to Node 2-2 as well, which is the backup node for Node 2-1. Finally, Node 2-1 (and/or Node 2-2) sends tuples to Node 3, and Node 3 outputs the results. Assume that

Node 2-1 is the primary node and Node 2-2 is the backup node, although in reality they are interchangeable.

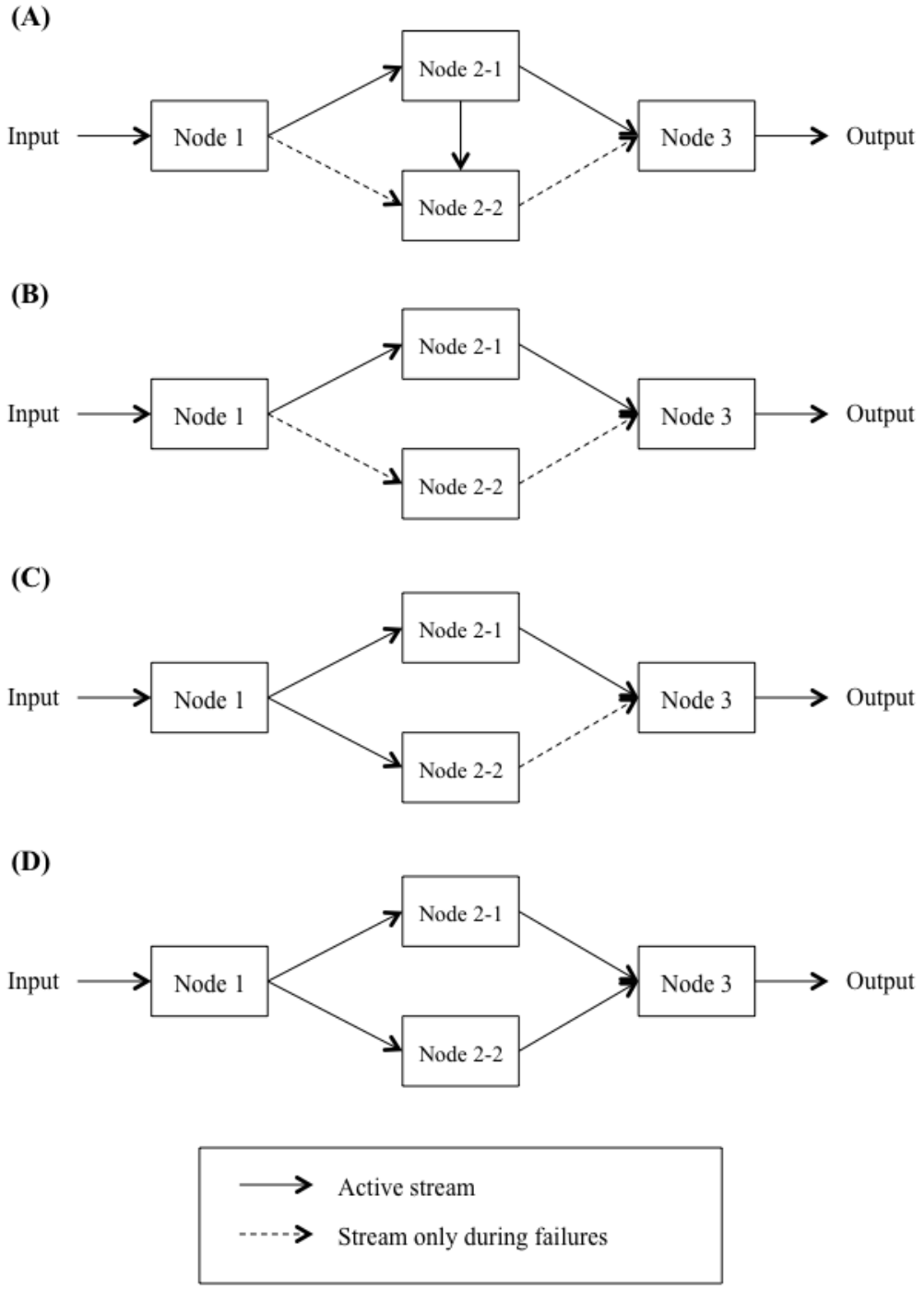

Figure 1. Comparison between high-availability techniques in stream processing

Passive standby, which is pictured in Figure 1 diagram **(A)**, was originally designed with the idea of having process pairs where multiple processes work together and complement each other to provide fault-tolerance (Bartlett, Gray, and Horst 1987). For example, the primary node in the diagram above, Node 2-1, is paired with a backup node, Node 2-2. The primary node periodically transfers a replica of its state to the backup node called a checkpoint (the arrow from Node 2-1 to Node 2-2). The source node (Node 1) stores its output tuples in a buffer until they have been checkpointed. When the primary node (Node 2-1) fails, the backup node (Node 2-2) takes over from the last checkpoint, and the backup node then becomes the primary node. In this technique, the duplication occurs when the primary node sends a checkpoint to the backup node. When the primary node fails, the backup node may output tuples that have previously been outputted by the primary node because the source node replays every tuple from the last checkpoint. Thus, the next node (Node 3) needs to perform tuple deduplication.

With upstream backup, the input tuples are saved in a buffer until all nodes downstream have finished processing them. Figure 1 diagram **(B)** pictures this technique. Node 1 saves the tuples in a buffer even after it sends them to Node 2-1, until Node 3 says that it is safe to discard them. When a failure occurs, a backup node takes over and the source replays all tuples in the buffer to the backup node. That is, when Node 2-1 fails, Node 1 resends all tuples in its buffer to Node 2-2, and Node 2-2 then becomes the new Node 2-1. In this situation, the source (Node 1) may resend a tuple that has actually been processed by all nodes downstream (though this depends on the actual implementation), in which case the downstream nodes (for example, Node 3) have to perform deduplication. Overall, this technique uses the minimum amount of duplication and deduplication.

The active standby technique involves process pairs as well (Hwang et al. 2005; Shah et al. 2003). Pictured in Figure 1 diagram **(C)**, this technique essentially makes the source node sends the same tuples to the primary and backup nodes at once. Both primary and backup nodes process the tuples even when their partner is alive. Then, only the primary node emits the output, while the backup node saves the output to its output buffer. Each tuple received by the downstream node is acknowledged (acked). The downstream node sends this ack to the backup node so that it knows that it is safe to discard the tuple from its output buffer. When there is no ack, it indicates that the primary node is dead, so the backup node begins emitting the tuples in its output buffer. In the diagram, Node 1 sends the tuples to Node 2-1 and Node 2-2 simultaneously, and both Node 2-1 and Node 2-2 process the tuples. However, only Node 2-1 emits the output to Node 3. Upon receiving a tuple, Node 3 sends an ack to Node 2-2 to inform that it is safe to discard the tuple. Depending on the actual implementation, Node 3 may receive duplicate tuples when a failure occurs such that it needs to perform deduplication. Hwang et al. showed that this technique has the fastest recovery time on a failure (since the backup node has the output tuples ready to be emitted at any time) at the cost of having the most redundancy compared with the two techniques mentioned above (2003).

These different techniques have different strengths and weaknesses. The upstream backup technique involves the least amount of overhead, but when a machine fails, the system needs some time to re-compute the failed tuples. The active standby technique involves more data overhead and more processing power but it has a faster recovery time due to having multiple nodes executing the same stream of tuples. The passive standby technique is the middle solution. Popular distributed systems (not limited to stream processing systems) have implemented these

techniques. Apache Hadoop and Apache Spark use the upstream backup technique (Borthakur 2007; Zaharia et al. 2012). Researchers have explored the passive standby technique using Hadoop as well (Wang et al. 2009). A model called Flux illustrates the usage of the active standby technique (Shah, Hellerstein, and Brewer 2004).

Our work is most similar to the active standby approach. The difference is that we do not use acks in our system, and both primary and backup nodes emit the output regardless of situation. Thus, the downstream node needs to perform deduplication at all times. In the example of our system portrayed in Figure 1 diagram **(D)**, Node 1 sends the same tuples to Node 2-1 and Node 2-2, and Node 2-1 and Node 2-2 both process all incoming tuples and both emit their outputs to Node 3. Thus, Node 3 has to deduplicate the tuples. As can be seen, we involve more duplication and deduplication compared to the previous three techniques, which can be approximately visualized by the amount of red color in the diagrams. However, this allows us to have zero recovery time when a machine fails. For example, when Node 2-1 fails, it does not affect the system at all since Node 2-2 continues to operate as a replica of Node 2-1. In summary, our technique is not necessarily better than the previous techniques, but rather it offers a solution for those who demand high-availability with zero recovery time at the expense of having more network traffic.

3. Our approach

Duplication

To accomplish k-safety, our system duplicates every tuple such that there will be $k + 1$ replicas of it on $k + 1$ different machines. Thus, the system is able to run normally when at

most k machines fail. All $k + 1$ replicas are treated like regular tuples and are processed in parallel on $k + 1$ machines. As a result, there will be $k + 1$ replicas of the same output that will be emitted to the next node. Hence, the downstream node must perform a deduplication.

Storm provides many methods to describe where the tuples should be sent. One of these methods is called "fields grouping". With fields grouping, the user can specify what fields in the tuple that they want to use as a key. Tuples with the same key are sent to the same machines. This is typically useful for aggregation operations. For example, if we want to create a word count application, we want the same words to go to the same machines so that each machine can completely compute the word count for a particular set of words.

We create our own grouping method called "*k*-safe fields grouping". While Storm's fields grouping guarantees that tuples with the same key are sent to the same machine, our *k*-safe fields grouping guarantees that tuples with the same key are sent to the same set of $k + 1$ machines. This set of machines is determined by the hash value of the key. To properly send the tuples to these machines, a correct scheduler is required, which is discussed in depth in Soroudi's paper (2015). Note that we do not need to send *all* tuples to a single set of $k + 1$ machines. We just need to ensure that tuples with the same key are sent to the same set of machines. It is important for correctness especially when performing aggregations. As mentioned earlier, word count may rely on having the same set of words on the same machines.

Deduplication

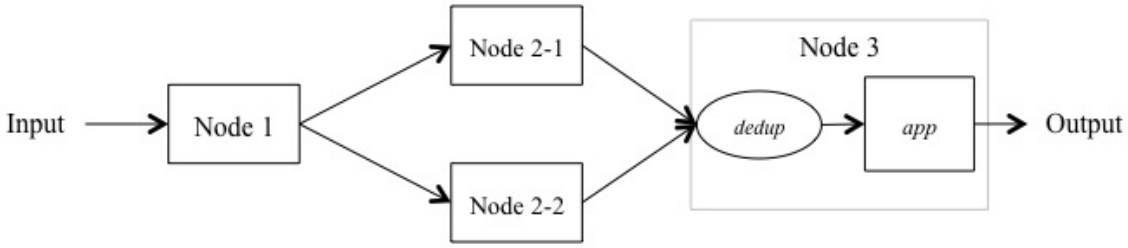

Figure 2. A closer look at diagram (D) in Figure 1

Deduplication can be performed upstream (without sending duplicate outputs) or downstream (after sending duplicate outputs). A system using the active standby approach such as Flux performs the deduplication upstream (Shah, Hellerstein, and Brewer 2004). In contrast, our system performs the deduplication downstream. Figure 2 shows that our system performs the deduplication downstream inside Node 3. For comparison, a system using the active standby approach usually performs the deduplication upstream, specifically in Node 2-2 (by avoiding sending its output unless Node 2-1 fails). By performing the deduplication downstream, our system burdens the network with a higher traffic, since Node 2-1 and Node 2-2 both send their outputs to Node 3. However, unlike active standby systems, our system benefits from not requiring heartbeats or acks. Furthermore, it has zero recovery time. This is because the upstream nodes continuously produce outputs and send them downstream regardless of situation. In the illustration above, Node 2-1 and Node 2-2 continuously send their outputs to Node 3, thus when either one of Node 2-1 or Node 2-2 fails, Node 3 still receives incoming tuples from the other node. Therefore, we chose this deduplication strategy to achieve the fastest recovery time possible at the cost of higher network traffic.

As can be seen in Figure 2, the deduplication (pictured by the oval labeled *dedup*) occurs before running the user or application code. Thus, the stream of tuples sent to the application code is guaranteed to be duplicate-free. As such, developers can write application code without worrying about duplicate tuples.

To uniquely identify each tuple and deduplicate, our system automatically adds three additional fields to each tuple: *seqnum*, *source*, and *primary*.

- *Seqnum*: the sequence number of this tuple. Each source node should assign a monotonically increasing sequence number to every tuple that it produces. This is useful to differentiate different tuples from the same source.
- *Source*: the source node that created this tuple. This allows differentiating tuples that have the same sequence number but come from different sources. Tuples coming from different sources should not be considered duplicates, as different sources are assumed to generate different data.
- *Primary*: the primary node for this tuple, which is the node that processes this tuple if $k = 0$. This field gets updated at every step. That is, after processing a tuple, a node must update this field in that tuple. This is described in a greater detail below.

To perform deduplication, each node (except source nodes) inspects the values of (*seqnum, source, primary*). Tuples with the same (*seqnum, source, primary*) are considered duplicates. Since each source generates a monotonically increasing *seqnum*, each node stores the maximum *seqnum* that it has seen for each (*source, primary*). Each node expects that tuples will arrive with increasing *seqnums*. Thus, if a tuple has a *seqnum* lower than or equal to the

maximum *seqnum*, it will be removed from the stream. Table 1 provides some examples to illustrate this concept.

Current state (*source, primary*) → max *seqnum*	**(*source, primary, seqnum*) of the arriving tuple**	**Deduplicate tuple**
(1,1) → 5 (1,2) → 3	(1,1,5)	Yes
	(1,1,6)	No
	(1,2,2)	Yes
	(1,2,4)	No

Table 1. Examples of deduplication decisions

The *primary* field is necessary because Storm does not guarantee the global ordering of tuples, although it guarantees that tuples will arrive in order when sent between two nodes. Figure 3 illustrates this concept. In the figure, the source generates four tuples with sequence numbers 1, 2, 3, and 4 with $k = 0$ (no duplication). Suppose that the source sends tuples 1 and 2 to Node A and tuples 3 and 4 to node B. Storm guarantees that Node A will receive tuples 1 and 2 in order and Node B will receive tuple 3 and 4 in order. However, Node B may finish processing tuple 3 before Node A finishes processing tuple 1. Thus, it is possible for Node C to receive tuple 3 before receiving tuple 1. In this case, even though tuple 1 has a lower sequence number than tuple 3, it should not be removed. The value of *primary* in tuple 1 is "Node A", while the value of *primary* in tuple 3 is "Node B". Since the two tuples have different *primary* values, they will not be removed. On the other hand, if tuple 2 arrives at Node C before tuple 1, then tuple 1 will be removed since it has a lower *seqnum* than tuple 2 and the same *primary*

value with tuple 2 ("Node A"). However, this scenario is impossible because Storm guarantees that tuples come in order between two direct nodes.

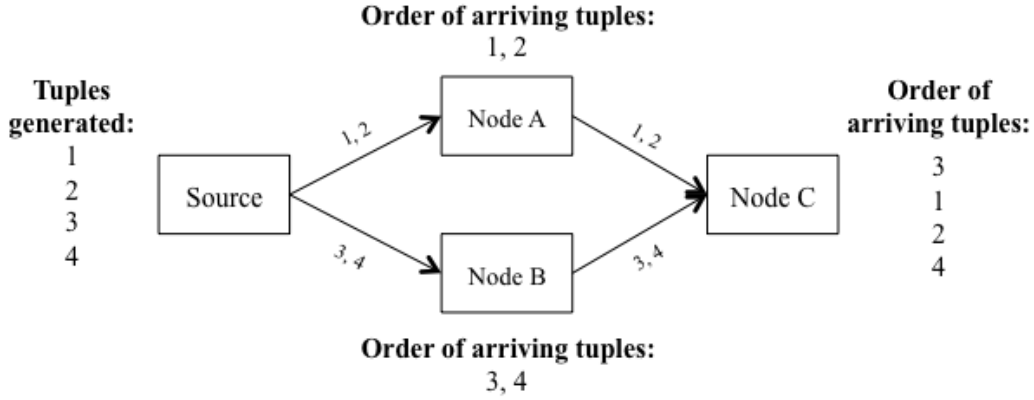

Figure 3. Example of tuples arriving out of order

For the system to deduplicate correctly and produce the correct output, aggregations must be executed carefully. The timing of emitting aggregation results must be determined by the tuples. A common mistake is to emit aggregations at a predetermined time interval, such as every five minutes or every hour. However, this approach would not work with our system because it emits aggregations not based on the tuple, but based on an outside source (system time) instead.

Figure 4 illustrates this problem. Suppose that we have two nodes, Node A and Node B, and suppose that they are a backup of each other such that they receive and process the same tuples. The goal of these nodes is to compute the sum of the values in the tuples that they receive. Suppose that the user programs this application such that these nodes emit the sum every second. However, these nodes do not receive the tuples at exactly the same time. Node A receives the tuples with values 2, 3, 3, and 5 within the first second, so it outputs 13. However, Node B only receives the tuples with values 2, 3, and 3 within the first second, so it outputs 8. These outputs are supposed to be the same, but they are different. They are supposed to be

duplicates, but now they are not, and they will not be deduplicated correctly. In the next round, these two nodes emit different sums again. Node A emits 23 whereas Node B emits 28. As can be seen, the result becomes erroneous. This can be easily avoided if the timing of emitting aggregations is determined by the tuples, such as by using the tuples' timestamps or sequence numbers. For example, if we make Node A and Node B emit the sum whenever the sequence number of an incoming tuple is divisible by 5, then they will deterministically aggregate the same tuples and emit the same sums all the time. It is important to ensure that the timing of emitting aggregations is based on the tuple, not based on an external source such as system time.

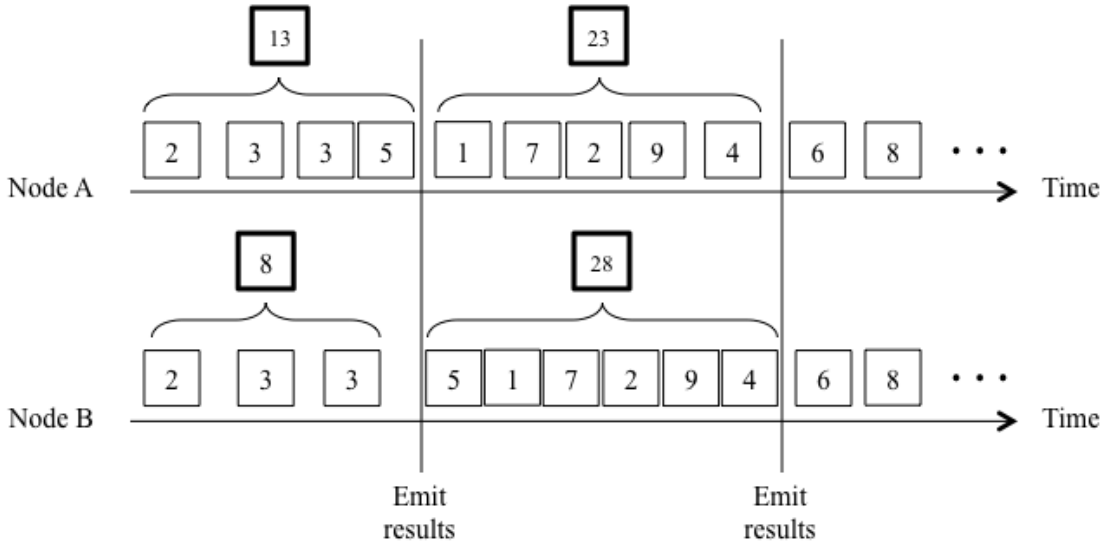

Figure 4. Illustration of a problem that occurs when aggregations is not deterministic

Finally, when performing aggregations, our system uses the (*seqnum*, *source*, *primary*) of the last tuple that a node has received, which should be the tuple that causes that node to emit the aggregation result (since the timing of emitting aggregations is based on a tuple). With this, deduplication can be performed correctly regardless of number of nodes, *k*, and aggregation behavior.

4. Useful topologies for demo and testing

There are two topologies that we consider useful for demo and testing, one with stateless nodes and one with stateful nodes. Nodes without states, such as those that only perform map and filter operations, are much easier to implement. Nodes with state, such as those that store any information in memory (for example, a counter), introduce new complications, especially when the node fails, since we need to recover its previous state.

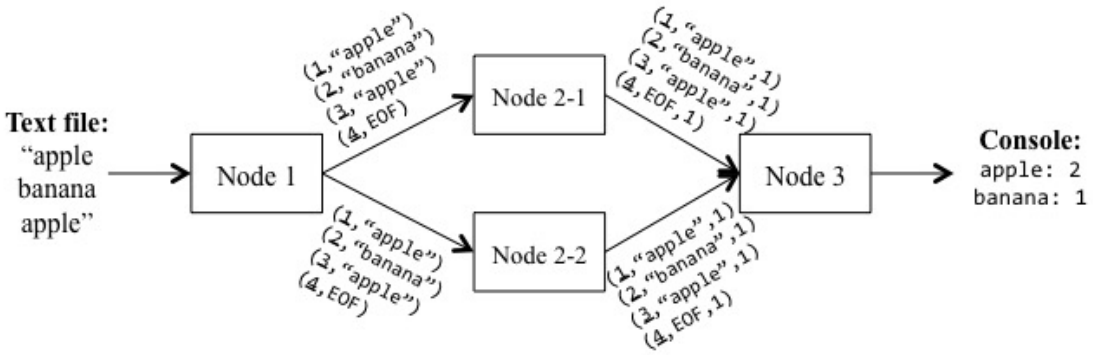

Figure 5. Example topology with stateless nodes

Figure 5 shows a simple but useful topology to test our system with stateless nodes with $k = 1$ (Node 1 and Node 3 are stateful, but again we assume that they are always up). The goal is to count the words in a text file and print the results to the console. In this setup, Node 1 reads a text file and splits the words into individual tuples. When it reaches the end of file, it emits an EOF tuple, and it starts reading the same file again from the beginning. Each tuple is given a monotonically increasing sequence number (*seqnum*), which is always the first field in the tuple and underlined (*primary* and *source* are emitted from the figure for simplicity). Node 2-1. Given $k = 1$, Node 1 emits the same tuples to Node 2-1 and Node 2-2. Then, Node 2-1 and Node 2-2 simply append a `1` to each tuple, which is the starting count. Node 3 performs deduplication and

aggregates the word count. Upon receiving an EOF tuple, it prints the word count to the console and clears its state. Using a text file containing "apple banana apple", this topology guarantees that Node 3 will continuously print `apple: 2` and `banana: 1` to the console, even if we shut down either one of Node 2-1 or Node 2-2. This is useful for demo and testing because it enables us to detect if there is something wrong with our system.

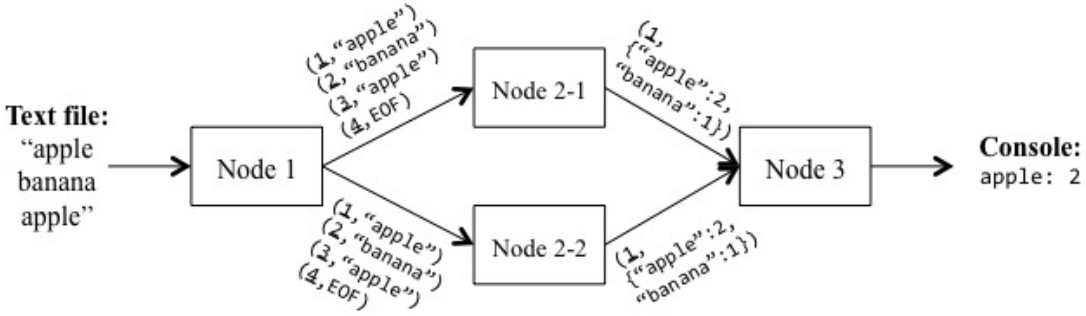

Figure 6. Example topology with stateful nodes

Figure 6 shows a similar topology with $k = 1$, but now the nodes have state. That is, Node 2-1 and Node 2-2 now compute the word count instead of simply appending a `1` to each tuple; hence they are stateful. These nodes send the word count to Node 3 upon receiving an EOF tuple. Node 3 performs deduplication and prints the word with the highest count to the console. As such, the output should always be `apple: 2`. Again, we can run experiments such as shutting down Node 2-1 or Node 2-2, and we expect that Node 3 will still continue to print `apple: 2`. If we find a different output, then we know that our system is erroneous. Therefore, this simple topology is useful for demo or testing, especially to show whether or not state recovery works.

Both of the two topologies mentioned above can be used to show the difference between our system and the original Storm. The difference will become apparent when we shut down either one of Node 2-1 or Node 2-2 when the system is running.

5. Results

We created a benchmark to evaluate the performance of our system and to compare it with other systems. Similar with other benchmarks for stream processing systems, such as Linear Road (Arasu et al. 2004) and IBM Streams (Nabi et al. 2014), the goal of our benchmark is to measure the maximum throughput that the system can sustain and the latency of processing each tuple.

We benchmarked our system along with other systems on a stateless topology and a stateful topology. In both scenarios, we used the following configurations:

- There are eight virtual machines (provided by Cisco), each having four cores, 8 GB of memory, and running CentOS 6.
- The number of machines that are actually used varies depending on the benchmark scenario.
- One of the machines are used solely for Storm's Nimbus, which is a master node that runs the scheduler and monitors the system.
- Each machine is set to have four worker slots, since it has four cores.
- The topology is shown in Figure 7. We use the topology labeled **(default)** unless "x2" is mentioned, in which case we use the topology labeled **(x2)**.
- Each tuple contains a payload of 256 bytes.

- We measure the system's performance over a period of 60 seconds.

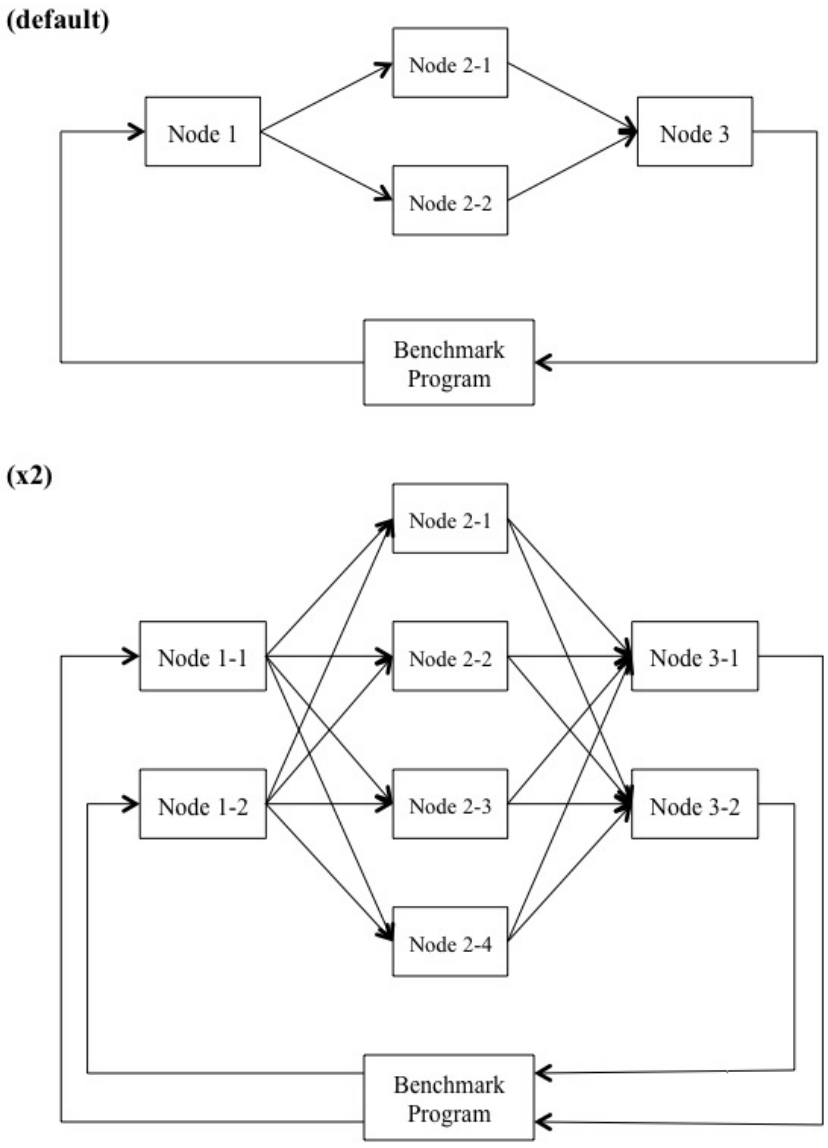

Figure 7. Benchmark topologies

First, we benchmarked our system on a stateless topology that does nothing but forwarding tuples. The goal of using this topology is to test the pure capabilities of our system not specific to any particular application.

Figure 8 shows the results, comparing our system with the original Storm with and without acking. Storm with acking guarantees a reliable delivery because the system acks every tuple, and a tuple that is not acked will be replayed. Storm without acking does not have this guarantee, but it has a lower amount of overhead. With $k = 1$, our system was able to sustain a throughput of about 65,000 tuples per second. Given more than 65,000 tuples per second as input, our system was not able to keep up, causing the latency to increase significantly at that rate of input. On the other hand, Storm without acking was able to sustain about 100,000 tuples per second. Cisco suggested us to increase the amount of machines proportionally with k because this is how they intend to use the system in real life. The rationale is to provide enough computing power to handle the increased amount of traffic. Therefore, we benchmarked our system again with $k = 1$ but using twice the amount of computing power. For this purpose, we used the topology labeled **(x2)** in Figure 7, and the result is shown with the label "x2" in Figure 8. With double the computing power, our system is able to match the throughput and the latency of Storm without acking. This is expected because our system has some overhead due to duplication and deduplication. As described in previous sections, our system adds an additional data to every tuple and performs a deduplication check for every tuple. Storm without acking does not have any fault tolerance, which allows it to run very fast due to having no overhead.

Note that there are nine nodes in the **(x2)** topology, but we only had seven available virtual machines. Thus, some machines acted like more than one nodes. This should not cause a significant performance impact because each machine had four cores so that those nodes can be run in parallel.

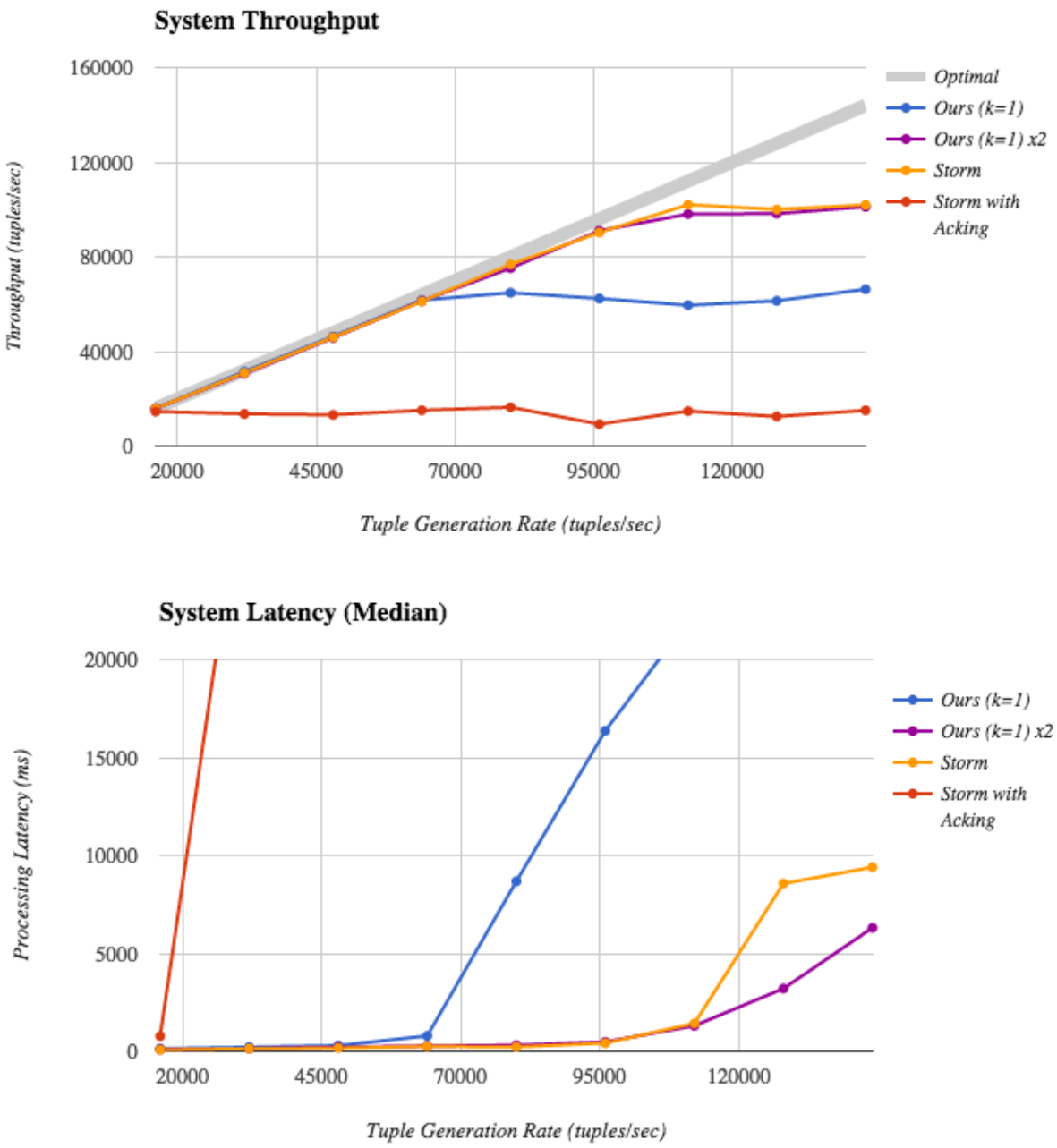

Figure 8. Benchmark Results on a Stateless Topology

Next, we tested our system on a stateful topology that runs a word count application. Figure 9 shows the results, comparing our system with Trident. Trident is an abstraction built on top of Storm that provides transactional support, which guarantees correct stateful computations

despite machine failures ("Trident Tutorial" 2015). Thus, Trident is commonly used for stateful applications. As seen in Figure 9, our system has a lower throughput than Trident even if we double the computing power (labeled "x2"). This is reasonable because Trident automatically optimizes batch computations. That is, Trident aggregates some tuples into mini batches before sending them to the network. Thus, it creates less network traffic. For example, in the word count application, Trident may aggregate 1,000 tuples having the same word, and then it sends the value "1,000" to the downstream node, telling it to increase the word count of that word by 1,000. With this, it saves network traffic, since it does not send the 1,000 tuples individually. For the latency graph in Figure 9, Trident's latency is measured as the latency per mini batch. It is expected that Trident has a higher latency than our system, since it does not immediately output tuples individually as soon as they arrive.

Finally, we simulated machine failures by shutting down some of the machines. This is where our system showcases its strength. When a machine fails, our system continues to run without affecting performance, while Storm or Trident must wait for a certain period of time before it can continue running, depending on the configuration. For example, the default timeout for Storm is 30 seconds; hence when a machine fails, Storm must wait for 30 seconds before it replays any tuple. Therefore, upon a failure, our system continues to operate normally, while Storm or Trident has some delay depending on the configuration.

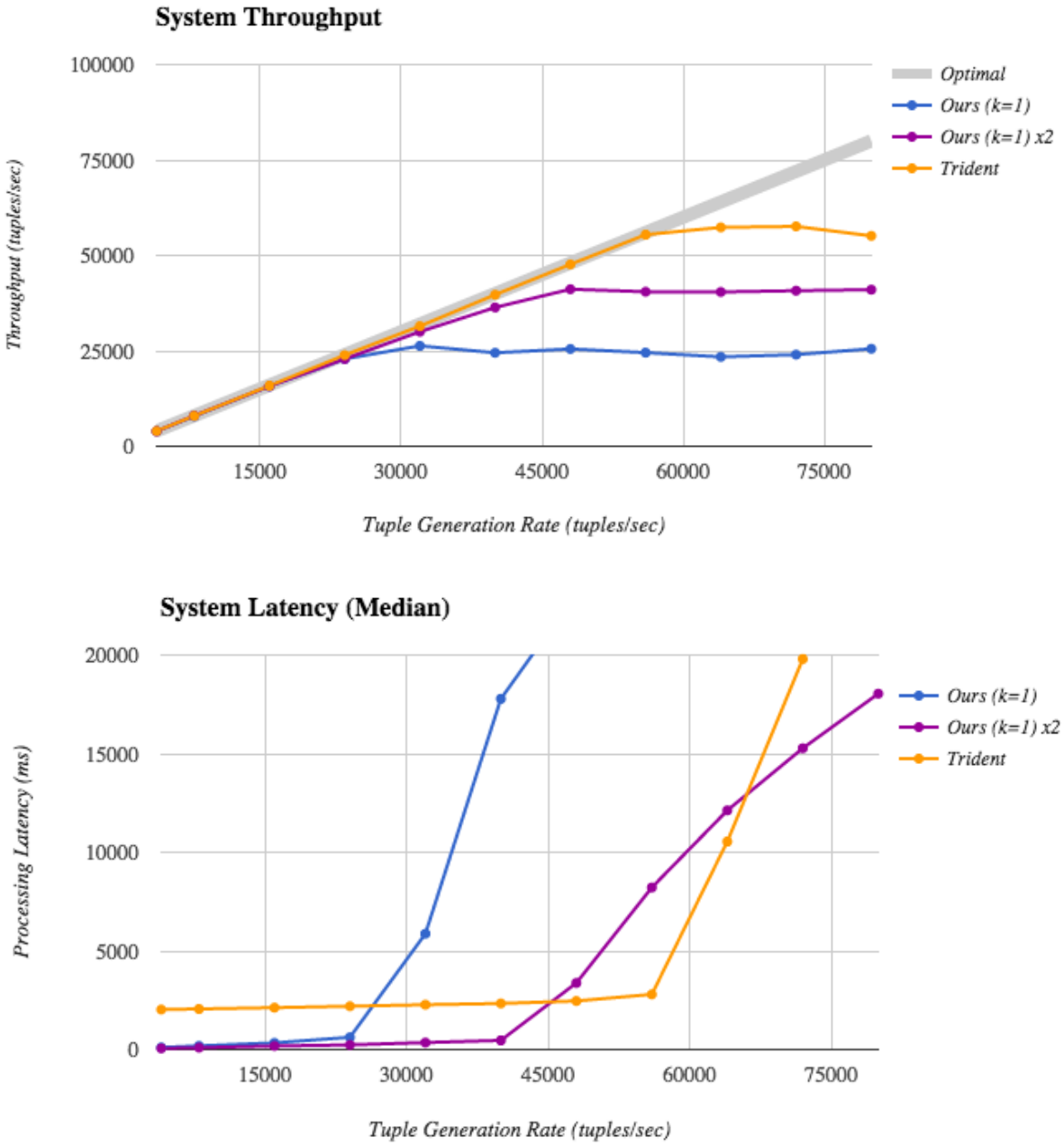

Figure 9. Benchmark Results on a Word Count Topology

6. Conclusion

In this section, I describe the technical aspect of our capstone project with a focus on duplication and deduplication design. To achieve *k*-safety, our system involves the most

duplication and deduplication compared with similar systems from previous work. As a result, our system incurs more network traffic and requires more computing power. On the bright side, our system does not incur any delay when at most k machine fails. In conclusion, our system excels in terms of high-availability, providing a new alternative for those who demand minimum latency upon failures at the cost of increased network traffic.

Concluding Reflections

We have implemented a fault tolerant method that enables a distributed stream processing system to tolerate machine failures without having any impact on performance at the time of failure. In other words, we have created a highly available system that processes data with low latency at all times. The drawback of our design is that it requires more computing power in order to handle the extra traffic due to data duplication. However, given enough computing power, our system is capable of operating smoothly with consistent low latency even in the event of machine failures.

There are still many rooms for improvements. First, it is important to reduce the duplication overhead if at all possible. For example, combined with a machine learning algorithm, we may be able to create a system that predicts when a machine will fail, such that it only starts duplicating data when it expects that a machine will fail in the near future. Next, it may be very useful to create a higher-level abstraction that can automatically create *k*-safe topologies without having the developers manually craft the topology components from scratch, since it is difficult to program the components given that so much parallelism and duplication are happening simultaneously. There are also many possible improvements to the current scheduler and recovery technique.

Nevertheless, the current trend is in our favor. The data processing industry has been rapidly growing in the past few years and is expected to grow at a similar rate over the foreseeable future. The market creates more and more applications with real time functionalities, demanding distributed systems in the back end that can process data in a low latency at all times.

With this trend, if given the right strategy and technical directions, it is possible to develop and expand this capstone project into something much bigger that can fulfill the real time processing demands of the current data-driven world.

References

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.

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.

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

Arasu, Arvind, et al. "Linear road: a stream data management benchmark."*Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004.

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

Babcock, Brian, et al. "Models and issues in data stream systems."*Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002.

Bartlett, Joel, Jim Gray, and Bob Horst. "Fault tolerance in tandem computer systems." *The Evolution of Fault-Tolerant Computing*. Springer Vienna, 1987. 55-76.

Borthakur, Dhruba. “Looking at the code behind our three uses of Apache Hadoop.” *Facebook*. 2010. 16 Web. February 2015.

Borthakur, Dhruba. "The hadoop distributed file system: Architecture and design." *Hadoop Project Website* 11.2007 (2007): 21.

Carney, Don, et al. "Monitoring streams: a new class of data management applications." *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002.

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

Chandrasekaran, Sirish, et al. "TelegraphCQ: continuous dataflow processing."*Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003.

Cristian, Flavin. "Understanding fault-tolerant distributed systems."*Communications of the ACM* 34.2 (1991): 56-78.

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.

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>.

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

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

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

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.

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

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.

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.

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

Koo, Richard, and Sam Toueg. "Checkpointing and rollback-recovery for distributed systems." *Software Engineering, IEEE Transactions on* 1 (1987): 23-31.

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

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-for-business-intelligence/>. accessed February 16, 2015.

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

Li, Jianneng. "Technical Contributions to High Availability on a Distributed Real Time Processing System." 2015.

Madden, Samuel, and Michael J. Franklin. "Fjording the stream: An architecture for queries over streaming sensor data." *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 2002.

Marz, Nathan. "History of Apache Storm and Lessons learned." *Thoughts from the Red Planet*. N.p., 6 Oct. 2014. Web. 17 Feb. 2015.

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

Nabi, Zubair, et al. "Of Streams and Storms." *IBM White Paper* (2014).

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

Postel, Jon. "Transmission control protocol." (1981).

Rychlý, Marek, Petr Škoda, and Pavel Šmrz. "Scheduling Decisions in Stream Processing on Heterogeneous Clusters." *Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference*. IEEE, 2014.

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

Shah, Mehul A., et al. "Flux: An adaptive partitioning operator for continuous query systems." *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 2003.

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.

Siewiorek, Daniel P., Gordon Bell, and Allen C. Newell. *Computer Structures: principles and examples*. McGraw-Hill, Inc., 1982.

Soroudi, Ashkon. "Technical Contributions to High Availability on a Distributed Real Time Processing System." 2015.

"Storm, Distributed and Fault-tolerant Realtime Computation." Apache Software Foundation, n.d. Web. 07 May 2015. <https://storm.apache.org>.

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

"Trident Tutorial." *Trident Tutorial*. Apache Software Foundation, n.d. Web. 07 May 2015. <https://storm.apache.org/documentation/Trident-tutorial.html>.

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, Feng, et al. "Hadoop high availability through metadata replication."*Proceedings of the first international workshop on Cloud data management*. ACM, 2009.

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

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

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.

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.