Received 24 December 2022, accepted 12 January 2023, date of publication 23 January 2023, date of current version 3 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3239365

RESEARCH ARTICLE

Interminable Flows: A Generic, Joint, Customizable Resiliency Model for Big-Data Streaming Platforms

BARA ABUSALAH[1], THAMIR M. QADAH[2], (Member, IEEE), JULIAN JAMES STEPHEN[3], AND PATRICK EUGSTER[1,4]

[1]Electrical and Computer Engineering Department, Purdue University, West Lafayette, IN 47907, USA
[2]Computer Systems Department, Umm Al-Qura University, Mecca 24382, Saudi Arabia
[3]IBM Watson Research Center, Yorktown Heights, NY 10598, USA
[4]Computer Systems Institute, Università della Svizzera Italiana, 6900 Lugano, Switzerland

Corresponding author: Bara Abusalah (baraabusalah1@gmail.com)

ABSTRACT The examiner of cloud computing systems in the last few years observes that there is a trend of the emergence of a new Big Data framework almost every 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. This is particularly crucial for streaming systems (compared to batch processing 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. Encapsulating these implementations into one layer and make 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 present Warden, a generic, multi-framework, flexible, customizable, low overhead protocol to ensure the resiliency of streaming applications running on streaming Big Data frameworks. Most reliability protocols carry out one rigid fault tolerance approach 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. 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 delivery semantics (Exactly Once / At Most Once) combined with two fault tolerance guarantees (Crash Failures / Byzantine Failures). To the best of our knowledge, such approach was never tried on multiple streaming frameworks before. We built a prototype of Warden on Flink and Samza (with Kafka) streaming frameworks. Our evaluations highlight the effectiveness of our approach in the presence of failures and without failures compared to other fault tolerance techniques (such as checkpointing).

INDEX TERMS Fault tolerance, reliability, replication, resource management systems (RMS), streaming frameworks.

I. INTRODUCTION

In recent years, Cloud Computing technologies have evolved rapidly and become one of the most researched areas in

The associate editor coordinating the review of this manuscript and approving it for publication was Chong Leong Gan.

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. A machine crash in a batch processing framework may result in restarting all batch processing tasks running on that machine. This task restart may not have a huge impact or a drastic effect to the end user if he/she is not expecting the output of the batch application to be delivered within a certain time window. On the other hand, this is not the case in 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.

Dealing with faults in streaming systems is much more challenging than batch processing systems. 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. The focus of this paper is targeted towards fixing this particular problem. In other words, we strive to enhance the fault tolerance and reliability of Big Data streaming frameworks by introducing a protocol we call Warden. There are many reasons and motivations that let us believe that fixing this problem is important for the Big Data and Cloud Computing communities. These motivations will be discussed in the following subsection.

A. MOTIVATION

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

1) 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 and weaknesses 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 drawback 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 network 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 [1]. Samza (until recently) didn’t have an option to run as a standalone system, instead it needed an execution engine such as YARN and a messaging system such as Kafka to operate. Each of these systems has its own constraints to use its built-in reliability features.

2) STREAMING FRAMEWORKS RECOVERY TIME

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. The user doesn‘t monitor the job online in real time, and he/she may accept some delays in job finish time if the batch job is not a Mission-Critical type. Hence if one of the job tasks fails and restarts the user may not notice that. 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.

3) REDUNDANT WORK

The observant of the Big Data frameworks and Cloud Computing technologies in general 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. 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 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.

4) 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 this paper we try to achieve Exactly Once in both Byzantine and Crash modes. 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 discussed later.

5) 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 streaming systems in particular.

These observations, among others, have motivated us to create Warden. In the following sections we will show the objectives, design and implementation of Warden that helped in overcoming the above shortcomings and achieve better reliability guarantees for Big Data frameworks in general and streaming systems in particular.

Contributions and Roadmap: Concretely, this paper makes the following contributions:

1) We provide a reliability approach that targets multiple Big Data streaming frameworks (not a single framework as most approaches do). Particularly, we focus on online realtime streaming systems since their reliability is more challenging than offline batch processing system.
2) We provide users an option to choose between different fault tolerance models (Byzantine/Crash) combined with different consistency options to choose from (Exactly Once/At Most Once) each of which is tailored towards particular applications and use cases.
3) We empirically show our evaluations on two well known streaming frameworks: Flink and Samza, with different consistency models (Exactly Once vs At Least Once).

The remainder of this paper is structured as follows. Section II presents the main objectives of Warden. Section III explains Warden‘s internal details followed by Section IV which shows evaluations on two applications running on two streaming frameworks. Section V discusses alternative approaches and why we ended up with our current approach. Section VI presents related work and Section VII concludes with final remarks.

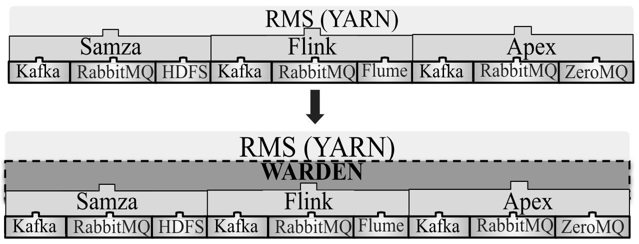

FIGURE 1. Vision of Warden.

II. WARDEN's OBJECTIVES

The primary objective of Warden is to provide a generic, multi-framework, flexible, customizable, low overhead protocol to ensure the resiliency of streaming applications running on streaming frameworks. In this section, we will go over the primary characteristics of Warden that help it achieve this goal.

A. MULTI-FRAMEWORK APPROACH

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 as shown in Figure 1. 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 Warden by using a predefined set of APIs to make the behaviour of Warden tailored towards their frameworks objectives as if Warden was built specifically and solely for their particular framework.

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 most streaming frameworks doesn't have task replication 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 at the level of applications). Whereas checkpointing has been widely implemented in many streaming frameworks, so choosing checkpointing as the main fault tolerance technique may not benefit a large subset of streaming 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.

B. MINIMUM RECOVERY TIME POSSIBLE

Compared to batch processing, stream processing applications expect lower response time and latency. Active replication helps achieve this 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 the stream 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 the 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.

Also Warden tries to minimize recovery time not only by blindly implementing replication but also by optimizing this technique to make its overhead as low as possible. This will be discussed later in Section III where we will show that all data streams will be flowing through an entity, called the verifier, which has enough data and information to process the replicated streams, make decisions and directly forward the data to the following stage tasks without blocking the flow of data between different execution stages.

C. FLEXIBILITY & CUSTOMIZABILITY

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. Warden 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 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. Another use 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 (such as counting votes in presidential elections day).

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. Combining both failure model and delivery guarantees give users a wide variety of options to choose from (please look at Figure 2):

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 spicific 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 Warden 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 Section III.

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 Warden 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. This will be discussed in Section III-B.

III. WARDEN's IMPLEMENTATION

In this section we will dive into the implementation details that helped in achieving Warden's objectives. But before that we need to define some terms. In this paper, the word tuple is used to indicate the abstraction of the data piece that is being sent in the stream. This tuple could be any data type (String, Integer, ... etc) or could be called different names like token or value. The word framework refers to a Big Data framework such as Hadoop, Spark, Storm, Heron, ... etc. There is a centralized component that we call the verifier where all tasks communicate with to send their data to be verified.

We deal with two types of failures: *Crash* and *Byzantine*. In Crash failures, the machine stops working (fail-stop) and the tasks running on the machine no longer produce or consume any data, maybe due to a power outage, network disconnection, operating system failures, ... etc. In Byzantine failure the machine and the tasks running on it will continue working, but the tasks could produce corrupted results and the processes running on the machine could behave randomly or arbitrarily. It is very hard to detect a Byzantine failure without comparing the output of the corrupted tasks with the output of the correct tasks, that is why verification (comparing outputs) is needed. Byzantine failures could happen due to many reasons: corrupted memory chips, CPU bits flip due to power transients, malicious user accessing the machine...etc.

We use an RMS called YARN as the wagon to carry our protocol (Warden). Concisely, YARN is an RMS that manages a cluster of resources. Frameworks (such as Hadoop, Spark, Storm, ... etc.) run on top of it to share the resources of the cluster. Each application running on top of each of these frameworks gets a specialized container called the Application Master (AM). From its name, the purpose of the AM is to be the head of the application where all application tasks communicate with to get more resources from YARN. Each task from these frameworks (Mappers and Reducers in Hadoop, Executors in Spark, Spouts and Bolts in Storm, ... etc) runs in what is called a YARN container (which is often a JVM). There is a centralized Resource Manager (RM) which acts as the central authority arbitrating resources among all the applications from all frameworks that run in the cluster. For more details about YARN please refer to its paper [2].

YARN is not the only RMS in the Big Data world, there are many other well known RMSs such as Mesos [3], Kubernetes [4], Docker Swarms [5], Omega [6], Fuxi [7], ... etc. There are many reasons for choosing YARN over other RMSs beside its popularity and it comes packaged already with Hadoop, some of these reasons are:

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 was actually the case for Samza until recently 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 [7] and Google Omega [6] 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.

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.

To achieve this, we introduce a Multi-Phase protocol where each phase is explained in details in each of the following subsections. Figure 3 shows these phases in sequence from left to right (the connected dotted region in the bottom and center of the figure represents the verification logic).

A. 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.

		Fault Tolerance Mode	
		Byzantine	**Crash**
Consistency Guarantees	**Exactly Once**	Byzantine Exactly Once	Crash Exactly Once
	At Most Once	Byzantine At Most Once	Crash At Most Once

FIGURE 2. Verification modes (optimistic byzantine not shown).

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 replicas of the same task running on the same machine.

Phase-2: Handshake

Once a framework task launches, the task communicates with the verifier to inform the verifier about some of this task's 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 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

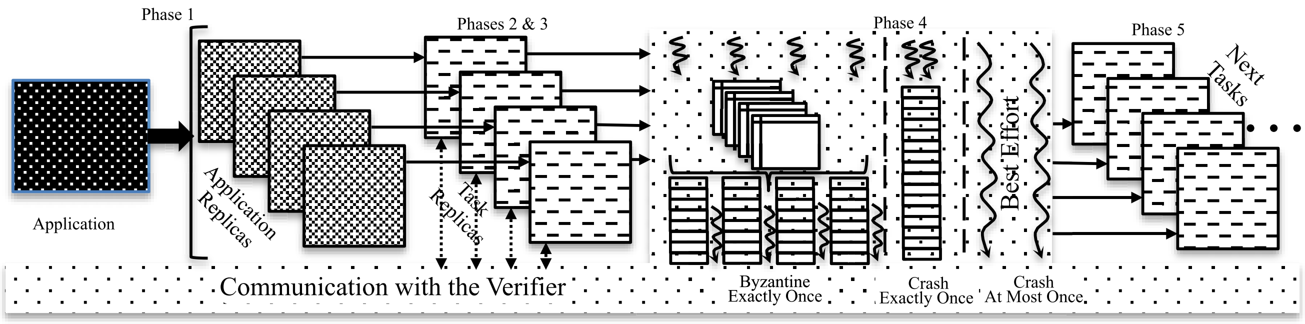

FIGURE 3. **Multi-phase protocol (byzantine at most once and optimistic byzantine are not shown).**

according to which mode (discussed in Subsection II-C) the tuples wish to be verified against. Figure 2 shows the Fault Tolerance Mode vs Consistency Guarantees grid. The details of each mode are as follows:

1) BYZANTINE EXACTLY ONCE

This is the most time consuming mode where each single tuple is verified against other tuples from replicas of the same task to form a majority ($3f + 1$). 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 send to next stage tasks which will be discussed in Phase-5.

2) 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.

3) 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, 100 tuples, . . . 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.

4) 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 mode the verifier doesn't compare the values of the tuples as in Byzantine mode, 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.

5) CRASH AT MOST ONCE

As stated in Subsection II-C 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 pulled 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 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.

B. GRANULARITY

As mentioned in Subsection II-C one of the main objectives of Warden is to achieve flexibility and customizability. This customizability is achieved not only by giving the users an option to choose between the five verification modes stated above 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. 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.

Due to these reasons we introduce a set of API calls detailed in Table 1 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.

Furthermore, one of Warden's objective is to target multiple Big Data frameworks (Subsection II-A). 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.

C. MULTITHREADING & SCALABILITY

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 continues 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.

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 do the operating system and the JVM deal with them. It is out of the scope of the paper 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.

IV. 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. We focused our evaluations on Warden while the verifier is running in the centralized mode to show the worst latency case for our verifier since in the centralized mode the tuples have to travel to the verifier machine then to next stages machines whereas in the distributed mode the tuples are sent directly to next stage tasks as will be discussed in section V.

TABLE 1. Warden's API.

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 data, 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.
`void` **`sendToVerifier`** `(string tupleAsString, int sequenceNumber, string taskIDOfNextLevel)`	`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. The return value is `void` but the API can be changed if there is a need to get a value back from the verifier.
`string tupleAsString` **`receiveFromV`**`()`	If this is a receiving task then Warden has 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.

A. 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(until recently) couldn't run standalone, instead Samza required two things to function: an execution engine to run Samza‘s tasks on (such as YARN), and a message passing pipleline to carry the streams of data for Samza‘s tasks (such as Kafka). 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 Flink and Samza please refer to their papers [8] and [9] respectively.

B. 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 (centralized) 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 has been run three times where error bars show the difference between runs. We used YARN (Hadoop) version 3.3.4, Flink version 1.16.0 and Samza version 1.5.0.

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. Another possible way for evaluation 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 other works [10], [11], [12], [13], [14], [15], 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 (Byzantine failures don't affect application finish time since corrupted results are compared and ignored in Byzantine mode. Hence the evaluation figures focus on failures that affect latency).

C. APPLICATIONS

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

1) 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 $\sim 500MB$). 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 the paper is on streaming frameworks, the evaluations are done on Flink streaming mode.

2) 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 pre-defined 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) to simulate different checkpoint sizes ($\sim100MB$ to $\sim500MB$).

D. 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 (until very recently) had only At Least Once consistency, 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.

Figure 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. From the figure:

1) Crash Exactly Once finish time (with checkpointing disabled) is faster than Samza with checkpointing enabled when the checkpoint size is $\sim500MB$/sec and almost as fast as the finish time when the checkpoint size is $\sim100MB$/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.

2) Increasing the checkpoint size will induce more time overhead. Even though both Flink and Samza strives 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 ($\sim1KB$/sec).

3) 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, some timeouts are part of YARN’s configurations and it is out of the control of Flink or Samza users. For example, most frameworks run their tasks inside YARN containers. The timeout for these containers is set in the NodeManager configs which is shared between all frameworks running on top of YARN. Changing this timeout may affect other applications from other frameworks running on top of YARN and most likely it is out of the control of the framework’s users. In fact, most of these configs are editable only by the admins of the cluster.

Moreover, 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 I-A1 for more details about the timeout problem.

4) 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.

5) 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

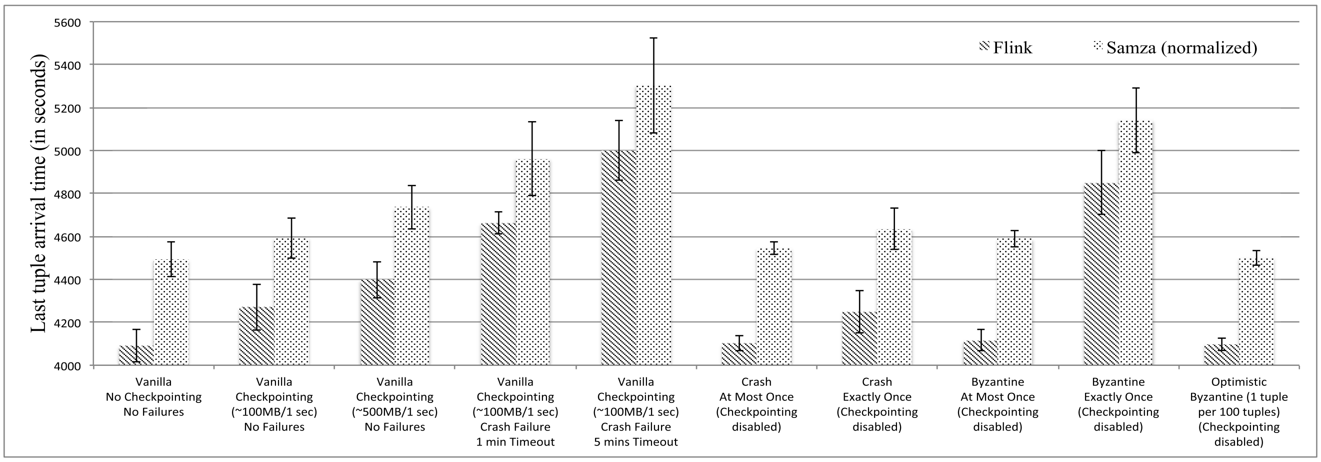

FIGURE 4. **Flink and Samza applications on Warden (merged into one figure due to large caption per column).**

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.

6) 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.

7) 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.

E. 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 I-A5, hardware is getting cheap, including decent memory and CPU chips. Moreover, many datacenters have idle commodotiy 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 II-C, 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.

F. 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 5 shows this case when Flink was tested on YARN, Flink version 1.6.0 failed to timeout and

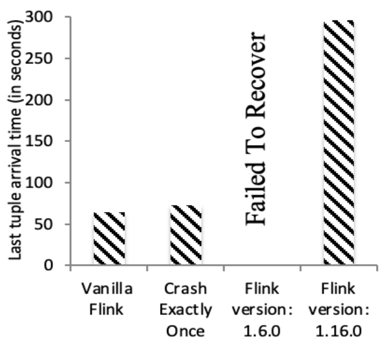

FIGURE 5. Unexpected behaviour from failing the AM of the same application running on different versions of the same framework.

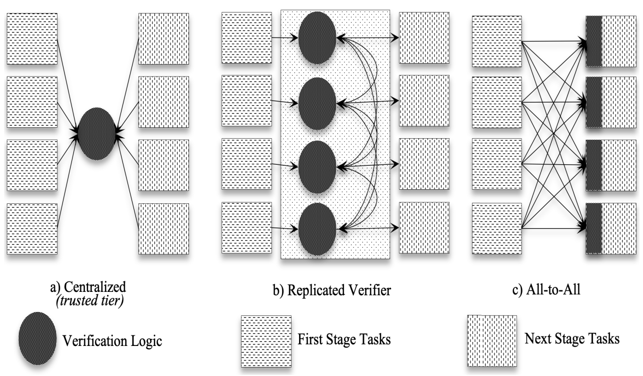

FIGURE 6. Replicating the verification logic.

restart the job when the machine that has Flink‘s AM crashed. Although this was fixed in later versions of Flink, 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.

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, 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.

V. DISCUSSION

So far, in all our implementations and evaluations, there is a centralized component that all traffic goes into, this component is the verifier. The verifier so far is a single point of failure. We assume that there is a single verifier running in a ‘trusted tier’. There have been some discussions on replicating the verifier itself.

Figure 6 shows some suggested designs to replicate the verifier in case it is moved out of the trusted tier:

- Figure 6-a shows the design that we used in this paper which a centralized verifier running in a trusted layer.
- Figure 6-b shows a design of a replicated verifier where four replicas of the verifier communicate with each other to reach a consensus on the data that will be pushed to the next layer.
- Figure 6-c shows a design of Warden where the verification is done inside the tasks themselves (i.e the verifier is inside Warden’s library that is attached to the task code). There is a ‘mini’ verifier inside each consuming task that is working as a ‘gatekeeper’ where the verification logic is applied. In this mode, the tasks will communicate with each other before they begin working on the input data received from previous stages.

There are several reasons why we didn’t consider implementing and evaluating these options:

1) Network communication overhead: The network overhead of sending tuples between the four replicas of the verifier will be much higher than processing the tuples locally. In Figure 6-b(Replicated Verifier), each tuple will be sent to its corresponding verfier then each the four verifier replicas have to communicate between each other to reach a consensus. A similar

overhead occurs in Figure 6-c(All-to-All) where the number of messages sent through the network is much higher than the number of messages sent in Figure 6-a (Centralized). This network communication overhead can be mitigated if all traffic are sent through a centralized component which has a global vision of all data coming from all tasks replicas and making consensus decisions without the need to communicate with external verifiers to reach a consensus.

2) Decoupling verification from processing: In Figure 6-c (All-to-All), the verification logic has to be loaded into every replica of every task in the application. This may cause some issues:
   - Resource provisioning will become more complex since the client has to count for both its application resource requirements and the processing power needed for loading and executing the verification logic in the background of the execution unit (virtual machine/container).
   - The verification logic has to be loaded into the memory of each replica of each task at every stage of the application execution. Whereas, in case of a centralized verifier, the verification logic is already loaded into memory (and cache) and there is no need to re-execute any init scripts related to the verification. In other words, the centralized verifier is more optimized to execute the verification logic than in the task itself.
   - It may not be suitable for some applications where developers prefer to decouple verification logic from the tasks processing logic completely (e.g. to reduce resource consumption in the processing machines).

3) Paper scope: The focus of this paper is on providing fault tolerance and reliability at the applications and tasks levels, not at the level of system components. We can investigate other approaches for providing system level reliability not only to the verifier but also to YARN components such as the Resource Manager and the Node Manager, or to HDFS components such as the Name Node and the Data Node. In fact there are some research papers that focuses on replicating system level components in particular such as Upright [13] and ZooKeeper [12].

Due to the above reasons we decided to stick with a centralized verifier that is placed in a trusted machine. A feasibility study can be made to quantify the pros and cons of the different approaches in Figure 6 but this can be done as part of the future work.

VI. 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 [16] and Spark [17]. 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 [18] and hence Apex will not release any new updates or versions and may actually stop supporting current releases. We tried to avoid Spark, although its popular, due to the fact that Spark has the RDD model [17] 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.

Some active projects such as Mesos Marathon [19] and Google Kubernetes [20] 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 and neither have any mechanisms to ensure the correctness of the streams.

Papers like Medusa [21], Arora [22], Borialis [23] are not comparable to our system since they don‘t target multiple frameworks (i.e. not at the RMS level). They do however use replication for fault tolerance. Dryad [24] 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 restart 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 [25] 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 [26] 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 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. [27] 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. [28] proposed a checkpointing mechanism where the checkpoints are saved and distributed in a replicated file system like HDFS. Guardian [29] proposed an active replication approach that targets batch processing frameworks only. Our 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 [30] 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 also investigated other systems such as Apex, Spark, Storm, Heron, Kafka Streams, Mesos Marathon and Google Kubernetes. We still have an edge over all of the related work due to three main reasons: First, none of the related work targets multiple streaming frameworks as we do. Second, task and application finish time 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 with 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 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.

VII. CONCLUSION AND FUTURE WORK

In this paper, we 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 mulit-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. In the future, we plan to build 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.

REFERENCES

[1] *Flink Checkpointing Constraints*. Accessed: Feb. 3, 2022. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.7/ops/state/state_backends.html/

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, and R. Evans, "Apache Hadoop yarn: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–6.

[3] 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," in *Proc. 8th USENIX Symp. Netw. Syst. Design Implement.*, 2011, pp. 1–4.

[4] D. Bernstein, "Containers and cloud: From LXC to Docker to kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[5] *Docker Swarms*. Accessed: Dec. 17, 2022. [Online]. Available: https://docs.docker.com/engine/swarm/

[6] M. Schwarzkopf and A. Konwinski, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. Eurosys*, 2013, pp. 351–364.

[7] 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," *Proc. VLDB Endowment*, vol. 7, no. 13, pp. 1393–1404, 2014.

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

[9] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "SamZa: Stateful scalable stream processing at LinkedIn," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[10] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Trans. Comput. Syst.*, vol. 29, no. 4, p. 12, 2011.

[11] Y. Wang, L. Alvisi, and M. Dahlin, "Gnothi: Separating data and metadata for efficient and available storage replication," in *Proc. 2012 USENIX Annu. Tech. Conf.*, 2012, pp. 413–424.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Annu. Tech. Conf.*, 2010, p. 9.

[13] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 277–290.

[14] 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," in *Proc. 11th USENIX Symp. Operating Syst. Design Implement.*, 2014, pp. 81–96.

[15] F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li, "Hadoop high availability through metadata replication," in *Proc. 1st Int. Workshop Cloud Data Manage.*, 2009, pp. 37–44.

[16] *Apache Apex*. Accessed: Feb. 3, 2022. [Online]. Available: https://apex.apache.org/

[17] 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 *Proc. 9th USENIX Symp. Netw. Syst. Design Implement.*, 2012, pp. 15–28.

[18] *Apache Apex Shutdown*. Accessed: Feb. 3, 2022. [Online]. Available: https://www.datanami.com/2018/05/08/datatorrent-stream-processing-startup-folds/

[19] *Marathon*. Accessed: Dec. 17, 2022. [Online]. Available: https://mesosphere.github.io/marathon/

[20] *Kubernetes*. Accessed: Dec. 17, 2022. [Online]. Available: http://kubernetes.io//

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

[22] 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," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[23] D. J. Abadi, "The design of the borealis stream processing engine," in *Proc. CIDR*, vol. 5, 2005, pp. 277–289.

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. 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," in *Proc. LSDS-IR*, 2008, pp. 1–14.

[25] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez, "StreamCloud: A large scale data streaming system," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst.*, Jun. 2010, pp. 126–137.

[26] 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 *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 1–14.

[27] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik, "A cooperative, self-configuring high-availability solution for stream processing," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 176–185.
[28] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 574–585, Aug. 2008.
[29] B. Abusalah, D. Schatzlein, J. J. Stephen, M. S. Ardekani, and P. Eugster, "Dependable cloud resources with guardian," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 1543–1554.
[30] 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 *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst.*, Jun. 2010, pp. 138–148.

BARA ABUSALAH received the master's and Ph.D. degrees from the Electrical and Computer Engineering Department, Purdue University During his Ph.D. degree, he did two research internships—one at the NEC Laboratories, NJ, USA, and the other at the HP Laboratories, CA, USA. His research interests include fault tolerance techniques applied to big data frameworks (Hadoop, Spark, Storm, and Tez) and cluster management systems (YARN, Mesos, Omega, and Fuxi).

THAMIR M. QADAH (Member, IEEE) received the Ph.D. degree from Purdue University, West Lafayette, IN, USA, in 2021. He is currently an Assistant Professor with the Computer Science Department, College of Computer and Information Systems, Umm Al-Qura University, Makkah, Saudi Arabia. His research interests include designing and implementing secure, dependable, and high-performance software systems that exploit modern hardware technologies and cloud infrastructures. His research on queue-oriented transaction processing was recognized with the Best Paper Award in Middleware'18. Since 2015, he has been serving the research community as a Reviewer for top-tier conferences, such as SIGMOD, VLDB, ICDE, ICDCS, ATC, EDBT, Middleware, and CIKM, and a Reviewer for the IEEE Access journal. Moreover, he served as a Committee Member of the Artifact Evaluation Committee for ASPLOS, OSDI, and SOSP.

JULIAN JAMES STEPHEN received the Ph.D. degree in computer science from Purdue University, IN, USA. He is currently a Scientist working as a part of the Security Department at IBM T. J. Watson Research Center, NY, USA. His work also involves improving security policies for cloud native applications. His research interests include building systems and models that solve real world problems without compromising security and privacy of data.

PATRICK EUGSTER received the M.S. and Ph.D. degrees from EPFL, in 1998 and 2001, respectively. He has been a Full Professor at the Universitàdella Svizzera Italiana (USI), since 2017. Prior to that, he was a Faculty Member of Purdue University (2005–2016) and TU Darmstadt (2014–2017) and still an Adjunct Faculty Member and a Researcher in those institutes. He has also been a Visiting Faculty Member of MIT (2012 and 2013). He is also the Co-Founder and the Chief Scientist of SensorHound Inc.