An Analysis of a Checkpointing Mechanism for a Distributed Stream Processing System

Mariano Vallés
Universitat Politècnica de Catalunya- BarcelonaTech
Kungliga Tekniska Högskolan

Master thesis dissertation for the European Master in Distributed Computing

Supervisor:	Flavio Junqueira (Yahoo! Research Barcelona)
Rapporteur:	Leandro Navarro (UPC)
President:	Felix Freitag (UPC)
Secretary:	Jordi Guitart (UPC)
Vocal:	Johan Montelius (KTH)

July 2012

Acknowledgements

I would like to thank my supervisors Flavio Junqueira (Yahoo! Research Barcelona) and Leandro Navarro for the constant help and the advice throughout this project. Matthieu Morel, a very active Apache S4 commiter, from the Yahoo! Research lab, was also always willing to help me, and provided me with great ideas for the development of this thesis.

My classmates from the EMDC course have also been a great source of inspiration and discussion through the development of this work. Last, but certainly not least, I am very grateful to my family and Flor who have always been there to support me and make me go one step further.

Hosting institution

Yahoo! Research Barcelona

Yahoo! Inc. is the world's largest global online network of integrated services with more than 500 million users worldwide. Yahoo! Inc. provides Internet services to users, advertisers, publishers, and developers worldwide. The company owns and operates online properties and services, and provides advertising offerings and access to Internet users through its distribution network of third-party entities, as well as offers marketing services to advertisers and publishers. Social media sites consist of Yahoo! Groups, Yahoo! Answers, and Flickr to organize into groups and share knowledge and photos. Search products comprise Yahoo! Search, Yahoo! Local, Yahoo! Yellow Pages, and Yahoo! Maps to navigate through the Internet and search for information. Yahoo! also provides a large number of specific communication, information and life-style services. In the business domain, Yahoo! HotJobs, provides solutions for employers, staffing firms, and job seekers; and Yahoo! Small Business that offers an integrated suite of fee-based online services, including web hosting, business mail and an e-commerce platform.

In 2006 the Yahoo! Research Barcelona lab was opened in cooperation with the Barcelona Media Foundation.Yahoo! Research Barcelona focuses on web retrieval, data mining, social networks, and distributed computing. Yahoo! Research Barcelona led the FP6 SEMEDIA project and is currently a partner of the FP7 Living Knowledge, WeKnowIt, Glocal, COAST and VIPER projects. This project was developed within the Scalable Computing Group led by Flavio Junqueira.

Abstract

In recent years the need for non-traditional data processing systems has led the way for new platforms in the area of distributed systems. The growth of unbounded streams of data and the need to process them with low latency are some of the reasons for the investment in stream processing systems. These systems are distributed to achieve higher processing performance and, additionally, their incoming data is distributed in nature. Despite the many advantages a distributed stream processing system offers over a centralized solution, its complexity increases. Providing high availability through fault tolerance mechanisms is one the complex problems these systems present.

In this work we provide an analysis of the checkpointing mechanism in the Apache S4 platform and propose alternatives for its checkpoint backend storage subsystem. S4 is an open source stream processing platform developed by Yahoo!. Checkpointing is included in S4, in addition to its partial fault tolerance mechanism, to decrease the loss of state by a processing element in case of a failure. The correct and accessible storage of the checkpoints is handled by a component known as a storage backend. We provide two alternative storage backends and compare them to the existing solutions. The first one is based on append operations, thus improving the previous file system solution by decreasing disk access times. The second one was developed using HBase, a distributed datastore optimized for write operations which are buffered and written in batches, while preserving their durability through write-ahead logging.

Considering read operations are infrequent, our solutions are optimized for high loads of write operations concurrently accessing the backend. Our evaluation shows that better access patterns for this particular load benefit the checkpointing mechanism in S4, and a constant trade-off between durability of data and performance has to be considered when choosing among the different backends.

Keywords

Distributed Stream Processing Systems
Apache S4
Fault Tolerance
Checkpointing and rollback recovery
Distributed data stores

Contents

List of Figures

1 Introduction

Since the introduction of Relational Database Management Systems (RDBMS), approximately 30 years ago, there has been a growing interest in the ways to efficiently store, organize and analyze data. With the advent of the Internet, Web-scale applications often collect and process data to serve users in various ways: learning usage patterns, filtering information and classifying content. The amount of data in these applications has been growing continuously, therefore, new techniques for the processing of it have been proposed. An interesting group of systems, developed after the need to process large data volumes with a strong focus on the low latency nature of it, are the Stream Processing (SP) systems.

As opposed to the traditional offline batch processing model found in RDBMS, where data is stored permanently and processed asynchronously once the user requests it; SP systems provide a model for prompt processing of data on-the- fly using a set of continuous queries over incoming streams. Given the high volume of the data, SP systems rarely store the incoming streams permanently, in fact, the computation is done while streams are flowing through the system. Moreover, important gains in performance and efficiency can be obtained by processing these streams in a distributed manner. Stock market trading systems [5], sensor based networks [24], online fraud detection[37], and online advertising optimization[41] are some examples of applications running on distributed stream processing systems. However, the complexity of these distributed systems increases when compared to a centralized solution. Therefore, fault-tolerance mechanisms have to be used to guarantee high availability of them. Furthermore, many of these applications are crucial in the daily operations they support, therefore, fault tolerance mechanisms must not affect the continuous operations of the systems.

S4 [36] is one example of a fully distributed SP system. It was initially developed by Yahoo! for efficient stream processing in search advertising personalization, and it was later released under an Apache open source license in 2010. The designers of the platform sought for a flexible architecture to allow easy deployment of new algorithms for research, while still enabling the platform to be scalable and highly available to be used in production environments. S4 currently offers a fault tolerance mechanism allowing faulty nodes to be replaced by an idle node in the cluster, and uses a coordination system as a failure detector. Moreover, a checkpointing mechanism has been recently

introduced not to avoid losing events, despite this is tolerable in the S4 design, but to avoid the loss of accumulated state over extended periods of time on the processing elements. The current checkpointing mechanism in S4 was implemented to asynchronously serialize and save the state of the processing elements (S4 operators) in stable storage backends. Hence, by processing the checkpoints asynchronously, the time required by these operations affects the stream processing critical path the least.

Nevertheless, the requirements different applications have regarding durability and the characteristics their loads impose on the backend has not been yet studied. An analysis of the properties offered by different backends will yield different results on the performance of the checkpointing mechanism.

This work explores the requirements stream processing applications impose on the S4 checkpointing mechanism and analyzes how different storage backends can support these demands more efficiently. We improve one of the current backends, that uses the file system, by leveraging sequential operations over magnetic disks. Our solution proposes a checkpoint mechanism where durability is guaranteed at all times, with a strong focus on high write workloads. This allows the system to improve the performance on the storage of the checkpoints, which is a frequent operation, while imposing a higher overhead on the recovery, which we assume is not as frequent.

Finally, an analysis of the properties that different open source distributed storage alternatives provide is included, to consider them as alternatives for a backend solution. We have selected one of these systems, evaluated and compared its performance to the available solutions. This allows us to observe the behavior of several alternatives in different scenarios by using a benchmarking tool specifically designed for the purpose.

1.1 Motivation

The area of stream processing systems has gained a lot of attention in the past few years. Many of the stream processing systems have been designed to run in a centralized manner[2], while many others use a distributed and decentralized model. Even though these distributed systems offer many advantages over a centralized solution e.g, better performance on higher loads, the introduction of more computing nodes increases the probability of a failure of these components and increases the complexity of their architectures.

A significant set of systems has been developed and much work has been devoted in the area of fault tolerance mechanisms in distributed SP systems[8, 9, 12, 28, 29, 49]. Since to continue processing streams of data, distributed SP systems have to provide resilience in the case of hardware, software or network failures. Moreover, machines undergo scheduled maintenance in large data centers comprising thousands of nodes, thus they must be shut down and re-started with high durability of the data and guaranteed consistent states.

Many distributed SP systems achieve fault tolerance by replicating the processing elements either actively, such that all the data processed by one node is

also sent and processed by a secondary node, or passively, by asynchronously sending the process state to a secondary node, which will be ready to replace a faulty node. Other work [29], has focused on exploiting the flowing nature of data. To achieve lower overhead in fault tolerance, than that offered by process replication, this work has focused on allowing only upstream nodes to act as backups for downstream ones. Finally, another way to achieve fault tolerance is to perform asynchronous and periodic checkpoints of the state of the nodes. This is a common practice in distributed systems which ensures that a system's state is stored in a coordinated or uncoordinated manner, and rollback recovery can be performed based on this saved state. To guarantee recovery, the checkpoints have to be stored in stable storage which must be accessible during failure-free execution and must also survive failures.

For the specific case of S4, the current checkpointing mechanism could be further improved by analyzing how its stable storage component suits each scenario better. The initial release of S4 included a failure recovery mechanism where if a processing element failed, it would be, upon recovery, instantiated in a separate node. However, the processes' state, created at that node, would be lost if a failure occurred. On the subsequent version of the system an uncoordinated checkpointing mechanism was implemented to avoid complete loss of this state over long running executions of the system. One scenario where the checkpoint/recovery mechanism would be useful is, for example, an application training a model by processing streams of events. If the model has accumulated data over many months, the loss of such data is unacceptable.

In this work we focus in the cases in which the current checkpointing mechanism is not able to perform as desired since the stable storage component, known as the backend, presents certain restrictions. Therefore, an analysis on how applications impact the platform as well as an implementation of different backend solutions are included as part of this work.

1.2 Contributions

This work addresses the problem of providing a durable and high performance stable storage system for the checkpointing mechanism in S4 according to the properties this mechanism has. More precisely, it analyzes the case where high load of events modify the state of the processes in the S4 platform, thus many checkpoints are triggered representing a problem for the checkpointing component. Consequently, this work makes the following contributions:

- An analysis of the problems presented in the asynchronous uncoordinated checkpointing mechanism in the Apache S4 stream processing system when its load increases substantially.

- A solution to the problem of high workloads in the checkpointing mechanism that needs no further dependencies other than a distributed file system. This solution relies on append-only operations to disk, exploiting

certain properties from magnetic disks that benefit regular write operations while penalize infrequent read operations. In order to locate the different records being appended, a index is kept in memory.

- An analysis of different distributed data stores that would allow the backend for the S4 checkpointing mechanism to support high concurrent write loads. After a detailed analysis, an implementation using Apache HBase is included and its performance is evaluated.

1.3 Results

The results produced by this work can be enumerated as follows:

- A write-optimized file system backend for the Apache S4 checkpointing mechanism which can write checkpoints to stable storage 4.8x faster in average than the current file system based solution while performing write operations over an NFS distributed file system.
- Micro-benchmark results show our file system based solution achieves higher throughput in write operations over the previous solution.
- Our write-optimized file system backend can tolerate as many as 1400 events every second, compared to a maximum of 400 events in the previous file system backend, while using a three-node S4 cluster.

1.4 Structure of the Document

The rest of the document is organized as follows:

Chapter 2 provides a description of different available stream processing systems and how they differ from each other. A description of checkpoint based recovery mechanisms, as background for this work, is also included. The details of the Apache S4 platform are also covered in this section. In addition, the analysis of different distributed storage systems is part of this chapter, in view of their implementation as a backend for S4.

In Chapter 3 the main contributions of this thesis are described. We start by justifying the need for a more adequate backend to support the checkpoint mechanisms in S4, due to the characteristics present in its load. Later, the proposed solutions are described focusing on their architecture and the differential design decisions that can solve the problem.

Chapter 4 Chapter 5 presents experimental results, including results of micro-benchmarks and a benchmark from a distributed setting. Finally, in Chapter 6 we present our conclusions and suggest further improvements in the S4 checkpointing mechanism in the Future work section.

2 Background and Related Work

This chapter is introduced with an overview of different stream processing systems and fault tolerance mechanisms employed in this area. This is followed by a description of the characteristics checkpointing mechanisms in distributed systems have, to contextualize our work. A detailed description of the design and architecture of the Apache S4 platform is also included. Finally, we review different backend technologies that were analyzed as candidates to be used as part the S4 checkpointing system.

2.1 Distributed Stream Processing Systems

The increasing interest in applications handling high-volumes of streams of data has driven the development of stream processing (SP) engines, where streams are processed obtaining results with the lowest latency possible. A broad number of SP systems exist nowadays, ranging from academic-driven projects to industry based solutions.

As an opening note, Stonebraker et al. proposed in [46] some requirements a real-time SP system should include. This work provides a set of requirements needed for the systems to be competitive and acts as a guide to consider the features of the different available alternatives. While some of these requirements are of utmost importance, others could be included in SP systems adding value to them, but are not completely necessary. For example, reducing the processing latency by avoiding costly storage operations, such as disk access in the processing path is of high importance for any SP platform, as it is the use of *active systems* where applications are not required to continuously poll for incoming data.

In the area of Distributed Stream Processing Systems (DSPS), the requirements proposed by Stonebraker stating scalability and partition of applications should be transparent for the end users are remarkably important. Furthermore, the guarantee for High Availability (HA) of the processing system e.g. using a Tandem-style [10] hot-backup and failover scheme is of great relevance to this work and to any DSPS.

Additional requirements for SP platforms proposed by the Stonebraker include the use of a Stream Processing query language, in analogy to what Structured Query Language (SQL) provides in the DBMS area, and the integration

of 'Archive' data into the system for processing, such that an initial state could be created for any platform.

In [18] a more abstract term *'Information Flow Processing'* is used to refer to the broader area of SP systems. Moreover, the authors claim that two main models can be found in this area: the *data stream processing* model and the *complex event processing model*.

The *data stream processing* (DSMS) model produces output from different streams of data and usually extends the SQL query language with stream related operators (e.g. time windows operators) as well as supporting typical SQL queries such as join, aggregate and count. It is clear that this model is closely related to the one present in the community of researchers devoted to databases, to the extent that some authors [7] refer to their systems as Data Stream Management Systems (DSMS), a term most certainly derived from DBMS. However, while DBMS process persistent data stored with infrequent updates, DSMS process online data, rarely persisted and frequently updated, and in contrast save the queries and operations on this data in a permanent manner.

Conversely, in the *Complex Events Processing* (CEP) model, processed data is not generic, as it was in the DSMS systems, but it represents notifications for different events with its associated semantics. Therefore, precise notifications of real world events constitute the events. The CEP platforms are in charge of filtering, aggregating and combining them to produce high-level events that describe higher levels of abstractions.

Data Stream Management Systems: DSMS

An academic project known as Aurora [2, 15] from Brandeis University, Brown University and MIT is a clear example of a DSMS. Aurora is a centralized stream processing system that considers data streams as an unbounded collections of tuples from different sources and uses a graphical representation of *boxes* and *arrows* to specify the processing flow. Accordingly, data, represented by arrows, flows in and out of processing operators (boxes) forming a directed graph that describes the complete application. A set of query operators are defined in its query algebra (SQuAl). Seven primitive operators are specified, many of which are analogous to the same relational query operators: filter, union and join. Aurora's single-node architecture is orchestrated by its scheduler component which determines which operator (box) to run and it additionally receives Quality of Service (QoS) information of the output tuples in the system. The latter is an important mechanism for the system to identify when to discard tuples about to be processed, a technique know as *load shedding*.

The Borealis stream processing engine [1] is a follow up project from Brandeis University, Brown University and MIT that also falls under the category of DSMS systems. Its core components are based on its predecessor Aurora, however, a system known as Medusa [15] is used to expand Borealis and achieve distributed functionalities. Borealis claims to be a second-generation SP sys-

tem, since its behavior diverges from the previous systems, mostly due to its distributed nature, but also since it offers extended functionality in the processing of data. Three main contributions are outlined by the authors of this work. The first one is the possibility to dynamically revision the query results, in order to correct previously observed errors in the data. The second contribution is the dynamic modification of the queries in runtime: allowing to operate in different ways on data depending on factors such as load of the system or changes in the application goals. The third contribution is a flexible and scalable optimization mechanism including resource management and fault tolerance, to handle higher loads of data efficiently, guaranteeing tolerable Quality of Service (QoS) metrics.

STREAM[7] is a centralized data stream management *DSMS* system from Stanford University which is focused in continuously querying data flowing through the system using a specialized query language. The system's architecture is focused mainly on the registration of Continuous Queries that drive the systems' processing logic.

By using CQL (Continuous Query Language), a superset of SQL with added constructs for *sliding windows* for example, the STREAM platform physical query plans are built. This allows for optimizations and fine grained scheduling decisions to be made.

An adaptive approach is taken in STREAM towards query execution so that a continuous query can be modified as the data, load and system characteristics change over time. StreaMon is the component of STREAM which monitors and re-adapts queries while the system is running.

In [6], the authors point that STREAM future directions plan to distribute their query plans across a number of nodes given the nature of the incoming data being distributed. However, it can be read in the project's official website, that further work on it has been abandoned as of 2006.

While the majority of the aforementioned systems rely on real time data, the operations applicable to this data are constrained by the set of operators defined by the platform's query languages. This is one of the major differences between the DSMS systems and Apache S4. S4 follows a model where users can specify the operations to perform on data. Additionally, it does not even include a query language as a default method to perform queries. Nevertheless, the work presented in [45] as S4Latin, presents an innovative approach to include a query language similar to PigLatin[39] in S4.

Although some similarities can be found between the boxes and arrow model in Aurora/Borealis and the Actors model in S4, many differences still exist. The most remarkable features of Borealis, that S4 lacks, are its capacity to replay tuples (i.e. events) to correct previously erroneous data and the dynamic modification of queries while data is processed in the system.

The Apache S4 project is a clear example of a CEP system since it differs greatly from the previous DSMS systems. In S4, the users are allowed to define their own operations on data as well as using predefined operators (e.g. join), and the focus is put on the processing of low-level events that output streams

as their results. Events are characterized and semantics relative to each event type can be specified.

The difference between CEP and DSMS can be better understood, given the origin of these systems. DSMS systems are derived from DBMS systems, hence SQL operators are extended to fit time-constraint datasets. In contrast, a system like S4 was developed after the necessity of detecting patterns in search engine users with regards to advertising.

Complex Event Processing Systems (CEP)

Amini et al. presented the Stream Processing Core (SPC)[4] in 2006. SPC is a distributed stream processing middleware that supports stream-mining applications using a subscription-like model. In addition, due to its supports for non-relational operators it can be considered a CEP system.

SPC offers the possibility to extend the relational operators offered by most stream processing platforms, with user defined operators. This enables the use of user-specified semantics in the processing of data. The main functional components in the SPC programming model are known as Processing Elements (PE), where these operators can be defined. Data from a collection of streams is sent to a PE using an input port. Once the incoming data is processed by the PE, the results will be output through an output port, hence creating new streams.

As opposed to most SP systems, SPC does not pre-compile an application's flow graph prior to its deployment. SPC dynamic stream connections are set using a subscription-like model. This allows the system to discover and connect to new streams, or determine, dynamically, relevant streams a PE instance's connection would benefit from.

SPC is a system comparable to Apache S4 in many aspects since they are both designed to handle high-volumes of data in a distributed manner and they both allow users to define operators on the data. An important difference is that, while SPC uses a subscription-like model, S4 is based on a combination of MapReduce[20] and the Actors [3] model. The creators of S4 argue in [36] that this allows their system to achieve a greater level of simplicity, since all the nodes are symmetric and no centralized component controls the cluster.

Twitter's Storm[35] was released in 2011 as an open source project and is focused on providing distributed real time processing of data. Much like Apache S4, Storm was developed after the need to provide computation, similar to the way Hadoop processes in a batch mode, but in a real time manner. Storm processing units are known as 'bolts', where filters, aggregates and user-defined operations can be specified. This makes Storm another system in the category of CEP. Spouts are the sources of streams and together with bolts, they form higher abstractions called 'tasks'.

A comparison between Storm and S4 highlights a set of interesting similarities and differences. Both systems use Zookeeper[27] in their fault tolerance mechanism, and as a tool to assign tasks to nodes in a cluster. Additionally, an important difference is that Storm offers a strong guarantee on message

processing, while S4 allows the platform to perform load shedding, thus losing the processing of some incoming events in the case of a load peak. Additionally, Storm allows the user to define the level of reliability on incoming tuples, so that a series of 'Acker' processes can follow their completion or failure to be processed in the system. This is a tunable parameter, and its algorithm is designed to consume as less memory as possible.

Finally, Storm's architecture is composed of several workers, multiple nodes, and a centralized component which assigns worker to nodes called Nimbus. Nimbus represents a single point of failure in the system, hence if it fails, no further worker-to-node assignment will be possible. In contrast, S4's architecture is completely symmetrical among nodes, guaranteeing that the crash of a node will not stop others from getting work done.

Fault Tolerance in Distributed Stream Processing systems

Much work has been devoted to the problem of fault tolerance in distributed systems in general. A failure is considered the situation where "a server does not behave in a pre specified manner" in [17]. Moreover, Cristian categorizes failures into: *omission failures*: a server omits to respond to an input, timing failures where the service response is functionally correct but untimely and *response failure* where the server responds incorrectly, i.e. the value or state transition are incorrect.

The term 'failure semantics' can be used to describe the behavior of a system in the absence of failures. In [17] it is exposed how stronger failure semantics (more restrictive) are harder to guarantee, and therefore, a system that offers stronger failure semantics is often more complex. In the area of DSPS, usually crashing failures semantics are guaranteed. Many alternative solutions have been proposed to achieve crashing failure semantics. Masking failures through replication of the processing elements using a tandem style [10] "hot-backup" replication scheme is the most common approach, given the low latency requirements stream processing systems must fulfill. In this approach, also known as process-pairs [10], a spare backup server frequently synchronizes the state of a primary server, and takes over in case of a failure of the latter.

In [28], a set of high availability algorithms for stream processing systems are compared. The authors firstly describe two different algorithms for process-pair replication. On the one hand, in *passive standby*, each primary server sends checkpoint messages, that include the state of each of the operators in it and the state of its input queues, to the backup node. In case of a crash, the backup node will recover from the last checkpoint saved. On the other hand, in the *active standby* algorithm, the replica not only saves all of the primary state, but it behaves exactly as the primary node, processing all the tuples in parallel with the primary node.

While both the passive and active standby mode offer a short recovery time, there might be occasions, where their mechanisms are too costly. One scenario is, for a passive replication, when the size of the state to be saved is too large, imposing high network bandwidth use. Moreover, if the checkpoint-

ing operation is performed synchronously, due to high consistency constrains, the system must block until the checkpoint is safely stored. To overcome these problems, Hwang et al. [28] proposed their own algorithm called *'Upstream Backup'*, which focuses on utilizing upstream neighbors as backup for downstream nodes. In this approach, upstream nodes must hold all the tuples in their output queues until a confirmation of their processing, from downstream nodes, is received. Moreover, if any of the downstream nodes fails, tuples can be replayed from any of its upstream neighbors.

The authors show how their proposed algorithm has a 10x improvement on bandwidth utilization over the passive and active replication schemes mainly due to avoiding sending duplicate tuples to backup nodes, although these results are achieved at the cost of increasing the recovery time for the system.

Balazinska et al. [9] propose a mechanism to manage the availability and consistency levels a stream processing applications desires to achieve, allowing their system to tolerate software failures, network failures and network partitions.

In this work, the level of latency an application is willing to sacrifice for the sake of improved fault tolerance can be specified on a per application basis. The approach is based on an ordering guarantee of the tuples sent downstream by the processing elements, which is obtained by using a data-serialization operator. Once a failure occurs, which is detected by the absence of heartbeat messages from an upstream node, a downstream processing element will try to contact a replica of its upstream node. As the replica's state should be consistent, only some tuples may be lost in the process. The system will mark these tuples as *unstable*, and will later try to heal them or not, depending on the level of consistency specified by the user with regards to this application. If an upstream replica cannot be found, then the system blocks waiting for tuples, converting a partial failure to a complete failure, therefore achieving a very poorly available system, but highly consistent.

In [29] the authors studied the use of a high availability algorithm where a set of servers act upon a failure based solely on checkpointing, i.e. incrementally copying the state of the processes to a backup node. Hwang claims checkpointing is preferred over process pairs, given that the former can be used in a larger set of workloads and use cases. In fact, in a parallel execution scheme, half of the process cycles in a SPE have to be devoted to replication, thus tolerating an increase in the load of the system, soon becomes unbearable. Since this work is based on the Borealis system, it uses partitions of the systems query graphs to form smaller subgraphs, and assigns them to different backup servers known as High Availability (HA) units. A balanced algorithm for assignment of HA units to backup servers is described along with a scheduling mechanism that determines the order and the frequency of the checkpoints. In addition, the load of the servers involved in the HA units assignment is also included, so that no extra latency is introduced in the stream processing path by requiring a heavily loaded node to perform additional tasks.

The cooperative algorithm proposed by Hwang et al. diverges from previous work since it offers lower recovery times by leveraging parallelization using a set

of servers in the checkpointing process, and it offers a no-tuple-loss guarantee, since if a tuple is lost, it can be replayed into the input queue from one of the backup servers. Nevertheless, checkpoints in the backup servers are stored in memory only, assuming the failure of one or more backup nodes is a rare condition, when it is not. In fact, the authors explicitly mention that saving the checkpoints to reliable storage should be trivial, when it is in fact the purpose of this work to demonstrate how this problem can be explored using many different techniques and it is not actually easy to solve.

2.2 Checkpoint-based rollback recovery

Checkpoint based rollback recovery is a commonly used mechanisms employed to provide reliability and high availability of distributed systems. Rollback recovery implies restoring a system from an erroneous state, into a previous correct state. To achieve this, the system must periodically save its state (or part of it), hence creating a 'checkpoint'. Understanding certain aspects of this topic are key for our work given its focus on checkpointing techniques to achieve rollback-recovery. An exhaustive study of these concepts is found in [22].

A checkpoint based approach for recovery of failed processes in a distributed system considers a system composed of multiple processes connected through a reliable network. The periodic state of the processes in a system is persisted to stable storage. In the case of a failure, the system can be restored to a consistent state represented by these checkpoints, thus reducing the amount of lost work.

Since checkpointing can be costly, at times, this mechanism is combined with message logging. Once a checkpoint has been saved, every operation following the checkpoint is saved to a log. The intermediate state, between two checkpoints can be recovered by replaying the logs in case of a failure.

Stable storage

Stable storage is an abstraction of where checkpoints can be saved while using a rollback-recovery mechanism. However, it is not to be confused with the disk storage it uses to achieve it. Stable storage is specific to the type of failures a system is designed to tolerate.

For example, for systems where only process failures are tolerated, stable storage can be sufficiently implemented by using the volatile memory of another process. Although, if transient failures wish to be supported, saving the state to the local disk in each node can solve the problem. Finally, to support non-transient failures, stable storage becomes more complex and must be located in an component external to the process, where a distributed file system can be used to support these failures. This last case defines precisely the concept of stable storage which the S4 checkpointing mechanism needs. We will refer to stable storage and storage backend interchangeably.

Checkpointing mechanisms

Each process participating in a rollback-recovery mechanism must save its state periodically to stable storage. To recover from a failure, a consistent global state must be recreated using the saved checkpoints.

Three different checkpointing techniques are described in [22] : *uncoordinated checkpointing*, *coordinated checkpointing*, and *communication induced checkpointing*.

In the case of uncoordinated checkpoints each process in the system performs a checkpoint of their state in an independent manner. Although, this requires less overhead while saving it, the recovery to a consistent global state becomes more difficult. This difficulty in recovery is illustrated in Figure 2.1. First, process P sends message M3 and later checkpoints its state. Afterwards, process Q receives the message and includes the reception of the message in its checkpoint. By recovering using this set of checkpoints, a consistent recovery line would be obtained. In contrast the checkpoints saved after message M4 has been sent, create an inconsistent recovery line, as process Q includes the sending of M4 in its checkpoint, but P does not. Hence, recovery using these checkpoints would leave the system in an inconsistent global state. Solving this problem requires rolling back to the previous set of checkpoints available which are consistent. However, this can lead to moving back along many inconsistent checkpoints, causing a cascaded rollback, what is also known as a 'domino effect'.

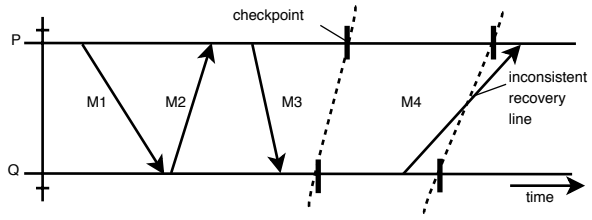

Figure 2.1: Uncoordinated checkpointing generating both consistent and inconsistent collection of checkpoint states.

In contrast, in coordinated checkpointing, all processes in the system jointly write their saved state to stable storage, providing it with a system wide consistent state and avoiding the possibility of cascading rollbacks. This requires a global synchronization mechanism. A simple algorithm to achieve synchronization is to block communication while the checkpointing is in progress. This approach requires a coordinator which starts by taking a checkpoint of its state and it later broadcasts a message inducing the other processes to do so. These processes must stop their normal processing when this message is received, take a tentative checkpoint and send an acknowledgement to the coordinator. The coordinator will finally reply with a commit message forcing the processes to atomically make the tentative checkpoint permanent. A more complex so-

lution is using the distributed snapshot algorithm described in [13] so that non-blocking coordination is achieved.

Finally, communication induced checkpointing (CIC) is a hybrid approach between the two previous approaches. Processes take two different checkpoints, local checkpoints that can be taken independently and forced checkpoints that enforce the recovery line to make progress. Basically, in CIC the checkpointing of useless sates is discouraged as it represents additional overhead to the system.

2.3 Apache S4

This section describes the Apache S4 platform in detail to gain full understanding of the design of the system and the component that this work relies on.

S4 system architecture

S4[1] is an acronym for "Simple Scalable Streaming System". The system goals are to provide a simple and flexible processing platform for unbounded streams of data where no central process exist and high scalability can be easily achieved by adding more nodes to the system. The design of the system is based on the Actors model [3] and allows it to offer location transparency and encapsulation mechanisms, thus building a highly concurrent platform. Moreover, by using the Actors model, the platform can be used in clusters of commodity hardware, since the use of shared memory mechanisms is avoided.

As opposed to offline processing applications (e.g. MapReduce), S4's on-the-fly processing nature provides some unique characteristics, namely, a low latency and high-throughput system with high scalability properties. Moreover, upon load spikes, S4 drops events from the processing queues on the processing elements. This is commonly known, in the stream processing community, as *load shedding*.

S4 offers the developers the ability to personalize the functionality of each processing elements in the system, and form directed graphs between them for the data streams to flow through. These basic units of computation are known as Processing Elements (PE). Data is sent in the form of events through different PEs. Hence, communication between PEs can only be done by emitting and consuming these events between different PEs.

Task to nodes assignment and configuration of the cluster is done by using a Zookeeper cluster management service. Allowing the system to be dynamically configured, and in addition, react upon the failure of a node, reassigning tasks to idle nodes.

[1] http://incubator.apache.org/s4/

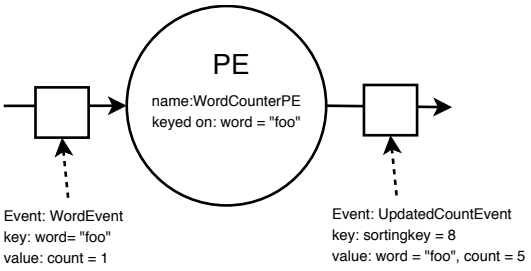

Figure 2.2: A representation of a Processing Element (PE) that counts appearances of words in S4. One instance of this element will exist for every occurrence of a word in the application.

Events

Events in S4 are simple Java objects which have a key and several attributes. Adapters convert external sources of information (e.g. data generated by Twitter stream feed) into S4 enabled Events. These events are passed into a PE in the system, then the PE state can be changed accordingly. After a PE state is updated, additional Events can be emitted on named streams.

Processing Elements

Processing Elements are the basic computing units in S4. Streams of events flow in and out of these elements and their state can be updated. Each PE instance is uniquely identified by four characteristics:

- *Functionality*: as defined by the class name and configuration parameters.
- *Type of events*: a list, of possibly one, streams from which the PE consumes events.
- *Keyed attribute*: on the received events
- *Value*: the value of the keyed attributes on the events it consumes.

Noteworthy is that there will exist one PE instance (in memory) per value of the keyed attribute. This instantiation is abstracted from the developer and handled completely by the framework. To illustrate this aspect, an example of a WordCounter PE instance is depicted in Figure 2.2. The PE class name is WordCounterPE, thus it receives events of type WordEvent and it is keyed on the tuple 'word="foo"'. Therefore, events with the corresponding key = 'word="foo"' will be sent to this PE, forcing the state to be updated to a new count value, and possibly outputting to other PEs. The emission of events to other PEs can be configured using two different mechanisms, either after a number of elements have been processed in that PE, or a certain period of time has elapsed.

A special type of PEs are those which have no keyed attribute or value, hence, they accept all events from the stream to which they are associated. They are known as "keyless" PEs and usually represent the first element in a processing path. If the system is running in a distributed cluster, there will be one instance of any keyless PE in every node in the cluster.

Given that many PE instances will be present in applications with large number of unique keys, a mechanism to remove the PEs from the system after a user-specified timeout is included in the system. The time to live (TTL) property of PEs allows elements which have not received any events after the TTL value has elapsed, to be removed from the system, with the subsequent loss of accumulated state for that PE instance. Further discussion on how this state can be saved is presented later in this work, specifically in Section 3.1.

Many useful PE implementations for typical data operations are bundled into the S4 framework as standard PEs. Allowing the developer to reuse common tasks in their applications. Some of the standard PEs provided are join, count and aggregate, which serve a similar purposes to those present in the SQL language definition.

Processing Nodes

Any PE instance runs within a Processing Node (PN). PNs are logical containers for PE instances and provide the means for events to reach the different PEs in the system, interact with the communication layer to dispatch events and emit output events.

Figure 2.3 provides an overview of a PN and the underlying layers in the S4 platform. The PN allows events to be received by using the Listener component. Later on, these are sent to the a Processing Element Container (PEC) which will deliver it to the corresponding PE. If specified in the PE, additional events are emitted using the event Emitter component, and/or events are sent to other PNs using the dispatcher component.

S4 events are routed to different PNs by using the Fowler-Noll-Vo hash function [26] of all the known keyed attributes in that event. Therefore, at most one PE instance will be present in the system for every keyed attribute.

A PE prototype is a special type of PE object which only has three of the components defined earlier: functionality, event type and keyed attribute. However, the keyed attribute is unset. An instance of a PE keyed on a previously unseen key attribute will be created by cloning the PE prototype object, although the key attribute will now be present. For example, the first arrival of an event with key: 'word="foo"' and value: count=1, as shown in Figure 2.2 will clone the PE prototype and create a new PE instance. This mechanism ensures that there will only be one PE instance keyed on a specific value per PN. In addition, all events containing such a key attribute will correctly arrive to the PN which contains the corresponding PE instance by the use of the previously mentioned hash function.

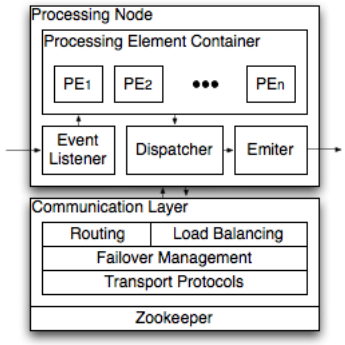

Figure 2.3: S4 Processing Node architecture. Extracted from [36].

Communication Layer

The communication layer handles the case where an event is emitted from one PE to another not present in the same PN. Since the communication layer manages the mapping of physical nodes to logical nodes, i.e. PNs.

By using Zookeeper as a cluster coordination tool, it is possible to assign tasks to a subset of active nodes present in an S4 cluster, leaving the remaining non-active as idle nodes that can be used in case of a failure of the former.

S4 Failure Tolerance Mechanism

The S4 fault tolerance mechanism is governed by the use of Zookeeper as a coordination tool. Zookeeper assigns one or more tasks to active nodes in the S4 cluster, and forms a pool of idle standby nodes with the remaining ones (to which no task has been assigned). A standby node can register to listen for one or more active nodes, thus replacing it in case of their failure.

In case of a hardware failure of an active mode, i.e. one holding a task, a failover strategy is adopted. An idle node will poll to see if it can acquire the corresponding task (represented in Zookeeper by using an ephemeral one of its data structures known as *znode*). If it succeeds, the task will now be assigned to this node. The newly active node announces the task assignment in Zookeeper, thus no other node can possibly acquire it.

As stated before, the loss of streamed data is an acceptable condition in many SP systems. For instance, S4 drops events in case of a load spike, also known as *load shedding*. Consequently, in the case of the failure of an active node, some events will probably be lost, however, that is allowed in most of the applications running on S4. This explains the reason for which S4 is regarded as a partial fault tolerance system.

Moreover, if an application has been running for several months, the accumulated state of the failed node would be lost upon a failure. Therefore, a

checkpointing mechanism has been designed to minimize the state loss in the case of a node failure and recover a crashed node state from the most recent checkpoint available.

2.4 Backend solution space

This section provides an analysis of a set of systems that can possibly be used as a storage backend for the S4 checkpointing component. The emergence of distributed data stores to be used in applications where traditional relational databases do not perform as expected [47] has opened a path for developing many different systems with different characteristics.

Some distributed storage systems relax the constrains set by the ACID properties (Atomicity, Consistency, Isolation and Durability) key in transactional databases. An alternative to the ACID properties was proposed by Brewer in [11], where to achieve better scalability, internet-scale systems should provide Basic availability, Soft state, and Eventual consistency. In such scenario, providing high availability and allowing a weaker consistency model results in a faster system.

Moreover, Brewer's conjecture, also known as CAP theorem [25], proposes that any distributed system can only provide two of three properties: Consistency, Availability and Partition Tolerance. A system with the ability to tolerate partitions of the network, as most distributed databases are, has to choose between serving highly consistent data or having data served with high availability (fast) but inconsistent in certain occasions. In the first case, a consensus protocol such as a distributed two phase commit algorithm can be used to achieve strong consistency, although this impacts greatly on the system's throughput, making it less available. On the other hand a highly available system, with a weaker consistency model such as eventual consistency [48] offers fast update operations at the price of having possibly inconsistent data returned by read operations.

A description of the different systems which could be used as a storage backend for S4 checkpointing is presented next. To conclude this section, Table 2.1 summarizes the analyzed systems with their characteristics which are relevant to this work.

Minimum requirements

We begin this section by highlighting certain requirements that must be present in any system to act as the checkpointing backend for S4:

1. Fault Tolerant: the supporting backend must tolerate failures such that the failure of one of its components does not affect the complete backend. Process pairing, logging of operations or regular checkpoints are some examples on how certain systems achieve this.

2. Distributed: in conformance to the fault tolerance requirement, a centralized backend would represent a single point of failure, therefore, by

having a distributed model the backend can, not only achieve fault tolerance, but also offer better performance, for example by aggregating read operations.

3. Durable: since data will be saved to support the fault tolerance mechanism of a system, durability of data must be a property present in the chosen system. For instance, in-memory databases achieve durability with a high level of replication of the datanodes or by periodically flushing its data to permanent storage, with the chance of losing intermediate states.

4. Consistency: a relaxed consistency model can be used, since a checkpoint read operation can return a stale value for a process. This would allow for all write operations to be performed with the lowest possible latency.

5. Client/Server architecture: deploying the backend system to each individual node would not allow recovery after the hardware failure of the node. Therefore, the system must me accessed using a client/sever architecture where the client is integrated into S4.

Distributed datastores

Distributed datastores are systems that are often regarded as NoSQL (Not Only SQL), since they differ from traditional RDBMS by relaxing their ACID properties and offering new data models such as columnar or key/value. Most of these systems have arisen after the need to provide higher availability to support operations from millions of users.

BigTable and HBase

In this context, BigTable[14] is a distributed data store able to handle petabytes of data across thousand of commodity hardware servers. Data is persisted by using a distributed file system known as GFS and coordination between the participating nodes is achieved by the use of Chubby, a distributed locking service. In addition, BigTable follows a structured data model where each row represents the key of a multidimensional sorted map. These rows exist in structures called tables, which are sparsely stored, hence each row can have an arbitrary number of columns.

HBase [2] is one of BigTable's open source counterparts. Its data model is very similar to BigTable's, allowing millions of rows to be stored in sparse distributed tables. To support its operation, it uses HDFS[43] similar to GFS. Having a distributed filesystem as the underlying layer, allows HBase to continue operating in the case of one of the storage nodes failing since data is replicated among the many datanodes in the file system.

[2] http://apache.hbase.org

As HBase is optimized for write operations, records are firstly written to a log using a Write Ahead Logging (WAL) technique for durability, later written to an in memory store (*memstore*), and finally flushed to disk once the *memstore* reaches a certain size. Files created on disk are immutable, thus two subsequent update operations on a record might be spread in different files. These files are created by using sequential I/O operations, which offer higher performance. Consequently, read operations are much slower than writes, since several I/O operations are needed to reconstruct a record from the different updates.

HBase's architecture is comprised of three main components: *HBaseMaster*, *HRegionServer* and *HBase client*. The *HBaseMaster* assigns regions to the *HRegionServer* and detects if any of the of the HRegionServers is unavailable. In that case, it will divide the WAL into the different regions the server was serving. Once finished, each region (HRegion) with its correspondent WAL will be reassigned to an available *HRegionServer*. The *HRegionServer* is the component that actually handles the read/write requests. An *HBase client* will connect to the *HBaseMaster* to retrieve the region server in charge of the key range of interest to it. On subsequent operations, clients will communicate directly with the *HRegionServers*.

In case of a failure of a region server, records that have not been persisted and remained in memory will be lost, however, by using WAL, HBase enables these records to be replayed. Using WAL for every write operations might increase their cost, however, it guarantees that all the changes to data will be present in the logs for future recovery.

Cassandra

Cassandra[31] is an open-source distributed storage partly inspired by BigTable and partly by Amazon Dynamo[21]. It resembles BigTable in its data model, where dynamic layouts for huge scale data sets are offered and each row can have multiple and variable number of columns. However, instead of using an underlying distributed file system, Cassandra uses a consistent hash algorithm to partition data across the cluster, enabling it to scale incrementally. Each node is assigned a random range within a "ring" generated with the hash key of each data item. Therefore, every other node is aware of who is responsible for each data item. This allows a read/write operation to get routed to any node in the "ring", which will determine the replicas for this key. Later, the request is served by one or more of the replicas depending on the consistency guarantees required.

Durability of data and high availability is achieved in Cassandra by replication. A particular data item will be present in N nodes, where N is a configurable replication factor. It is the job of the node responsible for the key (coordinator) of the item to replicate it to the $N - 1$ other nodes using one of the different replication policies, among "Rack-aware" replication, "Rack-unaware" and even "Datacenter aware" replication for deployments spanning multiple datacenters. On read operations, the consistency level can be specified

on a per-call basis, therefore, the request can be routed to the closest replica, or all of those containing the key, waiting for the complete quorum to return the results. Cassandra leverages each of the node's local disk to save records in an efficient manner. In fact, data is not immediately persisted to disk. Each update operation is saved in a commit log, which is similar to HBase's WAL. Once this log entry has been written, the item is stored in memory, and only when a memory buffer reaches a certain limit, it is flushed to disk, thus taking full advantage of sequential I/O operations. Since all write I/O operations are sequential, for faster disk access, an index is kept in memory for faster record retrieval based on the key. This index is also periodically persisted to disk along with the data files, to recover it in case of a failure.

Project Voldemort

Project Voldemort[3] [23] is self-defined as a distributed key value store developed by LinkedIn, and was also partly inspired by Amazon's Dynamo system. As opposed to HBase or Cassandra, it offers a simpler data model where the API defines as little as three operations: *put(key)*, *get(key)* and *delete(key)* operations. A typical Voldemort cluster can be seen as a consistent hash ring of nodes, where each of them is responsible for one or many partitions of the key space, and has one or more *stores*. A store is a structure similar to a table, however, configurable values can be specified to the *stores*. These values include: the replication factor, the number of required nodes to read a value from (before returning it) and the number of required write nodes for a successful *put*. In addition, Voldemort supports several storage engines, such as Krati[50] and Berkeley DB[38] where data is efficiently persisted to tolerate failures.

One remarkable aspect of Voldemort is its modularized architecture, allowing to interchange functionality in an easy manner. Each of the layers in the architecture are in charge of a specific function that implements *put* and *get* operations. Therefore, they can be assigned during runtime depending on the system needs. For instance, requests routing can be assigned to the clients or to the servers, in the first case reducing the load on the server, and in the second case, allowing for simple clients to be able to interact with the data store.

Voldemort offers eventual consistency guarantees, since it requires higher operational performance than the one offered by transactional based data bases. Allowing inconsistent data to be stored in the different replicas and repairing these inconsistencies at read time, which is known as read-repair consistency. This enables higher throughput of *put* operations at the cost of higher read times. Another approach that can be used in Voldemort, to tolerate node failures, is known as *hinted handoff*. When write operations are sent to a crashed node, a "hint" of the updated value is saved one of the alive nodes. Therefore, when the crashed node recovers, this value can be read by it. Both of these methods are further detailed in [21].

[3] http://project-voldemort.com/design.php

Finally, versioning is achieved by the use of vector clocks[32]. In this scenario every replica involved in a write operation has a list of the update operations it has observed from all the other replicas. The node responsible for the update will increment its version and pass the update and the vector clock to another replica. If two concurrent updates arrive from two replicas, the vector clock can be used to see which was performed first. If this cannot be determined, it remains for the application to solve the issue.

Redis

Redis[4] is an in memory persistent key value store. In addition to a one key-one value approach, Redis data model is expanded with several data structures. Maps, hashes, lists and sets are included in the system. For example, by using a hash, an extra level of indirection is given, thus a key is associated to many fields which can be accessed individually or as a whole. Lists and sets serve similar purposes associating a key to several values, and multiple unique values respectively.

Data persistency is configurable in Redis, and provided by two different methods: Append Only Files (AOF) logging and point in time snapshots, also known as *rdb* files. In the AOF model, write operations are logged sequentially to a log file using different policies (e.g. for every operation, every second, or not at all). This is performed using a background process of the system, thus allowing the main thread to continue serving incoming requests. In the case of a failure, a server can reconstruct its state by replaying the operations contained in the log with the probability of lost operations proportional to the frequency of the policy used to save the operations in the log. Using point in time snapshots allows an administrator to perform backups of the data base state that can be stored remotely. The frequency of the snapshots can also be configured, however, a separate process will handle the persistency of the state to disk, hence no additional performance overhead is imposed to the main process handling client requests.

Although Redis' current architecture is envisioned to run in a single node, replication is supported by using a master/slave model. One master server can have one or more slaves which can be used for increased scalability in read only operations or redundancy of data.

Additional datastores

Although the list of distributed datastore does not represent the entire ecosystem of solutions, the previous systems represent part of the most suitable alternatives to be used as an S4 checkpointing backend. Moreover, most of these systems have proved efficiency since they have been used in highly demanded production systems. Nevertheless, an additional set of datastores are worth describing since they offer certain characteristics unique to them and not covered previously.

[4] http://redis.io

Scalaris [42] is a peer to peer (P2P) based distributed key/value store that uses a structured overlay ($Chord^{\#}$) to achieve scalability and partition of data. In addition, its main contribution over projects like Cassandra or Voldemort, is that it provides a transactional layer with an adapted implementation of the Paxos algorithm[33]. Replication is achieved by using a symmetric replication scheme on the structured overlay. Finally, by leveraging the lexicographical order of keys used in $Chord^{\#}$, not only can range queries be supported, but partition and replica assignment can be tailored to support scenarios spanning multiple data centers.

An innovative work by Lim et al.[34], SILT, proposes the use of flash storage to achieve higher throughput on key/value stores in response to the high latency times a traditional disk has attached. SILT's architecture is composed of multiple stores (in memory indexes) where each of them serves a different purpose. A write operation is mapped to a write optimized store, where insertion order is provided and values are written sequentially to a log in the flash storage device. Once this write optimized store is filled, the key/value tuple will be passed to a hash organized store, which is more memory efficient. Finally, the values of the hash based store are processed and passed along to a sorted key store, where each key occupies the least memory possible. This flow of keys along the different stores, allows SILT to achieve high write throughput while having a sorted set of keys, thus guaranteeing fast lookup operations.

2.5 Summary

In this chapter several stream processing systems have been classified in two main categories: the database focused *Data Stream Management Systems*, where data processing is usually done by using an extension of an SQL language, and the Complex Event Processing (CEP) systems, which allow the users to define their own operations on the streams of data and provide semantics relevant to the events. Different systems, both from industry and academic backgrounds were described and compared to the functionalities that differentiate them to the Apache S4 platform.

In addition, an overview of the fault tolerance mechanisms available in DSPS systems is described. While most of these mechanisms have been developed under the idea of process replication, other techniques such as checkpointing has been studied. An example of this, is the work by Hwang et al. [29] where multiple servers participate in the saving and recovery of a failed node state to achieve higher performance.

Later in this chapter, a description of the checkpoint based rollback recovery mechanisms was exposed. To better understand the problem proposed in this work, a definition of stable storage, and the description of three different checkpointing mechanisms was presented.

The Apache S4 platform was described providing a detailed overview of the components in its architecture. The platform's design, focused in a totally symmetric system, allows it to be flexible enough to be used in different appli-

cations. Moreover, the use of a distributed coordination system, enables the platform to achieve fault tolerance using a warm failover schema.

This chapter is closed with the analysis of different distributed datastores that could be used as a storage backend for the S4 platform. Every system in this section has different properties. Some rely on distributed file systems at their core, while others borrow techniques from P2P systems to partition the data across multiple nodes. Their data model also differs broadly. Some of them support higher abstractions such as columns families and dynamic data models (Cassandra and HBase/BigTable), while others provide a simple key value storage model. A summary of these systems is shown in Table 2.1

Table 2.1: A comparison of the distributed datastores that can be used for S4 checkpointing mechanism.

	Write operations	Read Operations	Supports a large number of PE checkpoints	Durability	Consistency	Data Model	Other
HBase	Sequential write operations. Records are never overwritten	Requires multiple accesses (in memory) or disk, to reconstruct a record.	Designed to store hundreds of million of rows	Records are kept in memory, but uses WAL	Strongly consistent operations.	Column-group oriented	Underlying HDFS storage.
Cassandra	Sequential I/O operations.	Although consistent hash offers good random lookup times, many in-disk files may be accessed if the key is not in a node's memory.	Once memory is filled, records are persisted in disk. Allows a large number of records.	Commit log for failure resilience of in-memory data structure.	Tunable per call consistency.	Column-group oriented.	Consistent hashing algorithm to partition data.
Project Voldemort	Replicated writes in memory of several nodes with a write quorum.	Complete topology in every node, thus lookups have $\mathcal{O}(1)$ complexity.	Pluggable persistence layer. Default: Berkeley DB. Krati and MySQL also available.	Achieved by two techniques: versioning (vector clocks) and read repair.	Lower latency achieved by eventual consistency.	Simple key-value store with support for Lists.	Consistent hashing for partition of data. Modularized architecture and flexible design
Redis	Set operations have a $\mathcal{O}(1)$ complexity.	$\mathcal{O}(1)$ is the get operation time complexity.	Maximum 2^{32} keys. Probably RAM is the upper bound in this case.	Point in time snapshotting and append only logging of operations are offered.	Using a master-slave architecture strong consistency is not guaranteed	Key/ value store, where value can be one of: strings, lists, hashes, sets, sortedSet.	Replication in a master-slave manner through asynchronous replication. Not fully distributed.
Scalaris	Guarantees ACID using modified Paxos slows down the write operations	$\mathcal{O}(log)\, n$ for lookup followed by R reads for consistency.	Several DB backends available, Tokyo Cabinet as one of them.	Persistency is not available due to quorums.	Strong consistency guarantees using modified atomic Paxos.	Key/value	Chord# based key/value store with ACID properties.

3 Handling high workloads in the S4 checkpointing mechanism

This section starts by describing the checkpointing mechanism in S4 and its architecture. The problem statement subsection describes how certain stream processing applications require a new approach when they need to be checkpointed. Later in this chapter, the proposed solution to handle applications imposing an intensive write load on the checkpointing backend is described. This last subsection includes a discussion of the advantages of a new design in the storage backend, the justification for such design choices and the architecture of the implementations.

3.1 S4 Checkpointing mechanism

The S4 checkpointing mechanism aids the fault tolerance mechanisms in applications where the accumulated state stored in the PEs is critical. For example, applications that build models over extended periods of time, based on training data, cannot afford to reconstruct the entire model once a failure of a node occurs. Note that without the checkpointing mechanism, the complete accumulated state of the PE would be lost. The main goals of this subsystem are: impose low overhead on the processing path, scale accordingly given different sizes of checkpointing events and be adaptable to different technologies to be used as storage backends.

Checkpointing in S4 is performed in an uncoordinated manner to interfere as less as possible with the stream processing path. To achieve such a requirement, checkpoints, i.e. PE states, are saved asynchronously in a backend component, and can later be retrieved whenever a node fails and this state needs to be read by a newly active node.

Another possible scenario, where the checkpoints can be used, is when a PE has been removed from a PN, as a result from its TTL value being expired. In this case, a checkpoint of this PE can be read from the storage backend and the PE could be re-instantiated using its previously accumulated state.

This asynchronous mechanism, however, does not guarantee consistency of the checkpoints, i.e. the storage operation can fail, thus the last checkpoint state of a PE would not have been successfully stored. The consistency condition of the saved checkpoints is relaxed, since S4 does not guarantee the

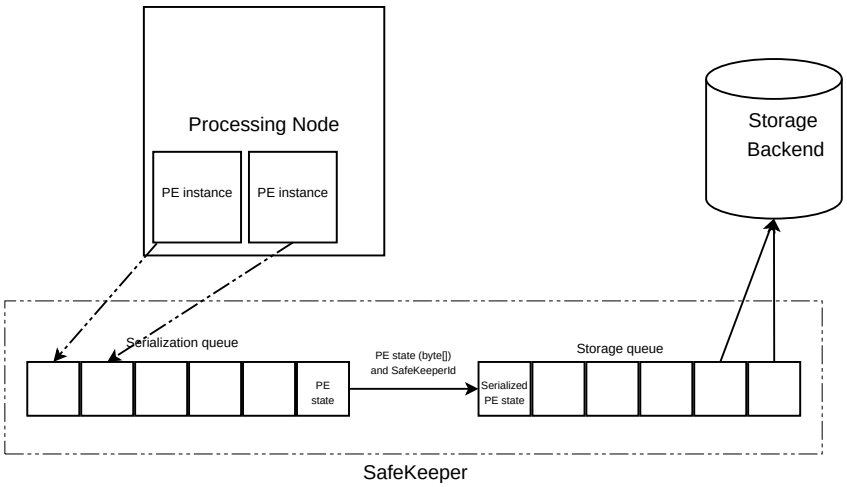

Figure 3.1: The S4 Checkpointing architecture. The SafeKeeper component handles the serialization and manages the PE's state queues. The storage backend is a completely separate and pluggable component.

processing of every event flowing through the system, neither does it guarantee that a consistent system wide state is checkpointed. However, by making the checkpointing mechanism uncoordinated and asynchronous, the stream processing performance is not affected.

S4 Checkpointing architecture

Figure 3.1 depicts the different components in the checkpointing component architecture which reside in each PE instance configured for checkpointing. Within a PN the checkpointing mechanism can be specified per PE, since some PE's state may be important to save, while other may not.

The checkpointing of a PE's state can be triggered in two ways: after a number of events have been processed, or on a regular period of time. Once a PE has processed enough events (given the configuration is using the first option) a checkpoint event will be triggered and handled by the same PE. S4 provides a flexible configuration schema, where some PEs can be scheduled for checkpointing while others not.

The PE will communicate with a component known as "Safekeeper", delegating the checkpointing task and immediately returning to its main task of processing event streams, reducing the latency introduced into it. When a PE state reaches the Safekeeper component, a unique identifier for that PE instance is created (known in S4 as a *SafeKeeperID*). Safekeeper is then in charge of the asynchronous serialization of the PE state by using a serialization queue from where the events are transformed into arrays of bytes. As a result, once the state is serialized, Safekeeper will generate a save-state task and put it in a

storage queue, elements will be consumed and saved in a storage backend from this last queue. A pluggable callback component asynchronously returns the results of the storage operations. These results can be logged, or in our case be used to measure write latency for each operation.

Storage Backends

A storage backend handles the storage and retrieval of serialized checkpoints from and to an S4 cluster. There is a clean interface so that different backend technologies can be plugged into a cluster by implementing write and read operations for the specific underlying storage technologies.

Since different applications will have different impact on the storage backend being used, the pluggable interface allows a broader set of use cases to be handled correctly. For example, an application might generate PEs with a considerable size, since each PE can contain much information, while other applications might create millions of PE instances and need to be safely (as much as possible) stored in a backend with high frequency.

Two different storage backends are already included with the S4 platform:

- A file system backend which creates separate files for each checkpointed PE state. The files are uniquely identified with the *SafekeeperId* as the file name. These files can be later stored in a distributed file system (e.g. NFS), to enable recovery for non-transient failures of a node.

- A backend using Redis, a simple in-memory key-value store that can be asynchronously replicated using a master slave schema and achieves data persistency by periodically flushing data to disk. Further description of Redis has been included in Section 2.4.

Recovery

When nodes are started they register in Zookeeper and acquire tasks, hence a mapping between tasks and nodes is saved in the Zookeeper ensemble for failover. When a node crashes, an idle node will enter the S4 cluster and acquire the task previously assigned to the failed node from Zookeeper. If it succeeds, all the events previously routed to the crashed nodes will onwards be sent to the new active node.

Consequently, the appearance of a key for which a PE instance did not yet exist, will force the platform to create a new PE instance. However, before returning the control to the PE for processing, a recovery mechanism will be attempted. The Safekeeper component will generate the identifier for that PE and will try to fetch a saved state for that identifier from the storage backend. If such a state exists, then it will be deserialized and used as the initial state for that PE. If it does not exist, the newly created PE instance will be created using a blank state.

Currently, the mechanism to recover a PE's state is triggered once a new instance of a PE is needed in a PN, i.e. lazy recovery. However, it would possi-

ble to implement a mechanism for the storage backend to retrieve, in advance, the saved state for PEs that would be instantiated on each PN. Therefore, an eager recovery mechanism, would reduce the impact of PE instantiation by pre-fetching PE states and storing them in the node's main memory. This is beyond the scope of this work and it is discussed as part of the Future Work section.

3.2 Problem statement

Different loads will be imposed on the checkpointing mechanism of S4 depending of the applications. Moreover, different applications will require distinct properties from the checkpointing backends. An analysis on the common operations in S4 reveals that durability is a property any of the backend solutions must provide.

Durability refers to the permanent safeguarding of data in persistent non-volatile storage, such that subsequent access to it returns the previously saved value, given no intermediate updates. This can be achieved with a series of techniques from in-memory replication of data before commitment, to using flash based storage or using magnetic disk drives.

Furthermore, when the volume of data grows, the use of inappropriate access patterns can strongly impact any system under analysis. Therefore, it is important to design algorithms that can tolerate high volumes of data while maintaining the durability guarantees. In this sense, the S4 checkpointing mechanism presents a scenario where write operations are very frequent, while read operations will only be needed upon recovery of a node.

For example, we can consider an application running on S4 with periodic checkpointing (e.g. every one second) with millions of PE instances on each S4 node. If these are simultaneously checkpointed, the load imposed on the storage backend should be tolerated with the minimum loss of checkpoint states. If the latency of write operations in the backend is high, most of these checkpoints will saturate the checkpointing component, resulting in undesirable loss of checkpoint states. A backend supporting low latency operations with strong durability guarantees can greatly benefit such an application.

Moreover, the serialization and storage queues follow a producer consumer model. When thousands of checkpoint events are concurrently processed by the Safekeeper component, it is highly probable that the number checkpoint states will outnumber the queues capacities. Therefore, once the checkpoint reaches the storage queue, the consumer (the storage backend) must perform a low latency save operation, otherwise, it will result in the loss of checkpoint states due to lack of space in the queue while events continuously arrive. By default in S4, the oldest state in the queue is removed and discarded in favor of the most recently created one to replace it. By providing a low latency mechanism for storage of checkpoints, we can reduce the number of PE states being replaced in the queues, and therefore, little to none of the PEs state will ever be dropped from these queues.

As exposed previously, the backend component in the checkpointing mechanism will act as 'stable storage', thus high durability must be guaranteed while high-performace in write operations must also be provided. The remaining of this work aims to optimize S4's checkpointing mechanism to enable write dominated workloads.

Failure Model

In order to determine a solution to the previously exposed problem, it is necessary to narrow down the different failures scenarios our solution would be able to overcome.

Our backend component will continue to operate as long as one of the nodes in an S4 cluster remains active. In fact, in S4, the case of many nodes failing can be tolerated by dropping incoming events, thus the events' processing rate will decrease and so will the PE state changes. The failure model followed in S4 is a fail-stop model with a Zookeeper ensemble detecting the failure of nodes. Therefore, our solution is subjected to use the same model as S4. Moreover, we also assume the communication channel, between the processes being checkpointed and the stable storage component, to be reliable.

In addition, failures of S4 nodes during the recovery process will not corrupt the process state in stable storage, and can possibly be recovered by a subsequent idle node. Since the S4 checkpointing mechanism is completely uncoordinated, inconsistent states between different processes in a system are tolerated. A process can recover using its last saved state before a failure, while another might use a state which is not completely up to date with the one before the failure. This can be translated into the following safety property: if a PE recovers its state based on a given checkpoint *ckp*, then *ckp* must represent a valid state at some previous time t.

Finally, the restart of nodes due to scheduled maintenance is also covered by the failure model proposed. While a node is restarted any of the unassigned idle nodes can take over its task seamlessly.

3.3 Solution Alternatives

Two different approaches have been considered. Both of these approaches are designed leveraging the use of commodity hardware, since S4 is a system designed to run on this hardware.

In the first approach we have optimized the file system based solution to tolerate high-write loads. Magnetic disks are a common component representing stable storage in commodity hardware, however, introducing improved access patterns for specific types of loads can yield positive results. In addition, by using a file system to solve the problem, there are no additional dependencies introduced into the S4 platform, and such a backend would require little additional maintenance in a production environment.

An application impacting the S4 checkpointing backend, with a high concurrent write load, will present issues for the current file system solution, where

every state is saved in its independent file on disk using a unique identifier as the file name. Therefore, we sought for an optimization of this backend providing a sequential write pattern to disk, thus serving faster write operations than the previous random access to it.

Sequential write operations can be obtained by using a single file where all the checkpoints are saved sequentially, along with their identifier. In this scenario write operations will have a reduced number of disk seeks, which usually dominate disk access. On the other hand, read operations will need to scan the complete file to find a particular record. To avoid this costly scan operation, an index, kept in memory with the different positions of the records, can be used. To guarantee its durability it can be periodically persisted to disk. Moreover, the index can also be reconstructed by scanning the append-only files to update any outdated index entries in the case of a node failure.

We have designed this solution envisioning that its read performance would be penalized both by the access to this index and its reconstruction in case of a failure. Since our goal is to provide better performance in failure-free execution of the S4 platform, slower read operations can be tolerated.

Our second proposal is to replace the file system backend with a distributed data store. Although, this would introduce yet another dependency for the S4 platform, these systems are commonly used to support high-traffic loads. Moreover, in the majority of the cases, these systems support data replication as one of their features, and provide different levels of consistency and availability to obtain lower latency in operations. An analysis of different systems to be used was provided in Section 2.4.

One of the most remarkable systems in this area is HBase, which offers a write-optimized model over a robust distributed file system. Write operations are performed in an append only manner, however, data is buffered in every storage node before being persisted to disk. We have implemented an HBase backend for the S4 checkpointing mechanism, which allows us to compare its performance with the bundled Redis backend.

3.4 A competitive file system storage backend

In [30] it is exposed that when the size of a dataset grows huge, efficient I/O access patterns are required to avoid them from becoming the system's bottleneck. In fact, in this work, Jacobs demonstrates that sequential access to disk can outperform random memory access for large amounts of data. The author also describes how random access to disk can be ten orders of magnitude slower than sequential access to it. In our design, we leverage this fact, and propose to minimize the disk accesses by writing sequentially to disk, and utilizing an in memory index to enable read operations for a specific record. To provide faster access to this data in the recovery process, this index is also flushed to disk periodically.

Our solution imposes a higher overhead on read operations when compared to the previous implementation, since the index entry has to be read for every

checkpoint state. Read operations are infrequent in the S4 checkpointing mechanism, they will occur either when a node has failed or when an expired PE (TTL=0) is re-instantiated, however, this is not the case for write operations, which will happen every time a PE needs its state to be saved.

In the following subsection a description on the reasoning behind using append only operations on disk compared to using random access while having multiple files (one per PE state) is provided. This is followed by a detailed description of the implementation of our solution to provide a more competitive checkpointing backend with the least number of additional dependencies to S4.

Sequential vs. Random operations

An analysis on the common workloads imposed on the checkpointing storage backend in S4 shows that write operations will be handled by different threads that consume the tasks present in the storage queue as shown in Figure 3.1. By using the current file system backend each write operation containing a PE state, will write one file per state on disk. In this section we present an abstract model for disk performance when it is accessed directly without any buffering of data by the page caches or disk controllers. A formula expressing the different time components in a single checkpoint write of size B on disk, is as follows:

$$CW_{time} = A_{time} + WT_{time}(B)$$

$$CW_{time} = (S_{time} + R_{lat}) + WT_{time}(B)$$

The checkpoint write time (CW_{time}) is formed by an access time: A_{time}, later subdivided to seek time (S_{time})(the time for the head to position on the correct track) and rotation latency (R_{lat}) (the time it takes for the disk head to locate the correct sector once positioned on the track), and the write transfer time $WT_{time}(B)$ which is a function of B: the number of bytes to be written.

For N checkpoints of size B bytes, the total write time is:

$$TotalCW_{time} = N * [(S_{time} + R_{lat}) + WT_{write}(B)]$$

The previous chekpointing backend worked by writing one file per checkpoint. Having an access time of: $N * (A_{time})$. However, the total number of bytes B to be written depends on the write transfer rate linearly, thus we can assume:

$$N * WT_{time}(B) = WT_{time}(N * B)$$

Since the number of bytes to be stored for N write operations remains constant, reducing the overall write time of a checkpoint can be achieved by reducing the number of disk seek operations per write. The disk transfer rate for N write operations (WT_{time}) will remain proportional to $N * B$, regardless of the number of disk accesses, thus remaining unchanged.

One possibility to reduce the number of disk accesses is to store the checkpoints sequentially in an append only file. In this scenario, each PE state

contained in a node is saved to the same file, and an index is kept in memory to locate the checkpoints in this file to alleviate read operations. Given the random nature of checkpoint's state storage, by forcing them to happen in a sequential manner, we can reduce the latency imposed by the disk heads moving to find the appropriate track, thus reducing the overall seek time (S_{time}). The total write time for N checkpoints can be expressed taking into consideration that only when a disk track is filled, will there be a head movement:

$$TotalCW_{time} = \frac{N * B}{BytesPerTrack} * S_{time} + N * R_{lat} + WT_{time}(N * B)$$

Where $BytesPerTrack$ represents the maximum number of bytes a disk can have on a certain track (although this varies, it is in general in the order of MB). We assume that once a track has been filled, the next track to write needs the disk head to spend a complete seek time period S_{time} until it reaches it. In this expression the reduction on the access time is of great significance since $\frac{N*B}{BytesPerTrack} * S_{time} <<< N * S_{time}$, thus this term can be ignored. We can reduce the overall write time significantly for the checkpoints by eliminating the seek time component, done by writing sequentially.

Moreover, it is important to add to this equation the overhead of flushing the in memory index to disk every T seconds. This term will impact the previously described value of $TotalCW_{time}$ only if $TotalCW_{time} > T$. The time cost for this operation can be expressed as:

$$IndexFlush = A_{time} + WT_{time}(IndexSize)$$

The total checkpoint time for the backend based on append only files, considering the index flush period T, can be expressed as:

$$TotalCW_{time} = \begin{cases} N * R_{lat} + WT_{time}(N * B) & if \quad TotalCW_{time} < T \\ N * R_{lat} + WT_{time}(N * B) + IndexFlush & if \quad TotalCW_{time} > T \end{cases}$$

No impact will exist on the backend regarding the index flushing operation, if the total time is lower than T.

On the other hand, read operations in recovery of a PE's state, will always be performed in a random manner. In the previous file system backend, the Safekeeper component generates the unique id corresponding to a PE instance, and tries to fetch the correspondent file with its id as a name. When entering the recovery process of a PE, it is uncertain which of the saved PE states will be recovered and which of them will never be re used, therefore, a lazy retrieval model has been adopted in S4. The saved state of a PEs is read only upon the re-instantiation of that PE in some, possibly, new node.

For the previous implementation of the file system based backend, the read operations will be less costly than for our solution. In analogy to the formula for write operations, a read from disk operation for a specific PE of size B can be expressed as:

$$CR_{time} = A_{time} + RT_{time}(B)$$

Where the access time is A_{time} and $RT_{time}(B)$ represents the read transfer time for B bytes. Since the access to the different checkpoints cannot be predicted, to read N checkpoints the time for the operation would be:

$$TotalCR_{time} = N * (A_{time} + RT_{time}(B))$$

$$TotalCR_{time} = N * A_{time} + RT_{time}(N * B)$$

Additionally, in our proposed solution, the cost of accessing the index in memory has to be added, along with the cost of reconstructing the index the when a node is started.

$$TotalCR_{time} = N * (A_{time} + R_{time}(B)) + N * IndexLookup$$

The *IndexLookup* operation represents an access to the in memory index, which will be done, every time a checkpoint needs to be read. According to [19], this memory reference operation should take 100 nanoseconds in average.

Moreover, an $IndexReconstruction$ task is performed for the initial recovery of a node. This is composed by two subtasks. The first one consists on the traversal of the index file in disk and its reconstruction in memory as a map. The second one is the update of the values from the append only files into the newly generated in memory index. These two tasks will depend both on the size of the index, determined by the number of different PE instances present in every PN, and the amount of PE states saved to the append only files not present in the memory index.

It is clearly observed how a read operation is penalized by the memory indirection and the index reconstruction time. However, this is a trade-off our solution can tolerate, since read operations only occur after a failure, while faster write operations will benefit the checkpointing system for most of the platform's operational time.

Solution Architecture

Figure 3.2 depicts the components of the proposed solution. Once a PE triggers the checkpointing mechanism of its state, after being serialized, the checkpoint is converted to an array of bytes; it is appended to the end of the current file, and the correspondent entry in the index is updated. The checkpoint state along with its key are safely appended to disk, before acknowledging the operation, thus durability of the checkpoint is guaranteed.

The in memory index holds one entry for each PE identifier (*SafekeeperId*). The value for each key is a combination of the position, the size and the filename where the checkpoint is located. This index is persisted to disk after a period of time has elapsed, hence enabling its reconstruction in case of a failure.

At any given time at least one append only file will exist, where the checkpoints are being written. However, once this file reaches a specified maximum size, the file is rotated and the references in the index are updated accordingly. Figure 3.2 illustrates the exact moment where the newly saved checkpoint for

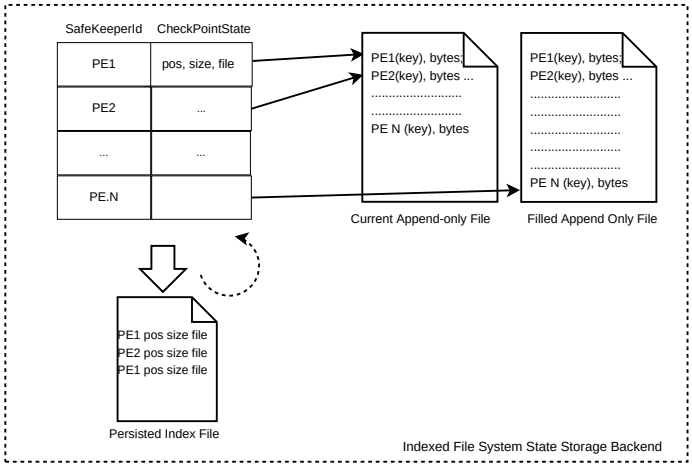

Figure 3.2: An indexed file system based storage backend to store S4 checkpoints. The PE state (in bytes) is written to an append only file, while an in memory index is kept to locate these values in the file. This index is flushed to disk periodically, and reconstructed in case of a node failure.

PE-N has been appended, and the index is about to be updated to reference the position in another file. Since only the latest checkpoint state is required, as the values stored in the "Filled Append Only File" become unreferenced, they can be garbage collected.

Recovery

In case of a crash failure, the checkpoint states will be located by reconstructing the index from the flushed index file. However, since this file will only be persisted periodically, it is highly probable that some index entries (resident in memory at the time of the crash) are not present in the file on disk. Figure 3.3 illustrates this scenario using time as a reference. In step 1, the index is persisted to disk, later, the addition of a new checkpoint to the append-only file updates an entry in the in-memory index. However, in step 3 the node's failure causes the loss of the index update, since the index flush in step 4 has not been reached.

To overcome this problem, the append only file holds not only the checkpoints but also the keys (*SafeKeeperId*) to the correspondent state. Once a node is restarted, it will firstly reconstruct the index from the flushed index file saving the last position to where the checkpoints are located in the append-only file. As soon as the index is completely reconstructed, the append only file will be traversed starting from the last position present in the index. The newly found keys in the append only file can then be updated in the in-memory index so that they reference the last checkpoint saved for that key.

This process will affect the starting time of a node when recovering from

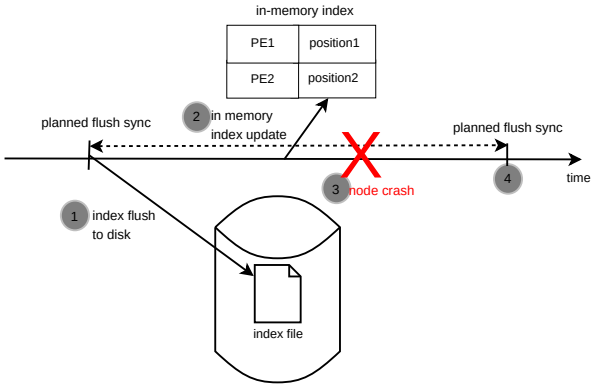

Figure 3.3: The index flushing mechanism and the problem of intermediate index updates not persisted to disk.

a failure, but guarantees that no checkpoint present in the append only files remains inaccessible.

Garbage Collection

The S4 Checkpointing mechanism uses the last saved checkpoint of a PE, therefore, once a checkpoint for a specific PE is updated, the previous value becomes of no use for the system. In order to avoid using additional space for values that will never be required, a Garbage Collection mechanism has been included in our solution.

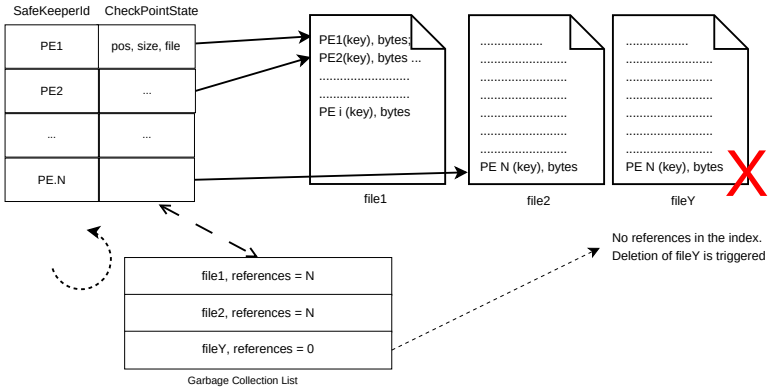

Figure 3.4: The Garbage Collection mechanism allows for outdated checkpoints to be removed keeping disk space usage to the minimum. A list of references (Garbage Collection List) retains the count on the references each file has. Once a file has no references from the index to any of its stored checkpoints (all of the entries have been updated), it can be deleted.

An overview of the garbage collection mechanism is depicted in Figure 3.4. As it is observed, once an append file has no entries in the index which references its checkpoint states, this file can be removed as it is the case for *fileY* in the diagram. A new checkpoint for *PE-N* is saved, updating the index and consequently decreasing the references counter for that file. Once it has reached zero, the file can be safely deleted.

3.5 A solution using a distributed data store: HBase

In Section 2.4 various distributed datastores were described. Focus was set on the desirable characteristics a datastore must have for the S4 checkpointing mechanism. HBase is a system optimized for fast write operations and in comparison to other systems, it is focused on storing millions of keys across multiple nodes.

Our main goal by implementing such a backend for S4 is to explore the possibilities offered by a system optimized for handling write operations. As aforementioned, fast operations in HBase are achieved by having immutable objects and storing the differences between them in batches of operations. HBase is usually ran using HDFS as a lower layer, therefore, it is well suited to handle batches of data written in append-only manner. By using this technique, records are buffered in memory in each node, and can later be written sequentially using a differential files mechanism.

This highly optimized scenario for write operations, implies that reads are de-optimized. Since if a record is constantly modified, a read operation will require to reconstruct the record by merging multiple parts of it to generate the latest version. Once again, this is a tolerable situation in the S4 checkpointing mechanism, since reads will only occur upon the failure of a node.

Implementation

Our PE states data model shares more characteristics with a simple key/value store such as the one Redis offers than the more sophisticated column group based model in HBase. However, we have adapted HBase's model for our work.

We have adopted a model where for every *SafekeeperId*, one row in the HBase system will exist. A representation of the column family model designed for the checkpoints storage is shown below using JSON[1] notation:

```
safekeeperID: {
        data:
                checkpoint:"checkpoint_byte_value"
 }
```

To reduce the number of checkpoints being stored in the system to the minimum, we have decided to only save the last version of every record in HBase. Given HBase provides strong consistency guarantees, once written, the

[1]Java Script Object Notation: http://www.json.org/

checkpointed state of a PE will remain up to date for later recovery, and in case it is lost due to a failure, it can be replayed by using the WAL mechanism.

4 Experimental evaluation

In this chapter we present experimental evaluation results focusing on the scenarios previously described and highlighting the contrasts the different backends present. We begin our evaluation with a series of micro-benchmarks that compare the two file system based backends, the one already present in S4 and our proposed solution. We then describe our performance measuring component, which allowed us to measure the latency time taken to save and recover checkpoints from a backend asynchronously. Finally, we provide a comparison of the system performance using a distributed S4 environment.

For our evaluation all the machines used in the experiments were equipped with 16 GB of RAM and Intel Xeon 2.5 Ghz processors connected using 1Gbit/s Ethernet network cards. Their disks were 7200 rpm SATA drives with 320 GB of capacity. In addition, all of the servers shared access to a common NFS v3 mount point where data was stored. Thus, recovery in the case of complete hardware failure of a node could be performed.

4.1 Micro-benchmark for the File System based backends

In a micro-benchmark we evaluated the performance of the two different file system based backends in isolation from the S4 platform. The environment where the values where read from and stored used an underlying distributed file system. In this case NFS v3. [40] was used since it is a common file system present in the Linux kernel since version 2.4 [1] and hence available in most server deployments.

By creating a tailor-made binding for the YCSB benchmarking tool [16], we were able to impose various loads on our systems varying the number of target operations on every run and adding more or less number of client threads on the YCSB load generator in order to saturate the systems. Initially, we started the experiments with only one thread and observed the number of operations it could provide us with. Then, we varied the number of threads on each run, so that we could obtain the targeted throughput. Moreover, we were able to specify the number of different PEs to be written or read. This is an

[1] http://nfs.sourceforge.net/nfs-howto/ar01s02.html *accessed 11 Jun 2012*

important parameter, since a write operation for a PE already present in the backend represents an update in the system (overwriting the previous value).

Figure 4.1 shows the values for an average of ten runs of a write dominated workload, where 90% of the operations correspond to writes and 10% to reads using 1KB as the size of the value to read/write. We injected 10000 operations into the system, creating a total of 1000 different PEs. The total number of operations does not affect each system's individual latency per operation, whereas the number of different PEs can, in fact, affect the previous file system based backend.

As it can be observed, the default file system solution can hardly surpass the limit of 700 operations per second with a major increase on the latency. On the other hand, our solution (Indexed FS) saturates at a throughput rate of approximately 2100 ops/sec with a latency value of 20 ms per operation. It is likely that, since these experiments were ran on an NFS mount, to allow any node to access the data, the cost of the metadata operations in the default file system backend imply several Remote Procedure Calls (RPC) as detailed in [44], namely *create_file* and *write*. In contrast, the indexed file system storage does not perform as many file system calls, and accesses the disk in a manner which reduces the access time by obviating the seek time per operation, as previously exposed in Section 3.4. These results describe how our append-only solution can yield better results for a write dominated workload over a distributed file system.

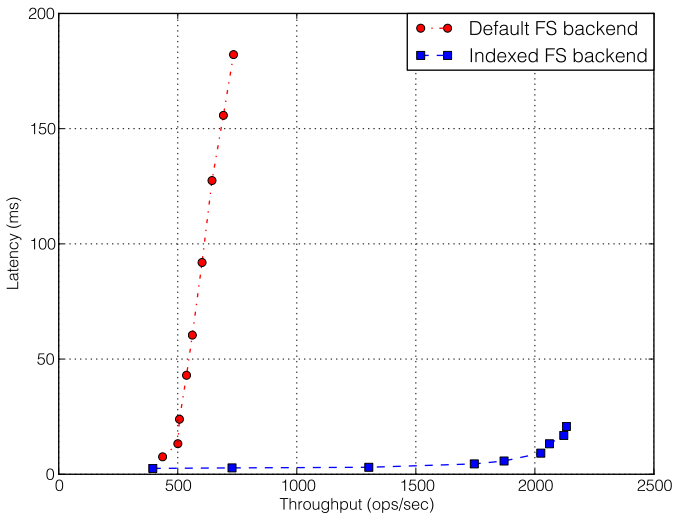

Figure 4.1: Micro-benchmark for the two file system based backends. Default FS is the previous backend, while Indexed FS is our solution. Values shown are the average over ten runs of writing/reading ten thousand 1 KB records.

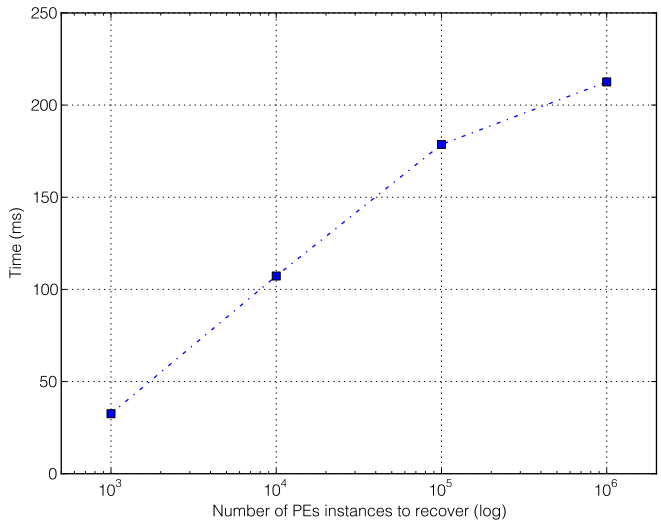

Figure 4.2: Index reconstruction time for different numbers of PEs in our proposed indexed file system backend.

Nevertheless, our indexed backend solution is dependent on an in memory index to locate the checkpoints in the append-only file. Therefore, it is important to understand the cost imposed by the reconstruction of the index in the case of a node failure. Figure 4.2 presents the index reconstruction time for various numbers of checkpoint states to restore. For an application with one million PEs being recovered, the index reconstruction time remains under 250 ms.

As the size of the index is directly proportional to the number of PEs to reconstruct, more index entries require more time to be reconstructed as shown is Figure 4.2. Furthermore, these time measurements are also dependent on the size of the checkpoint entries while scanning the append only file. In these experiments the entry size was approximately 70 bytes including the key, since the entries were the result of checkpointing PEs in a word count application used for the distributed experiment. This penalty on recovery will be imposed on the system while restarting a node, and the checkpoint read latency of our backend system will not be affected once the node has completed the restart procedure.

4.2 Distributed performance evaluation

We have also evaluated our solutions in a distributed environment. Since the S4 checkpointing mechanism saves the value of a checkpointed PE asynchronously, in order to obtain latency values for every single operation, a measuring com-

ponent was developed. Furthermore, for fair comparison of the systems, the two file system based backends used the same NFS v.3 mount point.

A simple word count application was deployed in the S4 platform, where every occurrence of a different word instantiated a different PE. A unique PE instance per word is distributed among the nodes in the S4 cluster. The checkpointing of the PEs state was triggered for every new event, thus for example, the occurrence of the word 'foo' would trigger the checkpoint of its PE state, and a subsequent occurrence would overwrite the previous value in the backend.

Measuring tool

Latency times, for individual asynchronous operations cannot be directly measured, as opposed to synchronous operations. Relying on a pluggable callback component present in the S4 checkpointing architecture, we developed a measuring tool that allowed us to specify which of the different backends to use in each case, in order to measure the cost of an asynchronous read or write operation in it. This component is shown in Figure 4.3.

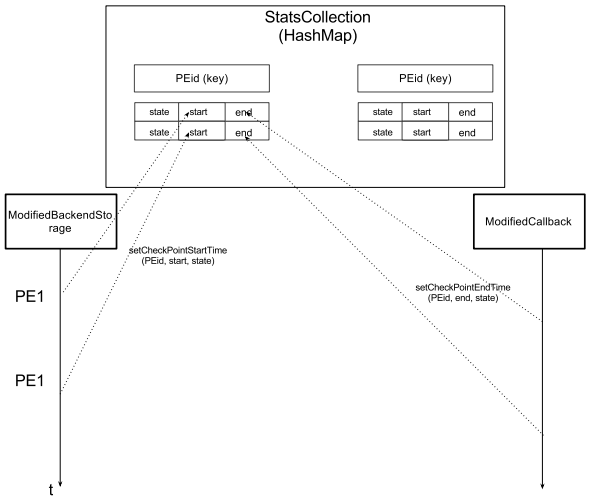

Figure 4.3: A component to measure the latency of read and write operations in the different backends

Write operations represent the asynchronous saving of a PEs state to the backend and once this operation is started, the start time for this checkpoint is saved in the component. Once the checkpoint has been successfully stored, the ending time will be reported back identifying every PE checkpoint, not only by the SafeKeeperId, but also by a hash value of its state. The hash of the state

allows the correct measuring of two consecutive checkpoints for the same PE without any interference between them.

On the other hand, in order to measure read operations for each backend, the recovery process has to be triggered. Therefore, our experiments initially sent events to the S4 platform and then restarted every node. The recovery of the PE's state was only triggered after an event keyed on the same value arrived to a Processing Node (PN) and the measurement tool took that into account. Once the state was read from the backend, the end time was saved and the total time computed.

Experimental setup

Our distributed environment consisted on several nodes with identical specifications. Figure 4.4 depicts the role each node had in the experiments. Three nodes composed the S4 cluster processing the event streams. An additional node acted as an adapter, in charge of transforming the streams of data into S4 events. All of these nodes were coordinated with the use of a Zookeeper instance running on a separate machine.

Furthermore, despite the load generator component ran in the same node as the adapter, as shown in Figure 4.4, during the experiments the average CPU load in this node never exceeded 2.5%. The load generator component was developed as a YCSB[16] biding where parameters for controlling the load imposed on the S4 platform were specified. In our case, an important parameter was the number of PE instances that the system would contain, since every instance will correspond to a different state in the backend.

In figure 4.4 the backend component was interchanged for every experiment, thus we were able to compute the latency times on the different backend solutions. For the file system based backends we relied on an NFS mount point, while for the distributed datastores, we decided to run a single node instance of both Redis and HBase to obtain our results.

Latency

Figure 4.5 and 4.6 present the average latency per operation on writes and reads on the file system based backends and the distributed datastores respectively by using our measuring tool. The figures represent the average over ten runs writing 10000 events and generating 1000 different PEs in the S4 cluster. In this case, 10000 represents the total number of events injected to the S4 platform, and the values where measured individually per operation. All of these experiments were performed by injecting 400 word events per second, since this is the maximum number of events the previous file system solution (Default FS backend) could tolerate without dropping any state from its queues. The checkpointing of a PE state was triggered for every single event, therefore we were able to observe the behavior of the checkpointing system for a period of 10000 events at a rate of 400 events per second using the different backends.

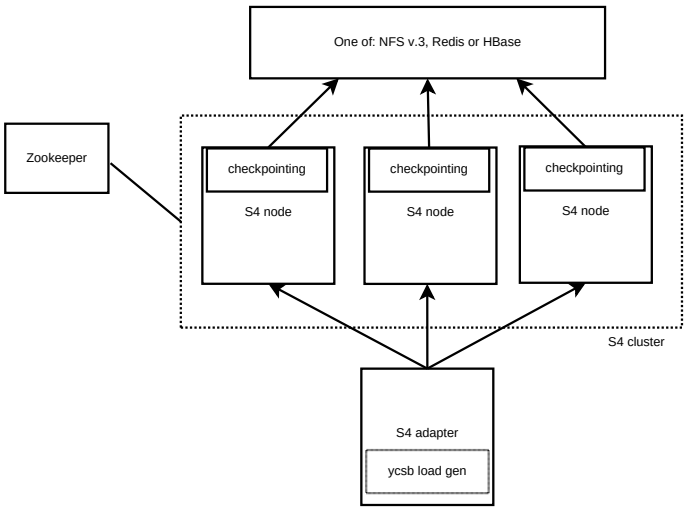

Figure 4.4: An experimental environment. Three nodes conform the S4 cluster and an additional node acts as the S4 adapter, where the YCSB load generator is also executed. One machine runs Zookeeper coordinating the system. All the checkpoints are stored using an NFS mount present in all nodes, Redis or HBase, thus data is accessible to all of them.

The read operations on the checkpoints were measured upon recovery of these nodes. Initially 10000 events were injected into the S4 cluster which generated 1000 different PEs. Once this was finished, the nodes were crashed and recovery took place. Ten runs of 1000 reads were executed to obtain an average on the read time per operation.

As it can be seen in Figure 4.5, our solution (Indexed FS) is 4.85 times faster in write operations than the Default FS backend. This can be explained by the reduction in the number of disk seeks needed to write a specific state. To our surprise, read operations also incur 3 times less time in our solution than in the previous system. However, these results are not completely fair, since our Indexed FS solution needs to reconstruct the index in order to read an entry from the backend as shown previously in Figure 4.2. We benefit write operations and penalize the read operations with the index reconstruction, nevertheless, this approach can yield better performance of the checkpointing mechanism in S4 in the regular operation of the system.

In Figure 4.6 an experiment was conducted using our HBase backend implementation and comparing it to the Redis backend storage, already present in S4. Both of these backends where run on a single node in a separate machine in the cluster. Redis performs slightly better in write operations, quite possibly due to implementation details, since both systems write values to a memory cache prior to persisting them to disk.

Moreover, in HBase, it can be noted how write operations are faster than reads since it is one of HBase's main optimizations. All records are stored in

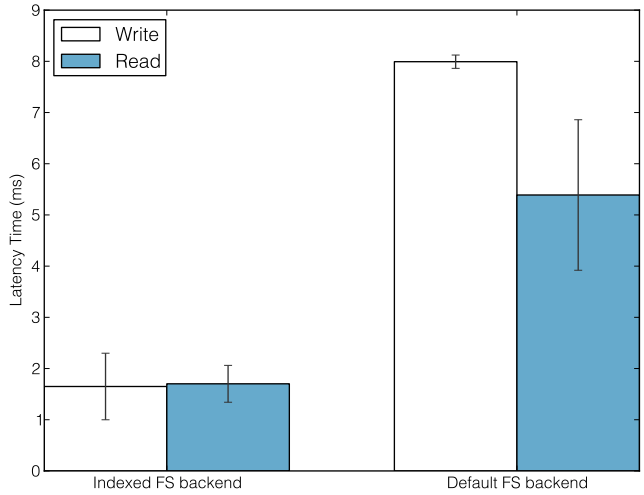

Figure 4.5: Distributed backend performance. Average latency per operation in checkpoint save (write) and recovery (read) comparing the two file system based backends.

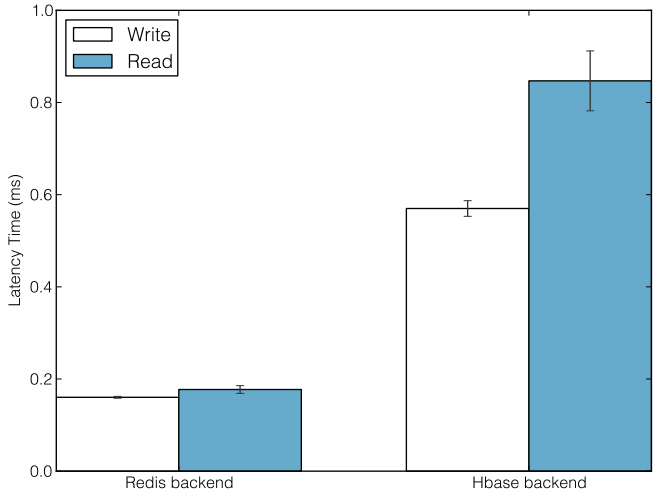

Figure 4.6: Distributed backend performance. Average latency per operation in checkpoint save (write) and recovery (read) comparing the two distributed data store backends.

an append only fashion, similarly to our Indexed file system based solution, thus a penalization is imposed on the reconstruction of the values to perform a read operation.

Finally, these observations represent our goal of achieving low latency operations while maintaining durability guarantees high. Redis has the lowest latency times in write operations, but it does so by storing the checkpoint states in memory and asynchronously persisting them to disk, thus reducing the durability of the solution. Likewise, HBase buffers the write operations in structures called *memstores*, however, higher durability guarantees are obtained by using a WAL mechanism, to reconstruct these values in case of a failure. Conversely, the file system based approaches achieve higher durability, but notable higher latency times per operation.

Throughput

To observe the behavior of the storage queue while using the file system based backends, we conducted an experiment were we observed the number of events being added to the storage queue per second. In fact, since these queues follow a producer/consumer model, once a checkpoint is added to a queue, it is either consumed by writing it to the backend, or dropped due to a newer checkpoint taking its place.

Figure 4.7 presents the average throughput of the two different backends when the number of different PEs to be saved is increased. The previously described distributed setting was used in this experiment as well. One million events were injected to the S4 platform, and we observed the behavior and averaged the incoming checkpoints to this queue every 5 seconds, taking into account no checkpoints were dropped from this queue. In this case, every checkpoint state consumption by a save operation allows another state to enter the queue. The throughput results are the aggregated number over the three nodes in the S4 cluster

In the case of the previous solution (Default FS), the number of checkpoints added to the queue per second cannot exceed 400 checkpoints per second without a significant number of them being dropped from the queue. Additionally, the number of checkpoints saved every second decreases to 150 when one million different files have to be created, since there will be one file for each PE state. The increase in the number of files is accompanied by a rise in the number of metadata operations in the file system, in addition to the higher latency of write operations due to its random access pattern, as described in Figure 4.5.

In contrast, our solution achieves lower latency in write operations and offers a sustained throughput of approximately 1300 checkpoint save operations per second, regardless of the number of PEs in S4. Thus, the use of the indexed file system backend would support S4 applications with thousands of different PEs producing little impact on the storage backend for the checkpointing component.

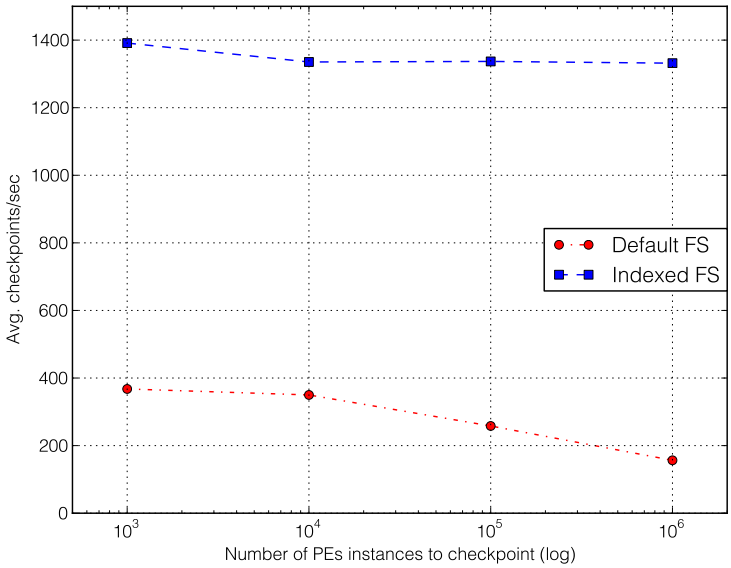

Figure 4.7: Average storage queue throughput for different numbers of PE instances in a distributed environment.

5 Discussion

The previously presented results provide us with useful comparisons between the different backend storage systems. However, additional details can be added to improve the understanding of our results.

5.1 File system based solution

We are aware that the use of a distributed file system (NFS) does not allow us to observe the complete disk behavior as modeled in Section 3.4 and fully quantify the benefits of writing sequentially and randomly to disk. NFS, in the experimental evaluation, represents a mount point to a complex dedicated storage component (commonly known as NetApp Filer). This Filer allows replication of data and provides a highly optimized environment for file operations. It is composed of many disks in a RAID configuration, and uses an NVRAM (Non-volatile RAM) layer for caching on top of the physical magnetic disks. These layers disallow the use of low level operations forcing persistent writes to disk. Nevertheless, all of our experiments still indicated that sequential write operations can yield better results than random ones, used by the previous backend, even in the presence of intermediate caching layers. Although this lack of control over the physical media exposes some differences between our model and the experimental results, using a distributed file system is the only possible way in which checkpoints can be accessed by many S4 nodes interacting with the backend. Therefore, we support our decision of using a distributed file system, even if our control remains at the higher level of the file system, provided by the NFS protocol.

Furthermore, to fully understand the behavior of our solution when writing directly to a disk, we conducted an additional micro-benchmark experiment using an *ext3* local file system (see Figure 5.1). As it can be observed, the number of operations per second decreases substantially when the disk is accessed directly and every operation is effectively flushed to the underlying media. The default file system backend cannot surpass 500 operations per second without introducing a major latency increase. On the other hand, our solution, *indexed FS*, provides better throughput with a lower latency value, and can scale up to 700 operations per second while maintaining the latency below 10 ms. It is our understanding that the effect of the NVRAM layer, present in the Filer com-

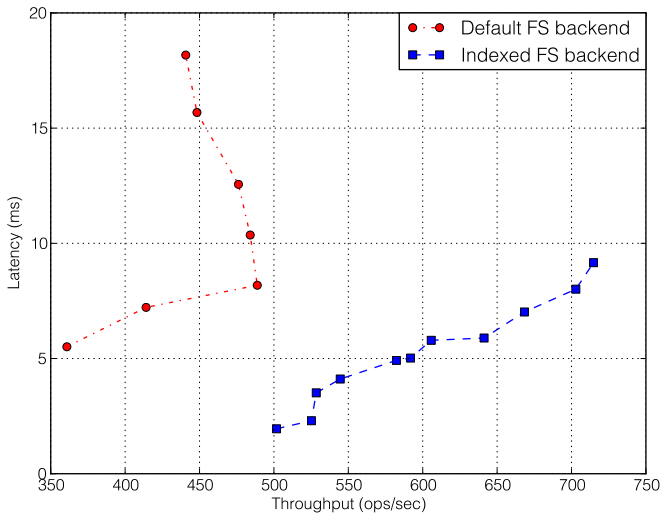

Figure 5.1: Micro-benchmark for the two file system based backends using a local file system. Values shown are the average over ten runs of a write dominated workload with ten thousand 1024 bytes records.

Table 5.1: Sequential and random times (in seconds) while writing to a file using NFS and the local file system. Different results are shown when forcing the filesystem to flush page caches to disk and not, using `fsync()`.

	1 million ops (0.96MB)	10 million ops (9.6 MB)	100 million ops (96MB)
local seq fsync	0.0353	0.3237	3.2741
local rnd fsync	0.3349	3.9525	1579.3410
local seq	0.0048	0.0414	1.1295
local rnd	0.3068	3.6825	1381.1431
nfs seq fsync	0.0315	0.3181	3.1816
nfs rnd fsync	0.3096	3.7492	44.7602
nfs seq	0.0048	0.0409	0.3870
nfs rnd	0.2887	3.3799	41.8916

ponent, allows it to yield better results than those shown in this experiment, as shown previously in Figure 4.1.

Finally, we include a simple experiment highlighting the differences while writing sequentially and randomly to a single file with NFS and the local file system. Table 5.1 lists these results with different numbers of operations using 10 bytes of data. The use of the `fysnc()` command after every operation in the two file systems, provides further insights on the control of the lower layers of the filesystems. For NFS, in most cases the results while using `fsync()` are

very similar to when it is not used, supporting the idea that layers such as an NVRAM prevent us from accessing the magnetic disks directly. In general, results exhibit that random write operations are penalized by the seek latency introduced by the disk, regardless of the file system in use. However, in the case of the local file system, the factor by which sequential operations outperform random ones is two orders of magnitude larger, while for NFS this difference is one order of magnitude. This re-enforces the idea that for the results presented in our experimental evaluation section, the use of NFS benefitted both of the file system based solutions, despite not providing sufficient control over the storage media.

5.2 Trade-offs

The results presented in Section 4 are not sufficient to make decisions on a backend solution that can fit all scenarios. Overall, increasing the performance of a given operation often causes the performance of other operations to degrade, as in the case of append and read operations.

Moreover, our results indicated that the lower latency times per operation were achieved by the distributed datastores, without implying they can fit all applications in need for durable checkpoints storage. The reduction in the durability guarantees included in Redis, for example, allows this backend to outperform the previous solutions. This is achieved by persisting data asynchronously, thus possibly losing information. To clarify these aspects, we summarize certain considerations in our backend alternatives that allow the reader to obtain a big picture of the trade-offs each solution presents in Table 5.2.

Table 5.2: A summary of the different trade-offs comparing the backend solutions.

	Default FS backend	Indexed FS backend	Redis backend	HBase backend
Additional dependency for S4	no	no	yes	yes
Favors writes over reads	no	yes	no	yes
Costly recovery	no	yes	no	no
Optimal scenario	Low number of PEs per node, infrequent checkpoints	High number of PEs per node. High frequency of checkpoints	Full durability of checkpoints is not needed. High frequency of checkpoints.	High number of PEs per node. High frequency of checkpoints.

6 Conclusions and Future work

6.1 Conclusions

Stream processing platforms are becoming prevalent in settings running applications that require event processing in near real-time e.g., online fraud detection or processing of sensor based networks data. In this scenario, the need for distributed and highly scalable systems is also growing. While these distributed stream processing systems provide improved performance over their non-distributed counterparts, certain complex problems arise with their distribution. Providing fault tolerance for these platforms is an interesting problem being constantly researched.

We have analyzed the Apache S4 platform partial fault tolerance model that masks failures through a failover strategy, and additionally offers uncoordinated checkpoint recovery to reduce the loss of a processing elements state upon a failure. Our efforts pointed towards developing a stable storage solution where checkpoints could be saved in scenarios of high loads of operations, albeit guaranteeing the durability of the saved information. In this direction, we developed two alternatives that provided us with different characteristics.

Our first solution consisted of a redesign of the current backend based solely on the file system. We benefitted from the write oriented workloads presented by the checkpoint mechanism to offer higher throughput in write operations and lower latency, maintaining durability of the data similar to the previous solution. We proposed an append-only system where the saved states were stored sequentially in a single file, as opposed to accessing one file for each of them. Finally, to access these states, an index was kept in memory with the location of the different checkpoints. This index was periodically flushed to disk, to allow its reconstruction in the scenario of a failure. With these techniques our solution offered lower latency operations when compared to the previous solution. A decrease by a factor of 4.8 in write operations and 3 times in read operations. It also yielded higher throughput results without dropping any checkpoint states of the queues when compared to the simple solution S4 includes. Finally, we described how by reducing the number of disk accesses on every checkpoint save, we were able to store states for applications with larger number of distinct processes in the S4 platform.

The second approach taken consisted on using a distributed data store to save the states generated by the checkpoint system. This approach was

preceded with an analysis on the available datastores that were suitable to develop our solution with. As an outcome of this analysis, we decided to implement a solution using HBase, a robust distributed datastore which is part of the Hadoop ecosystem. Since S4 already included a similar solution using Redis, our experiments compared the HBase backend to the Redis one. HBase provided higher durability guarantees than Redis, while at the same time negligible differences in write operations when compared to Redis.

In conclusion, this work describes different approaches to processing checkpointing states in a distributed stream processing system. By using more efficient access patterns, while considering the different requirements applications have, different alternatives can be used as the stable storage component of the checkpointing system.

6.2 Future Work

The Apache S4 platform is under heavy development at the moment, therefore, many of the components described in the architecture may change in the near future. However, the checkpointing component will remain similar to the actual solution, since it represents an effective and simple solution to fulfill this task in an asynchronous manner. Within this component, recovery would be enriched by using a configurable eager recovery mechanism. In this schema, checkpoints would be eagerly read from the storage backend and re-constructed prior to their correspondent PEs being instantiated, as it is done at the moment. This, however, is not easily achieved, since it would require prediction algorithms to be included, to maintain a low overhead by avoiding the reconstruction of unnecessary PEs.

Regarding the storage backends many other alternatives could be evaluated. One of them could consist on replacing the distributed file system in the experiments. HDFS could be used in replacement of the NFS mount point. This file system is highly optimized for the throughput of operations and not the individual latency of them, since it is designed to handle batches of operations. However, since it is utilizes clusters of commodity hardware, the nodes conforming the HDFS cluster, could possibly be the same as the S4 nodes.

Finally, another challenging problem to address would be a calibration mechanism, so that the frequency of the checkpoints could be regulated as the storage backends are saturated. This would allow a self-regulating mechanism reacting when the number of checkpoints approaching the storage backend, and its queues occur in an unforeseen manner. Therefore, reducing the frequency of the checkpoints or increasing the time period by which the mechanism is triggered, would decrease the number of state losses, while maintaining an acceptable level of state checkpoints being stored.

Bibliography

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA*, page 277289, 2005.

[2] D. J. Abadi, D. Carney, U. etintemel, 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*, 12(2):120139, 2003.

[3] G. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[4] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: a distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, page 2737, 2006.

[5] H. Andrade, B. Gedik, K. Wu, and P. Yu. Scale-Up strategies for processing High-Rate data streams in system s. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1375 –1378, Apr. 2009.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: the stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[7] D. P. Arvind, A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: the stanford stream data manager. *IEEE Data Engineering Bulletin*, 26:2003, 2003.

[8] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[9] M. Balazinska, H. Balakrishnan, S. Madden, M. Stonebraker, et al. Availability-consistency trade-offs in a fault-tolerant stream processing system. 2004.

[10] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in tandem computer systems. *The Evolution of Fault-Tolerant Computing*, 1:5576, 1986.

[11] E. A. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, page 710, 2000.

[12] A. Brito, C. Fetzer, and P. Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, page 173182, 2009.

[13] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):6375, 1985.

[14] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[15] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of CIDR*, volume 3, page 257268, 2003.

[16] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB, 2010.

[17] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):5678, Feb. 1991.

[18] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 2011.

[19] J. Dean. Large-scale distributed systems at google: Current systems and future directions. In *Keynote speech at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107113, 2008.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, page 205220, 2007.

[22] E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375408, Sept. 2002.

[23] R. Gao and A. Shah. Serving large-scale batch computed data with project voldemort.

[24] J. Gehrke and S. Madden. Query processing in sensor networks. *Pervasive Computing, IEEE*, 3(1):4655, 2004.

[25] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):5159, 2002.

[26] P. V. Glenn Fowler. *Fowler-Noll-Vo (FNV) Hash*. 2006.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, page 1111, 2010.

[28] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. A comparison of stream-oriented high-availability algorithms. *Brown CS TR-03*, 17, 2003.

[29] J. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, page 176185, 2007.

[30] A. Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):3644, Aug. 2009.

[31] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):3540, 2010.

[32] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558565, 1978.

[33] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79103, 2006.

[34] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 113, New York, NY, USA, 2011. ACM.

[35] N. Marz. Storm. https://github.com/nathanmarz/storm.

[36] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, page 170177, 2010.

[37] T. M. Nguyen, J. Schiefer, and A. M. Tjoa. Sense & response service architecture (SARESA): an approach towards a real-time business intelligence solution and its use for a fraud detection application. In *Proceedings of the 8th ACM international workshop on Data warehousing and OLAP*, DOLAP '05, page 7786, New York, NY, USA, 2005. ACM.

[38] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, page 183192, 1999.

[39] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. page 10991110, 2008.

[40] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, page 137151, 1994.

[41] S. Schroedl, A. Kesari, and L. Neumeyer. *Personalized ad placement in web search.* ADKDD, 2010.

[42] T. Schtt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, page 4148, 2008.

[43] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. page 110, 2010.

[44] P. Staubach, B. Pawlowski, and B. Callaghan. NFS version 3 protocol specification. http://tools.ietf.org/html/rfc1813#section-3.3.8.

[45] P. Stoellberger. S4Latin: Language-Based big data streaming.

[46] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):4247, 2005.

[47] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, page 11501160, 2007.

[48] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):4044, Jan. 2009.

[49] R. Wagle, H. Andrade, K. Hildrum, C. Venkatramani, and M. Spicer. Distributed middleware reliability and fault tolerance support in system s. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, page 335346, New York, NY, USA, 2011. ACM.

[50] J. Wu. Krati: a hash based high-performance data store. http://sna-projects.com/krati/, June 2012.