DEPENDABLE CLOUD RESOURCES FOR BIG-DATA BATCH PROCESSING & STREAMING FRAMEWORKS

by

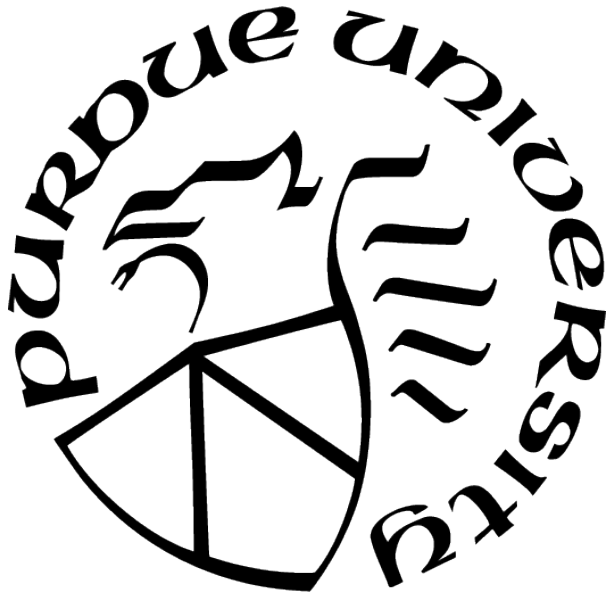
Bara Abusalah

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Prof. Arif Ghafoor, Chair

Electrical and Computer Engineering Department

Prof. Patrick Eugster

Computer Science Department

Prof. Samuel Midkiff

Electrical and Computer Engineering Department

Prof. Walid Aref

Computer Science Department

Approved by:

Prof. Dimitrios Peroulis

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The examiner of cloud computing systems in the last few years observes that there is a trend of the emergence of new Big Data frameworks every single year. Since Hadoop was developed in 2007, new frameworks followed it such as Spark, Storm, Heron, Apex, Flink, Samza, Kafka ... etc. Each framework is developed in a certain way to target and achieve certain objectives better than other frameworks do. However, there are few common functionalities and aspects that are shared between these frameworks. One vital aspect all these frameworks strive to achieve is better reliability and faster recovery time in case of failures. Despite all the advances in making datacenters dependable, failures actually still happen. This is particularly onerous for long-running "big data" applications, where partial failures can lead to significant losses and lengthy recomputations. This is also crucial for streaming systems where events are processed and monitored online in real time, and any delay in data delivery will cause a major inconvenience to the users.

Another observation is that some reliability implementations are redundant between different frameworks. Big data processing frameworks like Hadoop MapReduce include fault tolerance mechanisms, but these are commonly targeted at specific system/failure models, and are often redundant between frameworks. Encapsulating these implementations into one layer and making it shared between different applications will benefit more than one framework without the burden of re-implementing the same reliability approach in each single framework.

These observations motivated us to solve the problem by presenting two systems: Guardian and Warden. Guardian is tailored towards batch processing big data systems while Warden is targeted towards stream processing systems. Both systems are robust, RMS based, generic, multi-framework, flexible, customizable, low overhead systems that allow their users to run their applications with individually configurable fault tolerance granularity and degree, with only minor changes to their implementation.

Most reliability approaches carry out one rigid fault tolerance technique targeted towards one system at a time. It is more challenging to provide a reliability approach that is pluggable in multiple Big Data frameworks at a time and can achieve low overheads comparable with

single targeted framework approaches, yet is flexible and customizable by its users to make it tailored towards their objectives. The genericity is attained by providing an interface that can be used in different applications from different frameworks in any part of the application code. The low overhead is achieved by providing faster application finish times with and without failures. The customizability is fulfilled by providing the users the options to choose between two fault tolerance guarantees (Crash Failures / Byzantine Failures) and, in case of streaming systems; it is combined with two delivery semantics (Exactly Once / At Most Once).

In other words, this thesis proposes the paradigm of *dependable resources*: big data processing frameworks are typically built on top of resource management systems (RMSs), and proposing fault tolerance support at the level of such an RMS yields generic fault tolerance mechanisms, which can be provided with low overhead by leveraging constraints on resources.

To the best of our knowledge, such approach was never tried on multiple big data batch processing and streaming frameworks before.

We demonstrate the benefits of Guardian by evaluating some batch processing frameworks such as Hadoop, Tez, Spark and Pig on a prototype of Guardian running on Amazon-EC2, improving completion time by around 68% in the presence of failures, while maintaining around 6% overhead. We've also built a prototype of Warden on the Flink and Samza (with Kafka) streaming frameworks. Our evaluations on Warden highlight the effectiveness of our approach in the presence of failures and without failures compared to other fault tolerance techniques (such as checkpointing).

1. INTRODUCTION

In recent years, Cloud Computing technologies have evolved rapidly and become one of the most researched areas in distributed systems and computer science in general. The term 'Big Data' has emerged as a result of this evolvement to refer to Cloud Computing technologies that target applications which deal with very large data sizes. Since cloud application requirements are different from each others and the types of inputs are not the same, different types of Big Data systems have been developed to target the different requirements of these cloud applications. These Big Data systems are usually referred to as Big Data frameworks. Probably the best way to elaborate what does this term mean is by giving an example about it which is Hadoop. Hadoop is one of the most well known Big Data frameworks in the Cloud Computing community. It is a batch processing framework that uses Map Reduce algorithm to process input data files and uses HDFS as its distributed file system. The announcement of the first version of Hadoop in 2007 opened the door for developers to create new frameworks each of which is specialized in processing certain types of inputs and has its own characteristics that makes it unique from others. Some of these frameworks load data in memory, others are targeted towards database transactions, some others process data in real time and deal with continuous data streams.

In general, these Big Data frameworks can be split into two main categories according to the type of input they process; batch processing frameworks and streaming frameworks. From their names, batch processing frameworks deal with applications that process input data in bounded batches, whereas streaming frameworks specialize in processing continuous unbounded streams of data.

There are other characterizing factors for these two types of frameworks other than the type of the input data they process. The way these frameworks deal with security, fault tolerance, nondeterminism, ... etc. is tailored towards the type of the framework. For instance, the fault tolerance techniques applied to online real time streaming systems is different than how it is dealt with in offline batch processing frameworks. In the following subsection we will talk about these failures.

1.1 Failures in Cloud Computing & Big Data Applications

With the rapid development of *cloud computing*, distributed systems are becoming the *de facto* model of computation. Besides the emergence of new challenges for developers, fault tolerance has remained of paramount importance. According to Google's Jeff Dean, "*in each cluster's first year, it's typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours …*" [1]. Such failures are by no means limited to the first year [2]–[11].

A machine crash in a batch processing framework may result in restarting all batch processing tasks running on that machine. This task restart may have drastic effects for online streaming systems where end users are monitoring live data streams in real time where a machine crash may result in restarting the streaming tasks running on it which may end up halting the data stream for end users. This is not an acceptable behaviour if the end users expects the stream to be 'alive' and continuous without interruptions and to be delivered within a certain time frame.

As will be shown later, dealing with faults in streaming systems is also another challenging problem by itself. The difficulty of dealing with faults in streaming systems is it has to be transparent to the end user, i.e end users should not notice any interruption in the data stream as if the failure didn't happen at all.

Failures in datacenters are particularly onerous in the context of *big data processing*, which has recently advanced to become not only one of the dominant application scenarios for cloud computing, but also one of the most intensely researched areas in computer science. Big data applications are typically long-running applications that involve massive parallelization, achieved through big data processing frameworks such as Hadoop MapReduce (MR) [12] or Spark [13]; these are deployed on top of so-called *resource management systems* (RMSs) to best exploit the underlying hardware. Failures of subcomputations can easily hamper the response times and correctness of end results, and so such frameworks typically support recomputation of subcomputations to deal with partial failures. However, we observe several shortcomings with the current software system landscape:

1.2 Observations/Motivations

There are five main observations that have driven us to do this work. These motivations are:

Streaming & Batch Frameworks Recovery Time

Recomputation may not be sufficient anymore in many scenarios, with big data analytics no longer being a technology enabling new business models, but rather part of mission-critical decisions [14]–[17], and given the push towards online analytics.

As stated before, generally there are two types of Big Data frameworks based on the type of load they process; batch and streaming frameworks. In batch processing systems, the user submits a job and waits for it to deliver one final output. However, this is not the case in streaming frameworks. In streaming systems, some users are monitoring the live feed of updates online and in real time. If any task in the pipeline of that live stream fails and restarts, most likely the user will notice some interruptions in this live stream. Therefore, achieving low recovery time for streaming frameworks is much more needed and yet very challenging at the same time. On the other hand, some types of batch processing applications are Mission-Critical applications where any delay in the processing of the batch job is very undesirable for the end users. Hence, achieving low recovery time can benefit applications running on both streaming & batch processing frameworks.

Limitations of Checkpointing

In general, to achieve reliability for any system, there are two most common fault tolerance techniques that can be applied: checkpointing and replication. Most Big Data frameworks (both batch and streaming) already have checkpointing built-in and uses it to recover tasks progress in the case of machine failures. There are several shortcomings for the built-in checkpointing mechanism that makes replication more attractive and superior in many cases.

The first and most noticeable drawback of checkpointing is, in case of failure, there is always some time wasted in redoing the work that has been done from the latest checkpoint no matter how frequent the system takes checkpoints.

The second overhead of checkpointing is the overhead of consistent saving frequent checkpoints to persistent storage and the overhead of loading/transferring the checkpoint from the hard disk or any persistent storage device to the new machine where the new task has restarted.

Third, the time it takes the task to timeout and the time it takes the system to notice that the task/machine has failed. Note that this timeout could be a compound combination of timeouts of multiple systems working together. For example, as will be shown later, a streaming framework called Samza can run top of a resource management system (RMS) called YARN using Kafka as the datastream. There are multiple timeouts in this setup: Samza Application Master (AM) timeout, Samza container timeout, Kafka brokers timeout, YARN NodeManager (NM) timeout, YARN task container timeout, YARN AM timeout. Some of these timeouts can‘t be changed by the system users such as YARN NM and YARN AM timeouts because they are shared with other frameworks running on the cluster (managed by YARN).

Fourth, some systems have certain conditions and requirements to use their built-in checkpointing mechanisms. For instance, Flink is a streaming framework that constraints the user within certain preconditions to use Flink‘s built-in memory checkpointing mechanisms. These constraints are stated in [18]. Samza, on the other hand, doesn‘t run as a standalone system instead it needs an execution engine such as YARN and a messaging system such as Kafka to work with. Each of these systems has its own constraints to use its built in reliability features.

Redundant Work

Implemented fault tolerance support is hardwired and thus inherently geared at particular system models (e.g., asynchronous/synchronous) or failure models (e.g., benign / malicious failures). For instance, Hadoop deals with benign process failures and “slow” processes,

while several authors propose modifications dealing with correctness of component outputs to counter the insecurity of multi-tenant public clouds (e.g., [19]–[21]).

With support for recomputation being implemented across big data frameworks there is a duplication of mechanisms throughout frameworks. For example, Spark also implements recomputations like Hadoop (albeit with a broader scope given its support for incremental computing).

The observant of the Big Data frameworks and the Cloud Computing technologies in general (not only in Hadoop and Spark) will notice that some services and features are common between them. For instance, the examiner of the reliability aspect of these frameworks can see that almost all of them implement some kind of checkpointing inside the framework itself. Some parts of the checkpointing implementations are actually redundant and can be shared between different frameworks. It will be interesting to encapsulate these implementations into one shared component that can be shared between different frameworks. Although, as stated before, checkpointing has many shortcomings, so it is worth investing in another fault tolerance mechanism to make it shared between different frameworks.

Byzantine vs Crash Failures

Not all failures are fail-stop failures. Some failures cause the machine to eventually produce an output but this output is incorrect. Such a failure is called a Byzantine failure.

It is noted that checkpointing doesn't have any strategy to deal with such failures. In fact, most of the built-in reliability techniques in most modern big data frameworks doesn't have a way to verify the correctenss of the final output. Hence it is worth investing in a generic relibaility approach that can be used by multiple frameworks and cover a wider spectrum of failures (from Crash to Byzantine).

Models of Consistency in Streaming Frameworks

The way each streaming framework deal with failures is different but generally there are three common models of delivery/consistency in streaming frameworks: Exactly Once, At Most Once and At Least Once.

Briefly, in Exactly Once, a framework guarantees that, for all applications running on top of it, the receiver will receive the sent tokens exactly one time. In other words there are no redundant delivery of the same tuple or no missing tuples due to machine failures or out of sync snapshots. Even if failures happen, the system built-in fault tolerance mechanism still has Exactly Once guarantees. This model is the best delivery model between the three mentioned models because it will mask failures and make them transparent to the user in a way that no missing tuples or redundant tuples arrive mistakenly to the end user.

In At Most Once scheme, some tuples may not reach to the destination at all. This happens, for example, when a machine failure occurs and the sending source keeps sending tuples without getting back to the latest checkpoint. The exact opposite of this scheme is At Least Once where, in case of failures, the sending source has to rollback to the latest taken checkpoint and resend all tuples after that snapshot. This may resend some of the tuples that has already been received in the destination before the failure happen. The number of redundant tuples can be reduced by taking more frequent checkpoints.

In our work we try to achieve Exactly Once in both Byzantine and Crash modes (details later). We also give users an option to reduce the level of guarantees in both Byzantince and Crash modes from Exactly Once to At Most Once to give them faster delivery time as will be shown later.

Cheaper Hardware & Idle Machines

Another observation that can be perceived from latest advancements in computer hardware is that it is getting cheaper with every generation. Moreover, many clusters and datacenters have idle machines and commodity hardware that are not used and unutilized for a long time. These idle machines can be used to achieve better reliability for streaming frameworks with very low recovery time to overcome the aforementioned limitations of checkpointing and enhance the reliability of batch and streaming systems in particular.

These observations, among others, have motivated us to create this thesis. In the following sections we will show the objectives, design and implementation of two systems (Guardian

and Warden, details later ...) that helped in overcoming the above shortcomings and achieve better reliability guarantees for both streaming and batch Big Data frameworks

1.3 Objectives based on Observations

In a nutshell, the main objective of this work is to provide generic, multi-framework, flexible, customizable, low overhead systems to ensure the resiliency of both batch and streaming applications running on batch and streaming frameworks, respectively. In this section, we will go over the primary objectives that help it achieve this goal.

1.3.1 Multi Framework Approach

RMSs are deployed on different types of infrastructures involving different hardware (e.g, computers, network fabric) and software (e.g., operating systems). Clearly "the cloud" is hard to characterize exactly, and so in the cloud scenarios motivating RMSs, large differences can be observed in the underlying systems and infrastructure. Similarly, individual applications are affected differently or to a differing extent by various failure types (e.g., host crash failures, malicious processes). In summary, individual deployments and uses of same RMSs are subject to different (a) system models [22] and (b) failure models [23]. The design of our RMS is geared at supporting several of the respective categories as will be discussed.

Some research papers focus on achieving fault tolerance for one framework at a time, but it is more challenging when the target is multiple frameworks at a time. Moreover, having different implementations for different frameworks may end up having some redundant work. Whereas encapsulating all these implementations into one master project help in reusing some of the components that are shared between all these projects, and at the same time without sacrificing the performance of the generic approach (compared a single framework approach) by giving the users the ability to customize and modify the behaviour of the proposed RMS by using a predefined set of APIs to make the behaviour that RMS tailored towards their frameworks objectives as if it was built specifically and solely for their particular framework. This API also helps in the reusability of critical system components for current

and future frameworks by invoking this API during the execution of different applications running on different frameworks to connect to the proposed RMS components.

Also, since we are targeting multiple frameworks at a time, it will be wise to choose a fault tolerance technique that is not already implemented in most of the frameworks. One of the main reasons we choose replication as the fault tolerance technique of choice is the fact that many frameworks doesn't have task replication (with verification) built-in already. We are not aware of any framework that runs redundant execution tasks *and* verifies the data at the level of tasks (not applications). Whereas checkpointing has been widely implemented in many batch and streaming frameworks, so choosing checkpointing as the main fault tolerance technique may not benefit a large subset of frameworks as it is the case in replication.

Furthermore note that we are doing replication at the level of tasks, not the level of data. Some systems provide replication for their data partitions such as Kafka, and their data blocks such as HDFS. Replicating tasks is more challenging than blindly replicating data partitions, because the latter only ensures high availability for the data itself but it doesn't ensure fast recovery time or any form of verification to tasks outputs as it is the case in the former.

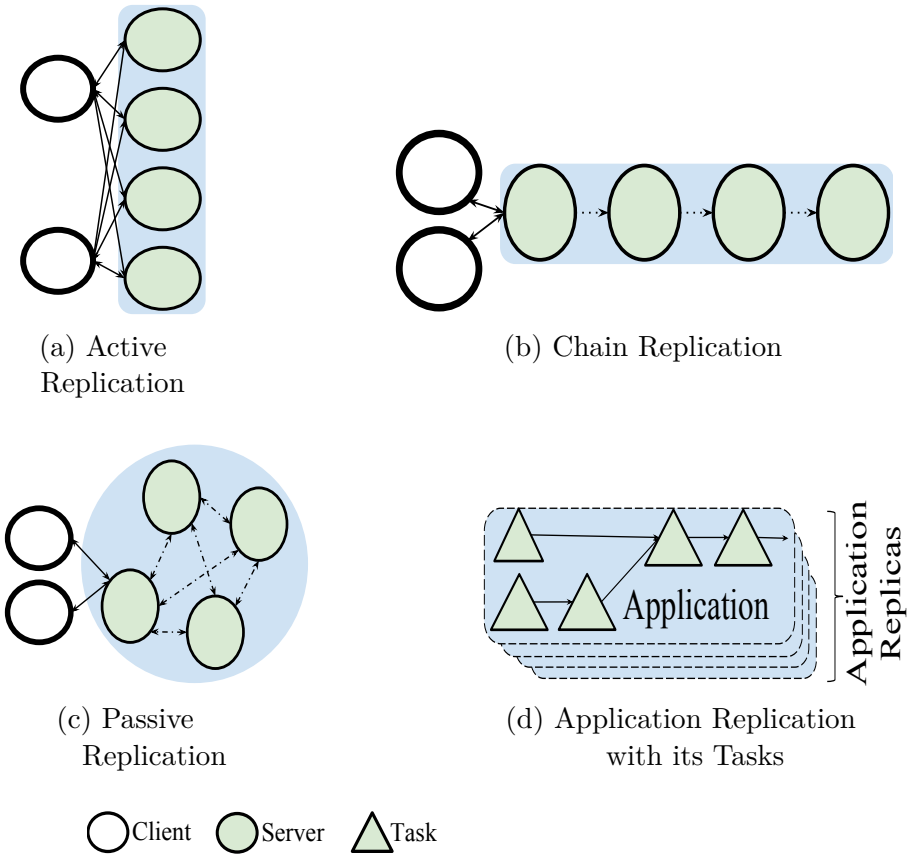
1.3.2 Minimum Recovery Time Possible

The goal of any fault tolerance mechanism is to keep the performance overhead (e.g., latency) as low as possible. Moreover, parameters that can be fine-tuned at runtime by system administrators to trade between guarantees and overhead are always desired in practice. The desire for overall low overhead and customizability further motivates the ability to adjust to different system and failure models, as some models subsume others, yet conservative solutions which are invariably geared at a weak system model (few guarantees) and pessimistic failure model (strong adversary) lead to unnecessary overhead in less austere deployments.

Active replication helps achieve low recovery time for both batch and streaming applications by having at least one other replica that can take over the responsibility of the failed component with minimum fail over time. In our context, this means, if a replica fails, there is at least one other replica that can take over control and continue the flow of data in a way

that makes the failure fully transparent to the end user. This fast recovery time performs better compared to some other fault tolerance techniques such as checkpointing where application has to rollback to the latest checkpoint and reprocess data from that snapshot. This reprocessing of data causes an undesirable delay for the tuples to arrive to the destination.

Figure 1.1 shows some of the most common replication techniques used to tolerate different types of failures. Compared to other replication techniques, our approach (Figure 1.1d) focuses mainly on replicating the application and its tasks.



(a) Active Replication

(b) Chain Replication

(c) Passive Replication

(d) Application Replication with its Tasks

Figure 1.1. Replication Methods

Also Warden tries to achieve minimum recovery time possible not only by blindly implementing replication but also by optimizing this technique to make its overhead as low as possible. This will be discussed in a later section in which we will show that all the data will be flowing through an entity, called the verifier, which has enough data and information to process the replicated streams, make decisions and in some cases forward the data directly

to the following stage tasks without blocking the flow of data between different execution stages and with very minimal overhead as will be shown in the evaluation results.

1.3.3 Flexibility, Customizability & Reusability

Many resilient/reliable systems claim that they can provide reliability to applications running on them by using a certain rigid/firm technique that only works in very certain conditions, and doesn't give the users much flexibility to customize this technique. Our work will provide its users many customizable options to choose from to give the users the best tailored reliable system that works exactly as they need. This is achieved by giving the users the option to choose which failure model they want their system to run on by changing the number of replicas ($r = f + 1$ Crash, $r = 3f + 1$ Byzantine).

Moreover, in case of streaming systems, we give users the flexibility to choose between two consistency models: *Exactly Once* and *At Most Once*. Most users prefer to work in Exactly Once mode in both Byzantine and Crash since Exactly Once doesn't drop or duplicate any tuple. But there are some cases where users are willing to sacrifice some tuples in the favor of receiving the fastest update of the current status of the system. For example, consider a security camera that sends a video stream which raises a security flag as soon as it detects a movement, users of such system are more concerned in getting an alarm about an intruder threat even if they skip some video frames. Another possible application for At Most Once delivery model is sensors in the field. Users who use these sensors are more interested in knowing the most recent sensor value even if some older values dropped from the stream. One other possible usage of the At Most Once delivery is watching a live sports match where viewers are more interested in watching the most recent live stream of the match even if that meant dropping some video frames.

On the other hand, Exactly Once applications vastly outnumber At Most Once applications. Most financial transactions and Stock Market applications can't tolerate dropping any transaction in the stream, hence it needs Exactly Once guarantees. Mission-Critical applications will also require Byzantine Exactly Once to ensure the correctness of the values of the tuples in the stream as will be shown later.

We didn't add At Least Once mode since almost all frameworks already have it built in. In general, most Big Data frameworks already use checkpointing as the main fault tolerance technique. Usually in checkpointing, the delivery guarantee is At Least Once since some tuples will be sent twice after a failure and getting back to the latest checkpoint. Hence we decided to focus on the other two delivery models. In case of streaming systems, combining both failure model and delivery guarantees give users a wide variety of options to choose from:

1. *Byzantine Exactly Once.* Byzantine Exactly Once is the slowest and the most resource heavy between all the options since it requires $3f + 1$ replicas, it needs to compare the actual values of the tuples and it has to ensure that each next stage replica receives exactly one copy of the verified tuple.

2. *Byzantine At Most Once.* In this mode, we need $3f+1$ replicas but it will be faster than Byzantine Exactly Once since some tuples can be dropped in case the end user desires to have the fastest and the most recent update of the data without the requirement of receiving each single tuple in sequence.

3. *Optimistic Byzantine.* In this mode, samples of data are taken between specific intervals (10 tuples, 100 tuples, ...) to make sure the data values between different replicas are as expected. This mode is usually used when users have high confidence/trust in the computations done on the data but the users want to verify *samples* of data during the application runtime.

4. *Crash Exactly Once.* This is the most common mode where the verifier acts as a bridge between sending and receiving tasks to tolerate crash failures of the sending tasks. If any sending replica fails, the tuples from the other sending replica will be sent to both receiving replicas. The values of the tuples are not verified as in Byzantine, instead the sequence numbers of the tuples from both replicas are tracked and the data is buffered in queues to maintain the Exactly Once guarantee.

5. *Crash At Most Once.* This mode is faster than Crash Exactly Once since in Crash Exactly Once there is a need to guarantee that next stage replicas receive one single

copy of each unique tuple, whereas users in Crash At Most Once are willing to sacrifice dropping some tuples in the favor of receiving the most updated changes to the data. It is similar to Byzantine At Most Once but the difference is this mode is faster since it doesn't compare the values of the tuples.

We'll elaborate more details on these modes when we discuss their implementation details in Chapter 4.

Furthermore, we give framework developers the flexibility to change the granularity of the execution unit that has to be verified by giving them the ability to *choose* what to verify. This is done by giving the framework developers the option to insert APIs any place they want inside the task, between tasks, or even after the whole application is done and before submitting the final output to the end user. In other words, reusability of critical system components for current and future frameworks is supported by providing application developers and framework users a common API that can be invoked during the execution of different applications running on different frameworks.

1.4 Background on RMSs

The goal of RMSs is to assign available *resources* (e.g., CPU, memory) on *nodes* to various applications, where a node refers to a single physical or virtual machine made available to the *resource manager* (RM – the front-end of an RMS). We use the term *framework* broadly to refer to any big data computation framework that runs on a *cluster* of nodes and uses a resource manager. A *job* is a specific program or instance of the framework, and an independently executed part or thread of a job is a *task*. For example Hadoop and Spark could be two different frameworks that run on the same cluster using a single resource manager; a specific MR program like word count is a job, and each map or reduce process in the wordcount job is a task.

Mesos [24], Omega [25], Fuxi [26], and Apache YARN [27] are examples of commonly used RMSs. Mesos [24] has an *offer-based* resource manager which provides a list of available resources to a requesting framework, and the framework decides which resources to use or not to use. Omega [25] provides decentralized scheduling which is good for scalability, but

makes it difficult to enforce global fairness and capacity constraints. Omega also expects the different frameworks that run on top of it to be coordinated. These constraints are not ideal for a more open system where a cluster could be shared by unknown entities. Fuxi [26] and YARN [27] are *request-based* approaches which means that frameworks make resource requests to the RMS and the RMS decides which resources are to be used to satisfy which request. Such a request-based approach facilitates location-based allocation optimizations since an application can specify its location preference when making a request as well as modify future requests based on usage and current allocations. Location optimality has a considerable impact on execution latency for big data processing frameworks like Hadoop.

The above RMSs have different takes at fault tolerance. YARN, just like Fuxi, uses checkpointing (of finished tasks). Mesos, Fuxi, and YARN use hot standby to deal with failures in the components of the RMS itself. However, in case a scheduled or running application task crashes, these RMSs delegate handling of such a fault back to the framework. For example in the Hadoop framework, this means restarting the map or reduce task that failed. Other frameworks may employ different strategies. None of the resource managers above handle incorrect responses returned by compromised components (e.g., due to multi-tenancy).

These four RMSs (Mesos, Omega, Fuxi, and YARN) are examples of commonly used RMSs that we can use to build our work on top of. In this thesis we'll show the design of our approach on the two most common and widely adopted RMSs out there: Mesos and YARN. We direct the reader to Mesos paper [24] for more details about its resource offers system, and YARN paper [27] for more details on YARN's design and its main components such as the Resource Manager (RM), the Node Manager (NM), the Application Master (AM) and HDFS main components such as the Name Node (NN) and the Data Node (DN). Figure 1.2 and Figure 1.3 (which is from YARN paper [27]) show the basic design of YARN. Whereas Figure 1.4 (which is from Mesos paper [24]) show the basic design of Mesos.

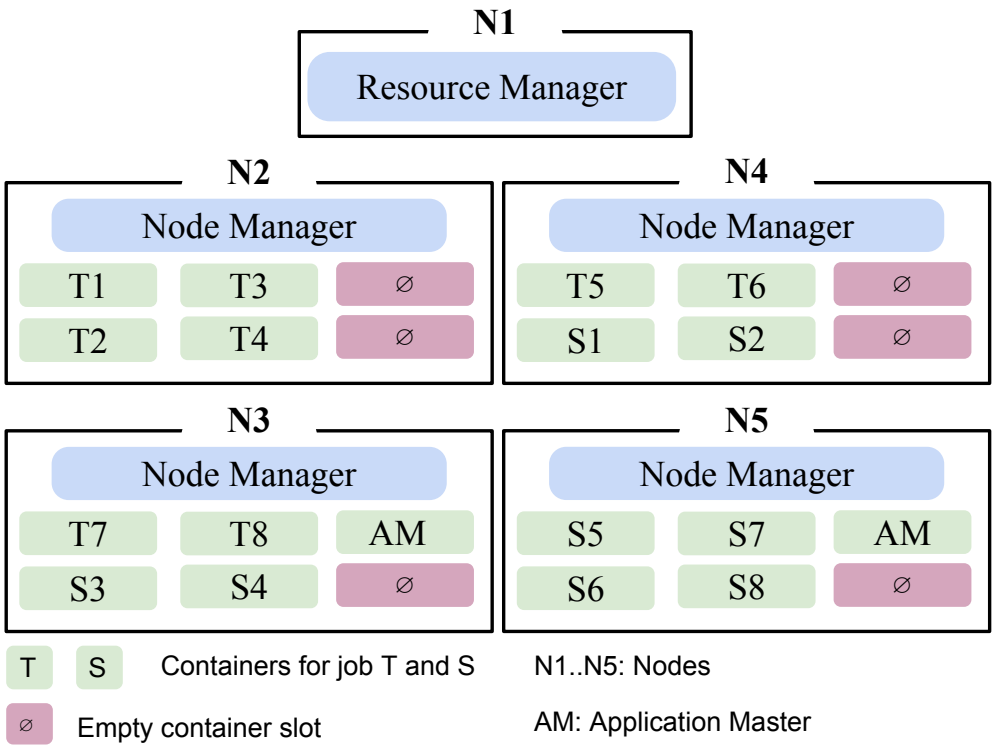


Figure 1.2. Vanilla YARN

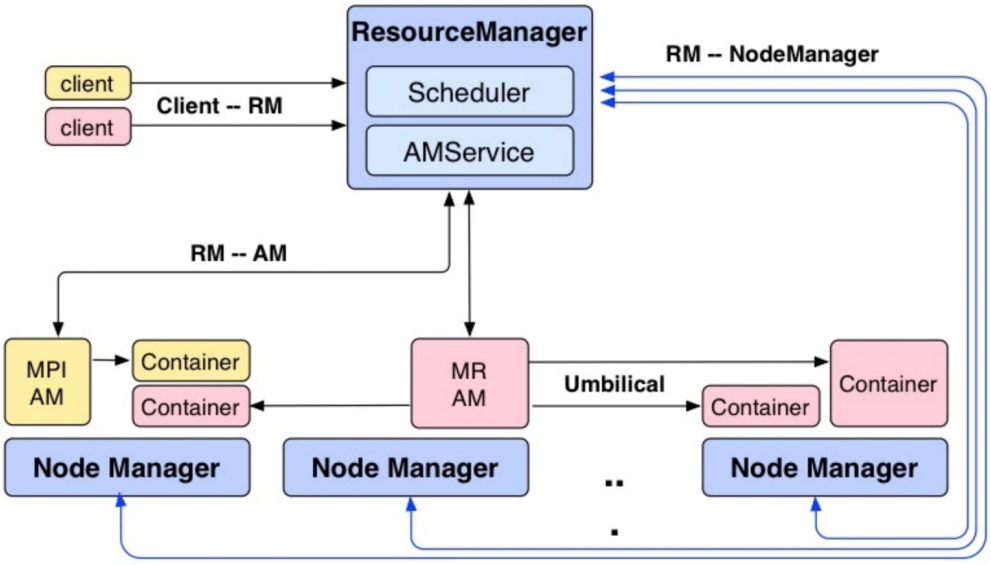


Figure 1.3. YARN Design (from YARN paper [27])

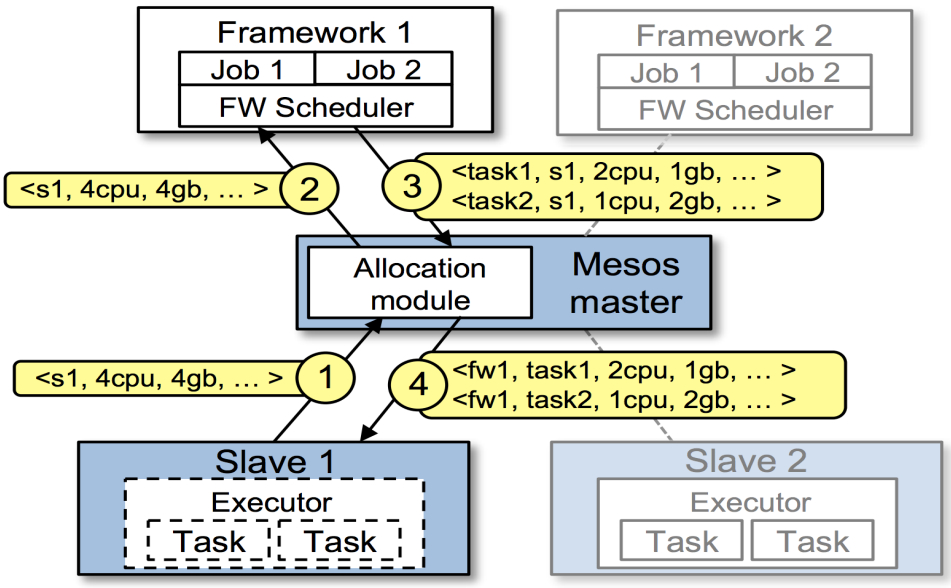


Figure 1.4. Vanilla Mesos (from Mesos paper [24])

1.5 Guardian & Warden

In this thesis, we will present two systems: Guardian and Warden. Guardian is customized towards providing generic fault tolerance support for Batch Processing Systems, while Warden is tailored towards achieving reliability for Stream Processing Systems. The reason for having two independent systems is mainly due to the fact that tackling the fault tolerance problem for Batch Processing Systems involves different challenges than Streaming Systems. In fact, our work on Batch Processing reliability was accepted as an independent publication [28]. Which motivated us to have another independent research project targeted mainly towards Streaming Systems which incur more challenges and ideas than Batch processing systems.

1.6 Thesis Organization

This thesis is organized as follows: Chapter 2 talks about related papers and research projects for the reliability of both batch and streaming systems. Chapter 3 introduces dependable resources for batch processing frameworks through Guardian. Chapter 4 focuses on reliability in steaming systems through Warden. Chapter 5 discusses a new hybrid verifica-

tion approach that takes advantage of both blocking and non blocking verification. Chapter 6 shows the design of Mesos since it is the main competitor of YARN in the cloud computing industry. And finally Chapter 7 concludes and summarizes the whole thesis.

2. RELATED WORK

There has been many research projects and papers on the reliability and fault tolerance aspects of both batch processing systems and streaming systems.

To make it easier for the reader, the related work section is split into two subsections; the related work for batch processing systems followed by the related work for stream processing systems.

2.1 Batch Processing Systems Related Work

One of the closely related works to our research is Google Kubernetes [29]. Done in collaboration with many leading software companies such as Google, Microsoft, IBM and RedHat, Kubernetes helps in orchestrating and managing Docker containers. It also provides replication for Docker containers that run as services. Mesos and YARN containers, on the other hand, does not have any option for active state replication and verification. Guardian tries to fill this gap by providing Mesos and YARN users an option to replicate their containers. Moreover, Kubernetes focuses on the replication of pods which can be an application or a service. Whereas Guardian has finer fault tolerance granularity that tolerates failures at the tasks level. Furthermore, Kubernetes doesn‘t verify the state and the correctness of the replicated tasks, instead it just blindly replicates a service or a pod to keep the service running in case the master has failed.

Active replication in batch processing frameworks has been studied before in several works [19], [21], [30]–[32], but was mainly targeted towards Hadoop. Costa et. al. [19] present BFTMapReduce, extension to MR that tolerates arbitrary and Byzantine faults. To this end, BFTMapReduce runs $f + 1$ replicas of map and reduce tasks in a normal run, in order to be more efficient than typical Byzantine replication. In case the replicas disagree on the output (e.g., a Byzantine failure), the tasks restart. This uses fewer resources in total, but suffers an additional cost of re-execution in the case of any fault. BFTCloud [20] uses dynamic replication techniques to tolerate various failures. Stephen et. al. [21] introduced ClusterBFT, a system to secure computations being run in the cloud for Hadoop. Similar to Guardian, ClusterBFT provides variable replication degrees to satisfy varying fault tolerance

requirements; however, unlike ClusterBFT, which is built for Hadoop specifically, Guardian yields a generic fault tolerance mechanism (through the Guardian foundation) that allows many frameworks to run their applications with configurable fault tolerance properties.

Veronese et. al. [33] introduced an efficient BFT algorithm, which improves upon the complexity of typical BFT methods by making assumptions about trusted services which remain correct even if the machine on which they are installed becomes faulty. Guardian does not attempt to implement such algorithms, but since Guardian does make assumptions about trusted services, there may be space for future development of these ideas. As well, work has been done to increase the performance of replication from an implementation standpoint. Serafini et. al.[34] proposed optimization techniques for replication systems, including speculative completion of some operations (even in the case of failures) and a new BFT protocol requiring $4f$ replicas. These techniques aim to increase the best-case performance of systems using replication. Since the resource requirements of Guardian currently increase linearly with the replication degree, it will be worth considering optimizations for future iterations.

While Guardian focuses on providing fault tolerance at the level of tasks, there are systems which seek to provide fault tolerance at the level of data. For example, UpRight [35] is a library designed to provide Byzantine fault tolerance to always-up services that need to store data and/or state. The case studies provided are HDFS and ZooKeeper [36]. While not directly related to Guardian, this shows that fault tolerance has applications in multiple facets of cluster operations, and not just at the task level. As mentioned in Section 3.1, we use UpRigth to replicate the verifier service to keep it running in the background in case of failure.

Group communication primitives are strong building blocks used for developing fault tolerant distributed systems such as state machine replication (see [37], [38] for a comprehensive lists of these primitives). A group communication primitive ensures certain guarantees among input messages (e.g., totally ordering all input messages) while tolerating certain number of failures. However, there is no need to (totally or causally) order concurrent inputs among replicas of a component in Guardian since components are very simple compared to processes in a group communication system. Consequently, Guardian does not need an ex-

haustive set of primitives for ensuring different ordering guarantees among messages under different assumptions.

We still have an edge over all the related work because, to the best of our knowledge, none of the related work targets fault tolerance at the RMS level and none of the related work targets the three objectives that Guardian has (genericity, low overhead and reusability). One may find some systems that achieve two out of three objectives but not the three objectives together. For instance, it is possible to find a customizable reliable approach that targets one particular framework at a time, but doesn‘t achieve reliability for any other framework other than that particular framework.

2.2 Streaming Systems Related Work

Warden can work on streaming frameworks other than Flink and Samza. There are no particular reasons for choosing Flink, Samza and Kafka in this project as the streaming frameworks in our case studies. But it is worth mentioning few notes about other streaming systems such as Apex [39], Spark [40], Storm [41] and Heron [42].

We started this project with Apache Apex because it is the most recent streaming framework at that time. Unfortunately, the company behind Apex announced its shutdown [43] and hence Apex will not release any new updates or versions and may actually stop supporting current releases. Storm is also a good candidate and a well known streaming system where application topologies consist of Spouts and Bolts. But Storm is slowly being replaced with Heron which is an updated framework of Storm by the same developers of Storm (Twitter). Think of it as Storm++. We tried running YARN on Heron but unfortunately we found out that there were many bugs in Heron integration code with YARN. This actually can still be seen in Heron‘s webpage [44] where they clearly say that Heron on YARN is still experimental.

We tried to avoid Spark, although its popular, due to the fact that Spark has the RDD model [40] in which Spark deals with ‘streams‘ of data as ‘mini-batches‘ of RDDs. So its streaming system is more of a mini-batching system rather than actual tuple-streaming system as in Flink. Hence, we prefered a more abstract streaming system such as Flink

where data streams consists actually of streams of tuples instead of mini-batches of RDDs. Nevertheless, Warden can still integrate with Spark but it will be more like providing Warden services to a batch processing framework rather than an actual streaming framework.

Other related works include Kafka Streams which is a library built on top of Kafka to provide steaming operations on data carried by Kafka. It takes advantage of fault tolerance features inhereted from both Kafka (replicating data partitions between brokers) and Kafka Streams (replicating state stores of tasks progress similar to Samza). However, one major constraint that Kafka Streams has is it has to work on Kafka as the messaging system and not any other messaging system. Our approach can achieve better guarantees than the fault tolerance provided by both Kafka and Kafka Streams combined. Compared to Kafka, our approach tolerates failures not only for the data but also for the tasks that operate on the data. Compared to Kafka Streams, active replication recovery time is much faster than what is provided in Kafka Streams since there is no reinitialization/restarting for the failed tasks on other machines. Needless to say that our approach also targets multiple frameworks (and hence multiple underlying message passing systems) whereas Kafka Streams is restricted to using Kafka as the underlying message passing system, which constraints its use cases to systems that use Kafka only.

Some active projects such as Mesos Marathon, Google Kubernetes has container replication but here 'replication' is stateless which means it can start a second container the same way the first was started. There is no state carried forward or verification or anything of that sort neither having any mechanisms to ensure the correctness of the streams.

Papers like Medusa [45], Arora [46], Borialis [47] are not comparable to our system since they don't target multiple frameworks at a time (i.e. not at the RMS level). They do however use replication for fault tolerance. Dryad [48] from Microsoft is more of a graph processing framework than a streaming framework. It combines computational 'vertices' with communication 'channels' to form a dataflow graph. In case of failures, Dryad restarts the failed vertex/task. It is worth mentioning that Dryad works on YARN out of the box, i.e. it is YARN-native, similar to Samza. StreamCloud [49] is a streaming framework but as the authors mention in their paper, fault tolerance is beyond the scope of there paper and they plan to investigate fault tolerance in their future work. Timestream [50] fault tolerance

approach doesn't cover the full spectrum of failures (Crash to Byzantine) as active replication does. Also application finish time in active replication is shorter since another active replica is running simultaneously. Moreover, the system that they propose in Timestream doesn't work on top of any RMS such as YARN which prevents it from sharing a cluster with other Big Data frameworks such as Spark or Storm. Finally, their approach doesn't target genericity as we do, i.e their fault tolerance approach can't be shared with Samza, Flink ... etc.

This is also the case for Hwang et al. [51] where they proposed a new checkpointing approach for stream processing but as mentioned earlier, checkpointing is not as effective as replication in terms of application finish time and the scope of failures that it can cover. Moreover, their approach doesn't target more than one framework at a time similar to our work. Kwon et al. [52] proposed a checkpointing mechanism where the checkpoints are saved and distributed in a replicated file system like HDFS. Guardian [28] proposed an active replication approach that targets batch processing frameworks only. Warden approach on the other hand targets multiple streaming frameworks which is much more challenging, since providing reliability for several online real time data streams is more difficult than dealing with offline batches of processed data. Zhang et al. 2010 paper [53] was released before YARN, Kafka, Storm and many other Big Data frameworks were released. The paper proposes an interesting approach for switching between active/passive replication. However, the paper doesn't target any genericity or how to target multiple frameworks at a time since most of the frameworks were released after the paper was published, but their approach can be integrated with our work to be used instead of active replication.

We still have an edge over all of the related work due to three main reasons: First, non of the related work targets multiple streaming frameworks at a time as we do. Second, task and application finish times will always be faster than most of the other fault tolerance techniques proposed since there will always be another active replica running simultaneously in the system. Third: The customizability that we give to the users is unmatched in any other fault tolerance approach; not only we give users to choose from the five modes discussed before (Crash Exactly Once, Byzantine At Most Once, ... etc.) but also we give users the ability to *choose* what to verify through a set of APIs that the users can insert in any place

in the task, after the task or even at the very end of the application. To the best of our knowledge, none of the related work targets the three objectives that Warden has. One may find some systems that achieve two out of three objectives but not the three objectives together. For instance, it is possible to find a customizable reliable approach that targets one particular framework at a time, but doesn‘t achieve reliability for any other framework other than that particular framework.

3. DEPENDABLE CLOUD RESOURCES FOR BATCH PROCESSING FRAMEWORKS

The focus of this chapter is on Guardian since it shows our vision for dependability focused on batch processing frameworks. Figure 3.1 shows Guardian vision.

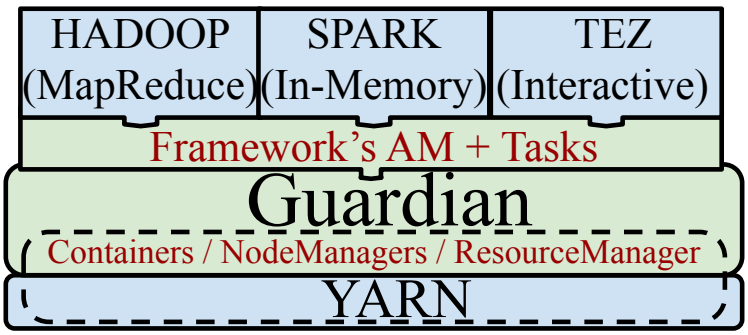


Figure 3.1. Vision of Guardian

In the following sections, we will go over the design of Guardian as will as the details of its implementation and its evaluations results.

3.1 Guardian's Design

In this section, we look under the hood of Guardian, and how it is designed on top of YARN. Figure 3.2 shows the top level design of Guardian, and communications between its core components. In a nutshell, Guardian provides dependable resources to frameworks, by replicating all YARN containers launched during job execution (the AM along with all components/tasks of the job submitted to the system). This container replication is the backbone of Guardian. We assume that the RM (with Guardian Scheduler) is in a trusted tier. The justification for this assumption is that the focus of this work is on providing fault tolerance at the application level (the AM and its tasks), not at the system level (the RM, NM, etc.). The number of application level components are subject to *scale* (as job/cluster size increases), and thus present a more difficult challenge for providing generic fault tolerance with low overhead. We leave fault tolerance level FT to other works, such as UpRight [35] and Apache ZooKeeper [36], through which these singleton components can tolerate failures (e.g., by using state machine replication techniques). It is worth mentioning that Mesos, the

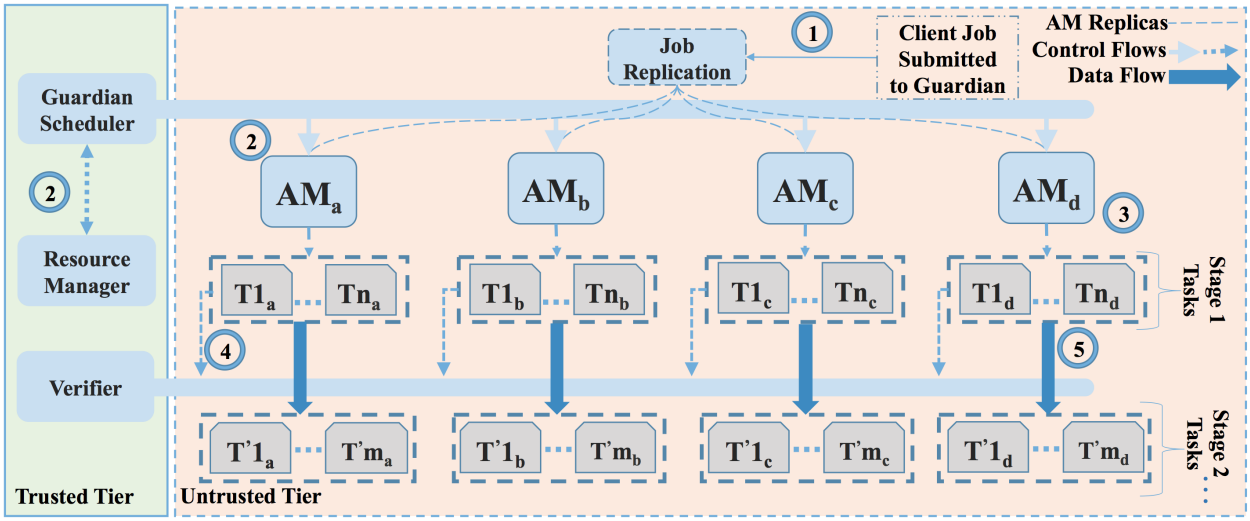


Figure 3.2. Guardian Overview

major competitor to YARN, already uses Zookeeper to have standbys for Mesos Master in case the master fails.

Overview

The main philosophy behind the design of Guardian is to replicate the AM along with all components of the job submitted to Guardian. Whenever a job is submitted to the system, the application submission component in Guardian submits r replicas of the job to the cluster. Hence, r AMs will be initialized in the system. Consequently, each AM replica will launch its own tasks belonging to the AM's job context. At the end of each task's lifetime, each task will communicate with the verifier process through a set of API functions.

Figure 3.2 shows the job replication flow of applications in Guardian. The primary components of Guardian include the job replication component (1 in Figure 3.2), Guardian's scheduler (2 in Figure 3.2), the verifier, the interface for the verifier (4 in Figure 3.2) and the interface for the frameworks (5 in Figure 3.2). The job replication component is actually part of the RM, but it is shown in the figure as a separate component for clarity of presentation.

Guardian's Scheduler

In YARN, the scheduling of tasks to a container for a framework is a *shared* responsibility between the RM and the framework's AM. The RM maintains a list of available containers

residing in different machines. The AM requests containers from the RM and schedules the tasks to the containers obtained from the RM. Each framework running on YARN has its own implementation of the AM to satisfy framework-specific requirements for scheduling. Therefore, vanilla YARN can schedule multiple, even all, replicas of the same task on a single machine if the machine has enough resources. For instance, the schedulers in the Hadoop framework try to assign mappers on hosts that have mapper data already stored to make use of data locality. This is a disadvantage for replication, as a single machine failure can wipe out all replicas of a single task. Guardian ensures that this scenario is avoided.

Accordingly, Guardian requires different replicas of the same task to be scheduled on different hosts, to prevent host failures from affecting multiple replicas of the same task. To deliver this functionality, Guardian contains a scheduler, an intermediate layer between the RM and AMs which enhances the default functionality of the RM. The Guardian scheduler sits between the RM and AMs; it receives information about the tasks and scheduling constraints from the AMs and makes requests to the RM on behalf of the AM (2 in Figure 3.2).

We have also modified YARN's RM so that it does not select containers from the same machine for every r requests, where r is the replication degree. This change guarantees that the Guardian scheduler will have enough containers 'diversity' to schedule replicas of the same task on different machines without compromising the AM's constraints. Since the Guardian scheduler takes care of the scheduling, the AMs will send container requests to it instead of the RM. Correspondingly the AMs will get back task-container mappings from the Guardian scheduler, instead of the RM as in vanilla YARN. The advantage of this is the Guardian scheduler has a global view of the task-container mappings from different AMs, so it will never assign replicas of the same task from different AMs on the same machine. Comparatively, the AMs can't handle this responsibility by themselves because each of them has its own local view of task-container mappings within its own application context.

Guardian's Verifier

Once the AMs receive desired resources from Guardian's scheduler, r AMs execute tasks of the first stage r times. The outcome of each stage must then be verified by a verifier service before being sent to the next stage (4 in Figure 3.2).

In a broad sense, the verifier service is responsible for verifying that the final outputs of a stage are correct before the next stage can start. This is done by comparing the hash values received from task replicas, comparing them and making a decision whether a task replica is corrupted or not. This will be accomplished in collaboration with Guardian's set of API functions (detailed in the following subsection) that can be injected inside a framework task wherever the tasks output need to be hashed and verified.

We use UpRight to ensure the resiliency of the verifier. Briefly, UpRight replicates the verifier process across different machines to ensure the correctness and the availability of the service that it is running (which is the verifier process). The motivation for choosing UpRight over Zookeper for maintaining the fault tolerance of the verifier service is that UpRight deals with a larger spectrum of failures (including Byzantine) compared to Zookeeper. More details about UpRight are not shown here for brevity, but can be found in the UpRight paper [35]. The verifier can run in different modes and can accept a different number of replicas r as will be shown later. The verifier has been implemented as a completely separate process that runs in isolation and is part of neither YARN nor the framework. Since the verifier uses UpRight cluster services to handle failures, it can be placed in either the trusted tier or the untrusted tier.

Interface Towards Frameworks

framework developers can use conventional YARN APIs along with the specific API functions introduced by Guardian to implement applications for their frameworks, and make their frameworks fault tolerant. In the following, we explain some of the API functions introduced in Guardian, and how programmers can use these to ensure various degrees of fault tolerance.

The primary set of functions in the Guardian API is used for sending the outputs of tasks to the verifier, in order to make sure that the output of tasks from different replicas is correctly verified before sending it to the next stage of computation. Additionally, Guardian provides functions for directing the output of tasks as input to one or more replicas in subsequent stages, in case failure does indeed occur. Specifically, Guardian provides the following three major functions:

`Boolean blockingVerify(byte[] output)`: This (blocking) function is used to send a byte array (e.g., hashed value of the task output) to the verifier, wait for the verification to be done and get the verifier response. If this function is used, there should be only one verification call at the very end of the task lifetime before sending its output to the next stage. The return value of this function is a boolean that the developer uses to check if the output of this task is correct and can be sent to tasks of the next stage or not.

`Future<Boolean> nonBlockingVerify(byte[] output,int seqNo)`: Instead of blocking the execution of the task, `nonBlockingVerify` will launch a new thread for each data chunk that needs to be verified. Each chunk of data should have its own sequence number (as an integer) that the verifier can use to compare it with its corresponding chunks from other replicas of the same task.

We note that in both cases, the framework developer needs to decide the input parameters. If the output of a task is small, the developer can pass the whole output (without hashing it); otherwise they can take a hash of the output and pass it as the first parameter. Also note that the verifier is capable of identifying which task belongs to which application context from YARN's TaskID, which is included in the message sent from the task replica to the verifier.

`IOChannel[] ConsultVerifier(int taskID)`: Whenever a task needs to send its output to the next stage, it should consult the verifier for the list of IO channels to which this task needs to send its output to. In the case of no failure, this list consists of channels of next stage tasks within the same AM. In the case of a task failure, the verifier will detect the failure, and when another replica of that task from another AM needs to communicate with the next stage tasks, it will consult the verifier to know which channel/address to use to fetch/send the correct verified output from the previous stage to the current stage.

These are the three major API functions that can be used in different applications running in different frameworks on top of Guardian. Some details about these functions and some other inferior API functions are not shown for conciseness.

3.2 Case studies

In this section, we discuss some of the details of the frameworks that we deployed on top of Guardian. These frameworks are Hadoop, Pig, Tez and Spark.

3.2.1 Hadoop

We have selected Hadoop as our first framework due to its popularity in processing big data applications. For the same reason we assume that most readers are familiar with its basics. We have connected Hadoop's AM with Guardian Scheduler through its API so that replicas of mappers and reducers for the same task from different AM's will be assigned to containers running on different machines.

Hadoop tasks primarily consist of mappers and reducers. In order to enable verification, we modified Hadoop's reducers, by adding some API calls to calculate the hash value of all keys and values that the reducers process. The hashed value is sent to the verifier through the `blockingVerify` API described above, before committing/spilling its final output to HDFS. This hash value is a byte array that is calculated by keeping one hash value per reducer. This value is updated whenever a new ⟨key,value⟩ pair is processed. All ⟨key, value⟩ pair hashes are updated into one variable, so that an error in any of the ⟨key,value⟩ hashes affects the final total value of that reducer's hash. The hash function used is SHA-256. There are many other alternatives to this design by using different set of API functions. For example, instead of using `blockingVerify` once at the very end of the reducer execution, one may use `nonBlockingVerify` to verify chunks of accumulated intermediate keys and values instead of one final hash value. Another possible way is to verify the outputs of the mappers by sending all of the mappers generated ⟨key,value⟩ pairs to the verifier, before sending them to the next stage. In our evaluation, the mappers do not calculate any hash values, and they do not send any hash values to the verifier. Instead, if a mapper fails or the output is

incorrect, this change/failure will propagate to the reducer that is processing that mapper's ⟨key,value⟩ pairs, and the verifier will catch that change/failure in the mapper by noticing that reducer's corrupted hash value. Allowing the verifier to compare the final hash value of the reducers, without comparing any intermediate values generated by any of the aforementioned components, cuts down the verification overhead by eliminating all intermediate comparisons while assuring the correctness of the final output. The verifier is capable of detecting which reducer under which AM replica is causing an inaccurate/inconsistent final result (because it can compare the stored hash values of all replicas). When all hash values are received, the verifier will compare them, and send a reply to all the replicas of that task confirming that their hash values are either correct or incorrect.

Pig

Apache Pig [54] is a data analysis platform which includes the Pig runtime system and the high-level language Pig Latin [55]. Pig Latin expresses data analysis jobs as sequences of data transformations (represented as a DAG), which are compiled to one or more MR jobs by Pig. Pig then submits the sequence of MR jobs to Hadoop for execution. Since data analysis jobs expressed in Pig Latin compile to MR, it was straightforward to adapt Pig to run on top of Guardian.

3.2.2 Tez

Tez [56] is an extensible framework for building high performance batch and interactive data processing applications, coordinated by YARN in Apache Hadoop. An application in Tez is formalized as directed-acyclic-graph of tasks for processing data. Tez uses YARN as its main resource management system. Briefly, Tez applications consists of a set of vertices and edges that are connected together to achieve a certain function. Each vertex has its own processor that will be the main execution unit for any task that is spawned from that vertex. Edges connect the tasks between the vertex using different communication patterns (One-to-One, Scatter-Gather, Broadcast). What makes Tez interesting to our work is that it can be used as a batch processing framework but without the MR paradigm used for

Hadoop and Pig. Since each task in Tez is a mirror of the processor function of the vertex that the task belongs to, the set of API functions mentioned in the previous sections need to be called on the processor output so that any task launched from that vertex uses this processor API; however, the edges that connect the vertices do not need to connect to the verifier. Instead, the edges will retain the normal flow of the data between the vertices after the data is verified by each vertex using our API.

3.2.3 Spark

In brief, Spark [13], [57] is a popular framework used primarily for in-memory, iterative computations. The primary benefit of Spark is the use of the RDD (Resilient Distributed Dataset) abstraction, which represents in-memory data across multiple machines. Spark consists primarily of two components: the Spark Driver and various *executors* tasks, which are simply threads run on each executor, typically perform some transformations on an RDD, but can perform other duties such as reading from files...etc. Spark applications developers need to inject Guardian's API calls discussed in the previous section in order to use Guardian in Spark tasks/transformations. We thought about alternative design options but it turned out that a fully transparent design is infeasible due to several reasons, Spark executors being one of them. These reasons will be discussed in details in Transparency section 3.4. Spark client (in its default configuration) running on YARN will submit an AM, which contains the Spark Driver, to run on YARN. The AM spawns a static number of containers on which to run executors. These executors remain alive for the duration of the application. The Spark Driver then schedules tasks on each executor which operate on RDDs distributed across the memory of the executor pool. This results in a more complicated application as a whole than MR (from the point of view of resource interaction), but still fits the basic mold set forth by Guardian. Spark can run in different modes (Spark batch processing, Spark streaming...etc). In this chapter, since the focus of Guardian is on the fault tolerance of big data batch processing frameworks, we use Spark as a batch processing framework.

3.3 Evaluation

3.3.1 Implementation & Synopsis

As we already explained, Guardian was implemented by extending YARN which is the main resource scheduling component for Hadoop starting in version 2.2. The main criterion that we measured for our benchmark applications is the job completion time of applications with different degrees of replication. Similar to some other works [30], [35], [36], [58]–[60], we used basic fault injection to study the effects of task failures and AM failures at certain times in their progress. The host failures were emulated by stopping all YARN JVMs (NM and containers) of a given machine. The main motivation to use job completion time as our main evaluation criteria is because applications that would use replication in Guardian would do so to avoid the long latencies caused by host failures. As a prime example, mission-critical applications have a high requirement for low latency, and cannot afford to delay computation (i.e., having to deal with host failures and timeouts). However, replication in Guardian comes at a cost. We measured the financial expense in using additional machines and the latency overhead, which mainly comes from creating replicas and verifying results.

Our results show that Guardian improves completion time in unsaturated environments by an average of 68% in the presence of failures, while incurring only a small increase in overhead compared to no replication (about 6% overhead in the highest replication degree).

3.3.2 Testbed

We have deployed Guardian on a cluster of 25 Amazon Elastic Compute Cloud (EC2) machines, each of which has 16 vCPUs, 30GB memory and 320GB SSD. Those 25 (slave) machines are treated as a cluster that is running on an untrusted tier (in the cloud). Each machine is running YARN NM and a Data Node (DN) for HDFS. Another machine is treated as the master machine (trusted tier), on which YARN's RM, Guardian's scheduler, Guardian's verifier and HDFS Name Node (NN) are running.

We specify the size of each container to have 1 vCPU and 2GB memory. The cap that we specify for each NM to work on is 20GB memory. This will result in each node having

a total computation power of 20/2=10 containers. If a machine is fully loaded, it will run all these 10 containers which will result in 20GB load and 10vCPUs load in theory. This will leave 10GB memory and 6vCPUs free in each machine just to give the NM and HDFS DN processes plenty of room to work without any congestion. In a fully saturated cluster situation, there will be 25 machines each running 10 containers which will result in a total of 250 containers running simultaneously.

We have done some experiments where we increased this cap up to 28GB memory (so a total of 14 containers per machine), but this setup led to congestion on CPU resources where most of the time the machine was not responsive since 14 vCPUs were reserved for containers and only 2vCPUs were left for the NM and the DN. So decreasing YARN resources cap will reduce the number of containers assigned to that machine but will assure that there will be no resource congestion, hence better performance for both YARN NM and HDFS DN.

3.3.3 Benchmarks & Applications

We have evaluated Hadoop MR, Pig, Tez and Spark on top of Guardian using the following applications. All of these benchmarks were run on datasets of $\sim 100GB$ size.

Elections: This application is used to count the votes of polls in an election process. The application goes through all election results that are collected from all poll locations in the country, running MR on them to know which president won the elections. This is a good example of a time sensitive big data application, where users are willing to give all their resources and computation power to finish their jobs in a very limited time, and they want to make sure that the results are not corrupted by a malicious user. Usually, elections happen early in the morning, and the results are announced before the end of the business day.

Dictionary: Takes English, French, German, Spanish, and Italian dictionaries for a set of words as an input. Outputs each English word with its translations in these languages in one line per word.

Word Median: Computes the median of records in a dataset.

TeraSort: This application samples the input data (produced by another application TeraGen) and applies MR to sort the data into a total order.

Pig Word Count: The word count application, written as a Pig Latin script takes in text input, tokenizes it and counts the number of occurrences of each word. The word and its corresponding count is written as output. This Pig Latin script gets compiled into a single MR job which is executed on top of Guardian.

Pig Word Count + Sort: This application performs the word count as described above and then sorts the word count output. This Pig Latin job gets compiled into a two stage MR job, where the word count output of the first phase MR job is given as input to the second phase MR job. This type of applications is important because it signifies the importance of frameworks that can run scripts/application that can be compiled and run into multiple MR jobs in sequence.

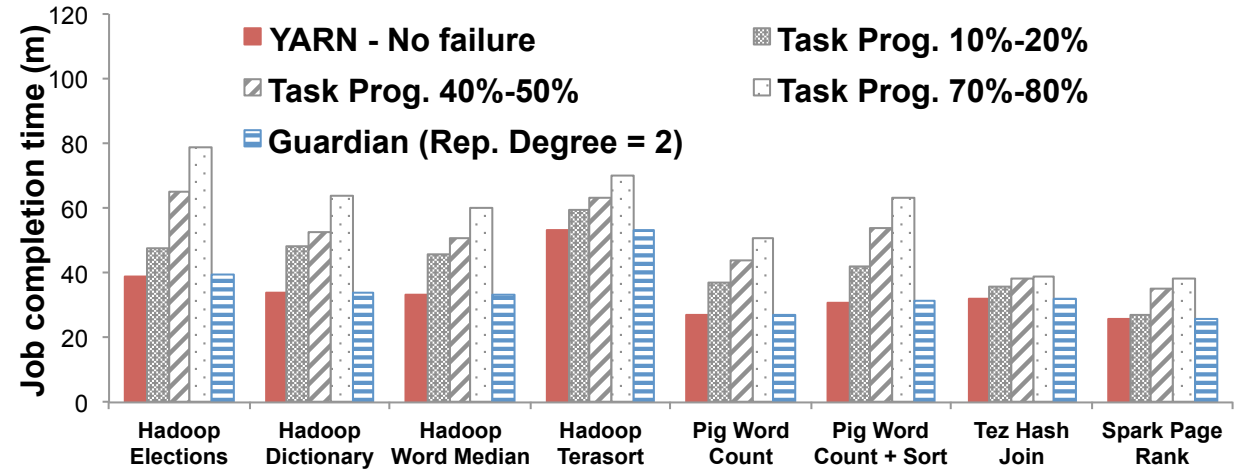
Tez Hash Join: An application written in Tez framework that joins two data sets using Hash Join. The application primarily consists of two input vertices that provides the input streams to the third output vertex that does the hash join.

Spark Page Rank: This is the well-known iterative page rank algorithm (for ranking websites) running on Spark (cluster mode) running on top of Guardian.

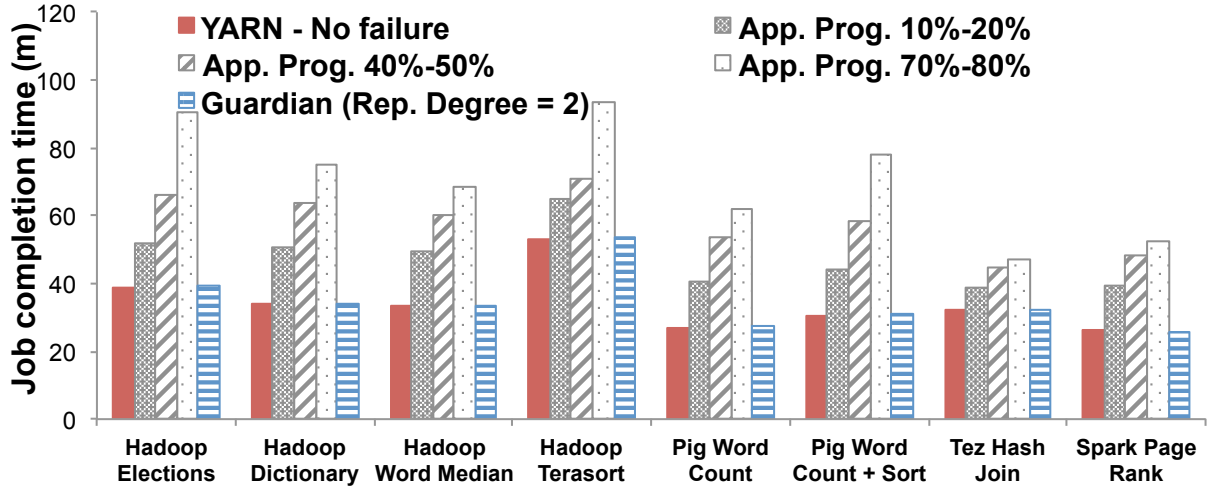
3.3.4 Completion Time

Figure 3.3a and Figure 3.3b show the delay in completion time when there are container failures and AM failures during the execution of the application, respectively. Container failure refers to the failure of the machine that has containers running on it but none of these containers is the AM container. AM failure refers to the failure of the machine that has the AM container running on it. We differentiate the two cases because AMs are *logically* centralized components instead of "one-of-many" components running worker tasks. In Figure 3.3a, 10%-20% corresponds to the event of machine failure that has containers running on it (none of which is the AM container) while the application progress is between 10% -20%. In Figure 3.3b, 10%-20% corresponds to the event of machine failure that has

containers running on it (in which one of them is the AM container) while the application progress is between 10% -20%. (Tez application numbers are scaled down 4 times to fit in the figures).



(a) Container failure during execution



(b) AM failure during execution

Figure 3.3. Effect of a failure on job completion time for YARN and Guardian. The job completion time is in minutes. The extent of application progress when the AM or container is killed is shown as a percentage.

From the figures we can note the following:

- The delay in replication for completion time (in Figure 3.3) in Guardian is negligible compared to the completion time of the same application without replication in the event of container failure or AM failure. This demonstrates the benefits of replication for latency-sensitive computation.

- Container failure and AM failure have more devastating effects on application completion time when the application has done more progress than early stages of the application progress. In other words, the effects of the event of machine crash that happens later in the life time of the application has more devastating effects than having it earlier in the lifetime of the application. This is expected, since YARN only saves the work of the finished containers, and does not have a checkpointing mechanism for saving the work of the running containers. So if a machine crashes while the containers are running on it, all the work that was done by these containers will be lost. Hence, if an application has containers that are working on long running tasks, crashing those containers later in time will have a worse impact than crashing them earlier.

- AM failure entails more delay than ordinary container failure if they both fail at the same time of the application progress. It is important to mention here that in the event of AM failure, all running containers will also fail. After the AM timeout and restart, the new AM that will take its place does not have any information about the containers that were running during the previous AM failure. Hence the progress of all containers that were running at that time is lost and they have to be restarted again. This explains why the AM failures have more overhead than a single container failure.

- Although not shown in the figures, we ran the same Tez Hash Join application on earlier versions of Tez and we noticed that the application failed to recover the AM after the AM failure. After investigating the issue, it turned out that that version of Tez had some problems in failure recovery that was under development. Since Tez was still in an incubating phase, it is not surprising that it faced such defects which were

fixed in later versions of it. But this highlights the importance of Guardian especially for big data frameworks that are still in incubating phase and are not fully stable.

3.3.5 Replication Overhead

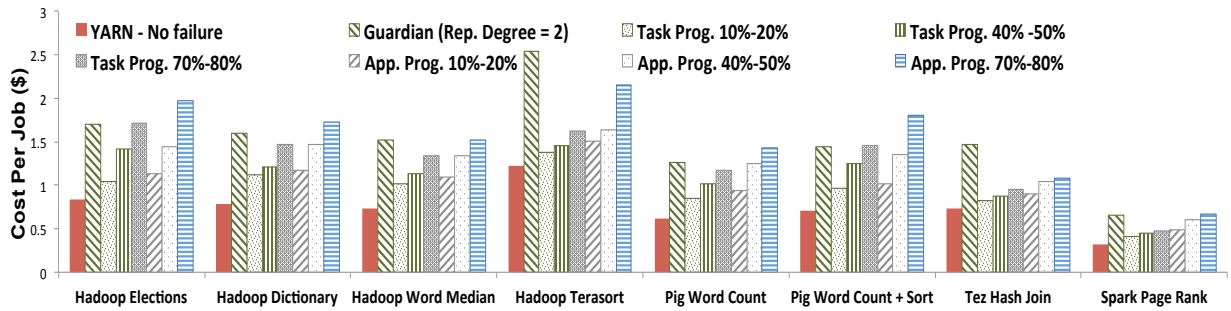


Figure 3.4. Cost of running a job on YARN with and without failure v.s. Guardian with Replication Degree 2

Figure 3.4 shows the financial expense of running specific jobs on Guardian with replication degree 2. We compute the cost for running a job by enumerating the cost of an EC2 instance over the time taken for completion of that job. As expected, the cost of running a job on Guardian with replication degree of 2 is higher than running the job on YARN (which does not make use of replication). In YARN, if the AM container fails after the application has progressed beyond 70%, the cost of the job becomes comparable to the cost of running the job on Guardian. To be precise, the *Elections*, *Dictionary*, *PigWordCount* and *PigWordCount + Sort* applications show higher cost for YARN when AM failures occur after 70%. The analysis also shows that the cost of container failures increases when failures occur during later stages of the application.

3.3.6 Job Completion Time vs Job Completion Cost

To show the trade off between the fast recovery of Guardian and the extra cost that Guardian uses, we compared the job completion time vs job completion cost in Figure 3.5 for some applications. In Figure 3.5 the x-axis is the same as the y-axis in Figure 3.4 which is the cost of job completion, whereas the y-axis in Figure 3.5 is the same as the y-axis in Figure 3.3a which is the job completion time under different task progress failures in YARN. Each of the applications in Figure 3.5 has 5 points in the plot. The points that grow linearly and are in the upper circle in the figure represent job completion times/costs in increasing task failure progress running in vanilla YARN. The points that are inside the lower circle

in the figure represent job completion times/costs for the same applications with one failure but running under Guardian.

The point of the figure is to show that the financial cost for completing a job keeps increasing linearly with task failures, and the increase of the financial cost is proportional with the progress of the task at the time of failure. The exception is in Guardian (with 1 failure) where the job completion time is almost as fast as vanilla YARN without failures but more expensive due to using more resources to run another replica.

This pattern also applies to AM completion time under different AM progress failures (Figure 3.3b) but not shown in another figure (similar to Figure 3.5) for conciseness.

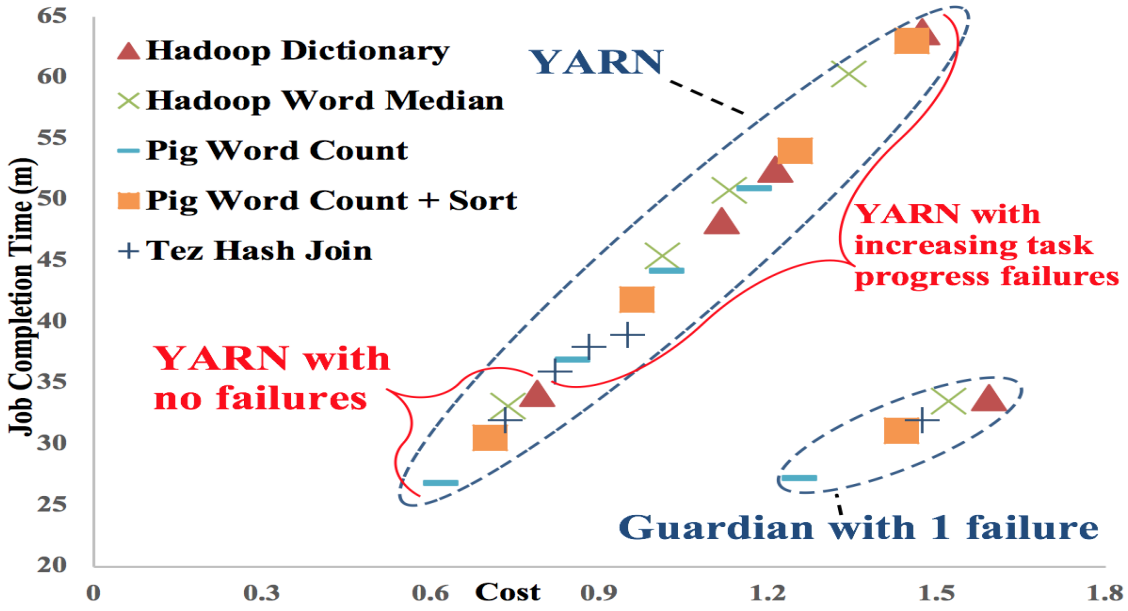


Figure 3.5. Cost Vs Time

3.4 Transparency

Guardian's design was carefully selected from multiple design options, each of which has its own advantages and disadvantages. The ultimate goal of achieving fault tolerance at the level of an RMS is to make the fault tolerance support for the framework as transparent as possible. In other words, the framework will run on Guardian without any changes being made to the framework code. This could be achieved, in theory, by intercepting all the communications between all the tasks, JobManagers, FW Schedulers, AMs, NMs and the RM. Practically though, intercepting all the network I/O and disk I/O traffic will lead to a

substantial overhead that could be avoided if we reduce the transparency of the design, i.e., by moving some of the fault tolerance logic to the framework itself. The reason why we need to intercept the communications in the first place is to be able to verify the output of the task replicas. This is done either by verifying the outputs on the fly, or checking them using a verifier process running in the background.

The other reason why a fully transparent solution is infeasible is the fact that some frameworks are developed in a way that will run multiple tasks inside one container in YARN. This means that the mapping between a framework's tasks and YARN containers is not a one-to-one mapping: each task will run on a separate container by itself without having more than one task in that container. This is true for some frameworks such as Hadoop and Tez; each mapper and reducer in Hadoop runs in a separate YARN container and each Tez task runs in a separate container as well. On the other hand, Storm and Spark are examples of frameworks that launch multiple tasks in one container. In Spark, each YARN container will run an executor that will keep running for the lifetime of the application. The Spark AM will communicate with the executors to launch Spark tasks (e.g. RDD transformations) on these executors, so it is not unusual to have multiple Spark tasks running on one executor (container). Running multiple tasks in one container makes a fully transparent solution infeasible, because even by intercepting all network I/O and disk I/O at each YARN container, that does not mean that we are intercepting the communication of individual tasks, instead we may be intercepting the communication of a group of tasks running in one container. This is also projected into Mesos as well, where multiple Spark transformations or Storm tasks can run in one Mesos Executor (similar to one YARN container).

Finally, another reason why a fully transparent solution is infeasible is due to the fact that starting multiple replicas of the same task (for frameworks that do not have a one-to-one container-to-task mapping) is not possible to be done in a transparent way. To discuss this, please look at Figure 3.6.

In this figure, steps 1 and 2 are already inherited from vanilla YARN. They resemble the steps that an AM takes to launch tasks on containers. Whenever an AM has some tasks that need to run in the cluster, the AM will ask the RM for some containers on which to launch these tasks. The RM will respond back to the AM with a group of allocated

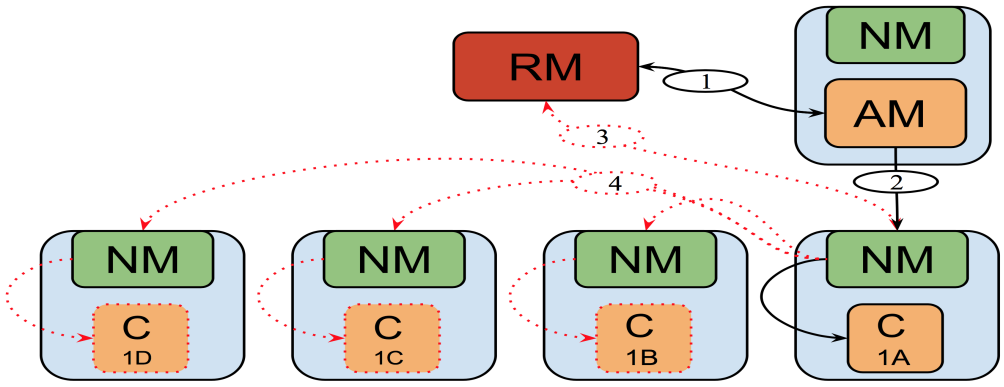


Figure 3.6. Replication through the Node Manager

resources (containers). The AM will then communicate with the NMs of the machines that have these containers to launch the tasks. One design option is to change the behavior of the NM such that whenever it receives a task launch request from the AM, the NM will try to replicate this launch request to other NMs automatically without any intervention from the AM. The NM will do that after it negotiates with the RM for containers on other machines (step 3). Doing this will help replicate the frameworks tasks in a transparent way (step 4) without any changes done to the framework itself and without any intervention from the framework developers. This design may work for frameworks that have one-to-one task-to-container relation, but for others (e.g., Spark) it will not work because the NM can no longer intercept the task launching event (step 2 in Figure 3.6) because the AM in these frameworks will launch tasks on the containers (e.g., executors in Spark) directly without going through the NM. In other words, the AM will launch Spark's tasks (transformations) by contacting the executor directly, so the NM will not have any information about the tasks that need to be replicated. This is also the case for Mesos (not shown in the figure) where some Framework Schedulers can launch tasks directly into their frameworks JobManagers while skipping Mesos Allocation Module all together.

Due to the aforementioned reasons, we conclude that having a fully transparent design for Guardian is infeasible. Hence, we moved to a less transparent solution in which some changes have to be done to the framework itself, through a set of simple API calls (Section 3.1), to have the benefits of Guardian fault tolerance.

3.5 Nondeterminism

Identifying sources of nondeterminism is important for the success of any framework running on Guardian. It is outside the scope of this paper to provide optimal techniques to deal with nondeterminism. Full implementation and evaluation of such techniques will be addressed in future work but here we highlight the sources of nondeterminism so application developers using Guardian are aware of them. The focus of this section is on nondeterminism resultant from replication, not from the explicit use of nondeterministic features such as random numbers or the usage of machine local environment variables (such as time). In all the following frameworks, if the application itself is nondeterministic then replication will also produce nondeterministic results. For instance, most Spark operations take as input a user provided function that can be any arbitrary function. It is important to realize that if these functions (which from the perspective of our system are black boxes) contain nondeterministic statements, the Spark application will also be nondeterministic.

Nondeterminism in Hadoop

There are two main sources of nondeterminism in the stages between the map and reduce tasks; one in the (optional) combine stage (Mappers→Combiners), and the other is in the shuffling stage where values for the same key are aggregated into lists (Shuffling→Reducers). Nondeterminism happens in the former case
(Mappers→Combiners) when values for the same key are being aggregated (combined). This aggregation is nondeterministic because there is no sorting of the values for each key. So each key will have a list of values, but these values could be combined in different orders in different runs. A similar problem exists causing nondeterminism in the latter case (Shuffling→Reducers) where values within the same key are not sorted in a specific order. Instead, different runs of the same MR application could produce different orderings of the same list.

These two primary sources of nondeterminism are not the only sources of nondeterminism in MR programs. Many approaches/papers already exist for analyzing nondeterminism in MR programs [61]–[63]. Xiao et. al. [62] found 5 major patterns of nondeterminism derived from 13,311 real-world MR applications. These five major patterns are: SingleItem,

IndexValuePair, MaxRow, FirstN and StrConcat. The authors do not mention explicitly how to prevent them, but what can be noted is that four of them can be resolved by having a secondary sort stage in the shuffle phase. Only one of them (the IndexValuePair) is not fixed by the secondary sort. Instead, the developer has to be careful in his/her applications not to violate this case. (Briefly, the problem in this case is the data are no longer produced in ⟨Key,⟨List_of_Values⟩⟩ form. Instead, there are multiple lists of values within the same key (e.g. ⟨Key,⟨List_of_Values1, List_of_Values2⟩⟩) so sorting just one of the lists of values will not solve the problem).

Nondeterminism in Tez

Contrary to Hadoop and Spark, Tez suffers from lack of documented resources due to the fact that Tez is a relatively newer platform than both Hadoop and Spark. Hence, Tez nondeterminism has not been addressed before in any work that we are aware of. To begin with, Tez is an execution engine that can run both MR jobs and jobs that are written as DAGs (of vertices and edges). There are three major sources in Tez that we suspect will cause nondeterminism in replication. The first source of nondeterminism in Tez is inherited from any nondeterminism operation that can happen from the MR applications that run on top of it. For example, the nondeterminism in shuffling that we discussed in Section 3.5 in MR will also be there for MR jobs executed via Tez.

The second source of nondeterminism in Tez is in applications that use different types of Tez edges. In Tez DAGs there are three major types of edges: One-to-One, Broadcast and Scatter-Gather. From these three types of edges, the last two could cause nondeterminism. In these two cases, the problem may happen when partitions from different input edges meet at the task input at the next stage vertex. The question here is whether those partitions are ordered in a certain way or are they first-come-first-serve. Note that the 'scattering' (in Scatter-Gather) is deterministic because producer tasks (i.e. tasks of the first stage) scatter data into shards which are gathered by consumer tasks (tasks of the next stage). The ith shard from all producer tasks routes to the ith consumer task. Tracking down Tez's code we found that multiple threads write to the consumers list of inputs without a specific order

(i.e. whenever the input is ready). Thus this list could cause nondeterminism for the *order* of inputs consumed by the consumer tasks.

Finally, the third source of nondeterminism in Tez is due to the Map-Reduce-Reduce (MRR) feature in Tez which allows jobs that have more than one reduce stage to be combined into one MR job. Nondeterminism may occur in this case if the second shuffling phase (between the two reduce stages) is also nondeterministic similar to the first shuffling phase between the mappers and the reducers (recall the first shuffling is nondeterministic for the values as we discussed in Section 3.5).

Nondeterminism in Spark

Spark, while providing an interface more general than Hadoop and Tez, still inherits several design concepts from MR. In particular, many Spark operations require a shuffle, similar to the shuffle found in Hadoop between the map and reduce stages, in order to co-locate data required to compute results on the same machine. Spark has classified the operations that trigger all-to-all communication warranting a shuffle. These include: repartitioning functions on RDDs such as `repartition()` and `coalesce()`, ByKey operations (e.g. `groupByKey`), and `join` operations [64].

Shuffling can cause nondeterminism because the ordering of keys within partitions is based on the order of arrival in the all-to-all communication, which may change based on network conditions. To counter this, Spark developers recommend using certain sorting functions (such as `sortBy()` to completely sort an RDD, and `repartionAndSortWithinPartitions()` to simultaneously sort partitions during a repartition) to provide consistent results across executions. The downside to this is that there is the performance hit involved in the extra sorting stage. As such, it is recommended to code applications to minimize the amount of shuffling wherever possible, and, when performing an operation that triggers a shuffle, to sort the relevant RDD(s) afterwards.

Apart from the shuffling, a few transformations and actions can lead to nondeterministic results due to the use of PseudoRandom Number Generators (PRNG). For example, the `sample` transformation uses `java.util.Random` internally and is thus inherently nondeter-

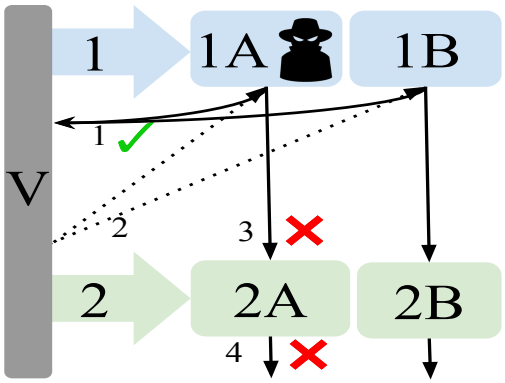


Figure 3.7. Sending Correct Hash Wrong Data

ministic. Similarly `randomSplit`, `randomSampleWithRange` and `takeSample` are nondeterministic operations.

3.6 Advanced Threat Model: Sending Correct Hash Then Wrong Data

One possible scenario that may happen when a malicious user is corrupting/hacking the system is that after a task in a certain stage finishes its work, this user will calculate the hash of the data and send this hash to the verifier. Then, assuming that this hash matches with other replicas of the same task, the user will send wrong/corrupted data to the corresponding task in the next stage. The verifier can‘t intercept this corrupted data (and can‘t even detect it) in any mode the verifier is working on (blocking, non-blocking, hybrid verify (will be discussed in Chapter 5)) because the verifier received correct hash from the malicious user and has no way to detect the corrupted data sent to the next stage by the malicious user. This scenario is shown in Figure 3.7 where in step 1 the malcious user sends correct hash to the verifier then he/she sends corrupted data to next stage in step 3.

This scenario will not cause any problems in our design in the intermediate stages but it may cause problems for the last stage. The reason for this is, for the intermediate stages, next stage tasks will be working on wrong data from the previous stage where the malicious user corrupted the data, so next stage tasks will produce wrong data and, respectively, a wrong hash. When next stage task sends this wrong hash to the verifier, the verifier will compare this hash with its corresponding replicas of the same task from other machines and

will notice that this hash doesn‘t match because the task itself was working on wrong input data from the previous corrupted task.

The reason why this scenario may cause a problem in the last stage only is that the last stage is not followed by any stages, so the malicious user can send correct hash to the verifier and write wrong data to the final output (end user (client), external hard drive, HDFS, …etc.). In the last stage, the verifier has no way to detect what was submitted to the end user because the verifier only deals with the hashes it received from the tasks and not the data itself. Hence the end user may receive unmatching outputs from the same task/application replicas.

There are several solutions that can be applied to tackle this advanced threat model. One possible solution is to send all data from the last stage to the verifier, so the verifier will take the responsibility of writing the final output to disk or show it to the end user (client). Another possible solution is to let the verifier double verify the final output written to disk by running another verification thread after the last stage tasks commit their outputs to disk. Yet another solution is what is called ALL-to-ALL: Send all data from the stage before the last stage (stage n-1) to all replicas in the last stage (stage n). Verification will be done inside the task itself in that stage. Finally, one last proposed solution is to make the tasks save their output to a trusted/reliable storage where the verifier can independently do the hashing on them and compare the hashes. That way even if the sender sends corrupted data it will be detected.

All these solutions have drawbacks but will deal with this advanced threat model problem. For example, sending all data (which could be in the size of GBs or TBs) from the last stage to the verifier through the network will cause a huge communication overhead and the verifier will take much longer time verifying the data compared to just sending and verifying the hashes (which is in the size of KBs).

In our evaluations in this paper, we assumed that the threat model will always send wrong hash to the verifier in case the data is corrupted. But in the case of a more advanced threat model as the one mentioned here, Guardian's users can deal with it by sending all the data itself to the verifer (instead of sending the hashes). They can send the data all

in one chunck using blockingVerify, or in multiple chunks using nonBlockingVerify (as was discussed in Section 3.1).

4. DEPENDABLE CLOUD RESOURCES FOR STREAMING SYSTEMS

This chapter talks about achieving our cloud reliability vision for streaming systems. This is done through a system called Warden (Figure 4.1).

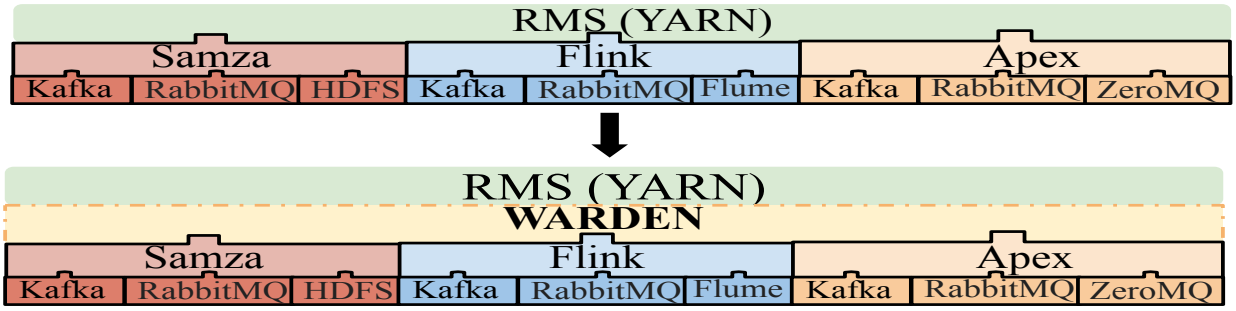


Figure 4.1. Vision of Warden

In simple terms, the general sequence of operations in Warden is as follows: Tasks process input data, tasks send the data processed to the verifier instead of next stage tasks (as it normally would) then the verifier send the verified data to next stage tasks to continue the normal flow of the application. The system relies heavily on which verification mode it is operating on: Crash Exactly Once, Crash At Most Once, Byzantine Exactly Once and Byzantine At Most Once. These modes were detailed before in Section 1.3.3.

To achieve this, we introduce a Multi-Phase protocol where each phase is explained in details in each of the following subsections. Figure 4.2 helps to show these phases in sequence of operations (from left to right).

4.1 Multi-Phase Protocol

The communication between the frameworks tasks and the verifier can be described in a 5-phase protocol:

Phase-1 : Initialization

Users submit their jobs to Warden normally as they do with vanilla YARN. Warden reads the Warden 's configuration file to know more about the properties of the submitted job. In this configuration file Warden knows how many replicas are desired by the user, and other job properties like Exactly Once guarantees or At Most Once guarantees, ... etc.

Warden then launches r AMs in different machines as if the user actually submitted r different jobs to YARN‘s RM. It is important here to launch the AMs in different machines so that if one of these machines fail with the AM in it, the other AM will continue working normally and it won‘t be affected by the first AM failure. If we leave this job to vanilla YARN‘s RM, then all AMs could end up running on the same machine. If that machine crashes or gets disconnected from the network then all AMs have to restart which affects recovery time drastically. This same modification is also done in later stages of execution, where Warden ensures that each task replica is launched on a different machine to avoid having multiple replicas of the same task running on the same machine.

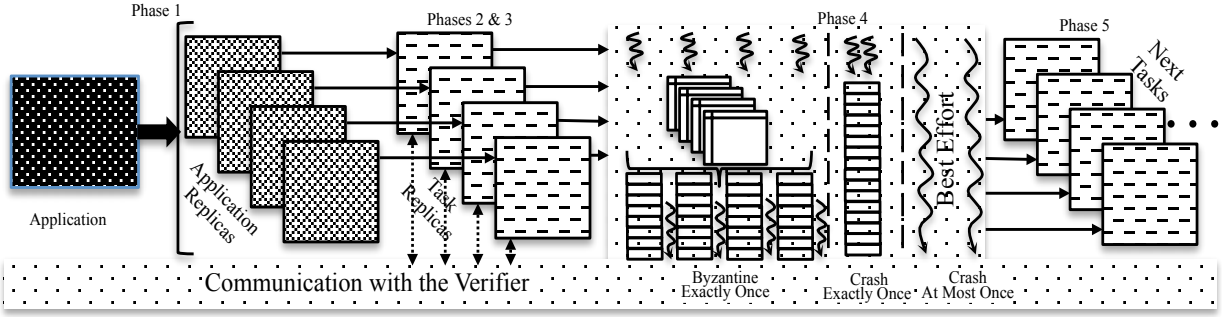


Figure 4.2. Multi-Phase Protocol (Byzantine At Most Once and Optimistic Byzantine are not shown)

Phase-2: Handshake

Once a framework task starts, the task communicates with the verifier to inform the verifier about some of this task information such as: which replica from which stage from which application from which framework does this task belong to, the hostname of the machine that this task is running on and the port that this task will be sending or receiving data from. The verifier stores each task information locally and use the tasks IDs to compare tuples received from tasks streams with their corresponding tuples from other replicas of the same task that belongs to the same application and the same framework.

Phase-3: Start-Sending Signal

Frameworks tasks shouldn‘t be allowed to send their tuples to the verifier until the tasks receive Start-Sending signal from the verifier. This means the tasks won‘t start processing data until they receive this signal. The verifier sends this signal when it has made successful handshakes with r task replicas, so all replicas of the same task has launched and ready to

send tuples to the verifier. Otherwise, if each task replica start right away without synchronising with other tasks, then this could lead to a situation where some replicas of the task has already started sending tuples whereas other replicas of the same task are still initiating and maybe didn‘t even finish the Handshake phase.

Phase-4: Multimodal Verification

This is the most important phase where the verification is actually done. The process of verifying the tuples is different according to which mode (discussed in Subsection 1.3.3) the tuples wish to be verified against:

Byzantine Exactly Once: This is the most time consuming mode where each single tuple is verified against $3f + 1$ other tuples from replicas of the same task. Once a majority is formed, the agreed-on tuple will be sent to its corresponding next stage tasks. The verification in this mode is done according to the following: once the verifier receives a first-seen tuple from the sending task, the verifier will save this tuple in a hashtable where the key is the counter (sequence number) of the tuple and the value is the actual tuple itself. Then the verifier will receive tuples with the same sequence number from streams of other replicas of the same task and it will save each tuple from each stream in its corresponding hashtable. Once a tuple forms a majority between the hashtables, the tuple will be pushed to its corresponding linked blocking queue from which it will be sent to next stage tasks which will be discussed in Phase-5.

Byzantine At Most Once: In Byzantine At Most Once, the verifier waits until the current tuple forms a majority from the four replicas streams then send it to next stage tasks. During that time, many tuples could have arrived from different streams. These tuples will be dropped in favor of sending a more recent tuple to the next stage. After sending the current tuple (after it forms a majority), the verifier will ‘grab‘ the next tuple with the highest sequence number (most recent tuple) at the current time interval, wait for it to form a majority then send it, and so on. Note that there is no queueing in Byzantine At Most Once but the values of the tuples will be compared with each other from at least 3 different streams.

Optimistic Byzantine: In this mode, data is not sent from the sending tasks to the verifier then from the verifier to the receiving tasks; instead sending tasks send the tuples directly to the receiving tasks but every few tuples (10 tuples, 100tuples,...etc) they send sample of the data (1 tuple) to the verifier so the verifier makes sure that the computation is correct up to that point. Users of this mode have high confidence in their data computations, but between time to time they still want to make sure that the computation is correct up to that point *but* without the overhead of sending the tuples to the verifier then to the receiving tasks as in Byzantine Exactly Once and At Most Once modes.

Crash Exactly Once: In Crash Exactly Once there is one major queue where tuples from both streams write into. Once a tuple with sequence number x is received, it will be put right into its position in the queue. It is possible to check whether there is already a tuple on that position or not but to speed up the process the tuple will be put into its position even if it overwrites the current value since both of them supposed to have the same value. In Crash the verifier doesn‘t compare the values of the tuples as in Byzantine, instead it assumes that the tuples values are correct and the verifier main concern is to place the tuples into their correct position in the queue according to their sequence number.

Crash At Most Once: As stated in Subsection 1.3.3 there are some cases where users are more interested in receiving the fastest most recent update in the stream and are willing to sacrifice dropping some previous tuples to do that. In Crash At Most Once there is no queueing (buffering), instead everytime the verifier wants to send a tuple to both replicas of the following stage tasks, the verifier will take the most recent tuple from both streams without checking its value or its sequence number. In most cases, the verifier won‘t drop any tuple and the system will actually achieve Exactly Once guarantees although it is running as At Most Once. But the verifier is not doing any queueing for the tuples so if a group of tuples arrives quickly in a very short time interval (burst), these tuples won‘t be queued, instead the last tuple (most updated value) will overwrite the tuple value that will be sent to next stage.

Phase-5: Send to Next Stage

After the tuples have been processed/verified in Phase-4, they will be sent to next stage tasks according to which mode they were processed in. For example, in Byzantine mode,

after a tuple is verified and is part of a majority (3 out of 4 in case r=4), the tuple will be pushed into its linked blocking queue. The threads which are responsible for sending the tuple will keep checking these queues for any new verified tuples to send them to their corresponding tasks in the next stage. The reason behind using a linked blocking queue data structure is it is a FIFO data structure (so order is reserved) and it is unbounded because the number of tuples to be held in the queue is unknown prior to the queue start, and blocking queues in general have the advantage of the take operation: this take operation blocks temporarily until data is available in the queue which is much more efficient than a busy wait in which the queue will be repeatedly polled until a tuple is available.

Another example is the Crash At Most Once mode where tuples are sent right away to next stage tasks without putting them in queues and without even comparing them to other tuples from other replicas of the same task. So sending to next stage is different according to which mode the tuples are being verified with.

4.2 Granularity

As mentioned in Subsection 1.3.3 one of the main objectives of Warden is to achieve flexibility and customizability. This customizability is achieved not only by giving the users an an option to choose between the five verification modes stated above (Byzantine/Crash Exactly Once/At Most Once, Optimistic Byzantine) but also by giving users the options to *choose* what to verify. Some users prefer to have a very fine granularity verfication model in which the output of each single task from each stage is verified, others may prefer to skip the verification of some tasks in favor of reducing the verification time overhead, other optimistic users are satisfied with a coarse grained granularity where it suffices to verify the very last output of the whole application before submitting the output to the end user in favor of further reducing intermediate verification overhead between different task stages. There is no way to predict the level of granularity preferred by different users of Warden, hence we decided to give the users the option to choose what to verify by providing them a set of APIs to interact with Warden according to the level of granularity they see fit for their applications.

API	Description
`Optional<int> portNumberFromV` **`handShake`** `(string taskDetails, string hostName, int portNumber)`	`portNumberFromV: Optional` return value that could later be used if this task is going to receive data from the verifier. It is `Optional` because not all tasks are receiving tasks, like first level tasks. This port number will be used later in **`receiveFromV`** API call. `taskDetails:` is space separated: "TaskUniqueID TaskLevel ReplicaNumber ApplicationUniqueID Framework" There is no particular reason for this order other than an ordered convention to help Warden parse the string according to a certain order. `hostName` and `portNumber:` are for the current sending task. We are assuming all tasks are sending tasks, if this is not the case then these values will be `Optional` as it is the case of receiving tasks.
`int portNumberToV` **`startSending`**`()`	`portNumberToV:` is a port in the verifier machine assigned for that task to send tuples to the verifier through. This is a blocking call that will only exit once the task receive a start-sending signal from the verifier as described in Phase 3. It is not `Optional` because the framework developer will not use this API call unless this is actually a sending task.
`Optional<int> slowDown` **`sendToVerifier`** `(string tupleAsString, int sequenceNumber, string taskIDOfNextLevel)`	`slowDown: Optional` value to slow down the speed of the fastest stream. `string tupleAsString:` there is no restriction on tuples to be strings, they could be changed to `byte[]`. `sequenceNumber:` local counter value of the tuple. `taskIDOfNextLevel:` The verifier already know which port in which machine to send this tuple to by only knowing the `taskIDOfNextLevel`, because all tasks give the verifier all their details in the handShake phase.
`string tupleAsString` **`receiveFromV`**`()`	If this is a receiving task, then Warden already saved the `Optional<int> portNumberFromV` locally from the **`handShake`** API call. Framework developers will place this API call exactly where the framework expects to receive tuples as if the framework is vanilla without Warden.

Figure 4.3. Warden 's API

Due to these reasons we introduce a set of API calls detailed in Table 4.3 to help developers achieve the best level of granularity for their frameworks and take the most out of Warden. In the table, the `handShake` and `startSending` APIs correspond to phases 2 and 3 respectively. `sendToVerifier` is called after receiving the `startSending` signal from the verifier, so it is done between phases 3 and 4. `receiveFromV` is called in next stage tasks after the verification is done so after phase 5. There are more details about these APIs not shown in the table for conciseness.

Another objective of Warden is to target multiple Big Data frameworks (Subsection 1.3.1). Each framework developer knows the best place to insert the API for each phase of Warden inside their framework. For example, the `handShake` API can be inserted in the launching code of the framework's task. Since the framework developer is the one who wrote this booting code, he/she is the one most knowledgeable to know where to insert the `handShake` API inside their framework task code.

4.3 MultiThreading

We have implemented the verifier in a multi threaded fashion where each phase in each task has its own independent thread. The reason why there is a new thread for each phase in each task is to prevent blocking the task execution sequence between different phases in the same task. For example; the Handshaking phase is done as soon as the task has started

running in its assigned machine, whereas the Start-Sending phase is done later in the same task after all the task built-in initialization and running code have been processed. If we block the task code in the Handshaking phase then the task has to wait for the verifier response before the task continue running its built-in initialization code which will cause unnecessary delay.

Also having a multi threaded design help in setting up the pace for receiving and sending threads in the verifier. For example, in the case of Byzantine Exactly Once mode, received data are written to linked blocking queues after they get verified, then sending threads send verified tuples from these queues as soon as they are ready. This is helpful in cases where sending and receiving threads are not working at the same speed.

4.4 Scalability & Flow Rates

The scalability of the verifier depends on the available resources of the machine that the verifier is running on. For example, the size of the queues that will hold the tuples until they get sent to next stage tasks can increase as long as the verifier can use more memory from the machine (until these tuples get sent to next stage, at which point they will be removed from the queues). Similarly, the number of threads the verifier can spawn depends on how many CPU cores the machine has and how does the operating system and the JVM deal with them. It is out of the scope of this work to optimize how does the JVM or the operating system schedule or pin threads between different CPUs, or how to optimize queues and hashtables memory allocations.

After a certain threshold (once the machine memory utilization crosses a certain value) and if the streams flowing through the verifier has different speeds (flow rates), the verifier can slow down the fastest stream speed by sending a signal to the fastest replica to slow down. This signal is actually the optional reply (return value) of `sendToVerifier` (shown in the API table) that the sending process will get back from the verifier after it sends tuples.

Usually this return value is 0 (no slow down necessary). Otherwise, if there is a need for the fast process to slow down then this slowing down value is measured by:

$$(Counter(f) - Counter(s)) * \delta$$

Where $Counter(f)$ is the tuples counter of the fastest stream, $Counter(s)$ is the tuples counter of the slowest stream, and δ is the average time between two consecutive tuples in the slowest stream. This will give the slow stream enough time to catch up with the fastest stream. Another alternative design is to set the return value to be the time between two consecutive tuples in the slowest stream (In other words, unify the time between two consecutive tuples (flow rate) to be as fast as the slowest stream). Once the fastest sending replica receive this value, it will stop sending tuples for that amount of time, then continue sending normally. For example, in the case of Byzantine Exactly Once, if one of the four replicas is very fast whereas the other three replicas are slow, the queues and the hashtables of the fastest replica will become very large in size to hold the tuples that still need to be verified against the three other slow streams. Without controlling the speed of the streams, an OutOfMemory exception could occur due to the fastest replica tuples occupying too much memory waiting for the tuples from the slow streams to arrive.

4.5 Evaluation

In this section we will go through the evaluation details of Warden and the applications that we ran to evaluate it, along with the details of the cluster used to run the evaluations.

4.5.1 Case Studies

We ran two frameworks (Flink and Samza) on top of Warden to evaluate it. For conciseness, we removed many details about these frameworks and the way they deal with failures when they‘re running in standalone mode and on top of YARN mode.

It suffices to say that Flink has Exactly Once guarantees whereas Samza has At Least Once guarantees. Flink can run as standalone and on top of YARN but Samza can’t run

standalone, instead Samza requires two things to function: an execution engine to run Samza's tasks on (such as YARN), and a message passing pipeline to carry the streams of data for Samza's tasks (such as Kafka). Samza uses YARN and Kafka out of the box, so from now on, when we say Samza, we actually mean Samza running on top of YARN using Kafka as the message passing system.

In both Flink and Samza, when a machine fails, the framework works with YARN's RM to restart the containers in a new machine, which is an expensive procedure especially for streaming frameworks because it will cause a long failover delay overhead.

For more details about the Flink and Samza please refer to their papers [65], [66], respectively.

4.5.2 Testbed & Synopsis

We have deployed Warden on a cluster of 21 machines, each of which has 20 CPUs and 120GB memory. 20 (slave) machines are treated as a cluster that is running on an untrusted tier (in the cloud).One machine is treated as the master machine (trusted tier), on which Warden's verifier is running on. Each container in the slaves machine is limited to 1 CPU and 3 GB memory, which totals to 16 container per slave machine (we left 4 CPUs idle to prevent resources contention). The input size for all applications is $\sim 2TB$. Each measurement in Figure 4.4 has been run three times where error bars show the difference between runs. We used YARN(Hadoop) version 2.9.1, Flink version 1.6.2 and Samza version 1.0.0.

To evaluate our applications deterministically, we made the input streams bounded. The main criterion that we measured for our applications is the arrival time of the last tuple in the stream. One other possible way to evaluate our work is to make the streams unbounded and measure the arrival time of the nth tuple in different modes. Either way will show the effectiveness of our approach in the presence and absence of failures. Similar to some other works [30], [35], [36], [58]–[60], we used basic fault injection to study the effects of task failures. The host failures were emulated by stopping all JVMs running in a given machine.

4.5.3 Applications

To test Warden we ran it with an application from Flink and another from Samza.

Flink Application

To evaluate Flink on Warden we use a Twitter application where it reads real time tweets from certain users and does some processing on the tweets (edits, merges, collect statistics,...etc). To make the evaluation accurate, the input was changed from real time tweets to a predefined size input stream of tweets and use this input across different evaluation runs. Otherwise the evaluations won't be accurate since the number of real time tweets by a certain user in a certain time window is different from one run to another. We changed the tweets size and the processing done on the tweets to simulate larger checkpoints ($\sim 100MB$ to $\sim 500MB$ in the figure).

All Flink evaluations have been done while Flink running streaming mode. Flink can run in batch mode where it will work as a batch processing framework but Flink's batch processing mode doesn't have any built-in fault tolerance technique (neither replication nor checkpointing). In other words if a task fails in batch processing mode it will restart from the beginning. Since the focus of this work is on streaming frameworks, the evaluations are done on Flink streaming mode.

Samza Application

This application merges real time edits streams from three wikipedia edits streams (wikipedia, wiktionary, and wikinews) into one major edits stream, parse these edits for further processing, get some statistics out of the information parsed from the streams and finally output these statistics to the end user. Both the input and output streams are Kafka streams. Instead of processing real time edits from real time streams, the input stream was changed to be a limited predefined input stream of edits to the application just to make the input deterministic for all evaluation runs. Otherwise the evaluations won't be accurate since the number of real time wikipedia edits in a certain time window is different from one

run to another. We generated a json file of wikipedia edits that Kafka reads and inputs into the Samza application. We create a synthetic checkpoint inside the statistics task. The checkpoint consists of the actual edits that have been done to a certain article within a time period. To change the checkpoint size we change the input file to have larger size edits (e.g multiple paragraphs added to the same wikipedia article) or smaller size edits (e.g one sentence added to different wikipedia articles). This can be shown in the figure where the checkpoint size changed from $\sim 100MB$ to $\sim 500MB$.

4.5.4 Applications Finish Times & Checkpointing Overhead

Note that the overhead of saving frequent checkpoints (state stores) in both Flink and Samza consists of two things: One to save the checkpoint locally then another one to save a checkpoint replica to another machine (distributed file system). In the runs that uses Warden we disabled checkpointing to see the pure overhead of Warden. Moreover, there is another overhead after loading the latest state-store checkpoint which is redoing the work from the latest checkpoint in the stream itself. In case a task fails in Samza, it will restart from latest Kafka offset that was recorded. This offset may not be accurate because the task could have processed some data after the latest offset checkpoint. Recall that Samza is At Least Once, so if a task fails in this application, the number of wikipedia edits and the statistics collected by the application could be inaccurate because some partitions have been processed twice due to the At Least Once policy. For example, in one of the test runs that finished without any failure, the output consisted of a total of 800000 tokens. Repeating the same run and crashing a machine with 10 seconds checkpointing interval ended up delivering 803809 tokens to the end user. So a total of 3809 tokens were delivered twice. As mentioned before, Warden can overcome this problem with two modes: Crash Exactly Once and Byzantine Exactly Once. On the other hand, the evaluations we ran on Flink was when Flink was running in Exactly Once mode which prevents sending redundant data but in the cost of higher checkpointing overhead as will be discussed later.

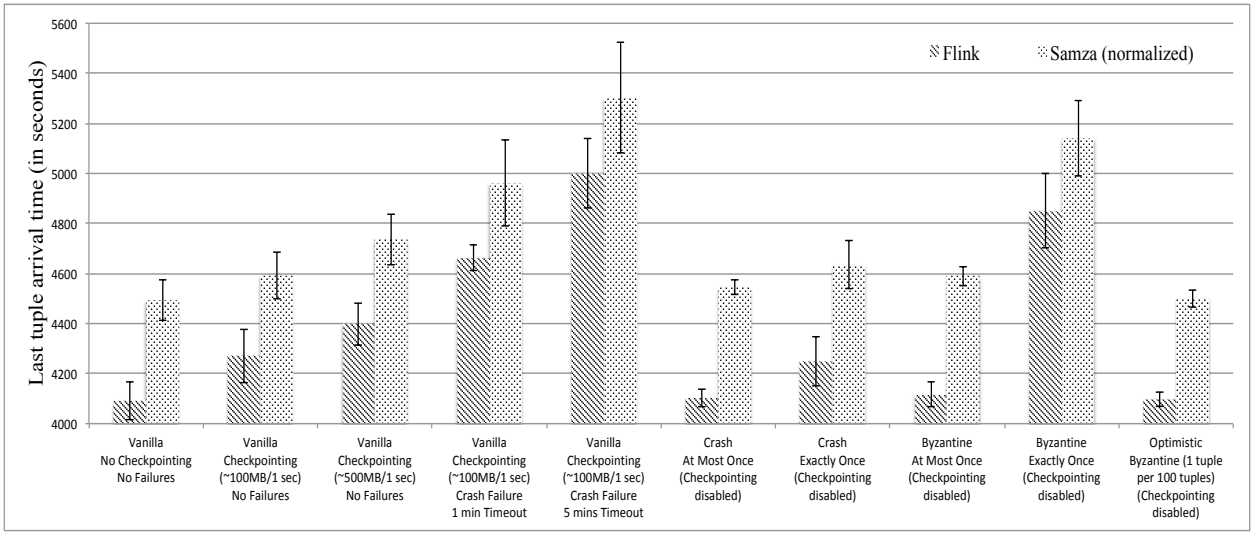


Figure 4.4. Flink and Samza applications on Warden

Figure 4.4 shows the application finish time in different runs for both Flink and Samza with and without Warden. Samza numbers were normalized to fit in the figure. Note that the y-axis in the figure starts from 4000 seconds. Notes from the figure:

- Crash Exactly Once finish time (with checkpointing disabled) is faster than Samza with checkpointing enabled when the checkpoint size is $\sim 500MB$/sec and almost as fast as the finish time when the checkpoint size is $\sim 100MB$/sec. And it is faster than both in Flink (discussed later). The overhead of Crash Exactly Once is due to the buffering that occurs inside the verifier. There is no buffering in Crash At Most Once that is why it's overhead is lower.

- Increasing the checkpoint size will induce more time overhead. Even though both Flink and Samza strive to optimize their checkpointing strategies, yet the overhead will be noticable for large size checkpoints over long periods of time. The problem is both Flink and Samza has to save the large checkpoint locally and send it to another machine for backup for each single task. So the overhead includes both saving the checkpoint to disk (twice) and sending the checkpoint through the network. These overheads won't be noticeable for small runs or for checkpoints of small size ($\sim 1KB$/sec).

- As expected, increasing the timeout will increase the recovery time. One may naively conclude that reducing the timeout will fix the problem, but the fact of the matter

is Samza‘s timeout doesn‘t belong to Samza in the first place, this timeout is part of YARN configurations which are shared between all frameworks and applications that run on YARN. So reducing the timeout may be out of Samza‘s users control since it may interfere with other applications and frameworks running on the cluster. Moreover, even for frameworks like Flink, there is a good reason why most frameworks and RMS systems have default timeouts of 5 or 10 minutes but not all the way down to few seconds. The reason for this is to reduce false positives by mistakenly marking a task as a failed task where the actual reason for the delay in heartbeat signals is due to some other unrelated reason such as resource congestion. For example, memory overload, busy CPU, slow disk or network congestion can cause delays for task heartbeats to reach to the master machine. Please refer to Subsection 1.2 for more details about the timeout problem.

- Note also that in case of failures, the overhead doesn‘t only include waiting for the timeout and then loading the remote checkpoint but also it includes the work that wasn‘t saved in the latest checkpoint and has to be redone after the latest checkpoint is loaded.

- Byzantine Exactly Once has the highest overhead between all the models. The reason for that is in this model, a majority of 3 out of 4 (assuming $f = 1$ in $r = 3f + 1$) values have to match before sending it to next stage. As mentioned before, we are saving the values in hash tables then after the values are verified they are saved again in sending queues to be sent by the sending threads. Note that in Crash Exactly Once mode there is no verification of values, instead it sends the tuples sequentially to next stage replicas without saving the tuples in hash tables for verification as in Byzantine mode. That is why Crash Exactly Once overhead is lower than Byzantine Exactly Once.

- Crash At Most Once, Byzantine At Most Once and Optimistic Byzantine are almost as fast as vanilla since there is no buffering in the verifier and some tuples being dropped to keep the stream updated to the most recent tuples. As discussed before, these three modes may not be suitable for some critical applications such as financial transactions

or stock market trades, but they could be useful in some other applications like reading the most recent correct values of field sensors, security cameras or fire alarm systems.

- One noticeable difference of the application finish times between Flink and Samza (without Warden) is that it takes Flink relatively slightly longer time to make checkpoints in both $100MB$ and $500MB$ checkpoint sizes compared to Samza. One possible reason behind this is Flink is running in Exactly Once mode, there could be some delay added to the application finish time due to the overhead of injecting barriers in the stream and the overhead of the alignment steps that Flink does to ensure Exactly Once semantics. Apart from that, both Flink and Samza have relatively similar application finish times (compared to their vanillas) in all the five modes (Crash/Byzantine Exactly Once/At Most Once, Optimistic Byzantine) since checkpointing is disabled in both of them once Warden is running.

4.5.5 Replication Overhead

It is expected that running replicas of the same application will require twice the resources in case of tolerating Crash, and four times the resources in case of tolerating Byzantine. This is inherited from replication itself. However, as mentioned in Subsection 1.2, hardware is getting cheap, including decent memory and CPU chips. Moreover, many datacenters have idle commodity machines that can be utilized to run replicas of the applications; particularly streaming applications where fast delivery is crucial for end user.

Nevertheless, as mentioned before in the objectives of Warden in Subsection 1.3.3, flexibitliy and customizability is one of the primary motivations behind designing Warden. Users of Warden are not forced to use 4 replicas of the application to tolerate Byzantine, instead they can reduce the number of replicas to tolerate Crash. In fact, users can disable Warden all together in case the cluster is congested (during daytime busy hours) then enable Warden when the machines in the clusters are mostly idle (during midnight, early morning hours). In general, the more resources used the better guarantees the system can achieve.

Compared to other reliability techniques, such flexibility doesn't exist in any other fault tolerance method in the literature. For example, checkpointing is already built-in almost

all streaming (and batch) processing frameworks. If any user who is using a framework that has checkpionting as the main fault tolerance technique wishes to enhance the speed of their applications (whether there were failures or not) they won't have any option to do so, even though they could have twice or even 4 times the resources of the cluster sitting idle, they won't have any option to utilize these idle resources in any way to enhance their fault tolerance guarantee, for example from At Least Once to Exactly Once in Samza, or to achieve faster recovery time.

4.5.6 Problems with Evaluations

While running some evaluations, we noticed some unexpected behaviour from some applications, particularly those that run on top of Flink.

One of these unexpected behaviours is how different versions of Flink deal with YARN's AM failure. For instance, in some runs we noticed that the Flink's AM failed to restart. Figure 4.5 shows this case when Flink was tested on YARN, Flink version 1.6.0 failed to timeout and restart the job when the machine that has Flink's AM crashed. Although this was fixed in the next version of Flink 1.6.2, yet it shows the importance of Warden for unstable frameworks or frameworks that are still in incubating phase (under development).

It is also worth mentioning that while evaluating some fault injection scenarios we noticed that Flink reports that the application finished correctly and the web interface shows that the application completed successfully. Where in fact, after checking the tasks logs, it turns out that the tasks actually failed due to some exceptions/errors related to the machine crash. This is a perfect example of Byzantine failure in which an application informs the user that the application finished correctly where actually it failed.

We are confident that such problems will be fixed in later versions of Flink. What makes such Byzantine failures hard to detect is the difficulty of finding the root cause of the problem. Since this problem happens sometimes when Flink runs on YARN, the root cause could be related to either Flink itself, or YARN itself or both of them.

In general, running different versions of systems on top of different versions of other systems introduces unpredictable problems and sometimes Byzantine failures. For example,

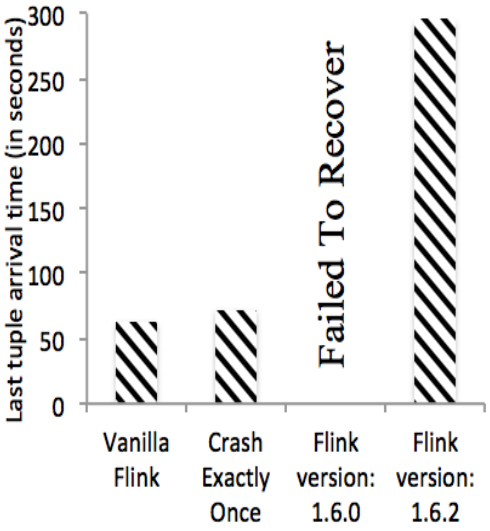


Figure 4.5. Failed To Recover

in our case, running the same version of Flink or Samza on top of different versions of YARN gives different behaviours when the machine that has the AM container fails: In YARN versions earlier than 2.4.0: all running containers will be killed if the AM container fails and the new AM has to restart the containers that were running once the old AM failed. But this behavior is different for YARN versions 2.4.0 and above. In such versions, YARN tries to keep running containers alive once the AM failed and will try to connect the new AM to the old running container in an attempt to minimize the damage of AM container failures. Furthermore, in more recent versions of YARN (version 2.6.0 and above), they changed the method used to measure the 'attempt failure validity interval' in YARN. This interval indicates when should YARN kill a failed application after the failed application exceeds the maximum number of application attempts it is allowed to have within a certain time window.

This is another advantage of replication compared to checkpointing. Checkpointing depends on the actual implementation of checkpointing inside the framework itself. This implementation may not be effective when running different versions of systems on top of each other. Replication, on the other hand, runs another replica of the application. This replica is completely independent of the version of the underlying system (YARN) or the version of the system running on top of it (Flink or Samza or Kafka) and how different versions of these systems interact with each others.

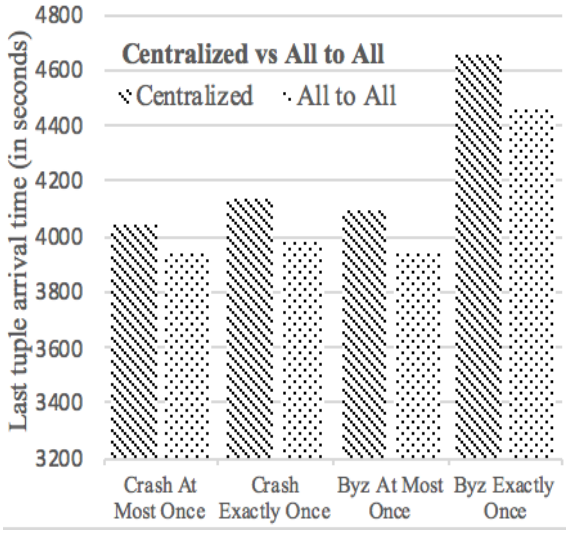


Figure 4.6. Centralized vs All to All

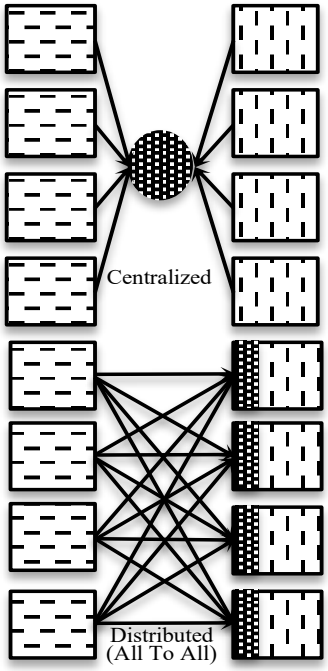


Figure 4.7. Verifier Modes

4.5.7 Centralized vs All To All Verification

The verifier can run in two modes (shown in Figure 4.7): centralized and distributed (All to All). In centralized, the verifier is running in a single trusted machine where all producing tasks send their output to, then the verifier send the verified data to next stage consuming tasks. Whereas, in distributed mode all replicas of producing tasks send their output to all consuming tasks (hence the term "All to All"). There is a 'mini' verifier inside each consuming task that works as a 'gatekeeper' where the verification logic is applied. The results shown in the previous sections were while Warden was running in centralized mode and under the assumption that there is a single centralized verifier running in a trusted machine. We focused our evaluations on the centralized mode more since it will show the worst case scenario that Warden can run on (in terms of latency). We compared between the two modes by running a streaming benchmark in which the results are shown in Figure 4.6. It is expected that All to All mode will be faster than having a centralized verifier since it'll remove the man in the middle (the verifier machine), but it may not be appropriate for all applications were developers prefer to decouple verification logic from the tasks processing logic completely (e.g. to reduce resource consumption in the processing machines.)

5. HYBRID VERIFICATION

In this section, we'll discuss two major problems in blocking verify and non-blocking verify and propose a solution that fixes these drawbacks. Figure 5.1a shows the normal sequence of operations in both blocking verify (numbers) and non-blocking verify (letters). Stages of execution are numbered 1, 2, 3. Each stage has tasks: 1A, 1B, 1C, 1D where 1 is the stage number and A is the replica number (ID). Tasks B, C, D not shown to preserve space. The column V on the left stands for the Verifier. Blocking verify works normally by not sending data to the next stage (step 3) unless current stage tasks get the verification signal from the verifier (step 2). Non blocking verify, on the other hand, can send data to the next stage directly (step b) without waiting for the verifier signal (step e). Note how the sequence of the steps is different from blocking verify to non-blocking verify, particularly when will the verifier send its response back to stage 1 tasks (step 2 in blocking verify, step e in nonblocking verify).

5.1 Blocking Verify Problem: Slow

Now consider the following scenario for blocking verify (which is shown in Figure 5.1b): assume two different Byzantine failures at stages 1 and 2 in two different execution paths, task 1A sends wrong data to the verifier whereas tasks 1B, 1C, 1D send correct data. Since it is blocking verify, all tasks will halt and wait for the verifier signal to either proceed normally to the next stage or make one of the replicas send its correct output to the execution path that has the corrupted task (the task with the wrong data). In this particular example (Figure 5.1b), the verifier will detect that task 1A is corrupted and orders one of the correct replicas (task 1B) to redirect its output to the next stage tasks of Task 1A. Similarly, in the next stage (stage 2), if there is another task from another execution path (task 2B) that produces wrong output, then that wrong output will be detected normally by the verifier and the verifier will again request one of the correct replicas that formed a majority to send its correct output to the next stage task (from task 2A to task 3B).

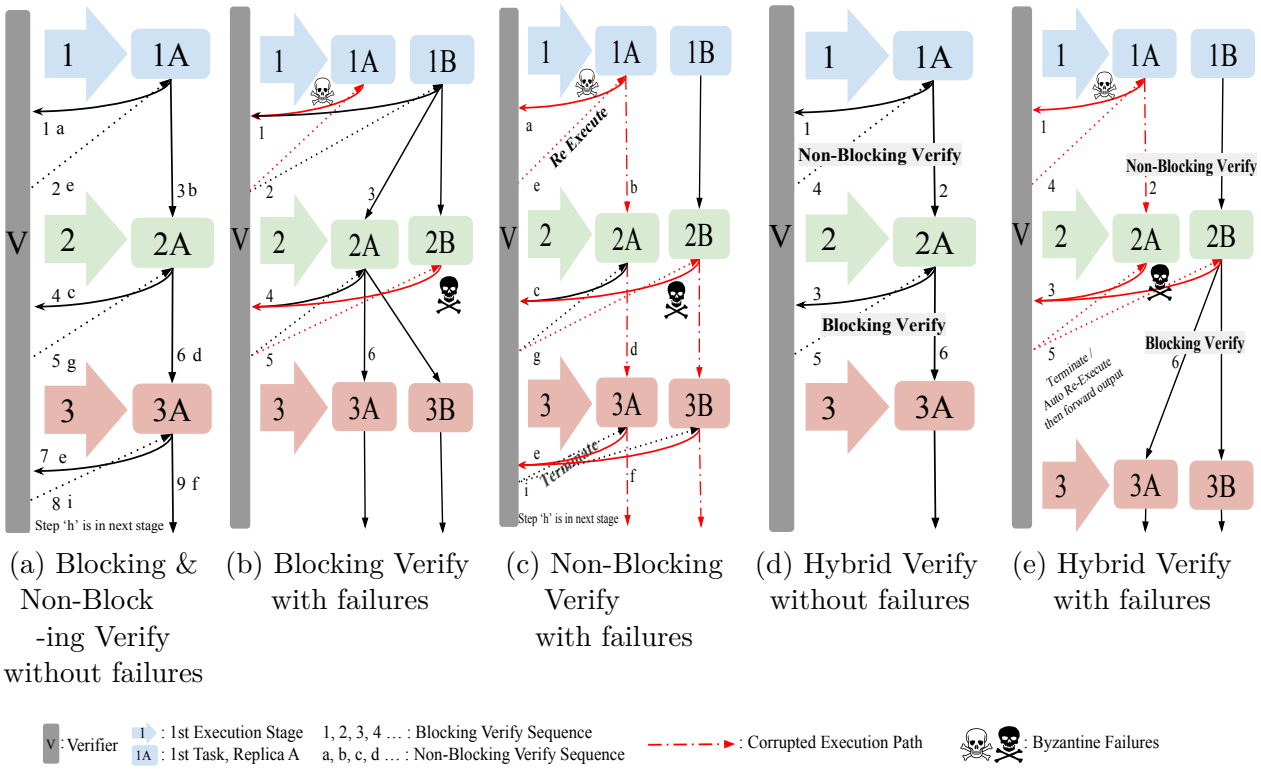


Figure 5.1. Blocking, Non-Blocking and Hybrid Verify

The advantage of this is it will always ensure that next stage tasks are working on correct inputs, but the main disadvantage here is that it is slow compared to non-blocking verify. The slowness will have worse effect when the common case is there is no failure in any replica, i.e there is no need to block and wait for the verifier because most likely the execution is correct (maybe based on history of no failures for some time, or the user has good trust in the execution environment) which encourages the move towards non-blocking verify.

5.2 Non-Blocking Verify Problem: Spread of Corruption

Let us take the case for non-blocking verify in Figure 5.1c. In this figure a dashed line represents a corrupted execution path. Task 1A in this case is corrupted, it will send its corrupted output to the verifier and then directly to the next stage task (task 2A) without

waiting for the verifier response. Now task 2A is working on wrong input so even if we assume that task 2A is functioning correctly then it will produce wrong outputs because the input itself is wrong. In this same stage (stage 3) another corrupted task (2B) is functioning improperly (although it received correct input for task 1B) and hence it will produce wrong output and send that output to the verifier and then directly to the next stage task (task 3B) without waiting for the verifier response. At this time, two tasks in stage 3 are working on wrong inputs: task 3A and task 3B. There is no way to get a majority in this case in stage 3 even if the tasks themselves are working properly because they received wrong inputs from stage 2 (*call this problem 1*).

At this time (step e in stage 1 in Figure 5.1c), the verifier will detect that task 1A produced wrong output and will act accordingly (either make task 1A re-execute or better to let task 1B forward its output to task 2A). The main problem here is that later stages are already working on wrong data, producing wrong outputs and forwarding their corrupted output to their next stages (spread of corruption). The verifier at this point have no choice but to terminate all later stages (because they can't form a majority due to spread of corruption) and re-execute the stages followed by the last stage that formed a correct majority. What makes the problem even worse is that the verifier has to know 'how' to kill a task. Recall that different frameworks are written in different programming languages and these frameworks tasks are not all the same type, the task could be a whole JVM, a thread, a C++ binary …etc. This will make the verifier job harder because not only it has to track all the tasks affected by the spread of corruption but also the verifier has to know how to kill/terminate these tasks (step i in Figure 5.1c) (*call this problem 2*).

There are different ways to technically terminate a task such as making the verifier send a signal to the JobMaster (JM)[1] to restart that particular task (need to change the JM code), or making the verifier itself restart that task (which means that the verifier has to know the launch command somehow from the JM). These kind of solutions are very undesirable because it will reduce transparency drastically.

[1]↑JM is just an abstraction for the single master entity that all tasks communicate with for job coordination, it corresponds to FW Scheduler in Mesos and AM in YARN.

Yet another consequence problem caused by the spread of corruption is that at this point stage 3 tasks and the following stages tasks may still be running, and since they received wrong inputs they have to be killed somehow. Technically this is difficult for the verifier to achieve since it has to maintain a track of all process IDs of the later stages and again it has to know how to kill them (*call this problem 3*).

Note that this spread of corruption phenomena doesn‘t happen in blocking verify because in stage 2 in Figure 5.1b all task replicas where already working on correct inputs (task 2A received correct input from task 1B instead of the corrupted output from task 1A).

5.3 Hybrid Verify

Until now we can see the main advantages and disadvantages for both blocking verify (correctness over speed) and non-blocking verify (speed over correctness). The question here is: Is there a way to do verification that achieves both correctness and speed at the same time? The answer to this is a proposed method called Hybrid Verify; from its name it is a hybrid between blocking and non-blocking verify, shown in Figure 5.1d. The main idea of it is: Never use non-blocking verify in two consecutive stages unless first stage output is verified. In Figure 5.1d, stage 2 tasks can‘t use non-blocking verify because non-blocking verify has already been used in its previous stage and first stage outputs are not verified yet. Lets see if the spread of corruption can happen in hybrid verify, this is shown in Figure 5.1e, task 1A will send wrong output to the verifier and send the output directly to the next stage tasks (task 2A) since it is working in non-blocking verify mode, then all tasks in stage 2 will use blocking verify because the verifier is not done yet verifying the non-blocking verify of stage 1. Note that task 2A will produce wrong output even if the task is not corrupted itself, because it is working on wrong input in the first place. Now assume that task 2B is also corrupted and will send wrong output to the verifier but it will not send that corrupted output to the next stage (stage 3) because it is working in blocking verify mode. At this point, the verifier will notice that it can‘t form a majority for stage 2 because 2 replicas match each other (tasks 2C and 2D) and the other replicas do not. At this point, the

verifier will order stage 2 tasks to restart and what will happen is the following: task 2A will produce wrong output again because it is working on wrong input originally. Tasks 2B, 2C, 2D should produce a matching output, and if they couldn‘t produce matching outputs the verifier may restart them again maybe on different machines, racks, datacenters until they form a majority (3 out of 4) (*solves problem 1*). Note that in blocking verify, since the task is actually still in ‘execution‘ mode while this task is waiting for the verifier reply, it is easier for the verifier to kill that task without ordering the JM to restart the task and without the need for the verifier to know the launch command for that task. It is enough in blocking verify to send back a kill order to the waiting task for that task to terminate. The JM will automatically notice that that task has failed and will auto re-execute that failed task without the need to change the JM code which will increase transparency of our solution compared to the case of non-blocking verify (*solves problem2*).

This way the verifier doesn‘t have to track down any later stages tasks that are working on wrong inputs and doesn‘t have to rollback and re-execute from the last correct outputs (that formed a majority) (*solves problem 3*). This will remove the disadvantage of non-blocking verify and will keep its advantage over blocking verify because it is not as slow as blocking verify.

The common case: Note that in hybrid verify, the common case is to use non-blocking verify at each stage, because most likely the verifier will send its signal back to the current stage replicas before next stage replicas finish working on its data. Think of big data frameworks where each task works on input data chunks up to 2GB whereas the verifier is dealing with hashes where each hash is relatively much smaller in size (around ∼1KB). So most of the time, the hybrid verify will be working as non-blocking verify so it will be faster than blocking verify, but in the less frequent cases where next stage tasks finished working before previous stage tasks got the verifier signal then hybrid verify will switch to blocking verify mode to avoid the catastrophic spread of corruption phenomena.

5.3.1 Testing Hybrid Verification

To test the effectiveness of the Hybrid Approach, we ran it on three applications from three frameworks (Hadoop, Tez and Spark).

We tested Blocking Verify, Non Blocking Verify and Hybrid Verify in both with and without failures scenarios and the results are shown in Figure 5.2.

- Without failures, Blocking Verify takes longer to finish than both Non Blocking Verify and Hybrid Verify. This is expected since Blocking Verify will never execute next stage tasks until current stage tasks are fully verified. Whereas, The Non-Blocking (Optimistic) verification is always faster than both Blocking where there are no failures. This is due to the fact that Non-Blocking mode will never hold and wait for the majority to form even if the previous stage haven't yet formed a majority (such as in the Hybrid approach).

- Without failures, Hybrid Verify is almost as fast as NonBlocking Verify since in most stages the verification of the previous stage is finished before the processing of the current stage task is done.

- With failures, Blocking Verify will take longer time than Blocking Verify without failures since it will need more time to restart the corrupted stage tasks and do their verification again.

- With failures, in case of Spread of Corruption, both Blocking Verify and Hybrid Verify are faster than Non-Blocking Verify with failures but not as fast as Non-Blocking Verify without failures. This is because Non-Blocking Verify restarts the application in case of Spread of Corruption whereas both Blocking Verify and Hybrid Verify can continue from the last correct stage.

- With failures, Hybrid Verify can detect where is the faulty stage since it switched the following stage to Blocking Verify. Once Hybrid Verify detects the corrupted stage, it will restart its tasks which will add more time to the total finish time by an amount of restarting and re-executing the corrupted tasks and redoing its verification (recall the

verification is done in parallel to next stage execution, so next stage execution time does not increase the overhead).

- Note that application re-execution never happens neither in Blocking Verify nor in Hybrid Verify since both of them have ways to detect failures and recover from them 'on the spot' while Non Blocking Verify may continue processing corrupted data which may lead to spread of corruption and restarting the application.

- It is worth mentioning that in this Non-Blocking implementation, in case it is discovered in later stages that the data is corrupted, then the verifier will order the whole application to start from the beginning. This is just to show the worst case scenario where it could be hard to know what is the optimal level to restart the application from. Because to know such information, the verifier has to know more about the application and all its tasks and their layers. Both Blocking and Hybrid finished around the same time in both failures and no failures. (recall that Guardian deals with not only different applications but also different frameworks (Hadoop, Tez, ...) that works on top of Guardian).

- Hybrid Verify is faster than Blocking Verify in case of failures because Hybrid Verify can take advantage of switching to pure Non-Blocking verification in case previous stage results were verified before the current stage finished processing; which is the most common case in big data frameworks. In other words, Hybrid Verify takes advantage of both words, Blocking Verify and Non-Blocking Verify. In the best case (no failures), it will be as fast as Non-Blocking Verify but in the worst case (with failures) it avoids the catastrophic phenomena of Spread of Corruption.

5.3.2 Tolerate two or more Byzantine Failures

In general, Byzantine Failures are tolerated by having multiple replicas that can form a majority when one (or more) replicas produce corrupted output. The majority can be formed by having 3 out of 4 matching results to tolerate 1 Byzantine Failure(where f=1, 3f+1=4), or 5 out of 7 matching results to tolerate 2 Byzantine Failures(where f=2, 3f+1=7).

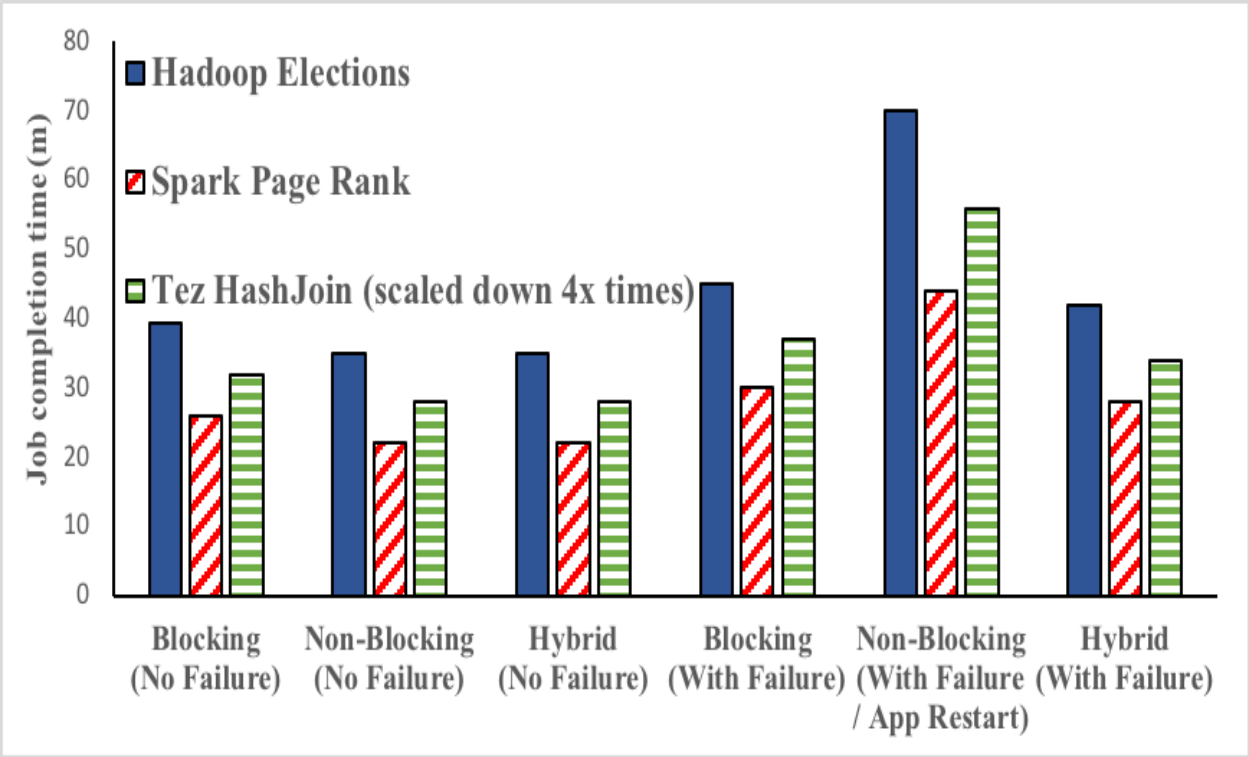


Figure 5.2. Hybrid Verification with and without failures

As long as there are at most 1 Byzantine Failure in a 4 replicas setup, or at most 2 Byzantine Failures in a 7 replicas setup, there is no need to re-execute any tasks in any execution stage. But the problem happens if there are more than 1 Byzantine Failure with 4 replicas, or more than 2 Byzantine Failures with 7 replicas. In that case, the verifier can't form a majority because 2 out of 4 is not a majority in the first case and 3 out of 7 is very close to 4 out of 7 in the second case. In these cases, the verifier has no option other than restarting this stage tasks in a hope that the new tasks will produce matching outputs.

Restarting the tasks in all verification modes

- **Blocking Verify**

  In Blocking Verify, restarting the task is done in a straightforward way; each task replica in each stage block the execution of the task until they hear back from the verifier. If the verifier can't form a majority it will send a re-execute signal to the current stage tasks.

- **Non-Blocking Verify**

  In Non-Blocking Verify, the damage of not forming a majority between the replicas is the highest damage between all modes. The reason for that is it is very hard to detect which replicas from which execution stage caused the problem of not having a majority. This is why in most cases the only possible solution is to restart the whole application from the beginning because later stages can't decide which output to send to the end user since there are different outputs and no majority can be formed.

- **Hybrid Verify**

  Recall how Hybrid Verify works: Never use Non-Blocking Verify in two consecutive stages unless first stage output is verified. In other words: follow a Non-Blocking Verify with a Blocking Verify until the output of the first stage is completely verified. In the case that there is no majority can be formed, Hybrid Verify switches to Blocking Verify. As mentioned before, Blocking Verify has the advantage of blocking the execution until the output is completely verified and a correct majority can be formed. In case a majority can't be formed, Blocking Verify will order current stage tasks to re-execute to form a majority.

To test the effectiveness of Hybrid Verify when a majority can't be formed, we evaluated the aforementioned benchmark with both 4 replicas setups and 7 replicas setups. We tested these modes not only for Hybrid Verify but also Blocking Verify and Non-Blocking Verify to compare the finish time in different modes. We compared the overhead between the following scenarios:

- 4 replicas with 1 Byzantine Failure.
- 4 replicas with 2 Byzantine Failures (no majority).
- 7 replicas with 1 Byzantine Failure.
- 7 replicas with 2 Byzantine Failures.
- 7 replicas with 3 Byzantine Failures (no majority).

Figure 5.3 shows the finish time of running 4 replicas and 7 replicas, respectively, in all the three modes: Blocking Verify, Non-Blocking Verify and Hybrid Verify without any failure.

Whereas, Figure 5.4 shows the finish time of running 4 replicas and 7 replicas, respectively, in all the three modes: Blocking Verify, Non-Blocking Verify and Hybrid Verify with different number of Byzantine Failures(as mentioned in the scenarios above).

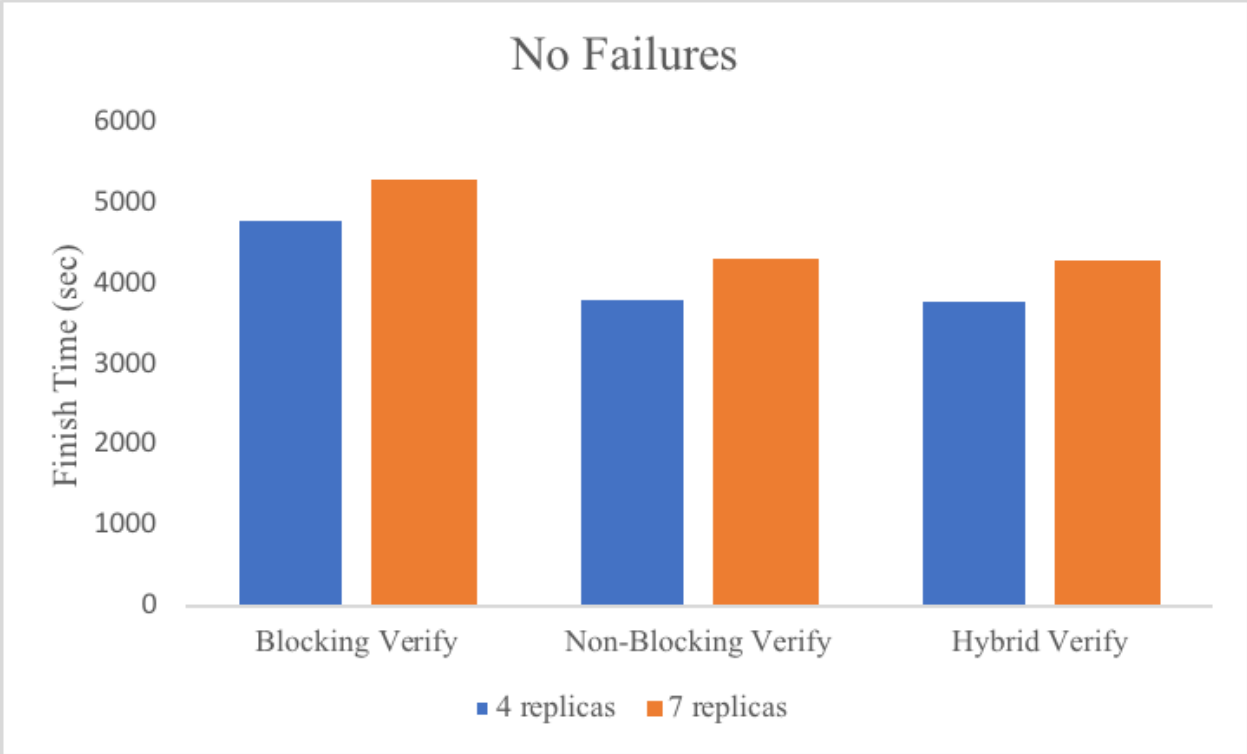


Figure 5.3. Hybrid Verification with no failures (4 vs 7 replicas)

From Figure 5.3 (No Failures)

- **Blocking Verify** takes longer time than both Non-Blocking Verify and Hybrid Verify. This is due to the nature of Blocking Verify in which it blocks the execution of the framework until the output of current stage tasks is completely verified.

- **Non-Blocking Verify** is faster than Blocking Verify since Non-Blocking Verify doesn't block the execution of current stage tasks waiting for the verifier response. Instead, the tasks in Non-Blocking Verify send their results to the verifier then send their outputs to next stage tasks directly without waiting for the verifier. At the end of the execu-

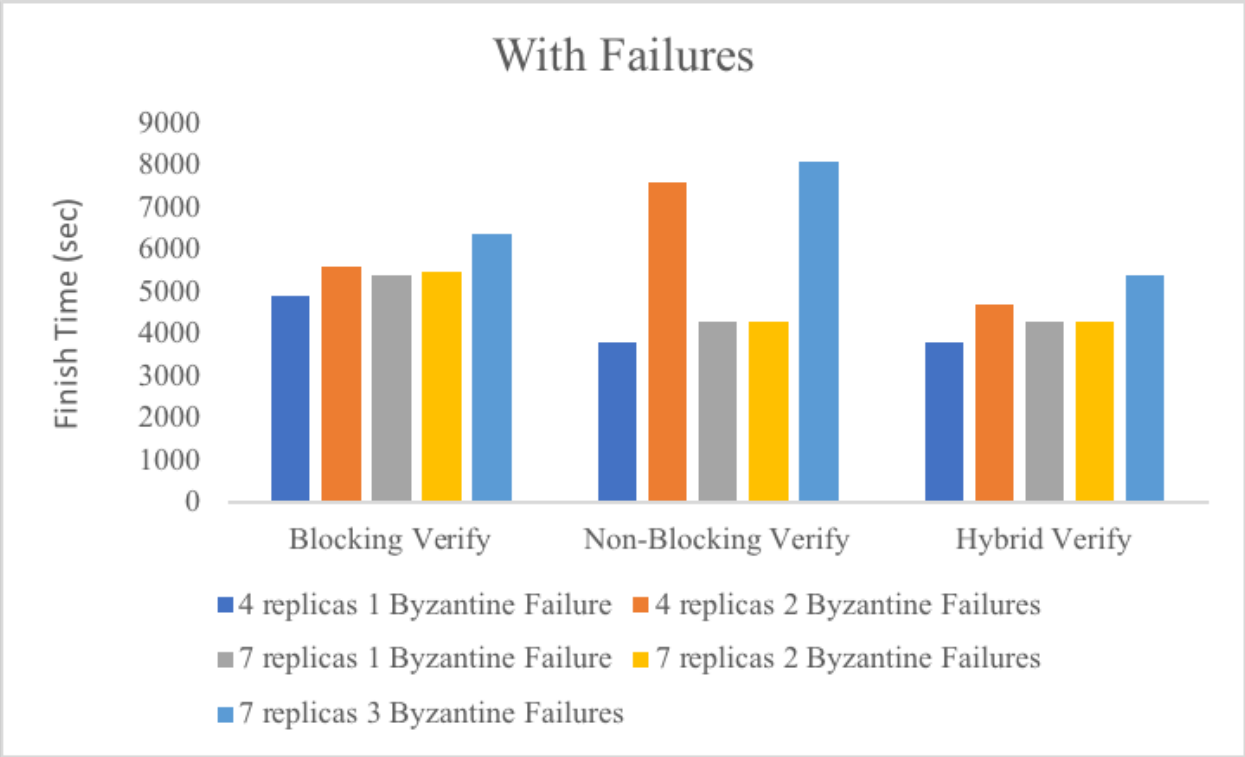


Figure 5.4. Hybrid Verification with failures (4 vs 7 replicas)

tion of the whole application, the end user will be notified if the final output formed a majority or not.

- **Hybrid Verify** is almost as fast as Non-Blocking Verify. The reason for that is Hybrid Verify works in Non-Blocking Verify all the time and it only switches to Blocking Verify when the verification of previous stage tasks is not finished by the time the current stage tasks finish execution. This gives the advantage of both worlds: the security of Blocking Verify to assure there will always be a majority, and the speed of Non-Blocking Verify by working in Non-Blocking Verify mode most of the time and only switch to Blocking-Verify when needed.

From Figure 5.4 (With Failures)

One common behaviour across all scenarios is the finish time will be very similar when you have enough resources. The only overhead that can be possibly added is due to the verifier waiting and verifying more task replicas outputs, or redirecting the output of a correct task to a corrupted task. We tried to simulate Hybrid Verify in a congested environment, where

there are not enough resources to run the replicas, but faced some problems that will be discussed in the next section.

Having more replicas will tolerate more failures in the cost of using more resources and adding a relatively small verification overhead since the verifier is dealing with more replicas.

Tolerating 2 failures will cause more overhead than tolerating 1 failure even with enough resources. One possible reason for that is the verifier has to redirect two I/O channels that have the corrupted outputs to the correct ones.

The problem with Non-Blocking Verify is it is very hard to know at which stage the tasks couldn't form a majority. That is why at the very last stage the whole application has to restart.

- **Blocking Verify:**
  - 4 replicas 1 failure: no need to re-execute since a majority can be formed: 3 out of 4.
  - 4 replicas 2 failures: there is a need to re-execute one stage tasks (the stage that can't form a majority) since a majority can't be formed: 2 out of 4.
  - 7 replicas 1 failure: no need to re-execute since a majority can be formed: 6 out of 7.
  - 7 replicas 2 failures: no need to re-execute since a majority can be formed: 5 out of 7.
  - 7 replicas 3 failures: there is a need to re-execute one stage tasks (the stage that can't form a majority) since a majority can't be formed: 4 out of 7. Eventhough 4 out of 7 is larger than 3 out of 7, yet there has to be at least 5 correct matching outputs for the majority to form a consensus.
- **Non-Blocking Verify:**
  - 4 replicas 1 failure: no need to re-execute since a majority can be formed: 3 out of 4.

- 4 replicas 2 failures: a majority can't be formed and there is a need to re-execute the application since 2 out of 4 can't form a majority. Moreover, in Non-Blocking Verify it is hard to detect which stage caused the problem so the whole application has to be restarted.
- 7 replicas 1 failure: no need to re-execute since a majority can be formed: 6 out of 7.
- 7 replicas 2 failures: no need to re-execute since a majority can be formed: 5 out of 7.
- 7 replicas 3 failures: a majority can't be formed and there is a need to re-execute the application since 4 out of 7 can't form a majority. Moreover, in Non-Blocking Verify it is hard to detect which stage caused the problem so the whole application has to be restarted.

- **Hybrid Verify:**
  - 4 replicas 1 failure: no need to re-execute since a majority can be formed: 3 out of 4.
  - 4 replicas 2 failures: a majority can't be formed but Hybrid Verify switches to Blocking Verify when the output of the previous stage has not finished the verification process, or the verification is finished but can't form a majority. In this case Hybrid Verify switches to Blocking Verify and will order current stage tasks to re-execute to get new outputs that forms a majority.
  - 7 replicas 1 failure: no need to re-execute since a majority can be formed: 6 out of 7.
  - 7 replicas 2 failures: no need to re-execute since a majority can be formed: 5 out of 7.
  - 7 replicas 3 failures: a majority can't be formed but Hybrid Verify switches to Blocking Verify when the output of the previous stage has not finished the verification process, or the verification is finished but can't form a majority. In

this case Hybrid Verify switches to Blocking Verify and will order current stage tasks to re-execute to get new outputs that forms a majority.

Replicas DeadLock Waiting for Resources

We tried to run 7 replicas to tolerate 2 Byzantine Failures but with congested resources used for the replicas. The problem in that setup is there weren't enough resources for the 7 replicas to run simultaneously to finish their work and send their output to the verifier. That is why if there are not enough resources for the replicas to run simultaneously, the finished tasks will hang indefinitely waiting for the verifier response, but the verifier can't form a majority yet because the verifier is still waiting for other tasks to finish their work, and these tasks didn't finish their work because they are waiting for more resources to be freed. This is similar to the DeadLock problem in Operating Systems.

6. MESOS DESIGN

6.1 YARN over Mesos

YARN is not the only RMS in the Big Data world, there are many other well known RMSs such as Mesos [67], Kubernetes [68], Docker Swarms [69], Omega [70], Fuxi [26], ... etc. But there are three main reasons why we choose YARN instead of any other RMS:

1. Some frameworks are YARN-native which means they already uses YARN out of the box as the main resource scheduler for the framework, and to run the framework tasks inside YARN containers. In other words, there is no standalone mode for this framework to run on, instead its 'standalone' mode already uses YARN as part of its main components. This is actually the case for Samza and some other frameworks such as Tez, Hadoop, Apex and Dryad.

   We are not aware of any framework that is RMS-native that uses any RMS other than YARN. For example, we are not aware of Mesos-native frameworks or Omega-native framework. So it will make sense to choose YARN as the main RMS since it will benefit both standalone frameworks and RMS-native frameworks.

   It is noteworthy that Flink used to be YARN-native until Flink version 1.4.0 where Flink's developers added a standalone option. In general, we noticed a tendency for many Big Data frameworks that are still in incubating phase or in its early stages to start with YARN as its main resource scheduler, then move on to provide a standalone option in later stages of development after the framework is mature enough.

2. Not all RMSs are open source. The fact that YARN is open source make it one of the few open source options out there to choose from. For example, Alibaba Fuxi [26] and Google Omega [70] are not open source although they have competitive properties to YARN.

3. The resource request system in YARN makes it attractive and appealing for us compared to some other resource management systems such as Mesos in which the resources are actually distributed in the form of resource offers to the frameworks JobManagers.

However, some developers may prefer to move our whole protocol from YARN and put it into Mesos. Maybe because their company is already using Mesos in their production systems and it will be easier to change Mesos code than changing all the company‘s production infrastructure to a completely new RMS such as YARN.

In this section, we will go over the design aspects of Mesos to make it achieve dependability for frameworks running on top of it.

6.2 Mesos Overview

Mesos [24] is a ‘resource offer‘-based RMS. To explain this, let‘s briefly go through the main design of vanilla Mesos. Figure 6.1 shows the main steps of the resource offers system in Mesos. Step 1 in the figure shows that Slave 1 is sending to the Allocation Module in Mesos Master the free resources in that machine. The Allocation Module in step 2 send these available resources to the framework scheduler as ‘resource offers‘. Accordingly, the framework scheduler uses these resource offers in assigning and launching tasks in the cluster by sending these tasks specs and requirements to the Allocation Module in step 3. What follows, in step 4, is that the Allocation Module will launch these tasks on the machines that has the resources offered in steps 1 and 2. For more details about Mesos please refer to Mesos paper [24].

It is important to mention that most components in Guardian and Warden are shared between different RMSs except the scheduler. Guardian and Warden Schedulers in YARN is different than Mesos. The reason there is a different design for Guardian and Warden scheduler with both Mesos and YARN is due to the nature of the built-in resource scheduling techniques used in either Mesos or YARN; Mesos deals with resource offers whereas YARN deals with resource requests. Hence, Guardian and Warden schedulers can’t be a shared component in both Mesos and YARN instead it has to be tailored towards each RMS built-in scheduling technique.

6.3 Mesos Scheduling

(From Resource Offers to Resource Bundles)

The idea of resource offers in Mesos is to give framework schedulers the knowledge needed to know how many tasks they can run on the cluster. But at the same time, we have to remember that replicas of the same task has to lunch on different machines so that in the event of one machine failure, the replicas from other machines will keep working. If we schedule all replicas on the same machine and that machine fails then all replicas will fail and has to restart which contradicts the objective of active replication. But the problem with the resource offers in Mesos is that it may give one framework scheduler many resource offers that belong to one or two machines only. These kind of resource offers are useless to the framework scheduler since the scheduler can‘t launch multiple replicas of the same task on different machines. Moreover these kind of offers will incur unnecessary network traffic that will not benefit any framework that uses replication.

To overcome this problem, we propose what we call ‘*resource bundles*‘. What a resource bundle is a group of resource offers that belong to different machines (number of machines = number of replicas) that were gathered and filtered out in the Allocation Module in Mesos master. Instead of sending one resource offer to the framework as in vanilla Mesos, we send a resource bundle (step 2 in Figure 6.2). The framework scheduler will accept this resource bundle and will launch a task replica that can be launched in all the machines that the resource bundle has. To do this, the framework has to choose the lowest resource offer in that resource bundle from each type of resources and launch a task that can be launched using these lowest common resources. For example, in step 2 in Figure 6.2, the resource bundle has resource offers from machines s1, s7, s4, s9. The lowest number of CPUs available from all machines is 2 cpus, and the lowest number of memory GBs available from all machines is 3GBs. Hence the task that the framework has to launch has to have resource requirements no more than 2 cpus and 3 GBs of memory. And this is exactly what happened in step 3. If the framework doesn’t have a task with these requirements then it will reject the offer and wait for another resource bundle (exactly as it used to reject resource offers in vanilla Mesos).

Once the Allocation Module receives the bundle of tasks that has to be launched on the slaves (step 3 in Figure 6.2), the Allocation Module will launch them on the cluster normally as if they were four independent tasks (step 4 in Figure 6.2), because technically they are four normal tasks that launch on four different machines. What happens after that is the tasks will do their normal processing and will connect to the verifier using an API similar to the API we used in Guardian and Warden.

The Allocation Module has the ability to change the size of the bundle according to the number of replicas desired by the user. Also note that at step 3 and step 4 there is no need to worry about launching the tasks at the same time since that should happen by default because the resources are already reserved for that framework.

One major drawback of this design is what is happening in steps 2 and 3 in Figure 6.2. In this design, the frameworks has to be aware of replicating its tasks according to the desired number of replicas by the user. We can improve the transparency of this design by introducing an intermediate layer between the Allocation Module and the framework scheduler as shown in Figure 6.3. The objective of this layer is to receive resource bundles from the Allocation Module (step 2a), find the lowest resource from each type, send this lowest resource as a single resource offer as if it is running in vanilla Mesos (step 2b in Figure 6.3). The framework scheduler will accordingly launch its task (step 3a in Figure 6.3) again as in vanilla Mesos, then the intermediate layer will modify that launch request by changing it to launch four tasks replicas on different machines and send these launch commands to the Allocation Module (step 3b). From there the Allocation Module will launch these tasks replicas as if they were normal tasks in vanilla Mesos. This design will make the frameworks schedulers completely unaware of replication compared to the previous design but on the other hand its main drawback is the overhead of this intermediate layer since all communication between all framework schedulers and the Allocation Module has to go through this layer, whereas in the previous design this overhead is distributed between framework schedulers since each scheduler is dealing with the Allocation Module directly.

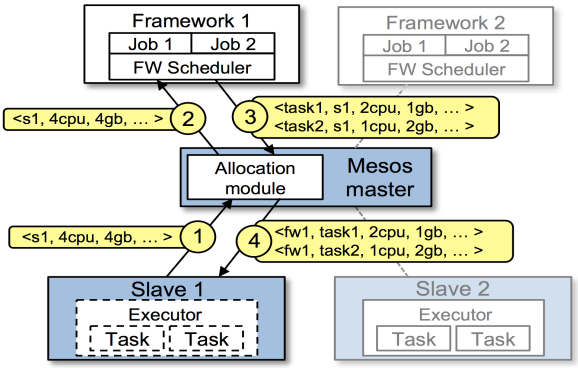


Figure 6.1. Vanilla Mesos (from Mesos paper [24])

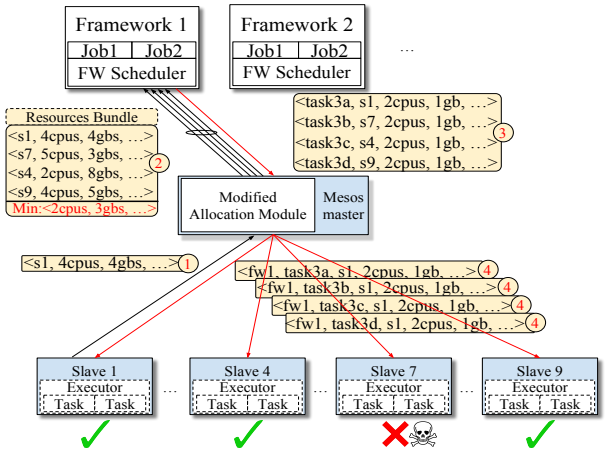


Figure 6.2. Mesos Replicaion 1

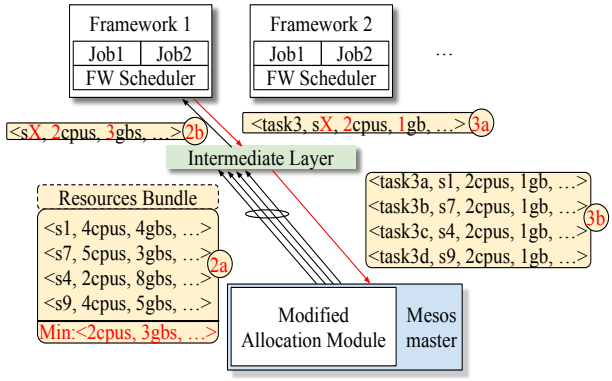


Figure 6.3. Mesos Replicaion 2

6.4 Mesos Prototype

In this section, we will go over a prototype of Guardian built on Mesos. The idea is to show a proof of concept of Guardian on more than one RMS. The design and the implementation of Guardian with Mesos is based on the design shown in Figure 6.2. The main difference between Mesos and many other RMSs (including YARN) is that Mesos uses the resource offers system compared to the resource request system used in other RMSs. This makes it more challenging to schedule replicas of the same task on different machines. To solve this problem, we propose the idea of 'resource bundles' that is explained in details in Section 6.3.

In this prototype, we used Hadoop on top of Mesos with Guardian. We used the well-known word count application in this prototype as a proof of concept. We ran the application on a similar evaluation environment used in YARN with Guardian (Section 3.3.2). It is worth noting that Mesos uses 'Executors' instead of 'Containers' as in YARN. Conceptually, they are both considered the 'Execution Unit' of each of the RMSs. A failure is simulated by killing all Mesos tasks and executors running on a slave machine just to simulate a machine failure in a cluster.

The general workflow for Mesos tasks working on Guardian is very similar to YARN tasks working on Guardian (shown in Figure 3.2). First stage tasks (e.g. mappers) send the output to second stage tasks (e.g. reducers) after getting the data verified from the verifier.

The main results for Hadoop on Mesos with Guardian are shown in Figure 6.4 (note that the y-axis starts from minute 20). The approach we used in injecting failures is similar to the approach we used in Figure 3.3 to simulate container and AM failures in different progress times. As expected, having another replica running on a different machine will improve recovery time since there is no need to wait for the task to timeout, restart on another machine and redo all the work that was done before the machine crash.

During the evaluations we ran on Mesos with Guardian, we faced some problems and issues that we didn't face in YARN. These issues are discussed in the following part.

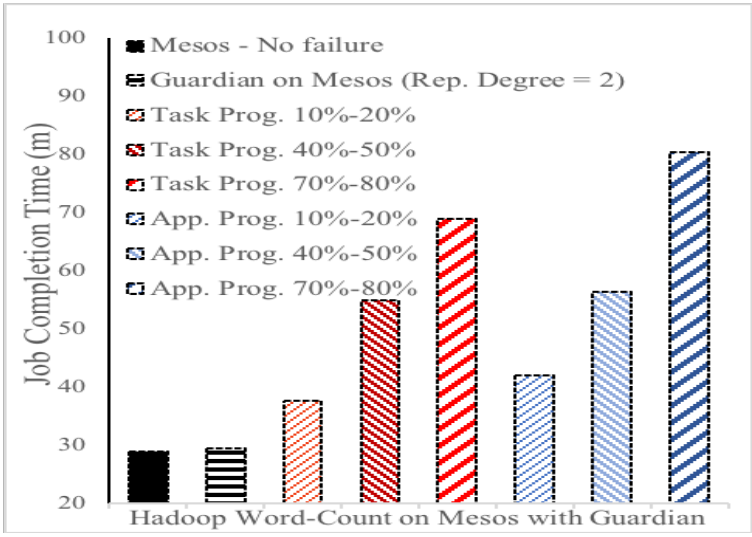


Figure 6.4. Hadoop on Mesos with Guardian

Over-Allocation of Mesos Resources

One of the problems we noticed while evaluating Guardian on Mesos is the amount of resources reserved compared to YARN. As noted before, Mesos uses the resource offers system compared to YARN (which uses the common resource request system). The problem that we noticed is that Mesos reserves more resources than YARN due to the resource offers system that Mesos uses. The root cause of this problem is the fact that Mesos 'reserves' the resources that Mesos uses in its resource offers before offering these resources to the frameworks.

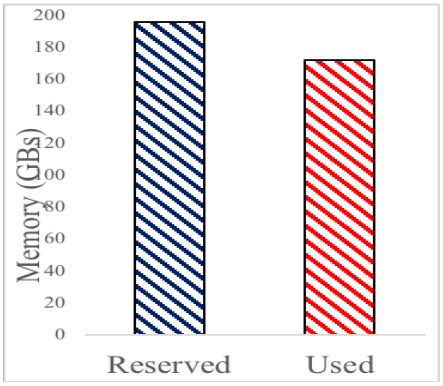


Figure 6.5. Memory reserved vs used in Mesos

For example, Figure 6.5 shows the total memory (in GB) reserved vs the total memory actually used for the Hadoop word-count application we used on Mesos. Sometimes Mesos reserves some resources that end up not used by any framework because no application in

any framework needs to consume all the resources offered by Mesos Allocation Module which results in temporarily 'wasting' some of the resources reserved. This problem doesn't happen in YARN since YARN uses the resource request system and YARN only reserves resources that are explicitly requested from a certain application running on a framework running on YARN. The same problem happens with other kind of resources (e.g. CPU cores), but it is not shown for conciseness. In all the evaluations that we had on YARN, we never noticed that YARN reserves any resources unless if the resources are being clearly requested by a certain framework.

Optimizing resource offers based RMSs is an interesting research direction for future work. But it seems that Mesos is the only RMS that uses this system compared to the more common resource request system that is used in YARN. Moreover, almost all frameworks that work on Mesos work on YARN, but not the other way around [71].

7. CONCLUSION & FUTURE WORK

Every year witnesses the emergence of new Big Data frameworks and Cloud Computing systems. As cloud computing becomes more ubiquitous, so does the need for robust fault tolerance systems. One crucial common objective that these frameworks in general tries to tackle is a better fault tolerance technique that achieves high availability and low recovery time. The rise of the RMS presents a new avenue for the development of generic solutions for fault tolerance that are also adaptable to many failure models.

This thesis represents one of the earliest and fewest works on providing generic fault tolerance support at the RMS level for both Batch and Streaming frameworks. This is achieved by providing two systems: Guardian and Warden. Both these systems achieve Genericity, Reusability and Low-Overhead in their own customized way.

Guardian proposes a new design for fault tolerance in the cloud, which promotes an emphasis on dependable resources, as well as configurability. We have shown that Guardian satisfies its goals: genericity, low-overhead and reusability. We also elaborated how can Guardian be adapted towards the two most common RMSs: Mesos and YARN. Our evaluations show that Guardian has low-overhead in unsaturated environments, and that the overhead of replication is far more beneficial than the penalties incurred from failures in RMSs without replication. From the evaluations we noted that Guardian improved completion time by around 68% in the presence of failures, while maintaining around 6% overhead. We also noted that AM failure causes more damage than task failure (at same fault injection time), and EC2 cost becomes comparable when application fails at $> 70\%$ progress. We also concluded that Guardian is good for frameworks in incubating phase. We‘ve also introduced a new ‘Hybrid‘ verification approach that takes advantage of both the Blocking and NonBlocking verification models while avoiding each of the models drawbacks. Moreover, we discussed three major reasons why a fully transparent design will not achieve Guardian's purposes. Many causes of nondeterminism has also been analyzed in three major big data frameworks (Hadoop, Tez and Spark). Finally we discussed some advanced thread models and how can Guardian handle them.

We've also presented Warden, a generic, multi-framework, flexible, customizable, low overhead protocol that strives to achieve different levels of fault tolerance guarantees in streaming frameworks with the lowest overhead possible. We described the multi-phased design and the multi-threaded implementation of Warden. We ran our evaluations on two instantiations (Flink/Samza(with Kafka)) and our results show the effectiveness of our approach in the presence of failures and without failures compared to other fault tolerance techniques; checkpointing in particular. We've also discussed how the verifier can avoid SPoF by introducing the All to All design.

In the future, this work can be expanded by building a trust management component that plugs into the verifier and computes a per node trust metric based on multiple system parameters (job completion time, CPU usage, etc.). Trust values can then be used by the scheduler to flag suspicious nodes or schedule time sensitive tasks.

Moreover, we plan to merge both Guardian and Warden into one main project and make it available on Github. Another orthogonal research direction is to explore other fault tolerance techniques such as lineages used in Spark and integrate it into Guardian and Warden to make it available to different frameworks.

Furthermore, non-determinism in streaming frameworks is an important field of research that will complement the work done in stream replication (Warden).

Finally, exploring the integration of reliability aspects into other types of RMSs such as Google Kubernetes with Docker Containers could be an interesting research direction.

REFERENCES

[1] *Google spotlights data center inner workings*, http://www.cnet.com/news/google-spotlights-data-center-inner-workings/.

[2] K. V. Vishwanath and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability," 2010, pp. 193–204.

[3] C. Beckmann, *Google App Engine: Information Regarding 2 July 2009 outage*, https://groups.google.com/forum/#!topic/google-appengine/6SN_x7CqffU, 2009.

[4] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure Trends in a Large Disk Drive Population," 2007, pp. 17–23.

[5] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can be Done About It?," 2003.

[6] *US Secret Service Report on Insider Attacks 2005*, http://www.sei.cmu.edu/about/press/insider-2005.html.

[7] *Victims of Lost Files Out of Luck Apr. 2002*, http://news.cnet.com/Victims-of-lost-files-out-of-luck/2100-1023_3-887849.html.

[8] M. Calore, "Ma.Gnolia Suffers Major Data Loss, Site Taken Offline," *Wired*, 2009, https://www.wired.com/2009/01/magnolia-suffer/.

[9] B. Cook, *Seattle Data Center Fire Knocks Out Bing Travel, Other Web Sites*, http://www.bizjournals.com/seattle/blog/techflash/2009/07/Seattle_data_center_fire_knocks_out_Bing_Travel_other_Web_sites_49876777.html, 2009.

[10] R. Miller, *FBI Seizes Servers at Dallas Data Center*, http:/www.datacenterknowledge.com/archives/2009/04/03/fbi-seizes-servers-at-dallas-data-center/, 2009.

[11] E. Weise, *Massive Amazon Cloud Service Outage Disrupts Sites*, http://www.usatoday.com/story/tech/news/2017/02/28/amazons-cloud-service-goes-down-sites-scramble/98530914/, 2017.

[12] *Apache Hadoop*, http://hadoop.apache.org/.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing With Working Sets," pp. 10–10, 2010.

[14] *Department of Defense Information Enterprise Strategic Plan 2011-2012*, http://dodcio.defense.gov/docs/DodIESP-r16.pdf.

[15] P. Nelson, *Search and Big Data are now Mission-Critical for Business*, http://www.searchtechnologies.com/search-and-big-data-critical-for-business, 2016.

[16] *The First Workshop of Mission-Critical Big Data Analytics (MCBDA)*, http://credit.pvamu.edu/MCBDA2016/, 2016.

[17] *Big Data - Why Transaction Data is Mission Critical to Success*, http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=IML14442USEN, 2014.

[18] *Flink Checkpointing Constraints*, https://ci.apache.org/projects/flink/flink-docs-release-1.7/ops/state/state_backends.html/.

[19] P. Costa, M. Pasin, A. N. Bessani, and M. Correia, "Byzantine Fault-Tolerant MapReduce: Faults Are Not Just Crashes," 2011, pp. 32–39.

[20] Y. Zhang, Z. Zheng, and M. R. Lyu, "BFTCloud: A Byzantine Fault Tolerance Framework for Voluntary-Resource Cloud Computing," 2011, pp. 444–451.

[21] J. J. Stephen and P. Eugster, "Assured Cloud-Based Data Analysis with ClusterBFT," 2013, pp. 82–102.

[22] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," pp. 225–267, 1996.

[23] F. B. Schneider, "What Good are Models and What Models are Good," *Distributed systems*, pp. 17–26, 1993.

[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," 2011, pp. 22–22.

[25] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers For Large Compute Clusters," 2013, pp. 351–364.

[26] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale," pp. 1393–1404, 2014.

[27] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache Hadoop Yarn: Yet Another Resource Negotiator," 2013, p. 5.

[28] B. Abusalah, D. Schatzlein, J. J. Stephen, M. S. Ardekani, and P. Eugster, "Dependable cloud resources with guardian," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2017, pp. 1543–1554.

[29] *Kubernetes*, http://kubernetes.io//.

[30] F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li, "Hadoop High Availability Through Metadata Replication," 2009, pp. 37–44.

[31] P. Costa, M. Pasin, A. N. Bessani, and M. P. Correia, "On The Performance of Byzantine Fault-Tolerant MapReduce," pp. 301–313, 2013.

[32] A. N. Bessani, V. V. Cogo, M. Correia, P. Costa, M. Pasin, F. Silva, L. Arantes, O. Marin, P. Sens, and J. Sopena, "Making Hadoop MapReduce Byzantine Fault-Tolerant (Fast abstract)," 2010.

[33] G. Veronese, M. Correia, A. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine Fault-Tolerance," pp. 16–30, 2013.

[34] M. Serafini and N. Suri, "Reducing The Costs of Large-Scale BFT Replication," 2008, p. 14.

[35] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright Cluster Services," 2009, pp. 277–290.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems," 2010, p. 9.

[37] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group Communication Specifications: A Comprehensive Study," pp. 427–469, 2001.

[38] X. Défago, A. Schiper, and P. Urbán, "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," pp. 372–421, 2004.

[39] *Apache Apex*, https://apex.apache.org/.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[41] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.

[42] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[43] *Apache Apex Shutdown*, https://www.datanami.com/2018/05/08/datatorrent-stream-processing-startup-folds/.

[44] *Apache Heron Experimental*, https://apache.github.io/incubator-heron/docs/operators/deployment/schedulers/yarn/.

[45] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable distributed stream processing.," in *CIDR*, vol. 3, 2003, pp. 257–268.

[46] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *the VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[47] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, *et al.*, "The design of the borealis stream processing engine.," in *Cidr*, vol. 5, 2005, pp. 277–289.

[48] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Kumar, G. Jon, P. K. Gunda, and J. Currey, "DryadLINQ : A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855742$%5Cbackslash$nhttp://portal.acm.org/citation.cfm?did=1855742.

[49] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez, "Streamcloud: A large scale data streaming system," in *2010 IEEE 30th International Conference on Distributed Computing Systems*, 2010.

[50] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[51] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik, "A cooperative, self-configuring high-availability solution for stream processing," in *2007 IEEE 23rd International Conference on Data Engineering*, IEEE, 2007, pp. 176–185.

[52] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 574–585, 2008.

[53] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "A hybrid approach to high availability in stream processing systems," in *2010 IEEE 30th International Conference on Distributed Computing Systems*, IEEE, 2010, pp. 138–148.

[54] *Apache Pig*, http://pig.apache.org.

[55] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language For Data Processing," 2008, pp. 1099–1110.

[56] *Apache Tez*, https://tez.apache.org/.

[57] *Apache Spark*, http://spark.apache.org//.

[58] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud Storage With Minimal Trust," p. 12, 2011.

[59] Y. Wang, L. Alvisi, and M. Dahlin, "Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication," 2012, pp. 413–424.

[60] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Physical Disentanglement in a Container-Based File System," 2014, pp. 81–96.

[61] Z. Xu, M. Hirzel, and G. Rothermel, "Semantic Characterization of MapReduce Workloads," 2013, pp. 87–97.

[62] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou, "Nondeterminism in Mapreduce Considered Harmful? An Empirical Study on Non-Commutative Aggregators in Mapreduce Programs," 2014, pp. 44–53.

[63] C. Csallner, L. Fegaras, and C. Li, "New Ideas Track: Testing MapReduce-Style Programs," in *SIGSOFT Symp. and Euro. Conf. on Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 504–507.

[64] *Spark Operations*, http://spark.apache.org/docs/latest/programming-guide.html//.

[65] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[66] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, 2017.

[67] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," 2011.

[68] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, 2014.

[69] *Docker Swarms*, https://docs.docker.com/engine/swarm/.

[70] M. Schwarzkopf and A. Konwinski, "Omega: flexible, scalable schedulers for large compute clusters," in *Eurosys*, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2465386.

[71] *mesosframeworks*, https://guidoschmutz.wordpress.com/2016/04/20/last-week-in-stream-processing-analytics-4182016/.