Benchmarking Fault-Tolerance in Stream Processing Systems

Master Thesis

by

Diana Matar

Submitted to the Faculty IV, Electrical Engineering and Computer Science
Database Systems and Information Management Group
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

as part of the ERASMUS MUNDUS programme IT4BI
at the
TECHNISCHE UNIVERSITÄT BERLIN
July 31, 2016

Thesis Advisors:
Dr. Tilmann Rabl

Thesis Supervisor:
Prof. Dr. Volker Markl

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Statutory Declaration

I declare that I have independently authored this thesis, that I have not used other than the declared sources/resources, and that I have explicitly marked all material that has been quoted either literally or by content from the used sources.

Berlin, July 31, 2016
Diana Matar

Acknowledgement

I would like to express my profound gratitude to my advisor Dr. Tilmann Rabl for his unstinted support. His guidance and feedback was of great help in formulating the overall organization of the thesis and design of the experiments.

I would also like to thank my dear friends and family for their love and encouragement.

Zusammenfassung

Die größte Herausforderung in der Verarbeitung von Datenströmen ist es einen Einblick in große Mengen an ungebundenen Echtzeitdaten zu geben mittels zuverlässiger und fehlertoleranter Architekturen. Zwei der verbreitetsten Datentromverarbeitungsysteme, Apache Flink und Apache Spark, bieten Ausfallsicherheiten in Bezug auf Performance und Fehler mittels verschiedener Fehlertoleranzmechanismen. Jedwede Technik für die Fehlertoleranz für Stromverarbeitung stellt einen Kompromiss zwischen Systemperformance und Mehrkosten für die Ausfallsicherheit dar. In dieser Arbeit wird die Performance von Apache Flink und Apache Spark in verschiedenen Fehlerszenarien ausgewertet. Unter den verschiedenen Datenstrombenchmarks gibt es nur wenige, die Werkzeuge für die Fehlerinjektion bereitstellen und die die Performance von Apache Flink und Apache Spark in realistischen Fehlerscenarios auswerten können.

In dieser Arbeit wurde der Yahoo! Streaming Benchmark verwendet um die Performance von Apache Flink und Apache Spark während der Injektion von transienten und permanenten Fehlern zu vergleichen. Die Fehler werden in Hadoop YARN injiziert mittels dem ArnarchyApe Fehlerinjektionswerkzeug

Abstract

The main challenge for stream processing systems is to provide insights into a huge volume of unbounded, real-time data using a reliable and fault-tolerant architecture. Two of the major stream processing systems, Apache Flink and Spark provide both performance and data quality guarantee against failure by using different fault tolerance approaches. Any fault tolerance approach in stream processing systems faces a trade-off between system performance and the fault tolerance overhead costs.

This thesis aims to evaluate the performance of the Apache Flink and Spark when subjected to different failures scenarios. Off the various streaming benchmarks, only a few provide fault injection tools and can realistically evaluate the performance of stream processing systems under real-world failure scenarios.

In the thesis, we used Yahoo Streaming Benchmarks to compare the performance of Apache Flink and Spark under transient and permanent faults injected into the Hadoop YARN environment using AnarchyApe fault injection tool.

Table of Contents

List of Figures

List of Tables

1 Introduction

Emerging class of data-intensive applications processes a huge amount of unbounded data on a daily basis for billions of global users. The most popular applications with real-time data updates are social networking applications such as Facebook and Twitter, where real-time data processing powers many use cases such as big data analytics, real-time reporting and dashboards that provides insight to the current state of such applications. With the large volume of data generated by these applications, the backend stream processing systems receive and process billions of events per day under different constrains such as scalability, performance, availability, and fault tolerance. Fault tolerance is critical to the reliability of stream processing systems, as they need to be operational 24x 7 while recovering from any failures with predetermined levels of data accuracy. The fault tolerance strategy in any stream processing system must establish the failures types that it tolerates, the guaranteed semantics for processed data and output, and the technique in which the system can store and recover before-failure state [1].

Majority of the evaluation benchmarks for stream processing systems are unable evaluate the system performance when subject to failures. Instead, such benchmarks employ evaluation techniques such as micro benchmarks and ad-hoc scripts that try to simulate failures that may not be representative of real world failures [2].

The aim of this thesis is to provide independent and verifiable experiments that evaluate the performance of Apache Flink and Spark when subjected to different failures scenarios. The fulfillment of the following research objectives will lead to the accomplishment of our aim.

- Design the experiments based on using an open source streaming benchmarks developed by Yahoo as a framework for performance comparison, and an introduction of open source fault injection tool called “AnarchyApe” to create failures in Hadoop YARN environment.
- Extend Yahoo Streaming Benchmarks with configurations that enable and disable the fault tolerance processing in both Apache Flink and Spark systems.
- Upgrade fault injection implementation in AnarchyApe tool to run on Hadoop environment with pre-defined failure scenarios.
- Conduct several experiments with different settings to evaluate the system performance upon failure and after recovery.

1.1 Outline

Chapter 2 introduces the main concepts of fault tolerance in real-time distributed systems and outlines their definition, structure, models and benchmarks. Chapter 3 discusses the related work was done to evaluate the recovery strategies of Apache Flink and Spark Streaming systems. Chapter 4 presents an overview about Apache Hadoop and YARN

infrastructure along with the fault tolerance of YARN components. Chapter 5 and 6 discusses the architecture of Apache Flink and Spark and their programming model and fault tolerance strategies. Chapter 7 describes the streaming benchmarks, fault injection tool and its failure scenarios and the experimental setup. Chapter 8 reports the results and evaluation of the conducted experiments and chapter 9 presents the conclusion.

2 Theoretical Foundation

This chapter reviews the main concepts of fault tolerance in real-time distributed systems and outlines their definition, structure, models and benchmarks.

2.1 Real-Time Distributed Systems

Real-time distributed system is a group of independent computer systems that communicate with each other within defined time constrains. The main goal of real-time systems is to operate correctly and accurately in a bounded time interval [3]. Real-time systems can be classified into hard real-time or soft real-time systems. In hard real-time systems, it is crucial to meet the timing constraints for its critical tasks even in the presence of a failure. On the other hand, Soft real-time systems can exceed the time limits in order to complete its essentials tasks. Hard real-time systems are typically mission critical embedded systems such as monitoring systems in healthcare applications while soft real-time systems are typically online business processes tracking and E-commerce applications [4]. In the business domain, real-time means "very fast" analytic processing especially at the decision layer where response time can be in milliseconds or microseconds [5]. Main source of real-time data can be social networks, banking transactions, network monitoring, and online marketing and advertising.

2.1.1 Fault Tolerance in Distributed Systems

Fault-tolerance is the ability of a system to continue operating properly and maintain its functionality despite the presence of faults [6]. Table 1 describes the three basic notations that define the inter-related problems that can occur in distributed systems [7]. Figure 1 denotes the relationship between the notations.

Problem	Description	Example
Fault	The cause of an error.	Bad transmission may cause damaged packages; A programmer may cause programming bugs.
Error	Part of a system's state that may lead to a failure or unexpected state.	Transmission errors lead to damaged packets at the receiving station.
Failure	Occurs when a component is not living up to its specifications.	A crashed program.

Table 1: Definitions of inter-related problems in distributed systems

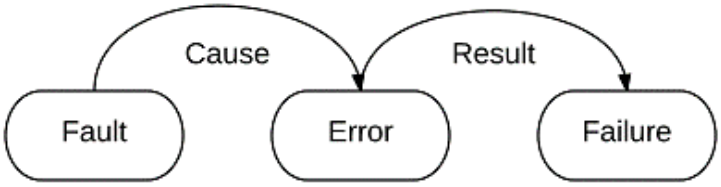

Figure 1: Relationship between fault, error and failure

2.1.2 Fault Types

Faults are categorized based on their longevity (occurrence with respect to time) and their impact on system resources during the processing time [8]. The three types of faults that can occur in real-time distributed systems and classified based on longevity are shown in Figure 2.

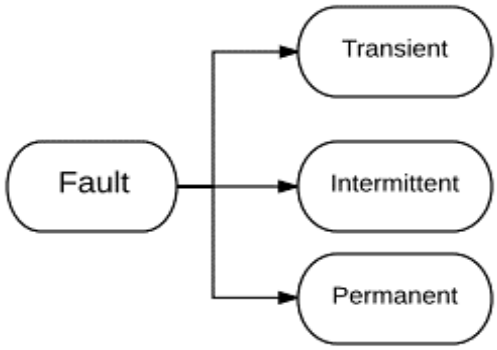

Figure 2: Different fault types in distributed systems

- Transient faults occur once and then disappear and typically, such faults are corrected by rolling back the system to its before-failure state.
- Intermittent faults occur occasionally and reoccur irregularly and typically, it is hard to predict the extent of the damage that they can cause.
- Permanent faults occur accidently and such faults persist until the replacement of the faulty component.

Failures that occur during execution time on system resources are classified as omission, timing, response, crash and arbitrary failures. Table 2 describes the behavior of system processors subjected to any of the different failure types [4] during their runtime. In real-time distributed systems, faults can occur at different hierarchy levels. At the physical level, faults can occur in hardware components such as CPU, memory, and storage. At the processor level, faults can occur due to operation systems crash and at the process level, faults can occur due to software bug or shortage of resources [4].

Failure Type	**System Behavior**
Fail-stop	Processor halts and remains in that state. Failure is detected by other processors.

Crash	Processor automatically halts and remain in that state. Other processors may not detect failure.
Omission	Processor does not respond to an input and fails to produce the expected output.
Timing	Processor responds with the correct value but outside the specified interval (either too soon, or too late).
Response	Processor responds with incorrect value.
Arbitrary / Byzantine	Processor exhibit arbitrary response with an incorrect result outside the specified interval.

Table 2: Description of the different types of failure in distributed systems

2.1.3 Failure Detection

The reliability of failure detector relies on the type of failure and the completeness and accuracy metrics of the failure detector [4]. Failure detectors in distributed systems can be implemented using several heartbeat mechanisms such as Centralized, Virtual ring based, All-to-all, and Heartbeat groups [4]. It is very critical for failure detectors in distributed systems to distinguish network failures from node failures in order to avoid poor performance of the whole system.

2.1.4 Dependability

Dependability defines the requirements for achieving scalable and maintainable distributed systems [9] and they are as follows:

- Availability
- Reliability
- Safety
- Maintainability

Availability and reliability of distributed systems assure immediate readiness of usage and continuity of service without any failure [10]. Safety guarantees the non-occurrence of catastrophic events that affect the system environment and maintainability evaluates the easiness of repairing the system after failure.

2.2 Batch Processing Systems

Batch and stream processing systems perform in a similar fashion. However, batch-processing systems processes data of a fixed size and the output usually occurs when whole data set is processed [11]. The drawback of batch processing is that it cannot provide low latency responses while processing a continuous stream of input data.

While Hadoop's MapReduce is a typical example of batch processing systems, some systems combine both stream and batch processing in a form of micro batch processing. Micro batch processing system, such as Apache Spark works on grouping the data into small batches and processes the small batches over a period based on a pre-defined batch interval.

2.3 Stream Processing Systems

Stream processing systems are systems that process and transform data streams from various data sources in order to analyze the data and respond to events as they occur and thereby enable better decision-making [9]. Stream Processing Systems are typically monitoring systems that apply data transformations such as counting, aggregation and filtering and then emit immediate outputs for further analysis.

2.3.1 Stream Processing Systems Requirements

Stream processing systems should be capable of the following: process a big volume of real-time data and produce fast, accurate and reliable results so that system users can react to changing business conditions in real-time. To meet such demands, it is essential to define system requirements for stream processing systems in order to evaluate and compare the available stream processing platforms.

The essential requirements for stream processing systems are high availability, scalability, minimum latency, and fault tolerance. The first three requirements ensure fast processing of data streams without any performance bottlenecks such as blockage or delay in the data stream while the last requirement guarantees efficient recovery without high recovery overheads. In case of failures, stream processing systems should continue to meet the real-time requirements and guarantee predictable and repeatable outcomes without getting adversely impacted by the recovery process or recovery overhead [12].

2.3.2 Stream Processing Architecture

Stream processing architecture provides a platform that integrates all of the data activities for an enterprise. The modern design of stream processing architecture uses a scale-up approach to build a universal stream-based architecture that integrates multiple systems in order to provides real-time insight from data streams [13]. There are mainly three components in stream processing architecture: producer, streaming systems, and consumers [13], as depicted in Figure 3. A producer is an application that has the capability to connect various data sources in order to collect and transform the data into a predefined format and publish it. Streaming systems receive the published data streams and deliver it to the consumers. Consumers are typically the stream processing engines that subscribe to data streams and analyze them by performing transformation and aggregation operations [13].

Figure 3: Overview of stream processing architecture

2.3.3 Stream Processing Model

Stream processing is a computation model for continuous processing and transformation of data streams [14]. The flow of the data streams is represented as a Directed Acyclic Graph (DAG) topology [15]. Figure 4 depicts the graphical representation of DAG topology where the operators are the vertexes and data streams are the edges.

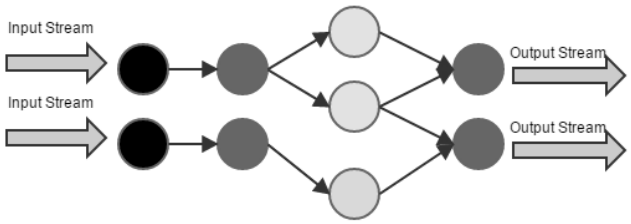

Figure 4: Stream processing graph

In stream processing, data Streams consist of events or records that consist of a key-value pair [15] and operators apply transformation on the input events and produce output stream for other operators.
Streaming operators are categorized into following groups based on their complexity [16]:

- Stateless operators such as filter operators that do not maintain a local state or time and order dependency.
- Deterministic stateful operators such as aggregate operators (sum, max, min and average) that maintain cumulative state but without the time dependency.
- Non-Deterministic stateful operators that maintain cumulative state and time and order dependency. These operators join multiple input streams to produce a new output stream.

2.3.4 Fault Tolerance in Stream Processing Systems

Large scale distributed systems like Hadoop are designed to run on low-cost commodity hardware where failure of its components is expected to happen at any given time. Stream processing on distributed systems should quickly recover from failures without adversely

affecting the processing capability and overall system performance. Hence, most of the failure recovery techniques work on finding the appropriate trade-off between their run-time overhead and recovery performance.

Fault Tolerance Strategies

The different strategies that steam processing systems rely on to improve the fault tolerance guarantee and achieve continuous processing are as follows.

Replication

In distributed systems, most of fault tolerance techniques depends on replicating the system components in order to recover from one of its copies in case of failure [17]. The two techniques for replication strategy are active replication and passive replication.

- Active replication: An expensive replication technique that replicates streaming operators on redundant components (replicas) and receive the same input in the same order with extra process cost for ordered multicast and output synchronization [16]. The advantage is that in case of a failure, a live replica immediately takes over the functions of failed components without any recovery time delay. The disadvantages are the added cost for duplication of storage and other hardware resources and overheads incurred in copying the redundant information.
  Passive replication: In this technique, replicas are on stand-by mode acting as a backup. Replicas takeover the role of active components only in the cases of failures. Passive replication results in overhead delays as all the input needs to be resubmitted in order to establish the before-failure system state [16]. In order to minimize the significant runtime overhead and recovery time, passive replication is used along with other failure recovery strategies such as checkpointing and upstream backup.

Upstream Backup (UB)

In upstream backup, processing node keeps a copy of its processed data in an output buffer (in form of events or tuples). This node acts as a backup for the data that are under processing at the downstream neighbors [16]. In case of any failures in the downstream node, the data stored in the upstream node is used to recover the failed operator state. Upstream backup guarantees that there is no data loss as the buffer data will be reprocessed. However, this technique requires extensive memory usage and results in a longer recovery time as the data needs to be retransmitted across the network for reprocessing.

Checkpoints

Checkpoints are used to save consistent state of running processes in persistent storage. In case of failures, the system recovers by restoring the state from the latest and completed

checkpoint. The two types of checkpointing techniques are [15]: uncoordinated and coordinated checkpointing. In uncoordinated checkpointing, running processes save their state independently into persistent storage. With this technique, recovery process can be very complicated and does not guarantee system consistency. In coordinated checkpointing, running processes jointly write to one global and stable state. This technique requires complicated global synchronization mechanism. A simple approach is to have a coordinator that communicate with the running processes in order to save to one global checkpoint. However, this approach can be very time consuming [15].

Recovery Types

The three types of recovery techniques that are used based on the requirements of streaming applications are [15].

- Precise recovery: Provides the strongest failure recovery guarantee as it completely masks the failure and hide its effect. This technique ensures that the output produced after a failure recovery exactly matches the output of an execution without failure.
- Rollback recovery: Provides weaker recovery guarantee than precise recovery. This recovery technique ensures that failures do not cause any data loss. However, the system may process duplicate data and produce imprecise output.
- Gap recovery: Provides the weakest failure recovery guarantee. A state loss is expected after failure as the system drops lost data and process only the currently available data.

Processing Semantics

Stream processing semantics guarantees completeness and correctness of the processed data and output results in a distributed system. A recent Facebook study [18] defined two categories of relevant semantics based on the nature of the streaming operators in use.

- State semantics: defines how many times can an input event be processed: at-least- once, at-most-once, or exactly-once.
- Output semantics: defines how many times can an output value appears in the result: at-least-once, at-most-once, or exactly-once.

State semantics specify the order of saving the in-memory state of operators and the position of the input event for stateful operators [18] as follows:

- At-most-once state semantics ([0,1]) in which the offset of input event is saved first to the checkpoint followed by the operator’s in-memory state. Upon failure recovery, the system drops lost input events in order to avoid processing duplicated events.

- At-least-once state semantics ([1, n]) in which operator's in-memory state is saved first followed by the offset of input event. Upon failure recovery, the system avoids data loss by replaying lost events but with the added cost of data duplication.
- Exactly-once state semantics ([1]) in which the offset of input event is saved with the operator's in-memory state at the same time. Upon failure recovery, the system avoids both data loss and redundant events processing but with increased latency.

On the other hand, output semantics, specifies the order of emitting the output value and saving both operator's in-memory state and the offset of processed event [18] as follows:

- At-least-once output semantics in which the system emits the output, and then saves in-memory state and the offset of processed event. Upon failure recovery, the system achieves high performance but with the added cost of data duplication.
- At-most-once output semantics in which the system saves in-memory state and the offset of input event before emitting output. Upon failure recovery, the generated output will be lost but without any data duplication.
- Exactly-once output semantics in which the system automatically saves in-memory state, the offset of input event and finally emits output in the same stage. Upon failure recovery, the system avoids both data loss and redundant events processing but with the overhead cost of increased latency.

Stream Processing Technologies

- **Apache Storm** is a stream processing framework that also provides batch processing of data streams. Apache Storm is widely-used by Yahoo, Twitter and Facebook for real-time analytics and machine learning use cases [19].
- **Apache Samza** is a stream processing framework that integrated with Apache Kafka and relays on Hadoop YARN for resource management and processing support [20].
- **Amazon Kinesis Service** is a paid Amazon Web Service (AWS) platform that loads and analyzes data streams from various resources [21]. Kinesis Service integrates with other AWS services and has an ability to scale to thousands of data sources.
- **Apache Spark** is a micro batch processing system with fault-tolerant streaming capability [22].
- **Apache Flink** is a native stream processing system that is known for its low latency and fault-tolerant stream processing [23].

Publish-subscribe Messaging System

Apache Kafka is a fault-tolerant scalable distributed messaging system that provide an API to consume events in real-time [24].

Coordination Service

Apache Zookeeper is an open-source centralized service that provides reliable coordination for distributed application. Zookeeper provide distributed configuration such as naming, configuration information, synchronization, and group services [25].

Persistent Storage

Redis is a non-relation and in-memory key-value store that supports different complex data structure and it is widely used as a database, cache and message broker. Redis has different levels of on-disk persistence that helps it to achieve high performance.

2.4 Stream Processing Benchmarks

With the increased complexity of modern streaming architecture, it is more difficult to evaluate the suitability of the various streaming systems. Therefore, stream processing benchmarks are designed to provide users insights about the configurations and performance of streaming systems. These benchmarks are designed based on predefined workload and evaluation metrics, which define performance measures for a specific streaming system. Benchmark experiments are conducted to run the benchmark workload, collect the performance measures, and finally provide an evaluation about the systems performance.

Usually, benchmark experiment consists of three stages: design, execution and analysis [26], which are described as follows:

- At the design stage, the design of benchmark experiment is defined based on the selected streaming systems, datasets, workload, performance measures and execution environment.
- At the execution stage**,** the experiments are executed to collect the performance measures.
- At analysis stage, the collected performance measures are analyzed.

2.4.1 Existing Stream Processing Benchmarks

Various stream processing benchmarks are publicly available. Most of these benchmarks focus on the latency analysis of Apache Flink, Spark and Storm. Moreover, the defined work load is quite simple and doesn’t present real-world use cases [27]. Recent benchmarks that evaluate stream processing systems are listed as follows:

- Yahoo Streaming Benchmarks (YSB) [28] are introduced to compare Apache Storm performance against Apache Flink and Spark. Chapter 7 presents more details about the benchmarks.
- Work done by S. Perera and A. Perera [29] provides reproducible benchmark experiment for Yahoo Streaming Benchmarks that evaluate the latency of Apache Flink and Spark in Amazon Elastic Computing Cloud (EC2). The evaluation concluded that Apache Flink was performing slightly better than Spark in many occasions.
- StreamBench [27] is an extensible stream processing benchmark with common API component and core set of workload that evaluate Apache Storm, Flink, and Spark. These systems were evaluated on three different workloads. The evaluation of the benchmark experiments reported that the throughput of Apache Flink and Spark was much better than Storm, but Apache Storm achieved a much lower median latency in most workloads [27].
- Work done by Córdova [30] provides latency analysis for Apache Spark and Storm on simple workload. The conducted experiment concluded that that Apache Storm was around 40% faster than Apache Spark.
- LinkedIn benchmark [31] is designed specifically for Apache Samza stream processing system and hence it is not a standard for other stream processing systems.

3 Related Work

Publicly available stream processing benchmarks provide a framework to compare the performance of the various stream processing systems including Apache Flink and Spark. Only a few of these benchmarks provide tools that help to experiment and analyze the dependability of stream processing systems [32]. In this chapter, we present benchmarks that provides a tool to evaluate the dependability of different processing systems.

3.1 MapReduce Benchmark Suite

MapReduce Benchmark Suite (MRBS) is considered as the first tool [32] for evaluating the dependability of Hadoop MapReduce system. The tool enables automatic fault load generation and injection in MapReduce environment at different rates and it provides a means to analyze the effectiveness of fault tolerance capability of Hadoop cluster in batch processing mode on Amazon EC2 private cloud [32]. The analysis showed that Hadoop cluster has 96% availability [32].

3.2 Netflix Monkeys: Chaos Monkey

Chaos Monkey is an Amazon Web Service (AWS) that runs in the cloud. It works on identifying the groups of systems and randomly terminates one of them [33]. The service assesses the impact of potential failures on normal operations during business hours. The service aims to find the system bottlenecks and assist the engineering team with the implement of mitigation measures by learning from failure outcomes and resolutions [33].

3.2.1 Evaluating Apache Spark Resiliency Using Chaos Monkey

Chaos Monkey ran resiliency assessment on Apache Spark application hosted on an AWS cloud cluster with the following configuration: Apache Spark V1.2.0, Apache Kafka V0.8.8, and Zookeeper V3.4.5 [34]. Chaos Monkey assessment examined the resilience of various Apache Spark components and revealed performance related issues of Apache Spark receivers that led to data loss [34] , as presented in Figure 5.

Post assessment, Apache Kafka team reported serious performance issues with the high level APIs [35] [36]. Spark V1.3.0 aimed to fix the identified issue and introduced a new low level Kafka API as part of Apache Spark packages [36].

Figure 5: Failure analysis in Apache Spark using Chaos Monkey (Source: [18])

3.2.2 Evaluation Apache Flink Resiliency Using Chaos Monkey

The team of Data Artisans (Company founded by the original creators of Apache Flink) assessed Apache Flink security detection application with YARN Chaos Monkey (Chaos Monkey-like logic to kill YARN containers) [37] on 30-node cluster. The team provisioned for enough spare worker nodes for the system to continue with full computational resources after container failure injection [38]. During the assessment, Apache Flink was in alignment with the Kafka generated events and after each failure, Apache Flink managed to catchup and validate the events sequence [38], as presented in Figure 6.

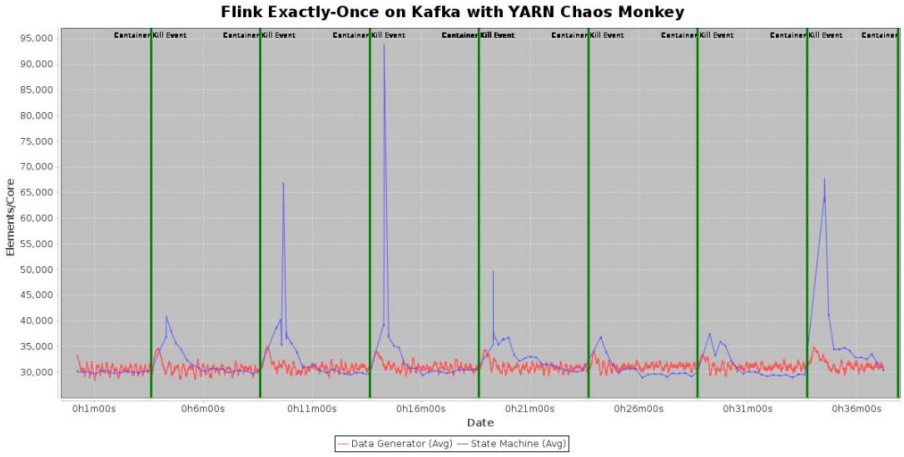

Figure 6: Failure analysis in Apache Flink using Chaos Monkey (Source: [38])

4 Apache Hadoop YARN

Apache Hadoop is a large-scale and distributed storage architecture that provides a platform for data processing on clusters that are built from inexpensive commodity hardware [39]. A cluster comprises of a group of node machines, which are of the following types: master nodes and slave nodes. Hadoop has the ability to reliably and effectively store, process and manage massive amount of data in any format including semi-structured and unstructured formats.

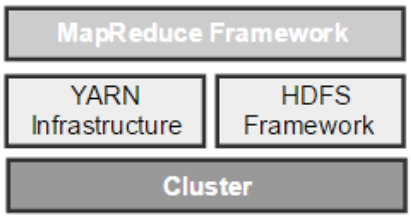

Figure 7: Basic illustration of Hadoop architecture

Hadoop infrastructure has four components inside its runtime environment, as depicted in Figure 7 and defined as follows:

- The cluster that is the hardware part in Hadoop infrastructure.
- YARN Infrastructure is the resource management layer required for the application execution.
- HDFS framework that provides persistent, reliable and distributed storage architecture.
- MapReduce framework is a programming model that provides an implementation methodology for processing large datasets across clusters.

In this chapter we present an overview of YARN Infrastructure and the availability model of its main components.

4.1 YARN Architecture

YARN (Yet Another Resource Negotiator) infrastructure provides computational resources such as CPUs and memory that are needed for application executions [40]. Figure 8 presents the YARN architecture and its components.

The three main components in YARN's architecture are the client, Resource Manager RM (the master), and Node Manager NM (the slave) [40]. To launch an application in cluster mode, the client submits the application to Resource Manager, and then the Resource Manager allocates computational resources in containers hosted by respective Node Managers. The Node Managers then launch the allocated containers to execute application's tasks [40].

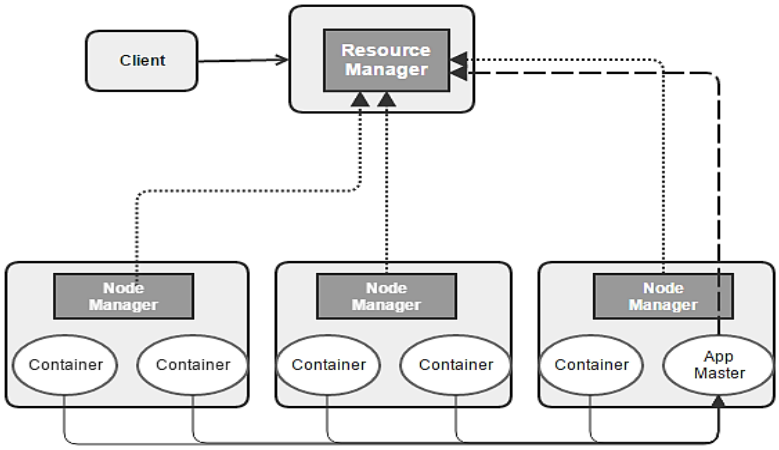

Figure 8: YARN architecture

4.2 Fault Tolerance and Availability

The responsibility of fault tolerance in YARN's architecture is distributed between the Resource Manager and Application Master [40]. Resource Manager is the cluster scheduler that controls the cluster utilization. Application Master is responsible for negotiating resources allocation with the Resource Manager and coordinating the running processes on the Node Managers' container. Both components need to assure the high availability and reliability of the cluster, particularly for streaming services.

Resource Manager

In active/standby architecture, Resource Manager's high availability (HA) feature is activated so that one of multiple standby Resource Managers can take over in case of any failures. The transition from the active to standby state can be done either manually or automatically. In the automatic failover option, Zookeeper controller uses Active/Standby Elector in order to decide which Resource Manager should be active.
Resource Manager has stateless restart where any in-progress work will be lost upon restarting the Application Master. However as of Hadoop V2.6.0, the problem has been resolved by restoring the entire running state of YARN cluster so that the applications can resynchronize with new Resource Manager and resume from the up-to-date state [41]

Application Master

The process of Application Master is run on the cluster's container just like those of other running applications. Upon the failure of Application Master, Resource Manager automatically detects the failure and then performs several attempts, if needed, to start a new Application Master's process in another container. Therefore, Application Master is not considered as a single point of failure and its process failure does not affect the cluster availability. However, since the Resource Manager kills the dead Application Master's

allocated containers, the work in those containers are lost [42]. Restoring the lost work left as responsibility of Application Master without any support from YARN platform. However, the recent work-in-progress aims to implement container-preserving restart of the Application Master that allows the Application Master to restart without killing its allocated containers and then resynchronize those containers with the newly launched Application Master [43].

Node Managers

Each cluster has serval Node Managers that launch and track applications' tasks. The Node Manager, which is the worker daemon in the YARN infrastructure registers itself at the Resource Manager by sending regular heartbeat responses to confirm its availability. Launched applications utilize computational resources that are allocated by Node Manager such as memory and CPU [44]. Figure 9 depicts the nature of computational resources in a worker node.

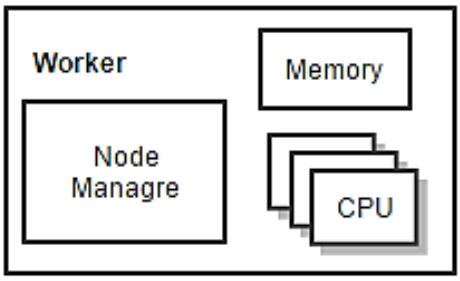

Figure 9: Overview of worker node structure

Resource Manager will automatically detect any failure of Node Manager after the timeout interval in which Node Manager stops sending heartbeats response. Default timeout interval in YARN configuration is ten minutes. Upon failure detection, Resource Manager removes failed Node Manager from the pool of available Node Managers, kills all containers running on the respective node, and then reports the node failure to Application Master. The responsibility of recovering lost work being done on the killed containers is left for the Application Master to react upon [44]. If failure recurrence of a Node Manager exceeds a certain threshold, it would be blacklisted and never used again by the Application Master [44].

Containers

Task's processes that are running on the container are the actual work that an application is expected to execute and finish. YARN does not handle container failures and it is left for the application to react upon. Resource Manager reports a containers' failure to the respective Application Master, which in turn notifies the application to reschedule and execute the lost work that was running on dead containers [44].

5 Apache Spark

Apache Spark is a batch processing system that has a streaming module to support fault-tolerant stream processing with a simple API. The Apache Spark model integrates with other Spark components and data sources.

5.1 Architecture

Apache Spark uses a master-slave architecture with a central coordinator (driver) that communicates with the distributed workers (executers) as shown in Figure 10. Each of Spark components run in their own Java processes on cluster nodes. Driver process runs "StreamingContext" which uses "SparkContext" to schedule and launch jobs on the executers across the cluster. Executers, on the other hand, are in charge of running individual tasks in a given job. Once the executer completes the task, it sends the result back to the driver.

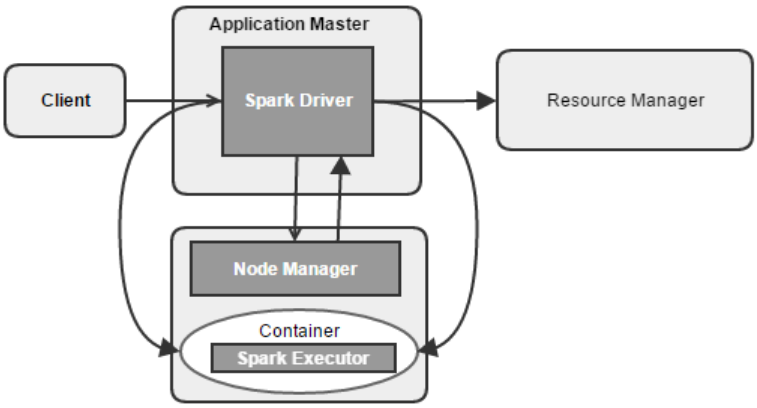

Figure 10 : Apache Spark architecture

5.2 Programming Model

Incoming data streams are represented as Discretized Streams (D-Streams) which are a series of stateless, deterministic batches of data (called micro batches) of predefined time intervals. Apache Spark treats each batch of data as Resilient Distributed Datasets (RDDs). RDDs is the main data structure in Apache Spark and it is an immutable distributed collection of objects stored in memory [45]. D-Streams can be processed using stateful operators (like mapWithState and updateStateByKey), windowed operators and other stateless transformation operators. Giving the volatile nature of RDDS, lineage is used to keep track of transformation information and re-compute lost RDDs in case of failures [46]. In order to avoid long and complex lineage, asynchronous checkpoints are taken periodically to prevent long lineages [45].

5.3 Checkpointing and Parallel Recovery

Apache Spark uses a checkpointing technique to save the state of RDDs periodically into persistent storage. In case of node failures, it re-computes in parallel the lost partitions on different active nodes. Checkpointing technique is also used in stateful operations to store the cumulative state of the dataflow. There are two types of checkpoints [47].

- **Metadata checkpoints** are used to save application configuration, define D-Streams operations and list the incomplete batches for data that has not been processed yet (just the metadata) [47].
- **Data checkpoints** are used to save intermediate RDDs into persistent storage if RDDs are used for some stateful operations. Data checkpoint is used to cut off the long lineage.

In case of node failure, Spark automatically re-execute in parallel the failed tasks on other active nodes as shown in Figure 11.

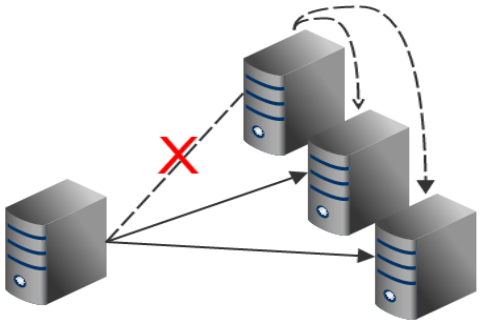

Figure 11: Parallel recovery in Apache Spark

Based on the Databricks evaluation (company founded by the creators of Apache Spark), asynchronous checkpoint technique has faster recovery than replication or upstream back techniques without the added cost of nodes replication [48]. Also this technique can mask the impact of straggler (slow nodes) on the progress of the system [48]. The evaluation outcome showed that for parallel recovery, the recovery time improves with more frequent checkpointing and addition of more computational resources [48] as shown in Figure 12.

Figure 12: Failure evaluation of Apache Spark done by Databricks (Source: [48])

With continuous stream processing, lineage and task, size gradually increases and affects the system performance. However, frequent checkpointing with all of the write requests to persistent storage will eventually slow down the system. Generally, checkpoint interval should be about 5 to 10 times of batch interval time [45].

5.4 Fault Tolerance and Reliability

Apache Spark has a support for recovering from failure of driver and worker nodes. However, guarantee against data loss doesn't come with the default configuration. Apache Spark V1.2 provides zero data loss guarantee [49] under the following configuration:

- Reliable data source such as Apache Kafka.
- Reliable receiver that applies in memory transformation on received data.
- Metadata checkpointing is enabled by the application driver in order to reconstruct the application.
- Write ahead log (WAL) is enabled and it synchronously saves the received data logs into persistent storage.

With the above configuration, the system guarantee at-least-once semantics but with a high overhead cost of enabling Write ahead log (WAL).

Direct Kafka D-Streams Approach

Apache Spark V1.3 introduced new Direct Kafka approach that ensure exactly-once processing semantics without the need to reprocess any duplicate data from WAL. Write ahead log (WAL) is no longer required as the data can be re-played directly from Kafka.

With Direct Kafka approach, Spark driver queries latest offsets from Apache Kafka and calculates the offset ranges for the next processing batch. Spark driver then directs the executors to launch the jobs and consume data directly from Kafka using the predetermined offset ranges [36].

For better reliability, the calculated offsets are tracked by Spark driver and saved in the checkpoints along with the application meta data. The disadvantage of direct approach is that Apache Spark does not update the offsets in Zookeeper. Hence, the progress of streaming jobs cannot be monitored using external monitoring interface.

External Shuffle Service on YARN

YARN Shuffle Service is an external shuffle service for Apache Spark on YARN. The external shuffle process runs independently on each node in the cluster and is separate from Spark application and its executors. Spark executors directly fetch the shuffle file output stream from external shuffle service. Thus, any shuffle state written by an executor may continue to be served beyond the executor's lifetime. Figure 13 depicts the communication channel upon executor failure.

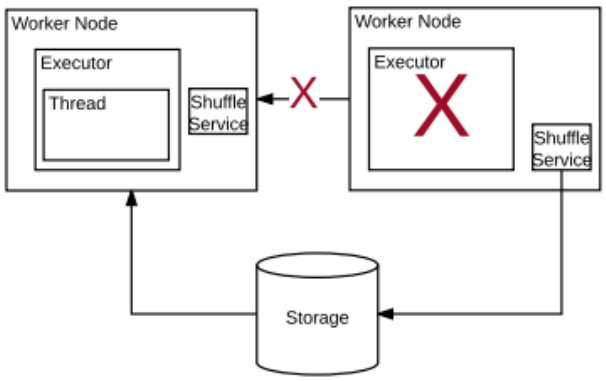

Figure 13: External shuffle services in Apache Spark

Enabling Backpressure

When batch processing time is larger than batch interval, Apache Spark will not able to read data faster than it arrives and will not keep up with data source throughput rate [50]. Enabling backpressure guarantees that the application receives data as fast as it can process and thereby by ensure system stability [50].

6 Apache Flink

Apache Flink has a native support for fault-tolerant stream processing. It also provides a batch processing model on top of its streaming engine as a special case of stream processing [27]. Apache Flink is known for its low latency and high performance [51].

6.1 Architecture

Apache Flink master-slave architecture has three main components: Client, Job Manager, and Task Manager. Client processes compile and optimize the submitted application, construct the job graph, and then pass it to Job Manager. The Job Manager that is the coordinator of the system creates execution graph, assigns the tasks to Task Managers, and manages execution states. Task Managers are the workers that has one or more task slots, each task slot executes the parallel instances of running operations. Each instance of an operation (pipeline of parallel tasks) is running on a separate task slot in the Task Manager. One task slot may run several tasks from different operators based on the scheduled jobs. Figure 14 illustrates Apache Flink architecture in which the Client submits a job to the Job Manager, which schedules and processes the job using the Task Manager.

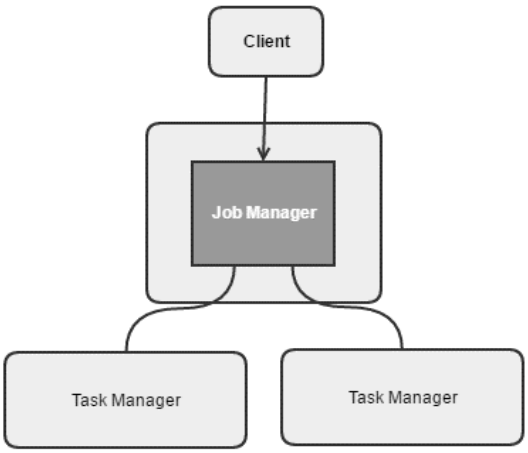

Figure 14: Apache Flink architecture

Apache Flink can run either in local or cluster execution mode. Local execution mode runs all of the Apache Flink processes on the same machine. This mode is very useful for testing and debugging. Cluster execution mode, on the other hand, uses YARN client to submit the job remotely to the Hadoop cluster. Figure 15 depicts YARN interaction with Apache Flink components under cluster execution mode [52].

In Cluster execution mode, YARN client uploads Apache Flink program and all its configurations into HDFS storage. Next, YARN client connects to Resource Manager and requests computational resources (Node Managers' containers). The client requests

YARN container to first start the Application Master and run Job Manager in the same container. Upon start, the Application Master allocates containers and prepare them to start Apache Flink Task Managers that can accept and start Apache Flink jobs [52].

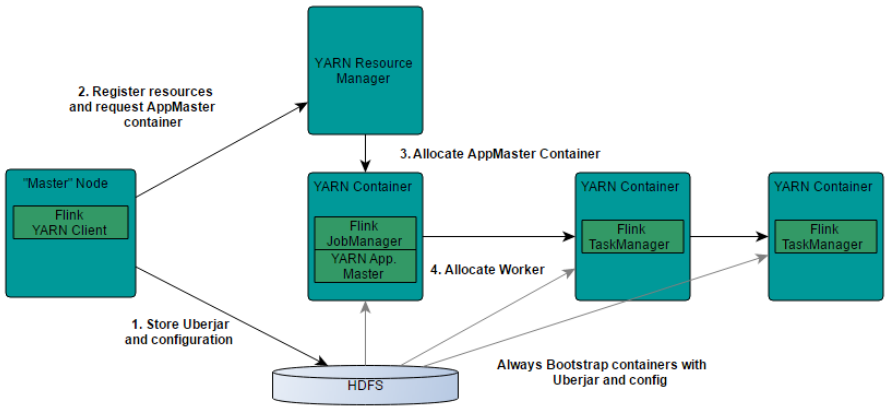

Figure 15: Apache Flink architecture under Hadoop YARN execution mode (Source: [52])

6.2 Programming Model

Apache Flink dataflow engine compile Flink programs into a dataflow graph (directed acyclic graph (DAG)) that is executed in distributed pipelined fashion [51]. Pipelined execution model is more complicated compared to other data processing systems such as Apache Spark. In such model, data streams flow through the system freely without unnecessary delays or intermediate materialization of large intermediate results [53]. To unify streaming and batch processing, Apache Flink represents input data using two different APIs: DataSet API for batch and bounded streams processing and DataStream API for unbounded stream processing [53].

By default, pipelined operators in the execution model are stateful operators but can become stateless in a special case of the execution logic [51]. Each operator produces intermediate results that are ready for other operators to consume.

Intermediate data is exchanged in parallel between running operators using intermediate buffer pools. A buffer is sent to consumer operator as soon as it is full or when a timeout condition is reached [51].

Both the buffer size and timeout value affect the system latency and throughput rate. Setting the buffer size (in kilobytes) to a high value increases the throughput rate. On the other hand, low latency can be achieved by setting the buffer timeout to a low value (in milliseconds).

6.3 Checkpointing

Apache Flink captures checkpoints by drawing periodical snapshots of the system topology and data flow state without stopping the stream processing operations. The mechanism for drawing the snapshots is called “Lightweight Asynchronous Snapshots for Distributed Data flows” [54], which is an implementation of Chandy-Lamport algorithm for asynchronous distributed snapshots [55].

Checkpoint Barriers and State Management

Different types of control events are injected into data streams at the operator level. One of the special type of control events is the checkpoint barrier that coordinate checkpoints by dividing the stream events into pre-checkpoint events and post-checkpoint events [51]. These checkpoint barriers are injected into the dataflow in preserved order at the stream source and each barrier carries a reference of its snapshot [56].

Different snapshots may happen concurrently by injecting multiple barriers in the data stream at the same time [56]. A checkpoint is considered as completed once its barrier passes through the entire data flow topology and all sinks operators (last operators at the end of a dataflow graph) acknowledge this snapshot to the checkpoint coordinator [56]. Figure 16 depicts the coordination among Apache Flink components to manage state snapshots.

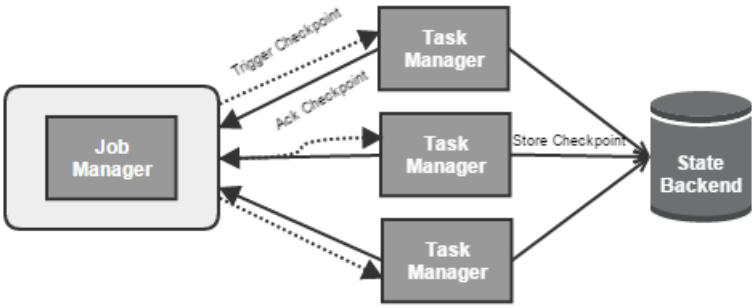

Figure 16: Fault-tolerant state management in Apache Flink

When an operator receives the snapshot barriers from all of its input streams, it automatically writes state snapshot to the Job Manager’s memory for smaller states or persistent storage such as HDFS for the larger states [57]. The operator state can either be buffered events with determined time interval (time window) or user-defined key-value state [57]. The snapshot contains a reference to the stored state and the last offset of processed event.

State Semantics

By default, operators with one input stream give exactly-once semantics [56]. However, operators with multiple input streams such as join operators have to ensure that snapshot barriers are received from all of input streams in order to guarantee exactly-once semantics. With streams alignment, the operator's state can be saved into checkpoint only when the operator finishes processing all records that belong to that state.

Failure Recovery

Upon failure, Apache Flink chooses the last completed checkpoint to restore the before-failure state of the dataflow. Based on the data source reliability, the source replays the input data stream starting from the events offset that was saved in the restored snapshot [56].

Savepoints

Savepoints is a feature that enables the system user to manually trigger a checkpoint and save a copy of the current state of the running job. This feature keeps a reference to a completed checkpoint that only expire upon user request unlike regular checkpoint that overwrite itself once new checkpoint is completed. Savepoints can be used for restoring the system from failure or after a maintenance or an application code upgrade in which streaming job needs to restart from defined time interval but without the need to replay the data stream from the beginning [58].

Enabling Backpressure

When streaming system receives data at a higher rate than it can normally processes, incoming data generates accumulated backpressure. Backpressure can also happen while checkpoint is being created. Apache Flink handles backpressure implicitly within its native streaming architecture and with the help of reliable data source like Apache Kafka [59].

7 Methodology

We conducted benchmark experiment to evaluate the performance of Apache Flink and Spark when subjected to failures. The benchmark experiment was designed based on using an open source streaming benchmark developed by Yahoo as a framework for performance comparison, and an introduction of open source fault injection tool called "AnarchyApe" to create failure in Hadoop YARN environment.

This chapter illustrates the architecture of the benchmark experiment and introduces more details about the framework, failure scenarios and evaluate criteria.

7.1 Yahoo Streaming Benchmarks

Yahoo Streaming Benchmarks were developed to compare the performance of Apache Flink, Spark and Storm streaming systems [28]. The benchmarks simulate a simple advertising application in which advertisement data for different campaigns is generated from Kafka, and then some common streaming operations are performed on data streams to check how many times an ad campaign was seen in a window. Finally, the related advertisements per campaign are stored into Redis [60].
The performed operations on ad campaign data streams are defined as follows [61]:

- Read ads from Kafka
- De-serialize JSON string
- Filter unnecessary ads
- Perform projection to remove unrelated fields
- Join ad id with campaign id from Redis
- Window count per campaign and output to Redis

Yahoo streaming benchmarks reported that Spark with its micro-batching design has poor performance compared to Apache Storm and Flink. On the other hand, Apache Storm and Flink both have very similar performance [61]. In response to Yahoo report, Apache Flink team extended Yahoo benchmark for tuning Apache Flink performance. With a cluster setup of one Gigabit Ethernet interconnection between interconnected nodes, Apache Flink achieved a throughput of 15 million events/second [62].

7.2 Fault Injection Tool

AnarchyApe is an open-source project developed by Yahoo [63]. The tool has about twelve failure commands that can be executed to inject faults in Hadoop 1.0 environment [64]. We extended AnarchyApe tool in order to inject failures in Hadoop 2.0 YARN environment. Failure scenarios are executed on master node using shell script where we specify the follows parameters:

- Fault type.
- Failure rate that specifies the frequency with which the system components and their processes fail.
- Failure duration.
- Maximum number of randomly chosen failed components.
- List of remote nodes IP address.

```
     java -jar ape.jar -remote cluster-ip-list.xml -k -F -M -T

Where cluster-IP-list is a list of worker nodes, the -k is a “Kill Node”
command, the -F specifies the failure rates, the -M specifies the maximum
number of failed nodes, and the -T specifies failure interval
```

Figure 17: Sample line in failure scenario script

A sample line to call fault injection script is shown in Figure 17. The used failure injection commands are killing, suspending and restarting the running processes on Hadoop YARN worker nodes. In real-world scenarios, node failure can occur due to hardware problems that cause a complete loss of data or failure on operating system level that lead to a loss system state. Table 3 shows the failure commands that we used for our different failure scenarios.

Command	Description	Note
Kill-node	This command kills a node process on the host.	Node process can be either: Data Node, Resource Manager, Name Node, Application Master, Node Manager or YARN Container.
Suspend-node	This command suspends a running Hadoop process.	
Continue-node	This command will start a Hadoop process that has been suspended.	

Table 3: Description of failure commands used in AnarchyApe fault injection tool

Table 4 describes the affected Hadoop YARN components by AnarchyApe and the expected behavior upon recovery.

Components	Failure mode	Behavior Upon Recovery	Tested by AnarchyApe
Resource Manager Failure	Single point of failure.	Recovered RM runs new instant of Application Master.	No

Application Master Failure	Runs on one of cluster's node. Single point of failure.	RM restart failed AM without restoring its state. Running tasks are killed and re-run.	No
Node Manager Failure		RM kills all containers running on the node.	yes
Container Failure	Failure handling left to the framework. AM sends regular heartbeat notifications.	Framework request new container from RM.	yes

Table 4: Description of failure modes for different Hadoop YARN components

7.3 Failure Scenarios

We focused on two cases that simulate real-time fail-stop failure scenarios:

- Failure caused by transient fault such as unstable network where a node disconnects and reconnects frequently within short time.
- Failure caused by permanent fault where a node disconnects frequently till resource manager fires timeout and the node fails permanently.

We used the following parameters in order to evaluate different failure scenarios:

- Failure interval: defines the failure duration of randomly chosen node and it is adjusted using AnarchyApe tool.
- Max Number of Nodes: specifies number of nodes that will fail in the same interval and it is adjusted using AnarchyApe tool.
- Failure time out: defines the wait time till the processes on the failed nodes restarted and adjusted in Hadoop YARN environment.

7.4 Evaluation Metrics

Four key metrics were investigated and recorded for the experiments. Table 5 shows these metrics along with the method used record them.

Metric	**Description**	**Method**
Throughput	The count of processed events over time interval.	Log file that tracks the counts of events for different campaigns over defined time interval.
Latency	Time in milliseconds for end-to-end processing time starting from when an event	Log file that tracks the end-to-end latency (in milliseconds).

	was emitted to Kafka until it is written into Redis.	
Overheads costs	Usage of CPU and Memory.	OS profiling tools (DSTAT) on individual nodes.
Scale	Data generation rate.	2000 -50,000 events/sec.

Table 5: Experiment Metrics

Fixed Parameters:

We designed the fixed experiment plan with the following parameter settings.

- Cluster settings and number of nodes (Environment settings): Maximum of 9 nodes
- Data scale: Maximum of 50,000 events/second.
- Maximum number of failed nodes during the experiment runtime: between 1 and 3.
- Experiment duration: 15 - 90 minutes.
- Failure timeout threshold: Adjusted in YARN.
- Fault injection start time: failure starts after 15 minute (900 Second) of experiment runtime.

7.5 Infrastructure

The experiments were run on Hadoop cluster (V 2.4) that consisted of 9 nodes where one node served as the master that runs the Name Node and Resource Manager processes and the other eight nodes configured as worker nodes that runs Node Manager processes. Table 6 defines the hardware specification of the used cluster.

OS	Fedora 23
CPU	IBM,8231-E2B model with speed of 3.72 GHz.
Memory	50 GB memory on master node and 60 GB of each worker node.
Software	Java V1.8.0, Apache Spark V1.6.0, Apache Flink V1.0.0, Kafka V0.8.2.1 and Redis V3.0.5.

Table 6: Description of cluster specification

8 Experiment

After demonstrating the design of the benchmark experiments in the previous chapter, we will present the execution and analysis of these experiments in this chapter.

The experiments are categorized into three groups: baseline performance without fault tolerance processing, performance with fault tolerance processing, and finally performance under failure for Apache Flink and Spark.

8.1 Benchmark Experiment for Apache Spark

The experiments used the following pre-defined measures for evaluation: latency, throughput, and CPU and memory utilization. These measures were captured by running the experiments with different runtime duration and data generation rate, which is the number of input events emitted from Kafka per second.

8.1.1 Benchmarking Baseline Performance

We based our stress factor on different events generation rate: starting from 2000 events/sec up to 50,000 events/sec. Figure 18 reflects end-to-end latency for different throughput rates. For smaller throughput rates, Apache Spark started with a small latency of 10 seconds for 2000 events/sec, and for higher throughout rates, latency gradually increased to 45 seconds for 30,000 events/sec. The gradual increase in throughput rates generated backpressure that affected the Spark performance and end-to-end latency.

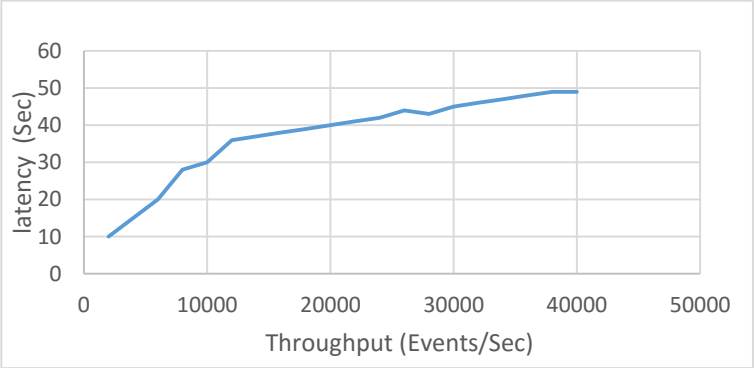

Figure 18: Baseline performance of Apache Spark for different throughput rates

We tracked system level metrics independently and captured CPU and memory utilization for baseline performance. It was observed that Apache Kafka already started computational resources utilization during the data generation phase. The peak utilization was reached during the transformation phase in which input data streams are processed in Apache Spark. For example, Apache Spark consumed a maximum of 16 GB of memory

and 38% of CPU capacity for throughput rate of 30,000 events/sec as shown in Figure 19 and Figure 20.

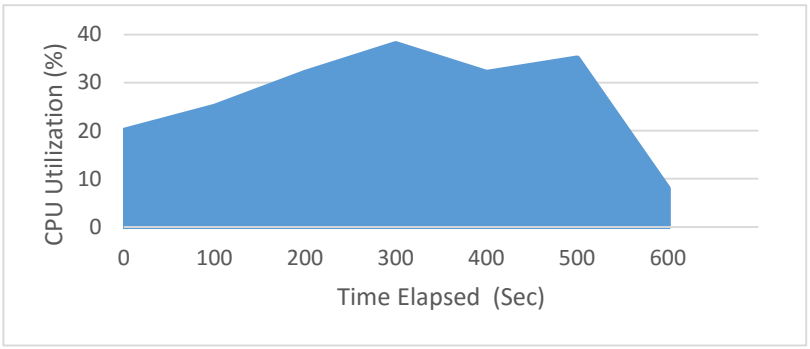

Figure 19: Baseline CPU utilization of Apache Spark for throughput rate of 30,000 events/sec

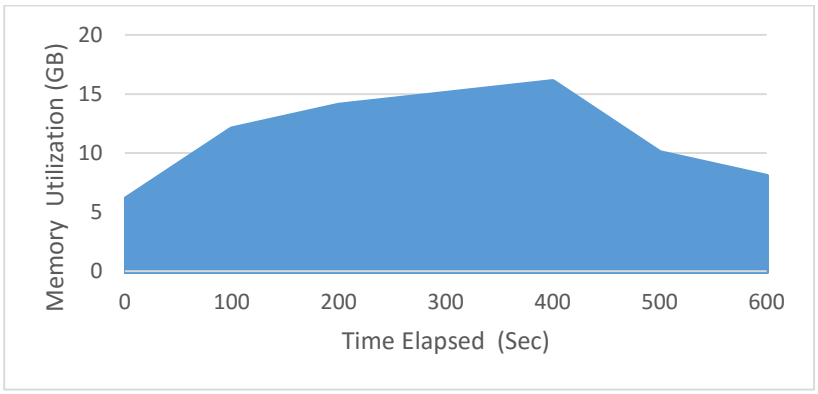

Figure 20: Baseline memory utilization of Apache Spark for throughput rate of 30,000 events/sec

Optimal Batch Interval

In Apache Spark, batch interval is considered as a performance bottleneck. It is important to ensure that the batch processing time is shorter than the batch interval in order to avoid backpressure. With backpressure, the system continuously falls behind incoming data streams and eventually reaches unsustainable level of performance.

We ran the benchmark with different batch intervals of 1 to 10 second duration. Using Apache Spark web interface, we monitored and captured the output and end-to-end delay for different batch intervals as shown in Figure 21.

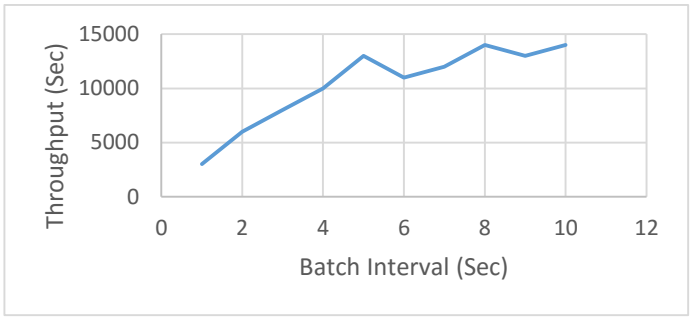

Figure 21: Experimental performance of batch interval in Apache Spark

8.1.2 Benchmarking Fault Tolerance Performance

The Yahoo Streaming benchmarks use direct Kafka stream approach that integrates Kafka receivers with Spark driver in which data is fetched directly from Kafka using Kafka API. Spark driver decides on the offset ranges for the batches and then launches the jobs using the determined offset ranges without updating it to Zookeeper. Direct Kafka API cannot handle any changes to Spark driver code.

We ran experiments for different throughput rates, similar to baseline experiment but with checkpointing enabled. The duration of each experiment was between 15 to 90 minutes. Figure 22 shows comparison between the Baseline and fault tolerance performance. Figure 23 reflects memory utilization with a peak consumption of 26 GB for a throughput rate of 30,000 events/sec.

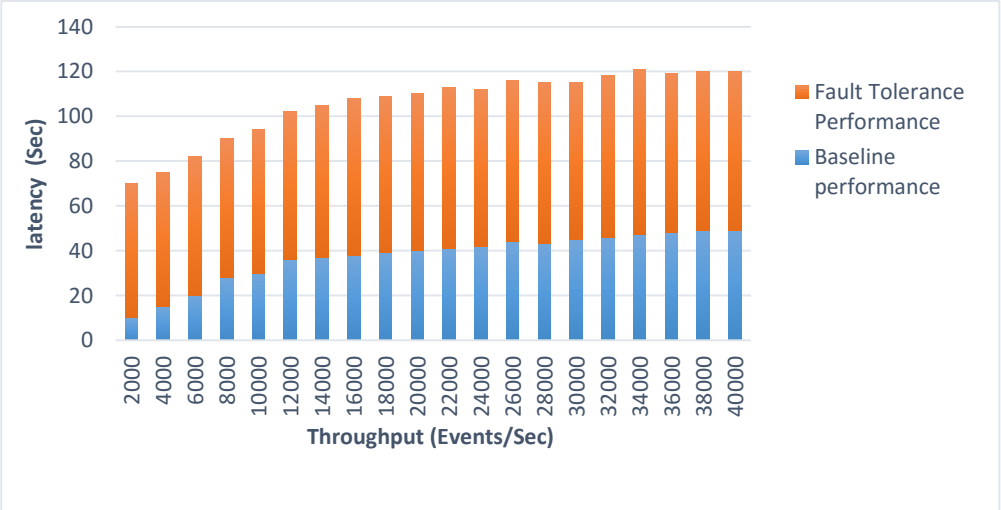

Figure 22: Comparison between baseline and fault tolerance performance of Apache Spark

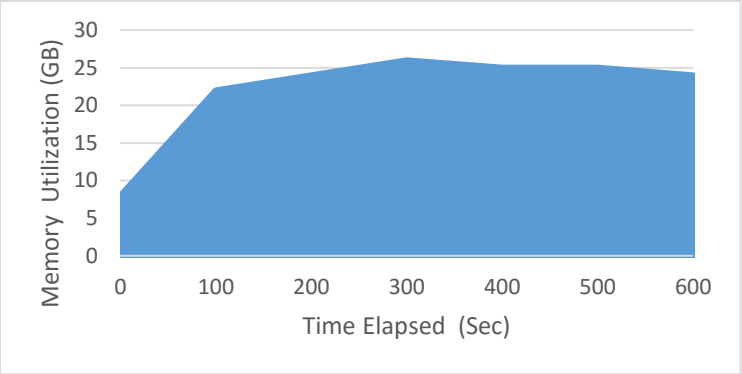

Figure 23: Memory utilization for fault-tolerant processing in Apache Spark for throughput rate of 30,000 events/sec

We observed that Apache Spark requires more time to prepare and store checkpoints into HDFS. Such overhead cost led to a dramatic performance degradation coupled with high scheduling delays. Figure 24 shows increased scheduling delay for throughput rate of 30,000 events/sec due to the overhead cost of preparing and storing checkpoints.

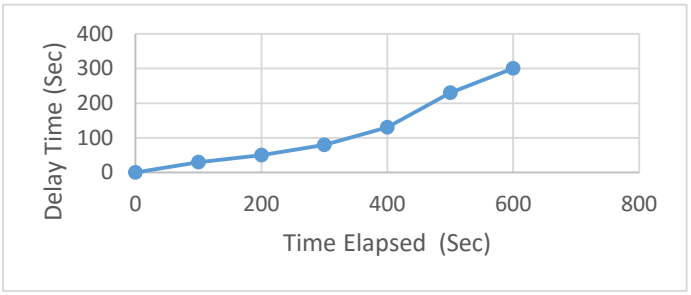

Figure 24: Scheduling delay of fault-tolerant processing for throughput rate of 30,000 events/sec in Apache Spark

Out of Memory Failure

Running the experiment for a duration that exceeded 90 minutes caused the system to eventually crash even before injecting any faults to the system. We observed an increase in delay that caused Spark to queue up the pending tasks.

With further investigation and tracking of output logs, we found that the implementation of data checkpointing technique is not fully optimized. For every checkpoint, data is computed twice, before triggering the checkpoint and while storing the checkpoint. Resolving this issue is still a work in progress under the future Spark releases [65].

8.1.3 Benchmarking Fault Tolerance Performance under Failure

Initially we ran a small scale experiment that analyzed the system behavior under failure. We manually killed Spark processes (driver and executers processes) and then re-started the application. We observed that Spark processed before-failure unprocessed events and updated the state accordingly with exactly once guarantee without loss of data or duplication. Through the aforementioned experiments, we found that Spark application can restore its state under the following constraints:

- Output operation must be idempotent (unchanged output) in order to saves results and offsets to data checkpoints [66].
- The class structure of driver code must not be changed, or else the application will reset the cumulative state.
- The saved offset must exist in the logs of Apache Kafka. The loss of these logs upon failure will stop the application recovery.

Next, we conducted benchmark experiments using the fault injection tool with the failure scenarios as follows.

Application Master Failure with a Data Generation Rate of 30,000 Events/Sec

We ran Spark application using dynamic resources allocation feature. The Node Manager fault was injected into the master node (Single point of failure) to kill the Application Master processes including Spark driver process. YARN recognizes the failure of Application Master after a certain timeout; Resource Manager waits for 10 minutes before considering the Node Manager as failed. After failure detection, YARN restarts the Application Master and Spark driver respectively; Apache Spark provides automatic restart for launching the driver in cluster mode.

The failure of Application Master forces Spark application to shut down and kill both driver and executers processes. After failure detection and automatic restart of Spark driver, the application starts to catch-up with unprocessed messages that are still coming from Kafka data source. With the driver catch-up effect, the application throughput rate increased dramatically after recovery (Maximum throughput rate after second failure was 120,000 events/sec) as shown in Figure 25.

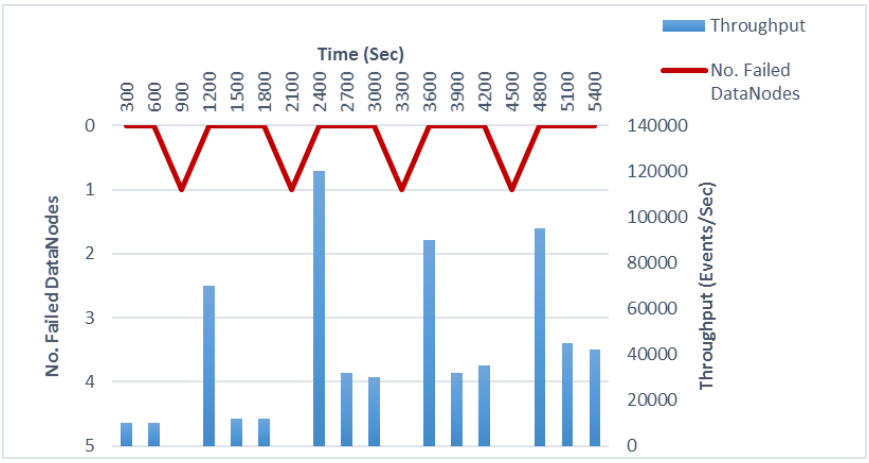

Figure 25: Performance of Apache Spark driver after recovery.

Executer Failure with a Data Generation Rate of 30,000 Events/Sec (Process Level Failure)

We injected fault to kill executors on different worker nodes that were chosen randomly on scale of 1 to 3. With the help of External Shuffle Service, running executors managed to get shuffle data, make progress and maintain reliable latency very close to fault tolerance performance without failure (latency time between 65-68 second) as shown in Figure 26.

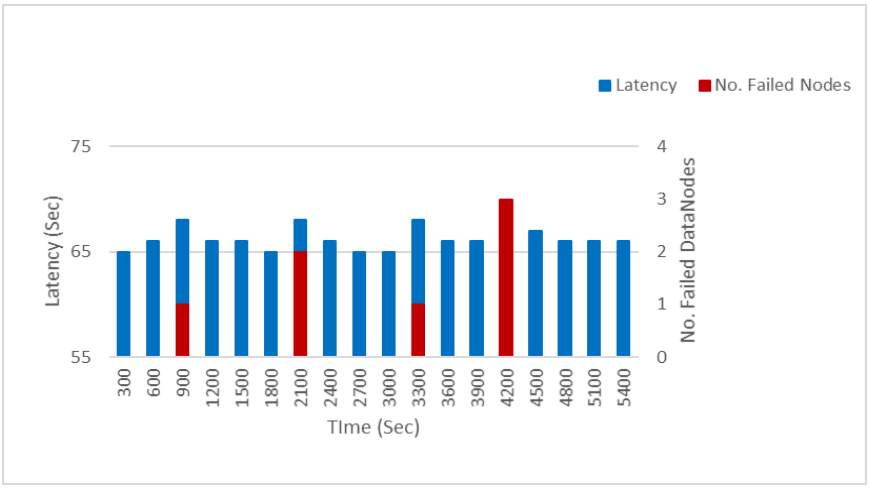

Figure 26: Performance of Apache Spark under process failure

Node Manager Failure with a Data Generation Rate of 30,000 Events/Sec

We injected fault Node Manager on different worker nodes that were chosen randomly on scale of 1 to 3. With the help of reliable direct Kafka API, Apache Spark maintained operator's state without any data loss upon the failure of the Node Manager. However, the overall latency increased from 65 seconds to 102 seconds as shown in Figure 27.

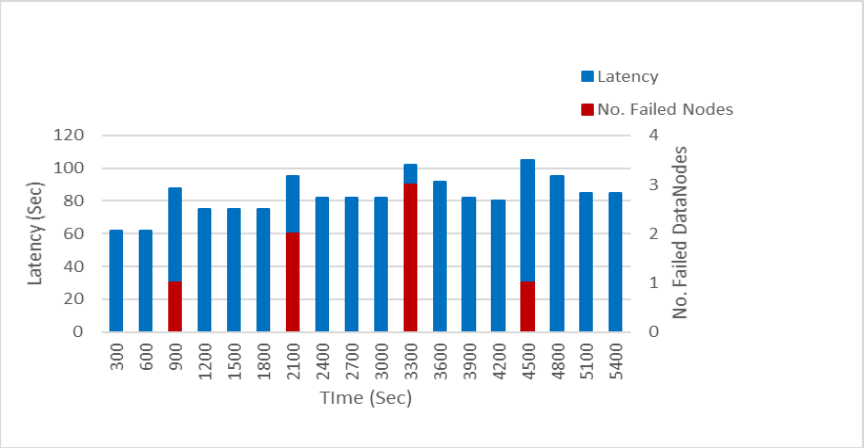

Figure 27: Performance of Apache Spark executors under Node Manager failure

8.2 Benchmark Experiment for Apache Flink

The experiments used the following pre-defined measures for evaluation: latency, throughput, and CPU and memory utilization. These measures were captured by running the experiments with different runtime duration and data generation rate, which is the number of input events emitted from Kafka per second.

8.2.1 Benchmarking Baseline Performance

By default, Yahoo Streaming Benchmarks enabled checkpointing mechanism for Apache Flink to ensure similar behavior to Apache Storm. For baseline performance experiment, we disabled checkpointing in Apache Flink. Figure 28 reflects the performance of Apache Flink under different throughput rates. Given the fact that both Apache Flink and Spark faced similar performance bottlenecks, we observed that Apache Flink with its native streaming support, performed better than Apache Spark. For throughput rate of 12,000 events/sec, end-to-end latencies for Apache Flink and Spark were 6 and 36 seconds respectively. Apache Flink exhibited stable latency for higher throughput rates. For example, with a throughput rate of 30,000 events/sec, Apache Flink achieved a latency of 23 seconds as opposed to 45 seconds for Apache Spark. Regarding resource utilization, Apache Flink reached a peak memory consumption of 12 GB and a maximum CPU utilization of 28% for a throughput rate of 30,000 events/sec as shown in Figure 29 and Figure 30.

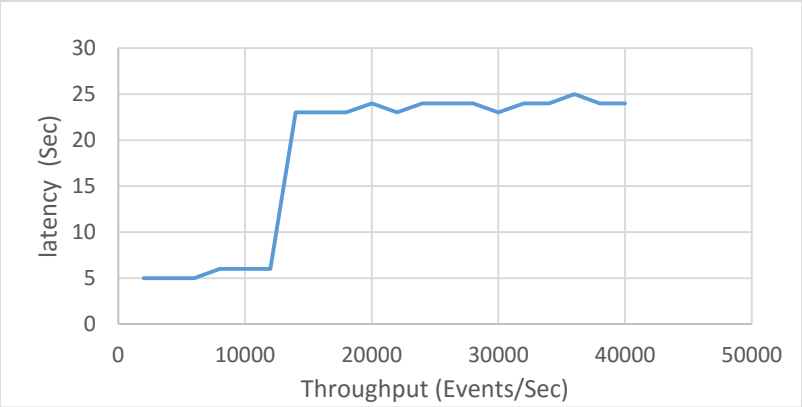

Figure 28: Baseline performance of Apache Flink for different throughput rates

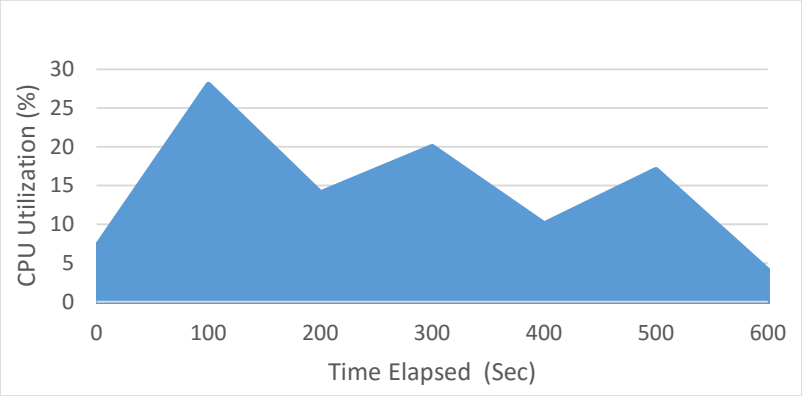

Figure 29: Baseline CPU utilization of Apache Flink for throughput rate of 30,000 events/sec

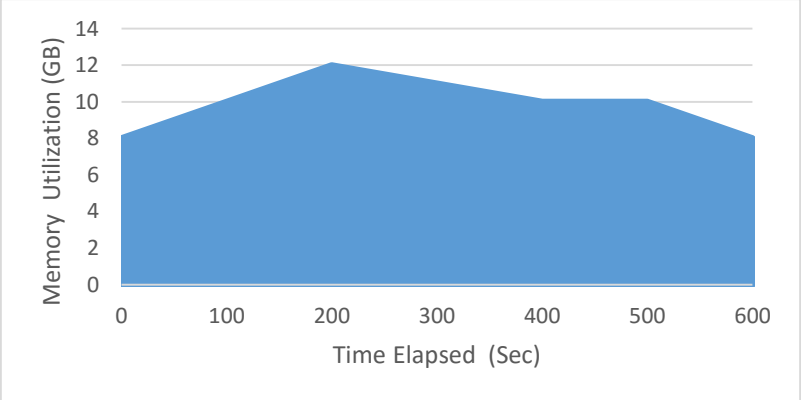

Figure 30: Baseline memory utilization of Apache Flink for throughput rate of 30,000 events/sec

8.2.2 Benchmarking Fault Tolerance Performance

We enabled checkpointing in Apache Flink by calling enableCheckpointing on "StreamExecutionEnvironment" with the adjustment of following parameters:

- Checkpoint interval in milliseconds
- Number of retries for restarting streaming job after a failure.
- Guarantee level that relates to the alligment of snapshots barrires, Exactly-once or at-least-once guarantee.
- Allowed number of concurrent checkpoints.
- Checkpoint timeout to abort a checkpoint if it is not completed.

Figure 31 shows comparison between the baseline and fault tolerance performance with checkpoint interval of 5000 milliseconds. Although the latency slightly increased, the system maintained stable performance levels with an increase in throughput rates.

Figure 31: Comparison between baseline and fault tolerance performance of Apache Flink

Figure 32 and Figure 33 shows CPU and memory utilization of Apache Flink for throughput rate of 30,000 events/sec. The figures reflect significant increase in the CPU and memory utilizations.

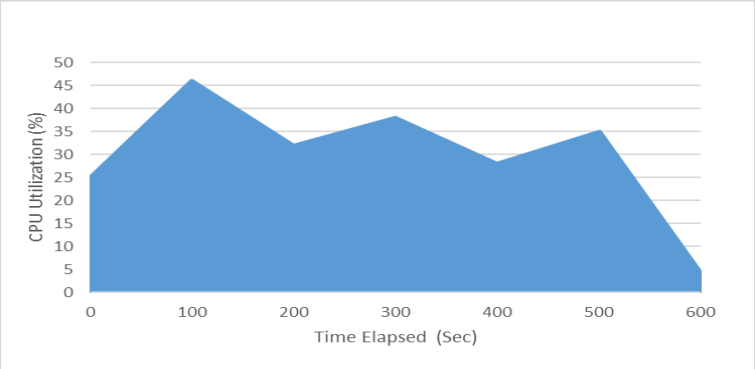

Figure 32: CPU utilization for fault-tolerant processing in Apache Flink for throughput rate of 30,000 events/sec

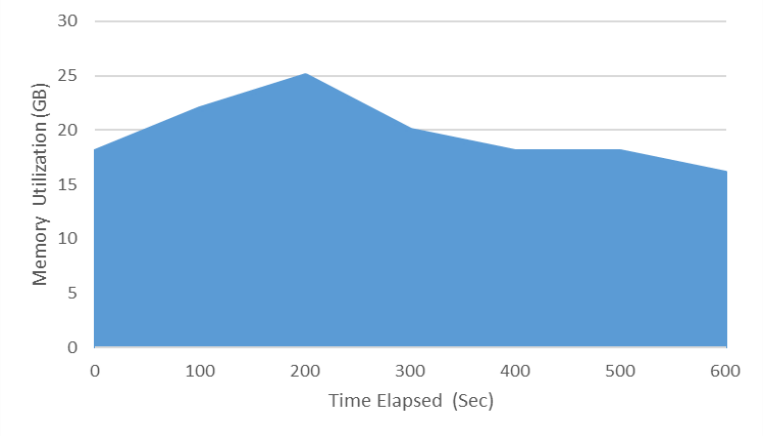

Figure 33: Memory utilization for fault-tolerant processing in Apache Flink for throughput rate of 30,000 events/sec

8.2.3 Benchmarking Fault Tolerance Performance under Failure

In our experiment, we excluded Job Manager failure (single point of failure) and executed different failure scenarios to suspend and kill Task Managers on different levels. We ran benchmark experiment with different runtime duration starting from 15 minutes up to 90 minutes and introduced transient and permanent failure using Anarchyape tool. Upon the failure of a Task Manager, Job Manager kills running tasks and marks the failed Task Manager as a dead process. Then Job Manager reschedules the canceled job execution on a different active Task Manager. In transient failure, the failed Task Manager is reactivated and it re-register itself at the Job Manager.

Transient Data Node Failure with a Data Generation Rate of 30,000 Events/Sec

We injected transient fault by randomly suspending Node Managers excluding the master node that is hosting Job Manager processes. Upon failure, the system stops the execution of the pipelined tasks under the Task Manager of the failed Node Manager, and therefore the system encounters a decrease in the throughput rate as shown in Figure 34. After recovery, the system enters a more stable state with a consistent throughput rate.

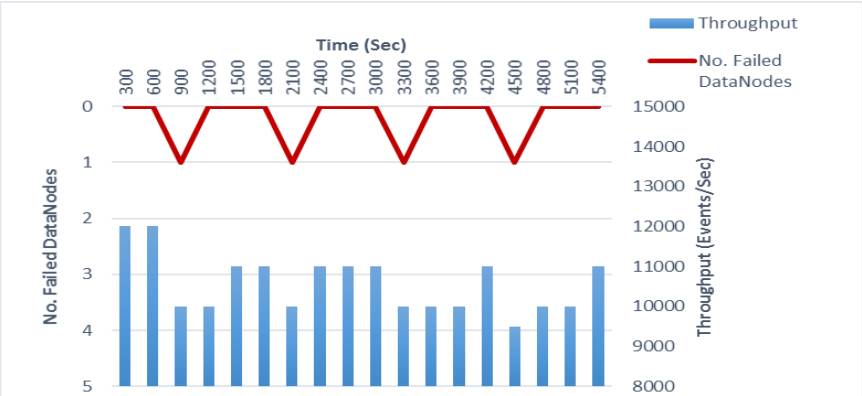

Figure 34: Performance of Apache Flink under transient failure

Permanent Node Manager Failure with a Data Generation Rate of 30,000 Events/Sec

During the execution of dataflow graph (DAG), it is not possible to scale up the application dynamically by allocating new YARN containers or increasing the number of available computational resources. Consequently, a permanent loss of Node Manager and Task Mangers means giving up the computational resources such as Task Managers' slots and YARN containers. Figure 35 reflects the gradual decrease of throughput rate due to permanent loss of computational resources.

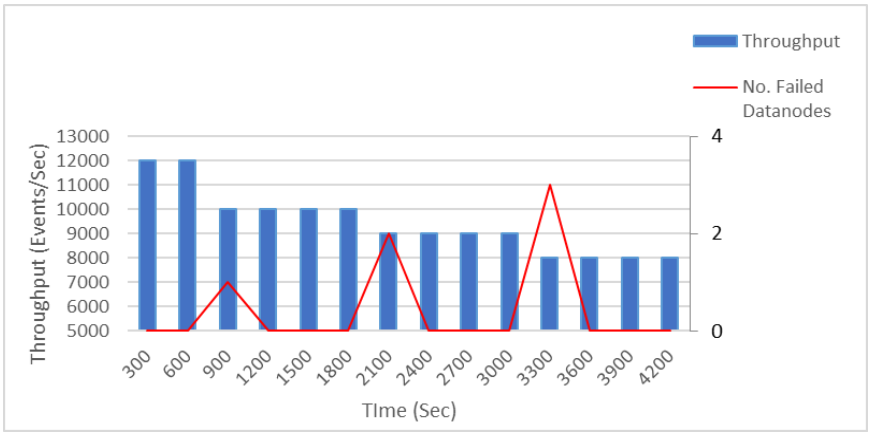

Figure 35: Performance of Apache Flink under permanent failure

8.3 Discussion

The experiment analysis was categorized into two groups: comparison between baseline and fault tolerance performances and evaluation of fault tolerance performance under failure. During the analysis of baseline and fault tolerance performances, Apache Flink showed more consistent and lower latency performance than Apache Spark when subjected to a gradual increase of throughput rate. Moreover, Apache Spark exhibited serious performance issues as we increased the runtime of the experiment. The system eventually crashed once the runtime exceeded 90 minutes. It was observed that the non-optimized implementation of checkpointing technique caused Apache Spark to compute its state twice before saving it to persistent storage. Such a behavior exhausted the computational resources and led to an out of memory exception error.

During the analysis to measure the performance of fault tolerance processing under failure, YARN infrastructure was injected with fault scenarios that caused transient and permanent failures in Node Managers processes. After recovery from the transient failure, Apache Flink maintained stable performance and consistent throughput rate. However, in the case of permanent failures Apache Flink exhibited a decrease in performance with an increase in the number of failed Node Managers.

For Apache Spark, both transient and permanent failures did not have any impact on the system performance because of Spark's dynamic resources allocation feature. However, the system exhibited different behavior for the failure of its driver and executors. Upon recovery from driver failure, the system showed a dramatic increase of in throughput that helped the system to catch up with unprocessed data. On the other hand, upon executor's failure, the system performed stably with a slight increase in latency after every failure.

9 Conclusion

Presently, there are no standardized and publicly available benchmarks to evaluate fault-tolerant stream processing with fault injection experiments. We presented a benchmark experiment that aimed to evaluate the performance of Apache Flink and Spark when subjected to failures. One specific contribution of the experiment was the introduction of a fault injection tool to evaluate the performance of fault-tolerant stream processing under failures. The experiment benchmarks the performance of two streaming systems Apache Flink and Spark and the experiment design was based on using the open source streaming benchmark developed by Yahoo for performance comparison and the introduction of an open source fault injection tool called "AnarchyApe" to create failures in Hadoop YARN environment.

The experiment analysis showed that both Apache Flink and Spark can process the reliable data stream generated by Apache Kafka with exactly-once semantics in which processed data is not lost or duplicated after failure. However, for both baseline and fault-tolerant processing, Apache Flink showed more consistent and lower latency performance than Apache Spark under a gradual increase of the throughput rate. For fault-tolerant processing under failure, Apache Flink maintained stable performance and consistent throughput rate for transient failures. However, Flink exhibited a decrease in performance with an increase in permanent failures in the computational resources. On the other hand, Apache Spark could not keep the performance at sustainable levels once the runtime of the experiment was increased.

In practice, the selection of stream processing systems depends on the application requirements. If the application requires low latency and exactly-once stream processing for streaming operation, Apache Flink would be one of the good choices. On the other hand, if the application throughput were the main criteria then both Apache Flink and Spark would be good choices.

The major limitation of the designed experiments is its inability to evaluate the effect of a single point of failure within Hadoop environment such as a failure of a Resource Manager or Job Manager within Apache Flink. The lack of flexibility to evaluate stream processing systems that are deployed on storage architectures other than Hadoop YARN is another limitation.

Further research is necessary to enhance the experiment and assess the performance of stream processing operation under complex workload. Secondly, scalability of the system can be tested using the "extended" Yahoo benchmark provided by Apache Flink team.

Bibliography

[1] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz, "Realtime Data Processing at Facebook," Facebook, 2016.

[2] Amit Sangroya, Damian Serrano ,Sara Bouchenak, "Benchmarking Dependability of MapReduce Systems," University of Grenoble - LIG - INRIA, Grenoble, France.

[3] Joseph, Zhiming Liu, Mathai, "Real-Time and Fault-Tolerant Systems," UNU-IIST, 2015.

[4] Arvind Kumar, Rama Shankar Yadav, Ranvijay, Anjali Jain, "Fault Tolerance in Real Time Distributed Systems," Motilal Nehru National Institute of Technology, Allahabad, 2011.

[5] M. Barlow, Real-Time Big Data Analytics: Emerging Architecture, United States of America: O'Reilly Media, 2013.

[6] "Fault Tolerance," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Fault_tolerance.

[7] Troubitsyna, Elena A., "Faults, errors, failures," [Online]. Available: http://users.abo.fi/etroubit/SWS13Lecture2.pdf.

[8] A. Christy Persya, T.R.Gopalakrishnan Nair, "SYSTEMS, FAULT TOLERANT REAL TIME," nternational Conference on Managing Next Generation Software Application (MNGSA-08), Bangalore, India, 2008.

[9] Steen, Andrew S. Tanenbaum and Maarten van, Distributed Systems: Principles and Paradigms.

[10] J.-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges," Toulouse, France, 1995.

[11] Zapletal, Petr, "Introduction Into Distributed Real-Time Stream Processing," Cakesolutions, July 2015. [Online]. Available: http://www.cakesolutions.net/teamblogs/introduction-into-distributed-real-time-stream-processing.

[12] S. Perks, Tolerating Late Timing Faults in Real Time Stream Processing Systems, University of leeds, 2015.

[13] T. Dunning and E. Friedman, Streaming Architecture: New Designs Using Apache Kafka and MapR Streams, O'Reilly Media, 2016.

[14] P. Zapletal, "Comparison of Apache Stream Processing Frameworks," Cakesolutions, [Online]. Available:

http://www.cakesolutions.net/teamblogs/comparison-of-apache-stream-processing-frameworks-part-1.

[15] André Leon Sampaio Gradvoh, Hermes Senger, Luciana Arantes and Pierre Sens, "Comparing Distributed Online Stream Processing Systems Considering Fault Tolerance Issues," May, Brazil and France, 2014.

[16] Nasir, Muhammad Anis Uddin, "Fault Tolerance for Stream Processing Engines," Stockholm, Sweden.

[17] S. Kamburugamuve, "Survey of Distributed Stream Processing for Large Stream Sources," 2013.

[18] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz, "Realtime Data Processing at Facebook," Facebook, 2016.

[19] "Apache Storm," [Online]. Available: https://storm.apache.org/.

[20] "Apache SAMZA," [Online]. Available: http://samza.apache.org.

[21] "Amazon Kinesis," [Online]. Available: https://aws.amazon.com/kinesis.

[22] A. Spark, "Apache Spark Streaming," Apache Spark , [Online]. Available: https://spark.apache.org/streaming.

[23] "Apache Flink," Apache Flink, [Online]. Available: https://flink.apache.org.

[24] "Apache Kafka," [Online]. Available: http://kafka.apache.org/.

[25] "What is zookeeper," Apache ZooKeeper, [Online]. Available: http://zookeeper.apache.org.

[26] "Benchmark Experiments," R package benchmark, [Online]. Available: http://benchmark.r-forge.r-project.org/.

[27] Y. Wang, "StreamBench: Stream Processing Systems Benchmark," Aalto University, 2016.

[28] "Benchmarking Streaming Computation Engines at Yahoo Engineering," Yahoo, 2015. [Online]. Available: http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at.

[29] S. Perera and A. Perera, "Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds," Erasmus Mundus Distributed Computing,Royal Institute of Technology, Stockholm, Sweden.

[30] P. Córdova, "Analysis of real time stream processing systems considering," University of Toronto .

[31] T. Feng, "Benchmarking apache samza: 1.2 million messages per second on a single node," [Online]. Available: https://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node.

[32] A. Sangroya, D. Serrano and S. Bouchenak, "Benchmarking Dependability of MapReduce Systems," University of Grenoble - LIG - INRIA, Grenoble, France.

[33] NETFLIX, "Chaos Monkey," NETFLIX, 2015. [Online]. Available: https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey.

[34] Bharat Venkat, Prasanna Padmanabhan, Antony Arokiasamy, Raju Uppalapati, "Can Spark Streaming survive Chaos Monkey?," NETFLIX, March 2015. [Online]. Available: http://techblog.netflix.com/2015/03/can-spark-streaming-survive-chaos-monkey.html.

[35] W. Guozhang, "Consumer Client Re-Design," cwiki.apache.org, 2015. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/Consumer+Client+Re-Design.

[36] D. Bhattacharya, "Near Real Time Indexing Kafka Message to Apache Blur using Spark Streaming," [Online]. Available: http://docplayer.net/4736945-Near-real-time-indexing-kafka-message-to-apache-blur-using-spark-streaming-by-dibyendu-bhattacharya.html.

[37] R. Metzger, "Yarn Chaos Monkey," [Online]. Available: https://github.com/rmetzger/yarn-chaos-monkey.

[38] Kostas Tzoumas, Stephan Ewen and Robert Metzger, "High-throughput, low-latency, and exactly-once stream processing with Apache Flink™," August 2015. [Online]. Available: http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/. [Accessed 2016].

[39] "Apache_Hadoop," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Apache_Hadoop.

[40] E. Coppa, "Hadoop Architecture Overview," [Online]. Available: http://ercoppa.github.io/HadoopInternals/HadoopArchitectureOverview.html.

[41] "Resource Manager Restart," Hadoop, [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/ResourceManagerRestart.html.

[42] J. He, "APACHE HADOOP YARN IN HDP 2.2: FAULT-TOLERANCE FEATURES FOR LONG-RUNNING SERVICES," Hortonworks, 2015. [Online]. Available: http://hortonworks.com/blog/apache-hadoop-yarn-hdp-2-2-fault-tolerance-features-long-running-services/.

[43] "Apache Hadoop YARN JIRA ticket for work-preserving ApplicationMaster restart (YARN-1489)," Apache Hadoop YARN, 2014. [Online]. Available: https://issues.apache.org/jira/browse/YARN-1489.

[44] V. K. Vavilapallih, A. Murthyh, C. Douglasm and S. Agarwali, "Apache Hadoop YARN: Yet Another Resource Negotiator," Hortonworks, Microsoft,Inmobi, Yahoo, Facebook.

[45] "Spark straming and Kafka," Cloudera, [Online]. Available: http://www.slideshare.net/jackghm/spark-streaming-kafkathe-future-of-stream-processing.

[46] W. Zhu, H. Chen and F. Hu, "ASC: Improving Spark Driver Performance with Automatic Spark Checkpoint," Shanghai, China, 2016.

[47] "Spark streaming programming guid," Spark, [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing.

[48] T. Das, "Discretized Stream - Fault-Tolerant Streaming Computation at Scale - SOSP," Databricks, 2015.

[49] A. Kawa, "Recent Evolution of Zero Data Loss Guarantee in Spark Streaming With Kafka," 2016, [Online]. Available: http://getindata.com/blog/post/recent-evolution-of-zero-data-loss-guarantee-in-spark-streaming-with-kafka/.

[50] "Apache Spark Backpressure (Back Pressure)," [Online]. Available: https://github.com/jaceklaskowski/mastering-apache-spark-book/blob/master/spark-streaming-backpressure.adoc.

[51] "Apache Flink™: Stream and Batch Processing in a Single Engine," Paris Carbone, Stephan Ewen, Seif Haridi, 2015. [Online].

[52] "YARN Setup For Flink," Apache Flink, [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-0.8/yarn_setup.html.

[53] P. Wagner, "Stream Data Processing with Apache Flink," 2016. [Online]. Available: http://bytefish.de/blog/stream_data_processing_flink/.

[54] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, Kostas Tzoumas, "Lightweight Asynchronous Snapshots for Distributed Dataflows," 2015.

[55] Lamport, K. M. Chandy and L., "Distributed snapshots: determining global states of distributed systems," ACM TOCS, 1985.

[56] "Introduction to Apache Flink," Data & Analytics, 2015. [Online]. Available: http://www.slideshare.net/mxmxm/introduction-to-apache-flink-51277423.

[57] "Flink Streaming and Checkpointing," Apache Flink, [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-master/internals/stream_checkpointing.html.

[58] "Savepoints. Flink Docs v1.1," Apache Flink, [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/savepoints.html.

[59] Ufuk Celebi, Kostas Tzoumas and Stephan Ewen, "How Apache Flink™ handles backpressure," 2015. [Online]. Available: http://data-artisans.com/how-flink-handles-backpressure/.

[60] Yahoo, "Yahoo Streaming Benchmarks," Yahoo, [Online]. Available: https://github.com/yahoo/streaming-benchmarks.

[61] A. Apex, "Extending-the-yahoo-streaming-benchmark-to-apache-apex-," Apache Apex, [Online]. Available: http://image.slidesharecdn.com/extendingtheyahoobenchmarktoapex-160505160128/95/extending-the-yahoo-streaming-benchmark-to-apache-apex-3-638.jpg?cb=1462464135.

[62] J. Grier, "Extending the Yahoo! Streaming Benchmark," Feb 2016. [Online]. Available: http://data-artisans.com/extending-the-yahoo-streaming-benchmark/.

[63] Anarchyape, "Anarchyape," Yahoo anarchyape, [Online]. Available: https://github.com/david78k/anarchyape.

[64] Faraz Faghri, Sobir Bazarbayev, Mark Overholt, "Failure Scenario as a Service (FSaaS) for Hadoop Clusters," University of Illinois at Urbana-Champaign, USA, 2012.

[65] "Optimize checkpointing to avoid computing an RDD twice (SPARK-8582)," Spark, [Online]. Available: https://issues.apache.org/jira/browse/SPARK-8582.

[66] "Spark Streaming + Kafka Integration Guide," Apache Spark, [Online]. Available: http://spark.apache.org/docs/latest/streaming-kafka-integration.html.

[67] "Big Data Processing with Apache Spark - Part 3: Spark Streaming," [Online]. Available: https://www.infoq.com/articles/apache-spark-streaming.

[68] "Spark Streaming + Kafka Integration Guide," Apache Spark, [Online]. Available: http://spark.apache.org/docs/latest/streaming-kafka-integration.html.

[69] "Apache Spark Streaming checkpointing playground for everyone to learn by example," Github, [Online]. Available: https://github.com/pygmalios/spark-checkpoint-experience.

[70] "Flink: General Architecture and Process Model," Apache Flink, [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-0.7/internal_general_arch.html.

[71] "Apache Hadoop YARN," Apache Hadoop , 2016. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[72] Tathagata Das, Matei Zaharia and Patrick Wendell, "Diving into Apache Spark Streaming's Execution Model," Databricks, 2015. [Online]. Available: https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html.

[73] G. Schmutz, "Apache Storm vs. Spark Streaming – two Stream Processing Platforms compared," 2014. [Online]. Available: http://www.slideshare.net/gschmutz/apache-stoapache-storm-vs-spark-streaming-two-stream-processing-platforms-comparedrm-vsapachesparkv11.