EXPLORATION OF FAULT TOLERANCE IN APACHE SPARK

BY

AKSHUN GUPTA

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Indranil Gupta
ABSTRACT

This thesis provides an exploration of two techniques for solving fault tolerance for batch processing in Apache Spark. We evaluate the benefits and challenges of these approaches.

Apache Spark is a cluster computing system comprised of three main components: the driver program, the cluster manager, and the worker nodes. Spark already tolerates the loss of worker nodes, and other external tools already provide fault tolerance solutions for the cluster manager. For example, the cluster manager deployed using Apache Mesos provides fault tolerance to the cluster manager. Spark does not support driver fault tolerance for batch processing. The driver program stores critical state of the running job by maintaining oversight of the workers; failure of the driver program always results in loss of all oversight over the worker nodes and is equivalent to catastrophic failure of the entire Spark application.

In this thesis, we explore two approaches to achieve fault tolerance in Apache Spark for batch processing, enabling promised execution of long-running critical jobs and consistent performance while still supporting high uptime. The first approach serializes critical state of the driver program and relay that state to passive processors. Upon failure, this state is loaded by a secondary processor and computation is resumed. The second approach narrows the scope of the problem and synchronizes block information between primary and secondary drivers so that locations of cached aggregated data is not lost after primary driver failure. Loss of these locations leads to a state from which computation cannot be resumed. Both approaches propose considerable changes to the Apache Spark architecture in order to support high availability of batch processing jobs.
ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Indranil Gupta who guided me through my Masters program making sure that I have all the resources and support I needed to succeed. I would like to thank my colleague, Nishad Phadke, who worked with me on this project before graduating. I would also like to thank Shubham Chopra and Sudarshan Kadambi from Bloomberg for helping me with the project.

Lastly, I would like to thank my parents for their continued support and love. I am where I am today because of them and I strive to make them proud.
# TABLE OF CONTENTS

Chapter 1: INTRODUCTION ............................................. 1
  1.1 Contributions of this Thesis .................................. 2
  1.2 Outline of this Thesis .......................................... 2
Chapter 2: APACHE SPARK BACKGROUND ............................ 4
Chapter 3: RELATED WORK ............................................ 7
  3.1 Hadoop MapReduce ............................................. 7
  3.2 Spark Checkpointing ............................................. 8
  3.3 Hydrosphere Mist ................................................ 10
Chapter 4: DESIGN ....................................................... 12
  4.1 Design of Restartable Cluster .................................. 12
  4.2 Serializing State ................................................ 18
Chapter 5: IMPLEMENTATION ......................................... 26
  5.1 Restartable Cluster ............................................. 26
  5.2 Serializing State ................................................ 27
Chapter 6: EVALUATION ............................................... 29
  6.1 Restartable Cluster ............................................. 29
  6.2 Serializing State ................................................ 32
Chapter 7: CHALLENGES OF APPROACH ................................. 35
  7.1 Increased Cost of Spark Development .......................... 35
  7.2 Sharing RDDs across Multiple SparkContexts ............... 36
  7.3 Replaying Logs Leads to Recomputation ...................... 37
Chapter 8: NARROWING THE SCOPE OF THE PROBLEM ............... 39
  8.1 Background ...................................................... 39
  8.2 Design .......................................................... 41
  8.3 Goals for Evaluation ............................................ 43
  8.4 Generality of Block Synchronization .......................... 44
CHAPTER 1: INTRODUCTION

The field of data processing and cluster computing has seen a lot of work for years including seminal ones like Dryad [1], Condor [2], and Google’s MapReduce [3]. The emergence of large scale internet services led to an increase in data aggregation and data analytics. Given the prevalence of rich data, many developers started developing custom applications to process raw data. Some of these applications include Word Count, Inverted Index [4] or counting degrees of a vertex in a large scale graph. The MapReduce project was part of an effort to create a simple and powerful programming model which could be used for parallelized applications.

Apache Spark [5] is an evolution of the MapReduce framework for batch processing (and later advanced to support stream processing). Apache Spark is an all-purpose computing system designed for accelerating computations on a distributed cluster of machines [6]. APIs in Java, Scala, Python and R, allow data scientists to specify high-level execution graphs that Spark then optimizes and parallelizes across its worker nodes. Spark contributors have created a wide variety of tools for various specialized purposes, like GraphX for accelerating processing of iterated graph algorithm [7], MLlib for machine learning tasks [8], Spark SQL for processing structured data with SQL-like semantics [9, 10], and Spark Streaming [11]. Spark is implemented in Scala [12] which is a statically typed function programming language which can be run on a Java Virtual Machine (JVM).

Apache Spark already tolerates the failure of the cluster manager and the worker nodes in the cluster. It does not, however, tolerate driver failure for batch processing. Failure of the driver process always results in the failure of the application and the loss of cached metadata in the Executors. This leads to wastage of resources as the developer is forced to restart potentially long running and/or resource intensive applications. In this thesis, we aim to make the driver program highly available so its failure does not result in catastrophic failure of the application.
1.1 Contributions of this Thesis

In this thesis:

1. We design and implement a target-agnostic serialization scheme to capture state changes in critical targets.
2. We create a simple replication-based fault tolerance patch to Apache Spark that allows for multiple “lazy” instances of driver program replicas to respond to primary driver failure. We employ failure detection and leader election to detect and overtake primary driver failure to resume computation.
3. We evaluate this serialization scheme and failover process by running experiments to test failover time and latency increase in application runtime for a variety of application types.
4. We discuss the challenges that this approach faces in detail and how to overcome them.
5. We explore a second design which solves problem of fault tolerance for online query use cases of Apache Spark. This solution aims to keep the cluster available and queryable by replicating RDD (see Chapter 2) locations.

1.2 Outline of this Thesis

1. In Chapter 2, we explore the background of Apache Spark explaining some key concepts and design decisions which makes Apache Spark a unique batch processing system.
2. In Chapter 3, we present some earlier work done related to fault tolerance and checkpointing in the area of batch processing.
3. In Chapter 4, we propose the design of an approach to solve the problem including serialization of state and the design of a restartable cluster.
4. In Chapter 5, we discuss how the design is implemented.
5. In Chapter 6, we evaluate the implementation of the restartable cluster and serialization of state using a sample of representative Spark applications and different failure scenarios.
6. In Chapter 7, we discuss the challenges of the solution proposed in Chapter 4 and discuss the changes which would need to take place in Spark and Scala to realize it.

7. In Chapter 8, we narrow the scope of the problem by trying to assume a different system model and different use case. We then discuss the design of a system which can be used to make the driver highly available for the mentioned system model.

8. In Chapter 9, we provide a summary of the ideas put forth in this thesis and look ahead to discuss some future work.
CHAPTER 2: APACHE SPARK BACKGROUND

Apache Spark uses an abstraction called a Resilient Distributed Dataset (henceforth RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel [13]. Spark and RDDs were created as a result of deficiency of the MapReduce model for two kinds of applications:

1. Iterative jobs: Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce job, each job must reload the data from disk, incurring a significant performance penalty [5].

2. Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [14] and Hive [15]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk [5].

RDD is a distributed in-memory data structure that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner [16]. RDDs (1) are read-only collections of objects partitioned across a set of machines that can be rebuilt when a partition is lost, (2) may not exist in physical storage — a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage—and (3) leverage their immutability to allow for easy replication across several machines. This allows for parallelized operations to be performed on RDDs and also helps provide fault tolerance for the workers operating on an RDD. RDDs support a fixed number of operations called transformations [17]. Rather than operating on the RDD in place, transformations yield new, transformed RDDs.

All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned
Figure 2.1: A typical Apache Spark cluster [19].

to the driver program. This design enables Spark to run more efficiently. For example, a dataset created through \texttt{map} will be used in a \texttt{reduce} and return only the result of the reduce to the driver, rather than the larger mapped dataset [18]. Operating on the RDDs in place would violate the immutability constraint. The lineage of these transformed RDDs is stored on the driver.

Spark applications run as sets of independent processes on a cluster, coordinated by the SparkContext (henceforth SC) object in the main program (called the \textit{driver program}) [19]. Specifically, a typical Spark deployment consists of a driver program, master, and worker(s). A typical Apache Spark cluster is shown in Figure 1.

Previous work has shown that the master processes can be made fault tolerant by deploying them using Apache Mesos [21] or Apache ZooKeeper [22]. The cluster manager is called the “master” in case it is not deployed using a resource manager and is deployed with the driver process itself. The cluster manager can communicate with the driver to allocate resources to complete the job. The workers receive tasks from the driver and master and execute them using their own compute power. The driver and workers regularly communicate with each other, which requires that the driver is addressable by the workers during
the lifetime of the application. Furthermore, the driver program is the entry point of the application and is where the directed acyclic graph (DAG) of sub-computation dependencies is created and managed to execute the job. Workers will compute a part of this DAG and will respond to the driver with results of the tasks, thereby, populating lineage of RDDs. The driver program keeps track of tasks and block information of workers which means that in case of worker failures, it can reschedule tasks at other workers. This does mean, however, that results of computations not reported back to the driver are lost.

In the case of a driver failure, not only do RDDs lose their lineage but workers lose a master from whom they can receive work. The failure of a driver not only means that all current computations the workers are doing will inevitably be lost but also means that all previous sub-computations that workers have completed are also lost. Driver failure is synonymous with application failure, and the Spark user is left with no solution other than restarting the application from scratch.
CHAPTER 3: RELATED WORK

In this chapter, we want to highlight some previous work in the space of fault tolerance and checkpointing in batch processing frameworks. We identify some key learnings from all examples to design our eventual solution to solve fault tolerance in Apache Spark.

3.1 Hadoop MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key [23]. The hallmark of MapReduce is the ease of programming and its ability to process large amounts of data with a high degree of parallelism.

Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner [24]. Some key architectural learnings can be derived from the Hadoop architecture. Similar to Apache Spark, Hadoop follows a master and slave architecture. The machine which houses the master node is called the NameNode. The NameNode keeps the directory tree of all files in the file system which in short means, keeps the mapping from file to location without actually storing the file [25]. This is similar to the Apache Spark model where the driver program keeps track of RDDs and its locations without actually storing the RDDs.

In Hadoop 1.0, the NameNode is a single point of failure which means that failure of this process results in loss of critical state for the application leading to termination. However, with the introduction of Hadoop 2.0, this single point of failure is removed by scaling horizontally the NameNode through HDFS Federation [26] as seen from Figure 3.1. In a typical HA cluster, two separate machines are configured as NameNodes. At any point in time, exactly one of the NameNodes is in an active state, and the other is in a standby
The Active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a slave, maintaining enough state to provide a fast failover if necessary. When any namespace modification is performed by the Active node, it durably logs a record of the modification to an edit log file stored in the shared directory. The Standby node is constantly watching this directory for edits, and as it sees the edits, it applies them to its own namespace. In the event of a failover, the Standby will ensure that it has read all of the edits from the shared storage before promoting itself to the Active state. This ensures that the namespace state is fully synchronized before a failover occurs [27].

As will be evidenced in this thesis, we derive inspiration from this architecture of horizontally scaling the master process and synchronizing state using a shared notification channel or a reliable file system.
3.2 Spark Checkpointing

Checkpointing in Distributed Systems is a technique for a distributed application to save its critical state so that the application can be resumed from previous failed state after failure. Many snapshot algorithms like the Chandy-Lamport Snapshot algorithm [29] capture all state for the processes in the distributed system periodically. The snapshot algorithm does this by periodically initiating a snapshot capture scheme in which each process records its own registers, variables, messaging channels, etc. Apache Spark provides the facility of checkpointing at a more abstract level with RDDs.

Apache Spark enables RDDs to be checkpointed to reliable storage like Hadoop File System (HDFS) [30]. Generally, RDDs are checkpointed when their computation takes a long time or the RDD has a long lineage of parent RDDs. When an RDD is checkpointed, its lineage is completely forgotten and the RDD is serialized to disk. When reading, the RDD is materialized which can be used for further computation. It is important to distinguish between RDD checkpointing and RDD caching. RDD caching is the process in which the RDD is materialized and kept in memory along with its lineage at the worker nodes. This way, upon worker failure, RDDs can be regenerated using the partial parent RDDs in the lineage stored in other worker nodes. Once the application finishes, all cached RDDs are wiped from memory from the worker nodes whereas RDDs checkpointed will remain even after application completion. This means that subsequent driver programs can read from checkpointed RDDs.

RDD checkpointing does not happen automatically, rather, it is a user initiated activity. This is because not all RDDs should be checkpointed as the cost of serializing and deserializing from disk may exceed the cost of natively constructing the RDD. A naïve solution to the problem of fault tolerance may seem like to checkpoint all RDDs and restart the job but this will be an extremely inefficient solution. Long running applications which handle excessive amounts of data will lead to prohibitive number of RDDs being checkpointed to disk.
An ideal solution would take advantage of the fact that RDDs are cached in memory during a Spark application runtime and not lose the application just because of driver failure. An ideal solution would not need to depend on RDDs being persisted on disk.

### 3.3 Hydrosphere Mist

There are solutions which enable running multiple Spark applications on the same set of machines. One solution we investigated was Hydrosphere Mist [31]. Mist implements Spark as a Service by managing multiple concurrent Spark applications. Mist coordinates state with each Spark application and can restart Spark applications upon failure. Mist allows for one machine to serve as a worker, driver, or master for 2+ active Spark applications simultaneously, which is notable because Spark’s design expressly prohibits running two drivers on the same virtual machine [32] and implicitly prohibits running two drivers even on the same machine, because the Spark master initialization script, start-master.sh, is only designed to have one Spark JVM running at a time.

![Figure 3.2: A typical Mist cluster](image)

Mist’s contribution to the Spark community, however, is insufficient to bring fault tolerance to the Spark driver. As shown in Figure 2, Mist allows users to segment executors for different applications - a feature not available in Apache Spark. In theory, a possible solution
to the fault tolerance problem in Apache Spark is to deploy the same application twice in a Mist cluster similar to the one shown in the figure. However, this solution uses twice the number of resources which makes it infeasible for large-scale batch processing jobs. Mist's framework does not allow for two Spark drivers from different Spark applications to communicate with each other, even if both drivers are running on the same machine, as this would violate the Spark-as-a-service semantics that Mist was created for [33]. The driver program cannot be replicated using Hydrosphere Mist as it treats each Spark application as a wholly independent entity.
CHAPTER 4: DESIGN

To bring fault tolerance to the driver program, the driver will need to be replicated. There are three ways to think about replication:

1. Live replication: We concurrently spawn multiple driver processes at the start of the application and propagate critical state change information from the primary driver program to the live replicas. Upon failure of the primary driver program, the replicas will detect the failure and use a leader election algorithm to promote exactly one of the replicas to be the primary driver.

2. Passive Replication: The primary driver state is lazily replicated. In this scheme, only one driver program is ever running and will periodically checkpoint its state to a reliable file system. An external failure detection mechanism will detect if this driver program fails and will respond by initializing a fresh driver program, connecting the existing workers to the new driver program, and restarting the execution of the Spark application.

3. A third approach is to remove the driver program from the Spark architecture altogether. This would revolve around assembling the worker nodes into a P2P system that internally coordinates computation without the need of a centralized driver program. This approach brings wholesale logic and architectural changes to Apache Spark which is prone to rejection from the community. Bringing fault tolerance to the Apache Spark batch processing system has to be iterative in nature and we need to think of a simpler solution. Therefore, we leave the conversion of Apache Spark to a P2P batch processing engine for future work.

Our approach is one that uses passive replication. This is done to minimize latency increase for the running application. If we used live replication, the primary driver would have to stream updates to all the standby drivers making it difficult to horizontally scale the number of standby drivers. Also, live replication would use more number of machines for every application which could make it wasteful when failures don’t occur during a run. The
only downside of using passive replication is that failover time (time taken after primary
driver failure until the new driver resumes computation) would increase.

However, our approach requires (1) viewing Spark applications as sets of independent
tasks completed sequentially by worker(s) and (2) viewing the driver as a deterministic state
machine, where the workers relay their incremental results periodically to the driver and each
update from each worker spurs a state transition in the driver. We provide fault tolerance
primarily through k-replication (see §4.1), but we also created a serialization framework for
“accelerated replay” by the newly elected primary driver of the failed driver’s state changes
(see §4.2). These “replay logs” are implemented using an ordered log of method invocations
in the primary driver so that replaying this log helps rebuild the state previously reached.

A critical step in bringing fault tolerance to Apache Spark is the creation of replica
drivers and the coordination infrastructure to perform Spark job computation. To achieve
correctness, we have created an infrastructure for a restartable cluster which will allow us to
restart Spark applications seamlessly and efficiently.

4.1 Design of Restartable Cluster

In order to restart an application, we need a system to restart the cluster with the same
configurations that were employed when the application was initially submitted. To create
the infrastructure for a restartable cluster, there are few considerations to keep in mind.

Replicating Configuration

Spark application can be run with custom configurations (e.g., setting a checkpointing
directory, setting the number of cores to run for the workers, setting ports to listen on and
ports to serve the web UI on, etc.). These changes are propagated via two methods:

1. Through the Spark application, itself. Regardless of whether the Java, Scala, Python,
or R API is used, each Spark application creates a SparkConf configuration object
through which it can configure how the SparkContext driver object (and thereby how
the application) will be run. Since we are running this Spark Application again with
the same startup scripts, any changes made in this file will also be executed in the new
cluster.

2. Through the command line interface. Users can set command-line flags for their ap-
plications when creating workers, creating the cluster manager, and submitting the
application to a cluster through the command line. For the sake of replicating those
settings, these commands are entered by the user in an external, on-disk configuration
file before the start of the application. These configurations are then used for starting
the master, the worker(s), and submitting applications to the driver.

Primary and Replica Drivers

During the start of any Spark application, the user will define a list of the IP address
of the replica drivers’ machines and a primary driver’s machine. The replica drivers act as
insurance for the primary driver in that if the primary driver fails, one of the replica drivers
will take lead and restart (ideally, resume) the computation. There is a tradeoff between the
cluster’s completion guarantee and the resource cost used for fulfilling that guarantee. More
replica machines implies stronger guarantees of the application being completed but also
implies a greater demand for resources by the application, which leads to higher provisioning
and logistic costs.

Another approach we considered is that of proactive replenishment. In distributed
systems, failure is the norm and not the exception. Failure of replica drivers will leave the
application in danger of non completion. $K - 1$ failures during the span of a long batch
processing job is highly probable when $K - 1$ is small. Proactive replenishment involves
dynamically increasing the set of standby drivers as standby drivers go down. With this
approach, one of the K drivers will relay stored critical state to the new replenishing
driver. The standby drivers will relay that information instead of the primary driver in order to avoid
overburdening the primary driver. We should remind ourselves that the standby drivers are
only collecting state relayed by the primary driver which means its resource consumption is not high. The downside of this approach is that this mode needs a global monitor which would initiate the process of adding a standby driver to the system introducing a single point of failure.

For the sake of simplicity, the current iteration involves us having a fixed number of standby drivers. In general, a replication factor of three is the standard. In our evaluation, we have used a replication factor of three as well (two replica drivers and one primary driver).

**Spark Application File Transfer**

The presence of a shared local file system between the replica and primary driver machines cannot be assumed. That is, the end user’s Spark application can only be guaranteed to be accessible by the primary driver, as would be the case if the end user were using a traditional non-fault-tolerant driver. During initial startup of the primary drivers and replica drivers, the primary driver must send the Spark application file (*.jar, *.py, etc.) to each
replica driver. Upon primary driver failure, the replica drivers need to be able to reference the application in order to (re-)run it. That said, if a shared file system does exist, we have included the provision of skipping the transfer of this file so that the overhead is not incurred.

**Failure Detection**

We need to have a failure detection mechanism so that when the primary driver dies, correct steps can be taken to overtake it and restart the application. A simple but effective solution is the gossip-style heartbeat protocol [34]. We chose this familiar protocol because it has proven to be accurate and fast while imposing low overhead in terms of both network bandwidth and development time.

The protocol is an epidemic protocol. Heartbeats from every live machine are sent to a random subset of the remaining live machines. A heartbeat from machine $M$ includes information about $M$’s current status as well as information about other live machines that $M$ has accumulated.

A correct implementation will allow us to accurately detect failure of the primary driver and of any failed replica drivers. We need an accurate failure detector with false positive rate of 0%. Should the failure detection protocol falsely indicate that the primary driver has failed, the primary driver (which is actively managing workers and making genuine computation progress) would have its workers forcibly revoked by a replica driver that believes itself to be the new primary driver, leading to an increase in latency of the application. This would mean that the replica driver either restarts the Spark application from scratch, or at least loses the subcomputations that the workers were performing at the exact millisecond that they were severed from the primary driver.

Consequently, we choose to remain conservative: we are willing to sacrifice detection time in exchange for this guarantee. A small increase in detection time is negligible when compared with the monolithic cost of running the Spark application for which the user is bothering to guarantee fault tolerance. We have modified the parameters of the gossip-
style heartbeat protocol to have extremely small false positive rate at the expense of failure detection time.

There are two states in which a driver machine can fail:

1. A driver fails while it is the primary driver. Upon primary driver failure, an elected replica (see §4.1) needs to overtake the just-failed driver (see §4.1).
2. A driver fails while it is still a replica driver. Upon a replica driver failure, no steps need to be taken as the application is still making progress on the primary driver. From the failed replica drivers missing heartbeats, all live drivers—both primary and replica—will eventually be made aware that the failed process is no longer eligible to be elected as a primary driver in the future.

**Leader Election**

Leader election allows us to elect a new primary driver upon primary driver failure (only) among multiple replica drivers. This driver is the process which will initiate connection with the worker nodes. Only one driver can be elected as there is only one process which will initiate the spark application. There are many leader election algorithms like the Bully algorithm and Ring Election. We could also use a third party service like Apache Zookeeper to manage the leader election.

However, we choose to do a primitive type of leader election which is effective and simple. We do leader election by nominating the live machine with highest hostname. All replica drivers know instantly which process is the new primary driver without communication. As soon as this implicit change of primary driver occurs, all other replica drivers update who their primary driver is so that further failures can also be dealt with accordingly. After this leader election occurs, the new primary driver begins the overtaking process outlined in §4.1.

**Overtaking Primary Driver**

After the new primary driver is elected, it initiates the following steps:
1. Start a new Spark master on the current machine. This master is the cluster manager from Figure 2.2.

2. Notify worker machines of primary driver failure and existence of a new master. The worker nodes disconnect from the previous master and connect to the new master. In the case of an Apache Mesos managed cluster manager, the user can provide the address of the master in the config file and that will be the address passed to the workers.

3. Submit application using the command given in the configuration file.

By running these three steps, the application will be run on the new primary driver. This means that results of the computation will be available on the new primary driver instead of the previous one (possibly where the application was originally submitted). Normally, users output the results to a reliable file system.Persisting results to a reliable file system is done via the Spark application.

**Cascading Failures**

Our system is fault tolerant upto $K - 1$ failures, where $K$ is the number of driver processes running (one driver and $K - 1$ replicas). When a primary driver fails, a new primary driver is elected. This repeats after each subsequent driver failure until the application is either complete or the application runs out of drivers. In the case of running multiple applications sequentially, the system follows a slightly different process as explained in §5.1.

**4.2 Serializing State**

This section centers the design decisions we made to capture critical state changes in the primary driver. We need to relay and apply these state changes at the secondary driver making sure that the standby driver is a replica of the primary driver.
**Approach**

In the open-source Apache Spark code base [35], `SparkContext.scala` houses the implementation of the driver program. A new instance of this object is created once a user creates and submits a Spark job. Our initial driver checkpointing implementation was to periodically serialize the entire state of the driver and dump it to a reliable filesystem. We attempted to leverage serialization tools like Scala Pickler [36] and Java’s `Serializable` interface, µPickle [37], BooPickle [38], Prickle [39], and Chill [40], but bootstrapping this naïve solution was tedious. After further investigation, we realized that several objects contained in `SC` neither implement the `Serializable` interface nor lend themselves to easy dynamic (i.e., runtime) pickling using Scala Pickler. An even larger issue is that the `SC` object is large and complex; dumping the entire contents of the object to disk and transferring such a large amount of data over the network at each checkpointing stage would be wasteful.

Our revamped approach is to only log changes in state of the `SC`. The advantage of this approach is that the log file will be substantially smaller, allowing our solution to scale to more machines and to more users of Spark. A replica driver program will now only need to iterate through the log file and replay the ordered state changes of the original driver program to replicate its state.

For the purposes of creating a replayable log, the `SC`’s state changes of interest are changes that occur as a result of the invocation of one of `SC`’s member functions. Naturally, a log file that contains an ordered listing of each function that was invoked on the primary driver program, along with the arguments passed into that function invocation, would contain information sufficient for a second driver program to replicate and completely resemble the creator of the log file. We installed software hooks in each of `SC`’s member functions that save to disk a few properties of the current invocation of that member function.

The process of recording this state is described here. The format for the log files takes into account Scala’s reflection library [41]. Scala’s reflection library allows for invocation of a function not known at compile time so long as the name of the function (as a `String`)
and the parameter types are supplied\(^1\). The data model that we follow to log the function, serialized function parameters, and parameter types is as follows:

\[
\begin{cases}
\text{name} & \vdash f_{\text{name}} \\
\text{args} & \vdash [x_1, x_2, \ldots, x_n] \\
\text{types} & \vdash [T_{x_1}, T_{x_2}, \ldots, T_{x_n}] 
\end{cases} \tag{4.1}
\]

where \(f_{\text{name}}\) is encoded as a human-readable Unicode string, each \(x_i\) is a machine-readable byte string, and \(T_{x_i}\) is the Scala type of \(x_i\) encoded in any machine-deserializable format.

**Assumptions and Guarantees**

There are a few considerations to keep in mind when developing this system and they are outlined below. For ease of explanation, we will discuss the specific use of our serialization system on SC, but the guarantees described here—which are specifically the guarantees needed to accurately log invocations in SC—extend to serialization of any Scala object.

**Total Ordering of Log Entries** It is imperative that the primary driver program appends to the log in exactly the same order in which its member functions are invoked. Conversely, the reader of the log must also replay the log in exactly the same order that the logged invocations appear in. Without these constraints, the second driver program may not have exactly the same state that the original driver program had when it failed.

**Nested Functions** We stated previously that all functions which change state of the driver program need to be logged. This was an oversimplification. In fact, if all functions are recorded blindly, the log will almost surely be incorrect. It is common in software design to use helper functions inside other functions. That means that function \(g_1\) could, for example, call function \(g_2\) and then function \(g_3\) twice. It is easy to see how logging every function

\(^1\)This is to disambiguate between overloaded functions of the same name.
invocation would create an incorrect log file. When a replica attempts to replay a log that resembles

\[ g_1 \rightarrow g_2 \rightarrow g_3 \rightarrow g_3, \]  

(4.2)

the replica would invoke \( g_1 \) (which internally calls \( g_2 \) once and \( g_3 \) twice) and then would invoke \( g_2 \) then \( g_3 \) then \( g_3 \) for a grand total of 1 invocation of \( g_1 \), two invocations of \( g_2 \), and four invocations of \( g_3 \). Calling a function multiple times is particularly dangerous when the function is not idempotent, and most are not.

In order to solve this problem, we look at the stacktrace for each function that we need to log. We log the function invocation only when no ancestor of this function resides in the same class. That is, we are logging the first function in \( \text{SC} \) that cannot trace its roots back to another function in \( \text{SC} \).

It is important to understand why we chose to look for any ancestor and not just the immediate parent. There may be a scenario where function \( f_1 \) (which is part of \( \text{SC} \)) calls function \( \Omega \) (not part of \( \text{SC} \)) which calls function \( f_2 \). If we only examined the immediate parent, \( f_2 \) would have been logged when it should not have been; only \( f_1 \) (\( \text{SC} \)'s causal root of the function invocation chain) should be logged. Therefore, it is important to check whether any of the ancestors in the function’s stacktrace is part of SparkContext.

The above solution appears to break down, however, when \( f_1 \) calls \( \Omega \) by means of a remote procedure call (RPC) and then \( \Omega \) calls \( f_2 \) also by means of an RPC. When \( f_2 \) is invoked, it is eligible for being logged. Unfortunately, \( f_2 \)'s stacktrace would contain no evidence of it being an RPC that was originally caused by \( f_1 \), one of \( \text{SC} \)'s own functions. The technique described above would log both the \( f_1 \) and \( f_2 \), instead of only logging \( f_1 \). When a replica attempts to replay the log, the replica will invoke \( f_1 \) (which calls \( \Omega \) which calls \( f_2 \)) and then will independently invoke \( f_2 \).

In fact, upon further examination, this argument is inherently flawed. There is no guarantee that an external agent will make an RPC call for \( f_2 \) during execution of \( \Omega \). The
external agent’s state may have changed between when the primary driver was alive and when the replica performed the replay. In fact, we cannot place any expectations on any external agent. The only way the replica can correctly replay the log is if it invokes $f_1$ and $f_2$ (in that order), without any dependency on external agents, and thus must replay the log in an isolated environment, one in which it cannot make network calls and inadvertently corrupt external agents’ states by communicating stale data (i.e. old function invocations) to them. Thus the solution we propose fully solves the problem of (networked) nested functions.

**Non-atomic Functions** We also need to worry about how logging and replaying will work if failures occur while the function is being executed. In other words, we cannot assume function invocations are atomic, but must rather assume that failures can occur while a function is being executed. This is an important note because this dictates how and where we log the function invocation. We can neither serialize at the beginning of the function nor the end; the former because if an error occurs after serialization is done, the replica drivers will infer that the function was called even though it was not, nor the latter because the driver may crash immediately before logging the function invocation but after all of its computation (and state change) has been completed.

This means that we need to implement a *two-phase commit serialization* logic in which we do a soft log of the function at the beginning of the function and we do a hard log at the end of the function. If the replica driver sees a soft log without a corresponding hard log, it can conclude that a failure occurred during the middle of that function’s execution. A very conservative approach to providing fault tolerance would be to abandon replicating the driver if such a scenario is encountered. This would force the user to restart the application from scratch.

A less conservative approach would require a replica driver program to deduce how far deep a function invocation the original driver died. The replica driver, upon completing its replay phase, would have to query each worker node for all messages it received from
the primary driver. Then either the replica driver fools itself into thinking that it, too, has sent those messages, or it forces each worker node to halt its computation and to rollback its state to one that is compatible with the replay log that the replica driver has received. Then the replica driver and worker nodes resume computation. This approach is much more taxing on the implementation side as it requires intimate knowledge of the worker and of the communication protocols between the workers and the DAG scheduler (the construct that sends execution instructions to the workers).

**Log File Format**  A naïve implementation of storing log data to disk would be to have one large log file and repeatedly append to this file. This poses two central problems. First, in case of a live replication of replica drivers, this monolithic log file can become very large. Implementing throttling protocols to transfer this file at intentionally slower rates between reliable storage and the replicas will add the complexity of download progress tracking. Secondly, appending to the end of the log will not be as simple as opening the file, seeking to the end of the file, writing bytes, and closing the file. No library we use to serialize the logging data structure will be simple enough to support this "quick" append. Consider JSON or Protobuf: both of these serialization formats will require deserializing the file contents into a data structure housed in memory, appending to this data structure, and then re-serializing the contents back to disk. This burdens the logging component with significant I/O demands as well as the risk of corrupting the file if failure occurs during this process.

To circumvent these problems, each function invocation will be logged to a separate file where filenames are of the form `functionLogXXX.log` where `XXX` is a monotonically increasing positive integer. Since the driver is a single process, we can maintain this monotonically increasing positive number. This way, the replica drivers can read from the smallest log file number to the end, which facilitates preservation of the requirements imposed in §4.2. This approach lends us flexibility in how critical state changes are relayed to replica drivers. If using “live replication”, we can either stream these logs as soon as they are written or batch
and send per time interval.

**Concurrent Functions** SC is thread-safe. That is, it is safe to assume that no two executions of state-changing functions can be interleaved by a well-behaved Spark application. Nonetheless, by creating a transactional system, where function invocation logs are prepared then committed, we can force serialization to be thread-safe even if SC’s thread safety cannot be relied upon. This also assists with meeting the requirements of target agnosticism.

**Target Agnosticism** The serialization functionality had to be designed in such a way that it is unaware of and unreliant on the implementation and design of the object that it is serializing. Employing such a design pattern enables the reuse of the same pieces of software to save state of any Scala object.

**Targets**

As stated previously, the SC is the main component in a Spark driver. It is the central object that holds handles to all state-capturing pieces of the driver. Naturally, serializing method invocations in this class is a necessity to capture state changes in the primary driver. This became our first serialization target.

SC also depends on several services, like an RPC Environment (RpcEnv) for managing RPC messages and a metrics manager for supplying measurements to the user, but these do not have to be serialized because they are implemented as services and can be requested from even a replica driver without needing to know the details of the service (knowing such details would violate the “black box” pillar of these very services).

There are a few things that SC depends upon that will need serialization in order to faithfully replicate driver state. Those components are namely the DAG Scheduler and Task Scheduler.

The DAG Scheduler does not contain any public documentation, as its implementation matters only to Spark contributors; end users never need to worry about the behavior of the
DAG Scheduler as it is entirely abstracted away from the user. We built our understanding of it by reading the in-code documentation and by reading Mastering Apache Spark 2 [42]. It is the reponsibility of the DAG Scheduler to take a lineage of RDDs (i.e., a specification of the flow of data through a Spark graph—a graph that is a directed acyclic graph, or a DAG) and convert it into a series of stages and jobs that can be assigned to and scheduled on executors. Most notably, the DAG Scheduler tracks which RDDs have been cached so that it can avoid recomputing these RDDs during the lifetime of the same Spark application.

The DAG Scheduler breaks individual steps in the data flow graph into jobs, which are sliced into stages. Stages are independent pieces of computation that can be reassembled into larger jobs, which are the larger, top-level units of computation.

When the DAG Scheduler submits a job, a JobWaiter, which extends JobListener, is created. It blocks until the underlying job either completes or fails. Our serialization functionality therefore should also target invocations of these callbacks in JobWaiter to detect when a job has completed. This completed job should be paired with the relevant subgraph in the DAG and the resultant RDD is serialized to disk so that a replica driver, upon re-creating the complete DAG, can prune the DAG based on what results have been completed—that is, based on what RDDs have already been computed—to avoid having to recompute results the cluster’s workers have already computed.

It is worth noting that serializing state changes in the DAG Scheduler is not necessary for correctness. Correctness of the application could be satisfied by having a replica driver restart the entire application from scratch, but this solution is no different from running the application on $k$ clusters simultaneously, hoping that at most $k - 1$ of those drivers fail. Serializing DAG Scheduler state changes only aids in avoiding recomputation by means of caching intermediate results.
CHAPTER 5: IMPLEMENTATION

5.1 Restartable Cluster

We implemented the restartable cluster infrastructure (RCI) in Python, outside the Spark stack. Writing code outside the Spark driver to bring fault tolerance to the driver ought to raise red flags because it is conceivable that the driver could fail without the RCI failing, or vice versa. Fortunately, such a scenario is impossible. Because the only way a driver can ‘fail’ is through network partitioning or outright machine failure, we can colocate a separate RCI process on the same machine as the replica driver. Failure of a driver is perfectly correlated with failure of this colocated process (since only machine failure or network partitioning can fail a driver) so we do not need to provide additional fault tolerance guarantees for this supplemental monitoring process. As a corollary, we note that our system is not robust to driver failures that come from semantic errors in the user’s application. For example, a NullPointerException that fails one Spark driver would indeed fail every driver in the restartable cluster. This is out of the scope of providing fault tolerance to the driver process.

Before the start of any Spark Application, the user would fill out a config.ini which would list important information about the cluster. The config file would list the driver addresses and the worker addresses. It would list the location of the Spark Application so that it could be transferred from the primary driver to the replica drivers in case a shared file system is not available. It would also contain a list of the spark submit commands which are needed to run the Spark applications.

When the config file is filled out, the worker and driver processes are started and the Spark application(s) listed in the config file are run in succession. It is important to note that this system is not fault tolerant towards application failures. Since we are restarting the job, any subsequent driver will also run into the same application error as was encountered before.
Allowing Multiple Sequential Batch Processing Jobs

Our restartable cluster allows users to run multiple Spark applications in succession. The user would just need to enter the Spark commands in a list in the config file and these applications would be run in succession. In the case of failures, already completed spark applications are not rerun as when a spark application is completed, replica drivers are notified of the job completion. This way the cluster keeps state of the current running job and only those jobs are restarted which have not been completed (either due to failure or non-commencement).

5.2 Serializing State

When recording state of the driver program, it is important to do so with no change to the interface of SparkContext.scala and minimal change to the implementation. In order to abstract this logic, we created a DriverSerializer (henceforth DS) singleton to be the central manager of all driver function invocation logging. We created hooks in each function in SC to request a transaction handle from the DS and to use the handle to initially perform a soft commit as soon as the transaction is fully built and later hard commit once the function completes.

Transaction Manager

The DS singleton serves to create handles to new transactions that SC can populate with data. It manages committing the transaction to a reliable file storage system and ensures that they maintain their total ordering.

```scala
private[spark] class DriverSerializer {
  def setCheckpointDir(dir: String)
  def createTransaction(): Transaction
  def incrementCommitCount(): Long
}
```
Transaction Abstraction

To keep the interface of a Transaction simple, we create a series of log(...) functions, each one overloaded with a different parameter type, and two different commit functions, one for soft commits and another for hard ones.

```scala
private[spark] class Transaction {
  def log[K: ClassTag, V: ClassTag](arg: Map[K, V])
  def log[T: ClassTag](arg: CollectionAccumulator[T])
  def log[IN: ClassTag, OUT: ClassTag](arg: AccumulatorV2[IN, OUT])
  def log[T: WeakTypeTag](arg: RDD[T])
  def log[T: ClassTag](arg: Seq[T])(implicit tag: ClassTag[T])
...
  def softCommit()
  def hardCommit()
}
```
CHAPTER 6: EVALUATION

In this chapter, we evaluate our implementation of the restartable cluster and serialization framework using a diverse set of applications and under different failure scenarios. The goals of our evaluation are to measure: 1) failover time under a variety of scenarios including cascading failures, cluster size and different configurations for the gossip-style heartbeat protocol, 2) runtime or latency impact of the state serialization system, and 3) average packet size which gives us an indication of required bandwidth for the solution.

6.1 Restartable Cluster

The primary metric to measure the performance of the restartable cluster is of failover time. Failover time is defined as the time taken between failure of the primary driver and successful restart of the application on a replica driver. For a deeper view, we have divided this period into failure detection time, time taken to start the master, reconnection time (between workers and master), and eventual start of the application.

Figure 6.1 shows the progression of failover time between varying sizes of clusters. As is expected, the time remains roughly constant across the cluster sizes. The time remains constant because all the intermediate times comprising of failover time are independent of the number of workers. However, one thing to note is that a large portion of failover time is comprised of failure detection time.

In the experiment leading to Figure 6.1, the timeout period for the failure detector was set to four seconds. Upon increasing time by two seconds, as seen from Figure 6.2, we can notice that the failover time increases by roughly 2 seconds.

As was mentioned before, we want to be conservative about labeling machines as failed due to danger of false positives. These experiments were done in an environment where the VMs were part of the same network leading to sub-second heartbeat latency. In a more realistic environment, a higher timeout is reasonable. This timeout period is configurable and depending on the network environment, the user can modify it to minimize failover time.
Figure 6.1: Failure Detection Period has majority stake holder in Failover Time. Timeout was set to 4 seconds. Number of drivers was 3. 10 experiments were run for each cluster size.

Figure 6.2: Failover Time increases by 2 seconds by increasing gossip-style heartbeat protocol timeout by 2 seconds. Number of drivers was set to 3. 10 experiments were done per timeout. Note that this is not measuring failover time but just detection time. Failover times are observed to be constant as per Figure 6.1
Figure 6.3: Average failover times are consistent between systems with varying numbers of replicated drivers. The failover times after each incremental driver failure were recorded and are shown here. A 2 + 2 second timeout period was used because of favorable network conditions.

During the hundreds of tests we ran, using the conservative settings discussed in §4.1, our gossip-style heartbeat protocol failure detector never experienced any false positives.

**Failover Time Over Cascading Failures**

Next, we measure the individual failover times over cascading driver failures during the lifetime of a single application.

A long-running application with varying numbers of replica drivers was created, and each driver was failed one after another, starting with the initial primary driver and ending with one driver remaining. Figure 6.3 shows the pattern of failover times after each driver. No matter how many failures occur in a single run of an application, the failover time remains constant. The failover time is, on average, invariant to the number of failures. The gossip-style heartbeat protocol was designed for scalability, so an increase in the number of replica drivers expectedly has no effect on the failover time.

Figure 6.4 shows that the failover time across multiple failures in a single experiment remains constant, *i.e.* that $\forall i, j < k, F_i = F_j$, where $F_i$ is the failover time after the $i$th
Figure 6.1 and Figure 6.3 strongly indicate that the design and implementation of the restartable cluster scale well.

6.2 Serializing State

To evaluate the runtime impact of the state serialization system introduced, we ran five different Spark batch jobs, namely PageRank [43], KMeans clustering [44], WordCount, estimation of Pi, and Sorting of 10 million integers. We chose these applications because they are a diverse set of applications which trigger all the endpoints set up in Transaction. We measured two characteristics: average packet size and increase in overhead of end-to-end latency in application runtime. Here, we use “packet” to mean each log file that contains the complete information of a single function invocation.

Analyzing Table 6.1, we can clearly see the average packet size is very similar for
Table 6.1: Percentage increase in application runtime and filesystem impact

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>14.9%</td>
<td>197.7 ± 925.0</td>
<td>77112</td>
</tr>
<tr>
<td>KMeans</td>
<td>16.6%</td>
<td>74.1 ± 132.5</td>
<td>4075</td>
</tr>
<tr>
<td>WordCount</td>
<td>6.1%</td>
<td>71.8 ± 158.4</td>
<td>2584</td>
</tr>
<tr>
<td>Pi</td>
<td>7.0%</td>
<td>185.8 ± 290.7</td>
<td>1858</td>
</tr>
<tr>
<td>Sorting</td>
<td>17.2%</td>
<td>75.7 ± 139.4</td>
<td>3934</td>
</tr>
</tbody>
</table>

Figure 6.5: Enabling driver serialization has a moderately sized impact on application latency
KMeans, Sorting, and WordCount but not for PageRank and Pi. This is because the PageRank and Pi applications create many more (generally large) RDDs. Logging these RDDs is an expensive operation in terms of bytes written to disk. However, even an average packet size of around 200 bytes is small enough to be efficiently transferred over the network without aggressive partial file transfer protocol overhead. Small packet sizes ensure bandwidth used by the application is low. It is important to note that total log file size is heavily dependent on the runtime of the application and nature of application.

Next, we measure the end-to-end latency impact of the new serialization architecture. We expect an increase in application runtime as serialization of objects and writing to disk are both in the critical path. This is evident from Figure 6.5 and Table 6.1. However, this cost is minimal compared to the cost of losing the whole job upon driver failure.
CHAPTER 7: CHALLENGES OF APPROACH

Chapter 3 discussed the architecture of the restartable cluster and serialization of critical state. These two concepts dictate how Spark applications can be restarted and how re-computation can be avoided. This chapter discusses challenges this design faces which makes it hard to solve the problem at this time. However, we argue that with some effort and a handful of changes to Spark and Scala, our design can solve the general problem.

7.1 Increased Cost of Spark Development

Our solution requires Spark developers to know about its existence and use which increases cost of development on the platform. The serialization framework (see §4.2) is unable to serialize inherently unserializable objects and is let down by the lack of features in the Scala reflection library [41].

A drawback of the TransactionManager and DriverSerializer (see §5.2) interface is that a developer of Apache Spark would be required to know about its interface and use while updating the Apache Spark code. If a developer wants to add functionality to \texttt{SC}, then he would need to make sure that state changes are captured using the above abstractions. This increased cost of developing on Apache Spark would make our design and implementation unpopular with the open source community. There is no one version of Apache Spark as each client (companies and individual user) has created their own version of Apache Spark with tweaks that benefit them which makes centralized changes hard to perform. As developers of Apache Spark, it is our responsibility to provide a solution which requires minimal intervention from users.

Python provides a feature called “decorators”. A Python decorator is a specific change to the Python syntax that allows us to more conveniently alter functions and methods (and possibly classes in a future version) [45]. Essentially, python decorators allow a pre-defined function to be called before the invocation of another function. This pattern could be used in our design which would make sure that any developer would not need to deal with the
Figure 7.1: This piece of code [46] shows a typical use case of decorator functions. It will print the time of invocation and the output of my-function.

interfaces defined. Currently such a clean implementation of decorators is not available in Scala.

Some state we want to capture is not inherently serializable. For example, there are functions in SC that cannot be replayed. Most notably, this includes methods that accept a callback function as an argument. Since functions cannot be serialized in Scala, we currently have no way to “replay” the invocation of such SC methods in the new driver. Creating custom serialization techniques for other non-serializable objects would require operating out of the standards set by the Apache Spark but can be achieved with time.

7.2 Sharing RDDs across Multiple SparkContexts

Default implementations of Spark RDDs cannot be shared between two Spark applications [47] and therefore cannot be shared between two Spark drivers\(^2\).

\(^2\)A second SparkContext instance is instantiated on the second driver after primary driver failure talked about in §4.1
Our multiple-driver solution depends on being able to transfer intermediate results between drivers on altogether separate machines, let alone between two drivers in a single JVM\(^3\), and RDDs are undoubtedly what will be used to deliver such inter-driver data transfers. One of the developers of Apache Spark explains that, “It’s not possible to take any user program in Spark and make this state entirely recoverable on process failure. If we started to go down this path, we would need to do things like define a standard serialization format for the RDD data, a global namespace for RDD’s, persistence, etc” [48]. Defining standard serialization format for RDD data and defining a global namespace would be out of the scope of this research project and they are not guaranteed to be the only additional changes which would be needed to effectively solve the problem.

Apache Ignite seems to offer a solution as it “provides an implementation of Spark RDD abstraction which allows to easily share state in memory across multiple Spark jobs, either within the same application or between different Spark applications.” [49]. IgniteRDD is an implementation of native Spark RDD and DataFrame APIs which, in addition to all the standard RDD functionality, also shares the state of the RDD across other Spark jobs, applications and workers [49].

### 7.3 Replaying Logs Leads to Recomputation

Blindly replaying logs will lead to re-computation of the Spark application and recognize that our solution of *only* replaying the log is insufficient.

We log all function invocations from our serialization targets (see §4.2) which means that all requests would get reissued from the driver to the cluster. The DAG scheduler relies on a live event loop to schedule jobs, stages, and tasks. A replica driver would have to go through the log file and will one-by-one create its DAG. Once the replica encounters a submitJob logged invocation, the replica will submit its own job, and the DAG scheduler will return a JobWaiter. The replica is, however, being replayed in an isolated network, and

\(^3\)In fact, even supporting two drivers on one JVM is not supported [32].
all outgoing packets are intentionally dropped. This is so as to avoid one worker receiving
one message from the original driver and then receiving the same (stale) message from the
replica after failure of the original driver. As a side effect of this decision, the replica driver
will now be waiting forever for the JobWaiter to finish waiting. The workers would need to
keep track of messages that have been received to differentiate between real task queries and
duplicate ones.

Another option would be to eliminate portions of the DAG that have been computed
and eliminate those subgraphs (i.e., to prune the DAG). Saving these subcomputations
would be done through installing hooks in the DagScheduler and store each return value of
each task, stage, and job and we will recompute the DAG from the sources to the sinks as
per usual, this time discarding vertices for which we already have computations. However,
results of subgraph computations would be in the form RDDs which means that this solution
depends on the one talked about in §7.2.

All these problems are not fundamental problems but ones which would take more time
and maturation of the Scala language to solve.
CHAPTER 8: NARROWING THE SCOPE OF THE PROBLEM

Until now, we have talked about achieving fault tolerance for Apache Spark Batch Processing which means achieving fault tolerance for arbitrary Spark applications. This makes it necessary to come up with a general solution which would work for all kinds of workloads and application types and that is why our proposed solution discussed in Chapter 3 involved trying to capture all critical state. Trying to capture all critical state turned out to be an impossible task with the current Spark architecture (as explained in Chapter 6) which meant that we needed to try to narrow the scope of the problem.

8.1 Background

Bloomberg, among other things, provides a data analytics platform which leverages Apache Spark for Batch Processing. Bloomberg’s primary use of Spark is in the area of online query processing and human-time interactive analytics. Apache Spark has been known for batch processing but Bloomberg uses it to perform interactive analytics. What this means is that they use Apache Spark as a service which is able to serve queries over a long period of time and perform low latency updates. Bloomberg has created a Spark server which is a single long running Spark application. Queries are served through a REST API and it can be deployed either through Spark Standalone or using a cluster manager like MESOS or YARN.

The analytics engine made available through the Spark server has two stages of its lifecycle. First is the ingestion stage where relevant data is ingested in the cluster and cached in memory of the executors of the Spark cluster. They use a separate data store to load the data in Spark. The second stage is the “query” stage in which queries are satisfied from RDD transformations using the cached data in the executors. This stage is from where the long running characteristic of their Spark Server is derived. These queries can come in during a long period of time whereas the ingestion phase is a short stage.
The ingestion phase includes ingesting high value data which is expensive to load into Apache Spark. Failure of the driver process leads to forced restart of the ingestion phase leading to high wastage of resources. Bootstrapping the Spark cluster (similar to the one talked about in §4.1) may result in expensive rehydration of any previously cached state, leading to severe performance hit to online analytics use cases. Since the query phase spans across a long timespan, the failure of the driver becomes more probable.

A running Spark cluster is made up of a Driver and a set of Executors. The basic abstraction of state is an RDD. When an operation is performed on an RDD, the result is a child RDD that has a reference to its parent RDD. This lineage between the RDD objects is maintained within the driver. The state within the RDD, if materialized, is kept at the executors within a component called the Block Manager. The driver maintains the association between RDD partitions/blocks and the executors they are hosted on.

It is important to note how this high value data is replicated across workers so as to ensure that worker failures do not result in losing that data. RDDs can be replicated across workers in a Spark cluster. Figure 8.2 clearly explains how executors request for peers to replicate the RDDs and how the driver keeps track of RDD block replications in executors.

Figure 8.1: Illustration of BlockManager which is used to house RDD blocks [50].
8.2 Design

As was discussed before, the Spark driver is an arbitrary application and so to narrow the scope of the problem, we decided to build a solution to provide fault tolerance for online interactive use cases. This narrows the scope of the problem because state (RDD blocks) created during the ingestion phase and its locality information is the only state we need to protect from failure.

Figure 8.1 shows that each Executor holds the BlockManager which manages RDD blocks and the BlockManagerMaster on each executor is an RPCEnv which is used to communicate with the BlockManagerMasterEndpoint. This endpoint is set up on the driver and using this endpoint, the driver keeps track of block information at each executor. This locality information is valuable and loss of the driver leads to loss of this mapping. We want to replicate this state across multiple standby drivers. If we increase the availability of this state, Apache Spark would be able to tolerate any failure during the query stage. One thing to note, however, is that this design would not tolerate failures during the ingestion phase. If failures do occur during this phase, ingestion is restarted. The cost of this restart is not
high as the ingestion process is a short lived process.

**Synchronization of State between Drivers**

The BlockManagerMasterEndpoint is an endpoint set up on the driver to which all registrations of BlockManagers set up on Executors takes place. All updates to RDDs, Blocks and Executors take place through this endpoint. This means all block locality information is gathered here. We create a notification channel between the primary driver and the standby drivers by instantiating a new endpoint on each standby driver called the StandbyDriverBlockManagerEndpoint. This instantiation would take place during the initialization of each process in the Apache Spark cluster through the SparkEnv. Internal code comments mentions that the SparkEnv object “holds all the runtime environment objects for a running Spark instance (either master or worker) including the serializer, RpcEnv, block manager, map output tracker, etc. Currently Spark code finds the SparkEnv through a global variable, so all the threads can access the same SparkEnv. It can be accessed by SparkEnv.get (e.g. after creating a SparkContext)” [35].

As a first iteration of this design, we would piggyback on this initialization scheme to set up a a standby driver process and standby driver endpoints on the workers itself. The BlockManagerMaster takes in its constructor the handle for the StandbyDriverMasterEndpoint. Whenever the primary driver receives an update from the Executors about blocks or RDDs (for example, registration of a new block or removal of an RDD), the primary driver relays that update to the standby drivers. All these updates are inherently serializable which means there is no added work other than creating the endpoints and initializing them in the SparkEnv during start up. Setting up the notification channel through SparkEnv means that the primary driver can discover the existence of the standby drivers and block information can be synchronized between the drivers.
Failure Detection and Leader Election

We can piggyback on the notification channel created between the standby driver and the primary driver to enable failure detection. Through the BlockManagerMasterEndpoint and StandbyDriverBlockManagerEndpoint, heartbeats can be relayed over this channel and after a certain timeout, failure can be detected. This means that the standby driver can detect failure of the primary driver. The standby drivers don’t need to be made aware of other standby drivers’ failure other than during leader election.

As a first cut implementation for leader election, we can employ the same scheme of leader election as discussed in §4.1. However, the driver processes would have to have knowledge about other standby drivers and that information can be passed through SparkEnv initialization. Once this design is proven to work, Apache Zookeeper will be used to handle failure detection and leader election between the drivers.

After all the above plans are implemented, the next and final step would be to determine how the standby driver overtakes the primary driver and connects with the workers so that it can start issuing queries to the cluster. The Executors need to be made aware (accept incoming connections from the new driver) of the change in driver to accept work from the driver and for heart-beating purposes.

Connection with Executors

After the leader election occurs, the new primary driver needs to broadcast its new role to the executors. The approach we are considering is for the Executors to accept connection from any standby driver from a pool of such drivers. This can again be implemented as part of the SparkEnv initialization. However, a more dynamic solution may need to be implemented if proactive replenishment is desired like creating an endpoint at the Executor to accept registration of new drivers.
8.3 Goals for Evaluation

In order to evaluate the performance of this solution, we would like to answer the following questions:

1. What are the bandwidth requirements of transferring block locality information between the primary and standby drivers. Does this requirement change with different types of data stored during the ingestion phase? If so, how?

2. How much time does it take for the cluster to quiesce after primary driver failure? Does this time change in varying failure conditions - before/during/after query execution?

3. How does the system perform under cascading failures?

8.4 Generality of Block Synchronization

We want to note that block synchronization is not a solution fit for the general problem of fault tolerance in Apache Spark. Block synchronization works based on the assumption that Spark is being used for online query use cases and failures do not occur during ingestion phase. In an online query use case, users submit queries one at a time. This means a fault tolerance solution for this use case does not have to solve for restarting a previously submitted query. The driver simply needs to be available for the next query. A general batch processing fault tolerance scheme would need to memoize computation in order to resume it - something which block synchronization cannot do.
CHAPTER 9: SUMMARY AND FUTURE WORK

This thesis presented a unique approach to achieve fault tolerance in Apache Spark. We discussed serializing function invocations in critical state targets to propagate state changes to standby processes. We discussed the design of a restartable cluster enabling failure detection and leader election for a primary driver overtake. We evaluated this new serialization scheme and system design by measuring application runtime increase and failover time with a diverse set of applications and scenarios. We outlined some of the challenges the approach faces and how they can be overcome.

Next, we moved onto the online query analytics use case of Apache Spark. We synchronized block locality information so that this data is not lost and future queries can be issued against it. We discussed our intended design for failure detection, leader election and overtake of the primary driver. We also highlighted some of the key questions we would like to answer while evaluating the solution.

9.1 Future Direction

As discussed in Chapter 3, a driverless Apache Spark is the ideal design for a fully fault tolerant batch processing engine. This driverless computation engine would be a P2P system of workers in which the workers would coordinate between themselves to compute the job at hand. Workers would have information about tasks being completed by other workers so that failures could be properly handled. This is a complete overhaul of the Apache Spark architecture and this part needs to be planned with a longer timeline.
REFERENCES


