SR3: Customizable Recovery for Stateful Stream Processing Systems

Hailu Xu*
Florida International University
hxu017@fiu.edu

Pinchao Liu*
Florida International University
pliu002@fiu.edu

Susana Cruz-Diaz
Florida International University
Miami, Florida
scruz065@fiu.edu

Dilma Da Silva
Texas A&M University
dilma@cse.tamu.edu

Liting Hu
Florida International University
lhu@cs.fiu.edu

Abstract

Modern stream processing applications need to store and update state along with their processing, and process live data streams in a timely fashion from massive and geo-distributed data sets. Since they run in a dynamic distributed environment and their workloads may change in unexpected ways, multiple stream operators can fail at the same time, causing severe state loss. However, the state-of-the-art stream processing systems are mainly designed for low-latency intra-datacenter settings and do not scale well for running stream applications that contain large distributed states, suffering a significantly centralized bottleneck and high latency to recover state. They offer failure recovery mainly through three approaches: replication recovery, checkpointing recovery, and DStream-based lineage recovery, which are either slow, resource-expensive or fail to handle multiple simultaneous failures.

We present SR3, a customizable state recovery framework that provides fast and scalable state recovery mechanisms for protecting large distributed states in stream processing systems. SR3 offers three recovery mechanisms — the star-structured recovery, the line-structured recovery, and the tree-structured recovery — to cater to the needs of different stream processing computation models, state sizes, and network settings. Our design adopts a decentralized architecture that partitions and replicates states by using consistent ring overlays that leverage distributed hash tables (DHTs). We show that this approach can significantly improve the scalability and flexibility of state recovery.

We realize the SR3 design on a prototype integrated with the widely adopted Apache Storm framework. Large-scale experiments using real-world datasets demonstrate SR3's scalability, fast recovery, and flexibility properties.

Keywords Stream processing, State recovery, Scalability.

ACM Reference Format:
Hailu Xu*, Pinchao Liu*, Susana Cruz-Diaz, Dilma Da Silva, and Liting Hu. 2020. SR3: Customizable Recovery for Stateful Stream Processing Systems. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3423211.3425681

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Middleware '20, December 7–11, 2020, Delft, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8153-6/20/12...$15.00
https://doi.org/10.1145/3423211.3425681

1 Introduction

Stream processing is proposed and popularized as a "technology like Hadoop but can give you results faster" [1], which lets users query a continuous data stream and get results shortly after receiving the data. Stream processing technology has become a critical building block of many applications,such as identifying spam campaigns from social network streams [2], making business decisions from marketing streams [3], and predicting tornados and storms from radar streams [4].

While in the early days most stream operators were used for simple computations which are *stateless*, such as `filter`, `sort`, today's stream operators are capable of powering more complex computations and evaluating more complex logic which are *stateful*, such as `mapWithState`. This requires today's stream processing systems to offer "*state handling*" — i.e., operators that can remember past input and use it to influence the processing of upcoming input.

However, stream processing applications may be highly dynamic due to factors such as variable data rates, network congestions, and application-specific data source characteristics. Stream processing applications are also often subject to instabilities and failures, where multiple streaming operators may fail at the same time, resulting in severe state loss that may break or hinder the progress of stream application workflows.

*Pinchao Liu contributed equally with Hailu Xu in this paper.

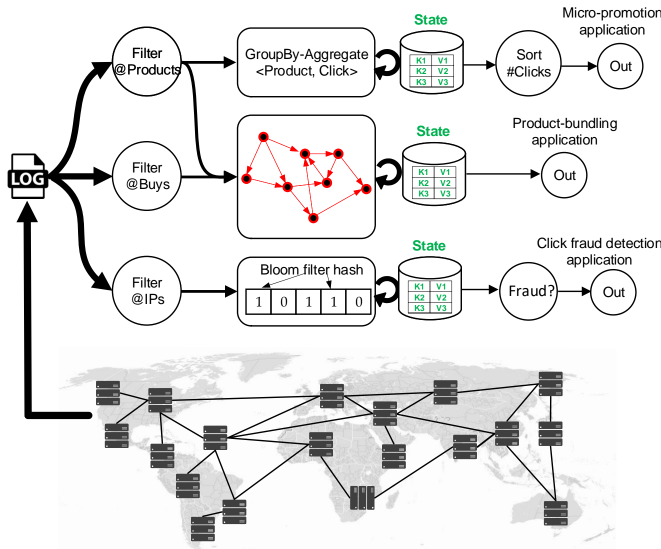


Figure 1. Real-world examples of stateful stream processing.

In this paper, we explore customizable state recovery mechanisms for protecting large distributed states in stream processing systems, in order to cater the needs of different stream processing applications that have different stream processing computation models, state sizes, and network environments.

Figure 1 shows real-world examples of stateful stream processing. When we are shopping at e-commerce websites, our user activities (e.g., *clicks*, *likes*, *buys*, *reviews*) are going to be continuously logged by these sites. On the backend, many stateful stream applications are concurrently running on top of these user activity streams to create insights and make business decisions. For example, Figure 1 top is a "*micro-promotion*" application, which analyzes the live page views of its products, `groupby-aggregates` them, and then *sorts* them to find the top-*k* products with the most clicks to apply a discount. Here the "state" is the stored knowledge base of key-value pairs consisting of product names and corresponding clicks. Figure 1 middle is a "*product-bundling*" application, which extracts users' buys, creates graphs of vertices and edges to get an idea of what products are usually purchased together, then makes online recommendations such as "*you like this, you may also like that*". Here the "state" is the stored knowledge base of connected graphs consisting of product names and bundlings. Figure 1 bottom is a "*click fraud-detection*" application, which identifies ad clicks as fraudulent by deploying a space-efficient probabilistic data structure like a Bloom filter [5] to memorize the IP addresses or the cookies of previous clicks, and comparing them to the new coming click stream to detect duplicate clicks in a short time. Here the "state" is the stored knowledge base in the Bloom filter.

However, we are facing significant challenges in managing these large distributed states in stream processing systems.

- *Challenge 1: how to recover from simultaneous failures of multiple stream operators for a large number of concurrently running applications?* Streaming computations are, by nature, long-running. Their workloads, as well as the runtime environment, may change in unpredictable ways. A stream computation is usually represented as a logical directed acyclic graph (DAG), where vertices denote operators and edges denote data dependencies between them. This means that if one operator fails and loses state, the dependent operators may also fail and lose their states. Multiple failures may frequently happen due to node fails, buffer IO error, or drive unfound. For example, in power outages, a non-negligible percentage (0.5%-1%) of nodes does not come back to life even after power is restored [6]. Multiple failures can also be easily affected by the location of devices (i.e., in different geo-distributed datacenters) and the type of cloud services [7]. What makes it particularly challenging is that many stream processing applications run concurrently on the same platform and consume the same data source. We need to be able to recover lost state for large numbers of concurrently running applications on the same platform.
- *Challenge 2: how to customize the failure recovery mechanism for different types of stream processing applications?* For example, Spark Streaming based systems [8, 9, 10, 11] treat streaming computations as a series of batch computations, whereas Storm based systems [12, 13, 14] treat streaming computations as a dataflow graph in which vertices asynchronously process incoming records. The state size for batch applications is usually large, whereas the state size for stream applications are usually small. Therefore, different stateful stream processing applications need different state recovery mechanisms that best meet their needs.

Since 2005, there has been a boom of stream processing systems, including Storm [12], Trident [13], Spark Streaming [15], Borealis [16], TimeStream [17], and S4 [18]. However, there is a lack of fast and scalable failure recovery mechanisms for protecting the large distributed states for these systems. The reasons are as follows: (1) they mostly inherit MapReduce's "single master/many workers" architecture, where the central master is responsible for all scheduling activities. As such, they do not scale well to a large number of concurrently running applications due to the inherent centralization bottleneck; (2) these systems offer failure recovery mainly through three approaches: replication recovery [19, 20], checkpointing recovery [8, 9, 12, 13, 17] and DStream-based lineage recovery [10, 11], which are either slow, resource-expensive or fail to handle multiple

simultaneous failures. Replication recovery adds high hardware cost because multiple copies must concurrently run on distinct nodes for failover. In distributed streaming, checkpointing recovery is known to be prohibitively expensive, leading users in many domains to disable this feature [21, 22, 23, 24, 25]. The third approach, DStream-based lineage recovery, is slow when the lineage graph is long (i.e., the streaming involves long sequences of operators) and falls short in handling multiple simultaneously failures; and (3) these systems are limited to a fixed computation model, e.g., asynchronous stream processing like Storm [12], synchronous mini-batch processing like Spark [15], and they do not have customizable state recovery mechanisms.

We make the following contributions in this paper. First, we show how existing techniques can lead to slow or resource-expensive state recovery that is not scalable and identify the causes (Sec. 2) of their shortcomings.

Second, we propose SR3, a customizable State Recovery framework that provides fast and scalable failure recovery mechanisms for protecting large distributed states in stream processing systems (Sec. 3). SR3 does not rely on a central master for recovering the state. The failure recovery process is deployed on a DHT-based peer-to-peer (P2P) overlay, scales to the size of the lost state, offers a significant reduction in failure recovery time and can tolerate multiple simultaneous node failures.

Third, SR3 provides three different failure recovery mechanisms (Sec. 3). An important novel aspect of SR3 is that it can host multiple distributed streams and offer each application the recovery mechanism that best fits its requirements. The goal is to cater to the needs of different stateful stream applications (e.g., different stream processing computation models, quality of service requirements, state sizes, and network environments).

Fourth, we present the integration of SR3 onto the Apache Storm framework and demonstrate its portability to other stream processing systems (Sec. 4). The source code of SR3 will be made publicly available.

Finally, we make a comprehensive evaluation of the scalability, fast recovery and flexibility of SR3 on a large cluster using real-world stream processing applications' datasets (Sec. 5).

2 Related Work

Distributed stream processing systems need to support stateful processing and recover quickly from failures to resume the normal processing. Table 1 summarizes the stateful stream processing systems and their state-of-the-art state recovery solutions.

2.1 State Management in Stream Processing Systems

Existing state management solutions can be divided into three representative categories: *in-memory*, *remote storage*, *in-memory+on-disk*.

Category 1: in-memory. Many industrial stream processing systems either do not support state (Heron [14], S4 [18], the early version of Storm [12]), or they rely on in-memory data structures such as hash tables and hash table variants to store state. For example, Muppet [26] and Trident [13] store state via hash tables. Spark Streaming [15] enables state computation via Resilient Distributed Datasets (RDDs) [31], the core data abstraction from Spark that distributes read-only multiset data items. These techniques rely on a central master for state management that results in a centralized bottleneck and, therefore, may be difficult to scale to large states.

Category 2: remote storage. Some systems such as Millwheel [27] and Dataflow [28] choose to separate state from the application logic. They have the state centralized in a remote storage [22, 32, 33] (e.g., a database management system, HDFS or GFS) shared among applications, periodically checkpointing it for fault tolerance. Using external storage can scale well to large distributed states, but it significantly increases latency in the critical path of stream processing. For example, Kafka can handle 100k–500k messages/sec per node at in-memory speed, however, the throughput of queries for remote key-value storage is often close to 1-5k requests per second — two orders of magnitude slower [29, 34].

Category 3: in-memory+on-disk. A few other systems such as Kafka [29], Samza [30, 34], Spark Streaming [15], Flink [8, 9] try to overcome this issue by using a combination of "soft state" stored in in-memory data structure along with "hard state" persisted in some on-disk data stores (e.g., RocksDB [35], LevelDB [36]). However, they sacrifice programming model transparency by requiring programmers to declare and maintain state using built-in data structures (e.g., Spark's RDDs [31], Muppet's slates [26]). The on-disk data store (e.g., Kafka [29], Samza [34], Dataflow [28]) incurs large I/O overhead due to well-known high write amplification [37]. Finally, scaling to large distributed states and recovering from failures in such systems is quite expensive, because when a single node fails, the in-memory state and on-disk state for all dependent nodes must be reset to the last checkpoint, and computation must resume from that point, resulting in significant time and space overhead.

2.2 Failure Recovery in Stream Processing Systems

Existing stream processing systems offer failure recovery mainly through the use of three approaches: *replication recovery*, *checkpointing recovery*, and *DStream-based lineage recovery*, which are either not scalable, slow, resource-expensive or incapable to handle multiple failures.

In the replication-based recovery approach, the system maintains a completely separate set of hot failover nodes, which processes the same stream in parallel with the primary set of nodes. The input records are sent to both. When there is a failure or multiple failures in the primary nodes, the system automatically switches over to the secondary

Table 1. Overview of state management and state recovery solutions in stream processing systems.

System	Data Structure	State Management	State Recovery Approach	Scale to large state	Handle Multiple Failures	State Recovery Policy	State Recovery Traits
Muppet [26]	Slates [26]	In-memory	Checkpointing	×	×	Static	Slow
Trident [13]	Hashtable	In-memory	Checkpointing	×	×	Static	Slow
Millwheel [27]	Hashtable	Remote storage	Checkpointing	×	×		Slow
Dataflow [28]	Hashtable	Remote storage	Checkpointing	×	×	Static	Slow
Kafka [29]	Hashtable	In-memory+on-disk	Checkpointing	×	×	Static	Slow
Samza [30]	Hashtable	In-memory+on-disk	Checkpointing	×	×	Static	Slow
Flink [8]	Hashtable	In-memory+on-disk	Checkpointing	×	×	Static	Slow
Flux [19]	Hashtable	In-memory+on-disk	Replication	×	✓	Static	High cost
Borealis [20]	Hashtable	In-memory+on-disk	Replication	×	✓	Static	High cost
Spark Streaming [15]	RDDs [31]	In-memory+on-disk	DStream-based lineage recovery	×	✓	Static	Slow for long lineages
SR3	Hashtable	In-memory	DHT-based parallel recovery	✓	✓	Dynamic	Fast, low cost

set of nodes and the system can continue processing with very little or no disruption. The replication recovery has been used in systems such as Flux [19] and Borealis [20]. The failover is fast and it can handle multiple failures. However, the replication recovery scheme doubles the hardware requirement.

In checkpoint-based recovery, all nodes periodically checkpoint their states to remote storage such as HDFS or GFS. Each node in the stream pipeline has an in-memory buffer to retains a backup of the data that it has forwarded to the downstream nodes since the last checkpoint. The system also maintains standby nodes. When a primary node fails, a standby node retrieves the latest checkpoint from the persistent storage, and its upstream node essentially replays the backup records serially to this failover node to recreate the lost state. The checkpointing recovery has been used in systems such as TimeStream [17], Trident [13], Drizzle [38], Flink [8, 9] and Multilevel Checkpointing [39]. It avoids the 2× hardware cost. For example, Flink [8, 9] can retrieve the checkpointed external state from remote database without the secondary set of nodes. However, the failover is slower than the replication recovery because it has to retrieve the checkpointed state from the remote storage and replay the buffered data on the last state to recompute the new state.

To achieve both fast recovery and small hardware overhead, the DStream-based lineage recovery was proposed. This approach has been used in Apache Spark-based systems [10, 11]. Its key abstraction is the *Discretized Stream* (DStream, for short), a continuous stream of Spark RDDs [31]. The most recent state is stored in each node's memory - using RDDs - together with the lineage graph, that is, the graph of deterministic operators used to build RDDs. When nodes fail in the system, instead of preparing a standby node for failover, DStream re-runs the lost tasks in parallel on other reliable nodes in the cluster using the lineage graph. However, the entire recovery processing is linear, that is, the lost tasks need to be executed strictly in line with the original lineage graph. As such, it may not work well for multiple failures because the recovery process is executed strictly in line with the original lineage graph. As such, the recovery process may be slow and incur multiple uploads of checkpointed state, incurring substantial network traffic in geo-distributed setting.

2.3 Preliminary Work

Our previous work [40] introduced FP4S, a fragment-based parallel state recovery mechanism that can handle many simultaneous failures for stateful stream applications. The key idea is to use erasure coding. For example, (32, 16)-Reed-Solomon (RS) code [41] divides a data object into 16 blocks and transforms these blocks into 32 coded blocks, guaranteeing that any 16 out of the 32 coded blocks are sufficient to reconstruct the original data object. Each stateful operator's state is divided into m fragments and then encoded into n blocks and checkpointed to n nodes in the leaf set in parallel ($n > m$), which guarantees that the original state can be reconstructed from any m blocks and it can tolerate a maximum of $(n - m)$ failures at a time.

Nonetheless, our preliminary work exhibits major limitations including: (1) **Expensive hardware cost.** Since it leverages erasure codes to fulfill state save and recovery, it will generate redundant states and cause additional storage space to save state fragments. For example, when using 16 raw fragments and 10 coded fragments to recover a 128MB state, a total of 208 MB (62.5% increment) is required in memory or disk to complete the state saving and recovery process. This causes a lot waste of resources and may be inefficient in systems with limited hardware resources. (2) **High latency.** Even though FP4S can tolerate many simultaneous failures by leveraging erasure codes, it incurs extra

computation overhead in generating coded fragments and computing state from fragments. The extra computational overhead may cause a long delay when recovering large sized state. For example, to recover a 128MB state from erasure codes, FP4S incurs extra overhead in the erasure code computation, which takes an additional 10s in recovering 128MB state. Therefore, the entire latency for saving and recovering 128MB state in FP4S will take around 90s (80s+10s). The time quickly increases with the increment of state size. (3) **No customization.** FP4S provides a single recovery mechanism for all stream applications. However, stream applications are quite diverse. They may have different QoS requirements, state sizes, computation models, or run in different network environments. Therefore, a single recovery mechanism may not work for handling all scenarios. For example, in diverse network environments, the availability of network resources varies dynamically depending on which stream application workflows are active at a given moment. therefore, the technique introduced in FP4S is not appropriate for supporting diverse stream workflows. In contrast, SR3 provides three different failure recovery mechanisms to cater the needs of different stateful stream applications.

3 Design

In this section, we define the problem (Sec. 3.1) and discuss the background of DHT-based consistent ring overlay (Sec. 3.2). We then introduce the SR3 framework, which includes the system overview (Sec. 3.3), the star-structured recovery mechanism (Sec. 3.4), the line-structured recovery mechanism (Sec. 3.5), the tree-structured recovery mechanism (Sec. 3.6), and how SR3 determines which mechanism to use (Sec. 3.7).

3.1 Problem Statement

We follow a generic stream query model [10, 21, 42, 43, 44]. A stream processing application's query is a directed acyclic graph (DAG) that specifies the dataflow, denoted as $Q = (V, E)$. DAGs can be implemented via many execution models, such as the partition/aggregate model which scales out by partitioning tasks into many sub-tasks (e.g., Dryad [45]), the sequential/dependent model in which streams are processed sequentially and subsequent streams depend on the results of previous ones (e.g., Storm [12]), and the hybrid model with sequential/dependent and partition/aggregate components (e.g., Spark Streaming [15], Naiad [21]).

A vertex $v \in V$ corresponds to a stream operator f_v that consumes input streams i from its predecessor (upstream) vertices and produces output streams o to its successor (downstream) vertices ($o = f_v$(i)). Each edge $e \in E$ represents a data flow between two vertices. The stream operator f_v can be *stateless* or *stateful*. A stateless operator consumes one input record at a time and outputs each result based solely on that last input record. A stateful operator maintains state that captures characteristics of some of the records processed so far and updates it with each new input, such that the output takes into account both historical records and the new input. Stateless operators are easy to recover because, by definition, input records are handled independently, and upon failure we can simply start a new operator instance. In contrast, stateful operators are much more difficult to recover.

The problem is: *how to achieve a scalable and fast failure recovery framework that protects large distributed states for concurrently running applications deploying diverse execution models?* These applications run concurrently on a shared distributed environment. Their operators are stateful. The applications comprise several DAGs, deploy diverse execution models, and vary on their requirements of CPU, memory, and network bandwidth.

3.2 Background

For maintaining and recovering state, our solution leverages peer-to-peer (P2P) overlay networks, more specifically, the Distributed Hashtable (DHT)-based consistent ring overlay with routing. The primary purpose of the P2P model (e.g., Pastry [46], Chord [47]) is to enable all nodes to work collaboratively to deliver a specific service. In such model, all nodes have similar roles, both serving and requesting services. For example, in BitTorrent [48], if someone downloads some file, the file is downloaded to her computer in bits and parts that come from many other computers in the system that already have that file. At the same time, the file is also sent (uploaded) from her computer to others who ask for it. Similarly to BitTorrent, where many machines work collaboratively to download and upload files, our solution enables all distributed nodes to work collaboratively to achieve state management, relieving the task scheduler (often implemented as a centralized service) from handling state. For this purpose, we leverage the following three data structures from DHT-based consistent ring overlays:

- *Routing table*: The routing table consists of node characteristics (*IP address*, *Node Id*) organized in rows by the length of common prefixes in the representation of a *Node Id*. When routing a message to `nodeId`, a node forwards it to the node in its routing table with the longest prefix in common with `nodeId`. State are associated with keys and nodes are responsible for a range of keys. In a system where N nodes store state, it is guaranteed that queries can be routed to the appropriate `nodeId` within $O(logN)$ hops. We use the routing table for locating state and in the line-structured recovery mechanism (Section 3.5).
- *Leaf set*: The leaf set for a node is a fixed number of nodes that have the numerically closest `nodeIds` to that node. This assists nodes in routing messages and in rebuilding routing tables when nodes fail. We use

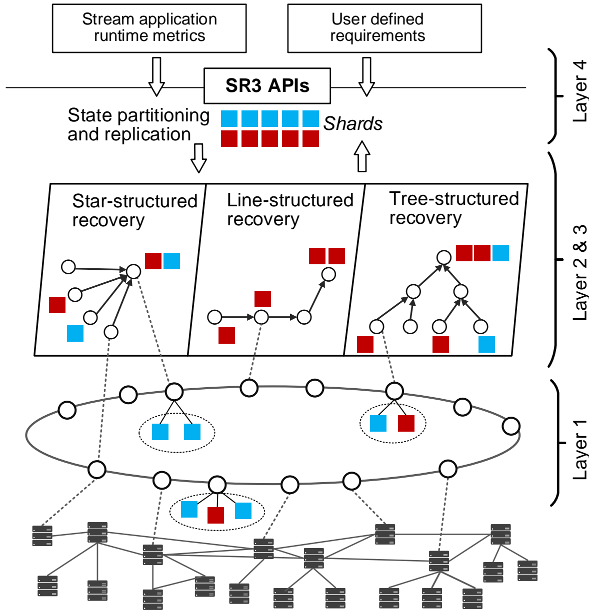


Figure 2. The overview of SR3 design.

the leaf set for the star-structured recovery mechanism (Section 3.4).

- *Multicast*: Any node in the overlay can create a communication group; other nodes can join the group and then multicast message to all members of the group. Multicast messages are disseminated through a multicast tree. We use multicast for constructing in the tree-structured recovery (Section 3.6).

3.3 The SR3 Overview

Figure 2 shows the overview of the SR3 system. It consists of several layers as follows.

Layer 1: DHT-based overlay. In our system, we introduce a new abstract concept called "node" to facilitate state management. Each stream operator is associated with a node. The association is unrelated to where operators execute; operators at the same vertex may be associated with different nodes. Each node is randomly assigned a globally unique identifier known as the "nodeId" in a large circular node ID space (e.g., 0-2^{128}). We organize these nodes into a P2P overlay network. The overlay is self-organizing and self-repairing.

Layer 2: State partitioning and replication. The node's state is stored in an in-memory hashtable data structure. Periodically, we divide each node's state into m shards, each of which is then replicated to n replicas and distributed to peer nodes. The peer nodes are preferably chosen as to enable high bandwidth communication. The parameters of m and n are determined by the adopted recovery mechanism (we offer three alternatives) and application characteristics. Our design ensures that when a failure happens, different sets

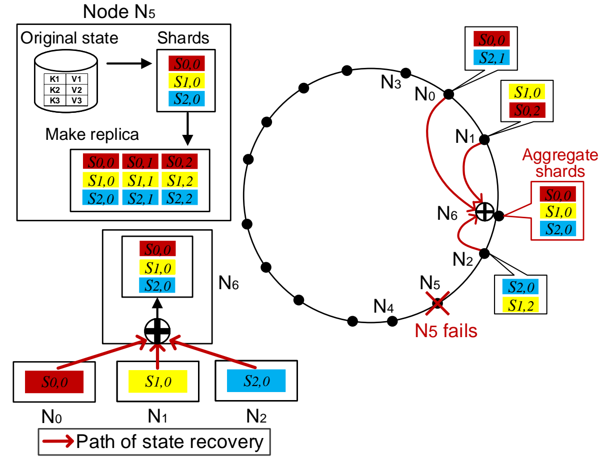


Figure 3. The star-structured recovery process.

of available shards can reconstruct failed state in parallel, thereby reducing the failure recovery time while tolerating multiple simultaneous node failures.

Layer 3: State recovery. Applications differ in state sizes, execution models and QoS requirements such as latency and throughput. Some applications, such as simulations that can adjust to data errors, can tolerate lower accuracy in exchange for efficiency in accessing state and quick recovery other tasks, such as state visualization for application debugging, cannot. We design three state recovery mechanisms to satisfy the needs from different applications. SR3 tracks user-defined requirements (e.g., latency sensitivity) and the application's characteristics (e.g., size of the state) to select the most appropriate mechanism (see Sec. 3.4, Sec. 3.5 and Sec. 3.6 for more details).

Layer 4: SR3 API. SR3 is currently integrated into Apache Storm [12]. We provide a high-level API that exposes to users configuration parameters and enables SR3's portability to other stream processing systems.

3.4 The Star-structured Recovery Mechanism

Figure 3 shows a straightforward implementation of star-structured recovery mechanism. Each node has a routing table and a leaf set. In this example, the state of node N_5 is divided into 3 shards and each shard has two replicas. They are distributed to the leaf set to ensure that the original state can be reconstructed from 3 shards of the 9 shards, where each shard among these 3 shards is the one from the its 3 replicated shards. As shown in Figure 3, the nine shards $s_{0,0}$, $s_{0,1}$, ..., $s_{2,2}$ are stored in N_0, N_1, ..., N_5 respectively. When N_5 fails, N_0, N_1, and N_2 upload $s_{0,0}$, $s_{1,0}$, and $s_{2,0}$ to N_6 to recompute the state of N_5.

The benefits are: (1) the recovery process is fast. Different nodes from non-overlapping leaf set nodes can work in parallel to recompute the lost state, which is much faster than retrieving the state from the remote storage (e.g., HDFS). (2)

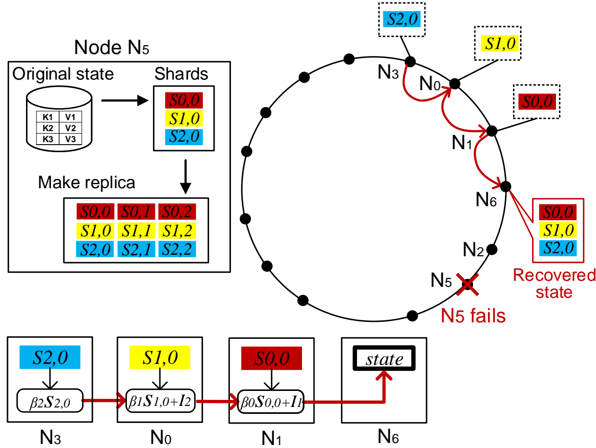


Figure 4. The line-structured recovery process.

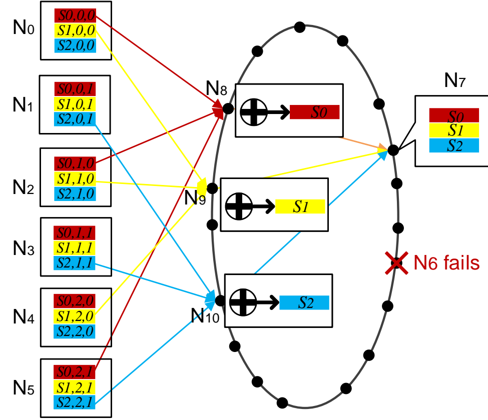


Figure 5. The tree-structured recovery process for single failure.

We achieve data locality because the leaf set contains nodes that are geographically close to the original node (e.g., within the same rack in the same site) that have abundant upload bandwidth.

3.5 The Line-structured Recovery Mechanism

The star-structured recovery works fine when the state is small. However, when the state is large, the replacing node needs to do all the downloading and reconstructing work, suffering a centralized bottleneck that increases the recovery latency, which we aim to avoid. We design the line-structured state recovery to fix this issue, where shards are transmitted and combined through a line covering the replacing node and all providing nodes. As shown in Figure 4, the nine shards $s_{0,0}$, $s_{0,1}$, ..., $s_{2,2}$ are stored in N_0, N_1, ..., N_5 respectively. When N_5 fails, N_3 uploads $s_{2,0}$ to N_0. N_0 merges $s_{2,0}$ with $s_{1,0}$, reconstructs it, and then uploads the result to N_1. N_1 merges the result with $s_{0,0}$, reconstructs it, and uploads the final result to N_6 to replace of N_5. The benefit is that, the downloading and computing load are well balanced among all involved nodes which helps recover large state. However, it can only recover one node at a time. When recovering multiple node failures, it may incur multiple times of network traffic and recovery time. Besides, the line-structured recovery disregards the bandwidth asymmetry in cloud environment.

3.6 The Tree-structured Recovery Mechanism

We design a shard-based parallel recovery mechanism to tolerate multiple node failures, where shards are transmitted and combined through a spanning tree covering the replacing node and all providing nodes. This spanning tree is built on top of a scalable application-level multicast infrastructure, called Scribe [49]. The key idea is to divide the state into many shards (e.g., based on key ranges), and use different sets of available replicas of shards scattered across leaf set nodes to reconstruct unavailable shards in parallel. By doing this, all nodes storing available shards can work as providing nodes and each of them only needs to participate in the recovery of some unavailable shards. This means a providing node only needs to upload some of the shards it stores. Thus, the amount of data a providing node uploads is reduced in a way that respects bandwidth asymmetry. The downloading and computing load are well balanced among all involved nodes without any centralized bottleneck.

Figure 5 shows the recovery process from a single failure in the tree-structured mechanism. N_7 is the replacing node for recovering the state when N_5 fails. We can see that the state is divided into 3 shards, s_0, s_1 and s_2. Each shard is further divided into 3 sub-shards and the replication factor is two. So for one shard, it has 6 sub-shards in total. For example, $s_{2,0,1}$ denotes the second replica of the first sub-shard in s_2, and $s_{2,1,0}$ denotes the second replica of the second sub-shard in s_2. In the tree-structured recovery process, the providing node only needs to upload 3 out of the 6 total sub-shards to reconstruct each shard. Here each shard refers to the shard of different partitions (different colors in Figure 5).

Figure 6 shows the recovery process from two failures in the tree-structured mechanism, where N_6 and N_7 are the replacing nodes for recovering the state when N_4 and N_5 fail. Similarly, the state is divided into 3 shards, and each shard is divided into 3 sub-shards and the replication factor is two. The recovery from multiple failures is similar with the recovery from a single failure, that is, the providing node only uploads 3 sub-shards to construct each shard. The difference is that every reconstructing node needs to reconstruct multiple shards and sends them to multiple replacing nodes.

3.7 Mechanism Selection

Which mechanism to use? Determining the optimal state recovery mechanism is difficult since it needs to consider various factors and application specifics. Thus, we rely on a heuristic approach that adapts mechanism based on (1) state

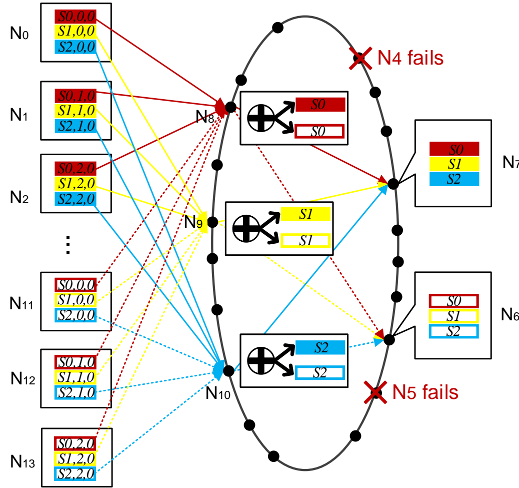


Figure 6. The tree-structured recovery process for two failures.

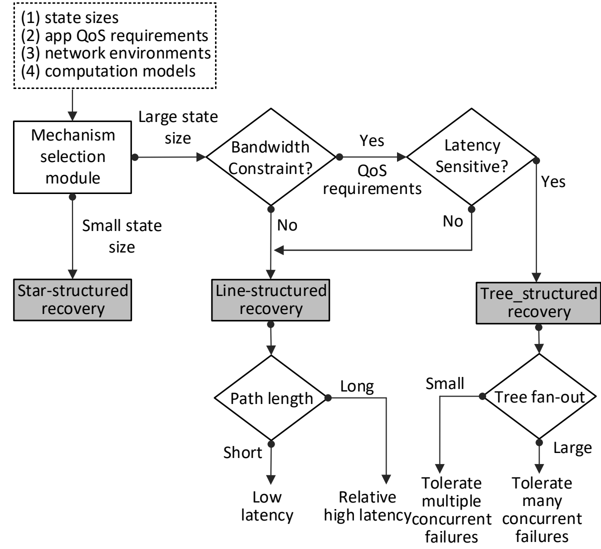


Figure 7. Determining which state recovery mechanism.

sizes, (2) application QoS requirements, (3) network environments (e.g., bandwidth bottleneck), and (4) computation models (e.g., synchronous micro-batch processing model or asynchronous stream processing model). SR3 focuses on fast recovery (tens of second scale) for large distributed states in concurrently running applications rather than real-time recovery.

Figure 7 shows how we determine which mechanism to use. In the case of stateless operator failures, it will simply resume the whole execution pipeline since there is no overhead for recovering states. In the case of stateful operator failures, SR3's state recovery mechanisms may not always outperform the traditional checkpointing recovery if the state size is too small or if the application can tolerate the checkpointing overhead of writing state to the remote storage. Thus, we use SR3 only with stateful operators for (1) applications that have strict QoS requirements for low recovery latency and (2) high probability of simultaneous failures that will involve large distributed states, such as many concurrent failures of Twitter trends due to servers' failures after power outages.

This information about application's QoS requirements and state size is typically available as part of the job submission information. If the state size is small, we choose star recovery in priority. On the other hand, if the state size is large, we further consider if the execution is constrained by the network bottleneck. In the case of abundant bandwidth, we choose line-structured recovery in priority by adjusting the recovery path length to handle different sizes of state and latency requirements. In the case of limited bandwidth, we further consider application's QoS requirements. If it is latency insensitive, we still choose line-structured recovery in priority. Otherwise, we choose tree-structured recovery in priority by dynamically tuning the runtime parameters such like tree fun-out, depth, number of replicas, or number of nodes. Therefore, it can cater for different runtime requirement and handle different sizes of state and concurrent failures that occur simulteneously.

4 Implementation

We implement the SR3 system on top of Apache Strom [50] (v.2.0.0) and Pastry [51] (v.2.1) software stacks. Instead of implementing another distributed system core, we leverage Storm's excellent runtime system (e.g., basic API, code interpreter, objects exchange layer) and Pastry's DHT routing substrate and event transport layer. Although some of the implementation details may be related to certain specifics of Storm, the overall architecture and designs of SR3 can fit into many other stream processing systems (e.g., Trident [13], Spark Streaming [15], Kafka [29], Samza [30], Flink [8], Wukong+S [52]). For example, Trident [13] can avoid to write state to in-memory store or remote storage such as Cassandra [53] by directly writing state into SR3 via its APIs. The APIs of SR3 can be easily cooperated into many other platforms so that they can replace the original state failure handler by SR3.

In Storm [12], stream processing applications are deployed and executed as topologies. The topologies contain the business logics. These logics are formed as a DAG and implemented by spouts and bolts. Spouts are the data sources of the stream, which accept input data from raw data sources like Twitter Streaming API [54], Apache Kafka queue [29], etc. Bolts are the logical processing units. Spouts pass data to bolts and bolts process and produce a new output stream. *IRichBolt* is the common interface for implementing bolts.

We made three major modifications: (1) SR3 interacts with the `IRichBolt` interface in Storm [12]. If SR3 is enabled, SR3

Table 2. SR3 APIs.

API	Description
List<>StateSplit(Sting state, Integer numberofShards, Integer numberofReplicas)	This function is invoked to partition the state into shards and create replicas. The inputs are the original state, number of shards, and number of replicas for each shard. The output is a list of replicas that contain various parts of the original state.
Boolean[] Save(List<>shard, DHTNetwork overlay)	This function is invoked to save shard into the overlay. It creates many concurrent threads to save the replicas of shards.
Boolean StarDefine(String appName, Integer starFanout)	This function defines the star-structure recovery. Users can define the star fan-out.
Boolean LineDefine(String appName, Integer lengthofPath)	This function defines the line-structured recovery. Users can define the length of recovery path.
Boolean TreeDefine(String appName, Integer fanout, Integer branchDepth)	This function defines the tree-structured recovery. Users can define the tree fan-out and the branch depth.
Integer Selection(String appName, String requirement, Long stateSize, Long networkBW)	This function is invoked to select a specific mechanism for recovering state. The inputs are the application's name, runtime QoS requirements, the size of state, and the available bandwidth. The output indicates which mechanism is selected.
List<Shards>Recover(String stateName, DHTNetwork overlay, Mechanism structure)	This function is invoked when a state recovery request is issued.

Table 3. Real-world application's dataset.

Application	Dataset	Size
Bargain Index	Google Finance [55]	>1TB
Word Count	Wikimedia Dumps [56]	9GB
Traffic Monitoring	Dublin Bus Traces [57]	4GB

periodically saves state into the DHT-based ring overlay for all stateful operators (bolts). The recovery mechanisms and aggregated state size are configurable in order to satisfy different real-world stream applications' requirements; (2) we implemented the star-structured mechanism and the line-structured recovery mechanism on top of Pastry's overlay, and implemented the tree-structured mechanism on top of Scribe's topic-based publish/subscribe trees. We added several new data structures: a list of operations for managing shards (i.e., `replicate`, `merge`, and `save`), a list for tracking the locations of each shard and routing shards, and a list for recording the dependency path between nodes; and (3) we implemented state version control by adding timestamps and sequence numbers to the messages, thereby avoiding state inconsistency during the state saving and recovery process.

The SR3 prototype will be publicly available at GitHub. It adds 1800 lines of Java across 17 files. We provide high-level interfaces exposed to users for configuring parameters. Table 2 shows the details of SR3 APIs.

5 Evaluation

We evaluate SR3 on emulation testbed in a distributed network environment. We explore its performance for a variety of real-world stream processing applications. Our evaluation answers the following questions:

- Does SR3 improve the state save and recovery performance when deploying different stream applications with various sizes of states?
- Does SR3 support flexible state recovery in handling various sizes of states with different network environments?
- Does SR3 scale with the number of concurrently running stream applications?
- What is the runtime overhead of SR3?

5.1 Setup

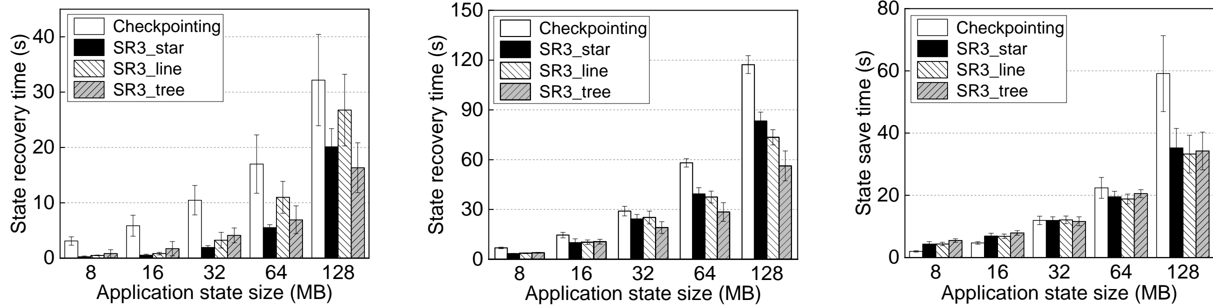
Evaluation deployment. Emulation experiments are conducted on a testbed of 50 virtual machines (VMs) running Linux 3.10.0, all connected via Gigabit Ethernet. Each virtual machine has 4 cores and 8GB of RAM, and 60GB disk. Specifically, to evaluate SR3's scalability, we use one JVM to emulate an SR3 node and emulate up to totally 5,000 SR3 nodes in our testbed. Linux VMs are equipped with LANs with high bandwidth diversity set by traffic control.

Baseline. We used Apache Storm [12] as the stream processing engine baseline. We use Apache Storm 2.0.0 [50] configured with 10 TaskManagers, each with 4 slots (maximum parallelism per operator = 36). We use Pastry 2.1 [51] with leaf set size of 24 and transport buffer size of 32MB.

Benchmark and applications. To demonstrate generality across diverse computations and streaming operators, we evaluate SR3 in state recovery with the real-world stream applications (see Table 3). These stream applications contain various streaming operators: stateless streaming transformations (e.g., `map`, `filter`), stateful operators (e.g., `incremental join`), and various window operators (e.g., sliding window, tumbling window and session window). We deploy these benchmarks to generate real-world application in Apache Storm to generate different sizes of state and various topologies.

We compare SR3 with a state-of-the-art failure recovery solution: the checkpointing recovery approach commonly used in TimeStream [17], Storm [12], and Trident [13]. We choose the checkpointing recovery approach as the baseline approach because of two reasons: (1) the replication recovery already costs 2× hardware, and (2) the DStream-based lineage recovery approach is not generalized for users. Because DStream-based lineage recovery sacrifices programming model transparency by forcing programmers to manage state using RDDs [31].

Metrics. We focus on the performance metrics such as latency of state save and recovery. The latency measurement is separated by the state save and recovery. The latency is

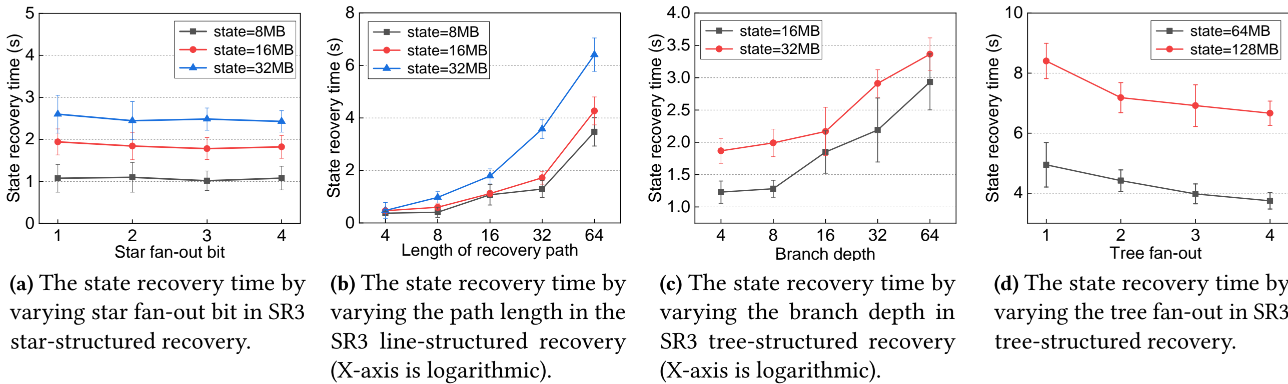


(a) The state recovery time by varying the size of state with no bandwidth constraint.

(b) The state recovery time by varying the size of state with bandwidth constraint.

(c) State save time by varying the size of state.

Figure 8. Performance evaluation of SR3 in terms of the time of state save and recovery in checkpointing, star-structured recovery, line-structured recovery, and tree-structured recovery.



(a) The state recovery time by varying star fan-out bit in SR3 star-structured recovery.

(b) The state recovery time by varying the path length in the SR3 line-structured recovery (X-axis is logarithmic).

(c) The state recovery time by varying the branch depth in SR3 tree-structured recovery (X-axis is logarithmic).

(d) The state recovery time by varying the tree fan-out in SR3 tree-structured recovery.

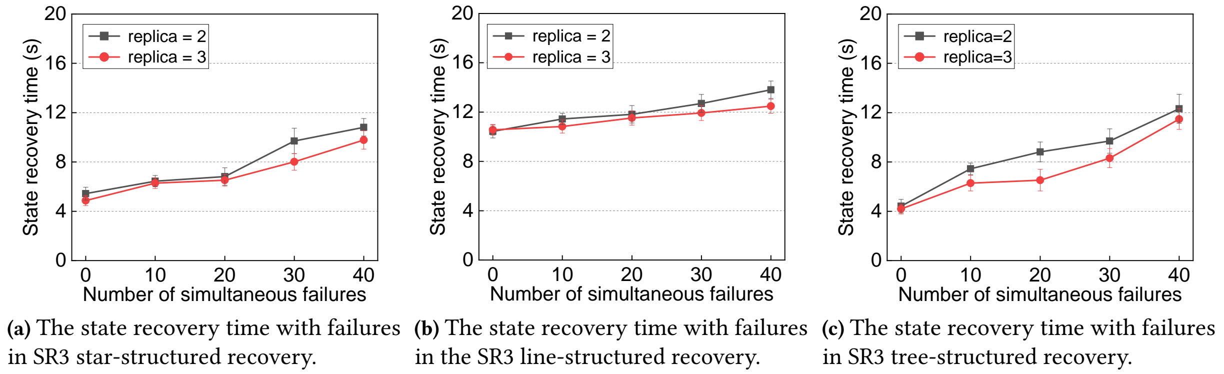
Figure 9. Performance evaluation of SR3 in terms of the state recovery time in star-, line-, and tree-structured recovery.

evaluated by deploying different size of state shards and various size of states of stream application. To evaluate the scalability of SR3, we measure how much state shards are distributed in each node with deploying different stream applications. To the runtime overhead of SR3, we focus on the CPU and memory utilizations during the state recovery.

5.2 SR3 vs Checkpointing Recovery

We compare the state recovery time of SR3 with Storm by varying the size of state with no bandwidth constraint. As Figure 8a shows, SR3 generally achieves 35.5% ∼ 65% less state recovery time compared to Storm. More specifically, when a state is relatively small (<32MB), the star-structured recovery mechanism achieves the fastest recovery. Line-structured recovery and tree-structured recovery take a little longer due to the introduction of redundant calculations in their state recovery paths. When the state grows larger than a threshold (e.g., 64MB), line-structured recovery leads to the longest recovery time due to the longest lineage path. On the contrary, since tree-structured recovery has many paths for recovering at the same time in parallel, the time is reduced.

Figure 8b shows the state recovery time comparison of SR3 with Storm under bandwidth constraint. Note that the upload bandwidth is limited to 100Mb/s per server. Results show that SR3 generally achieves 29.8% ∼ 42.5% less state recovery time compared to Storm. More specifically, when the state is relatively large (> 64MB), due to the constraint of the upload bandwidth, the star-structured recovery has a centralized bottleneck because all traffic flows to a single node, which leads to the slowest state recovery. On the contrary, the line-structured recovery and tree-structured recovery avoid this bottleneck, and thus are much faster. However, when the state becomes extremely large, the tree-structured recovery performs the best because it has many paths to recover state at the same time in parallel. This gives us insight that we should decide which mechanism to use based on the application characteristics, the network environment, and the size of state.

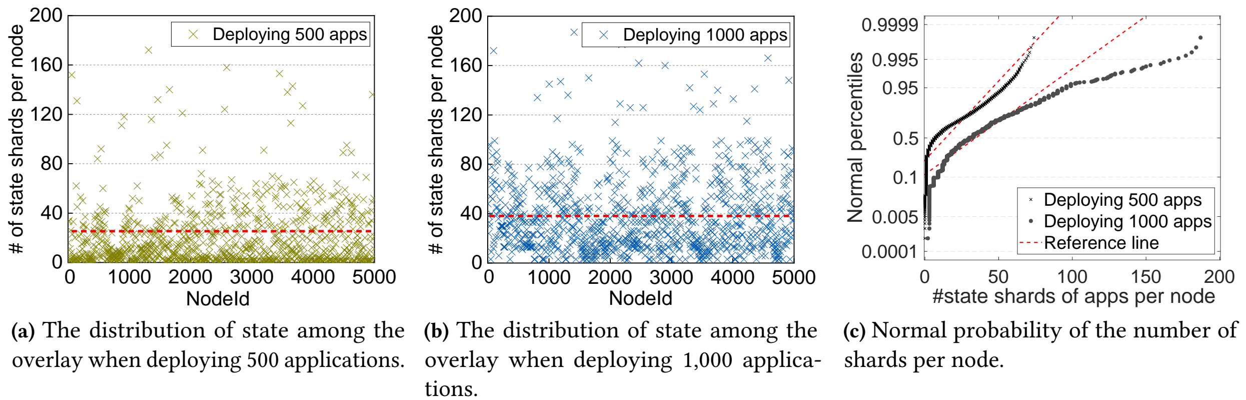


(a) The state recovery time with failures in SR3 star-structured recovery.

(b) The state recovery time with failures in the SR3 line-structured recovery.

(c) The state recovery time with failures in SR3 tree-structured recovery.

Figure 10. State recovery time with different number of failures.



(a) The distribution of state among the overlay when deploying 500 applications.

(b) The distribution of state among the overlay when deploying 1,000 applications.

(c) Normal probability of the number of shards per node.

Figure 11. Performance evaluation of SR3 in terms of load balance and scalability.

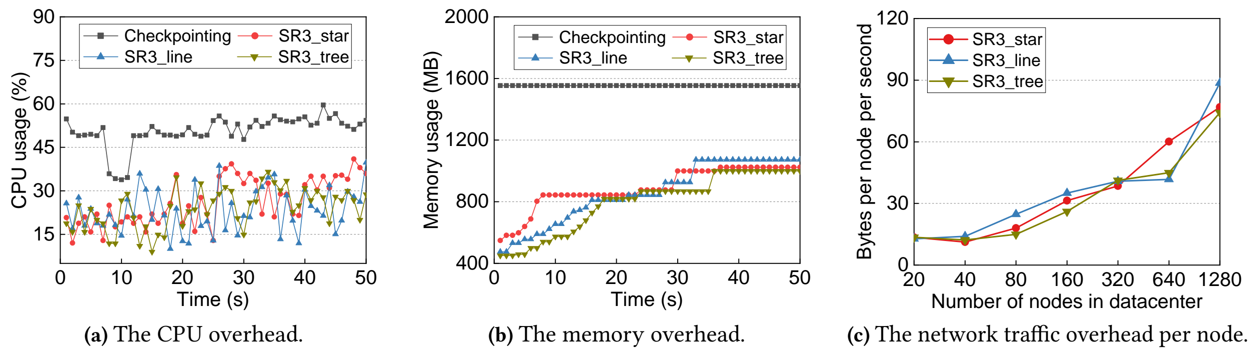
Figure 8c shows the state saving time comparison of SR3 with Storm. The state saving cost includes the time cost for dividing the state into shards, replicating each state, and then writing the shards into leaf set nodes. We write them into the leaf set nodes serially to enable a fair comparison with the checkpointing recovery. We can see that for a small state (<64MB), it takes more time for SR3 to save the state, while for large state (>64MB), it takes less time for SR3 to save the state. This is because, for small state, the overhead of partitioning and replication is not negligible compared to the total time. However, in the case of large state, many nodes in the leaf set take part in the partitioning and replication that balance the workload.

Figure 9a shows the state recovery time by varying star fan-out bit in SR3 star-structured recovery. Results show that the state recovery time does not change much as the star fan-out changes. This is because the depth of the star structure always equals to one and thus the latency is only related to the state size and the transmission speed. However, in extreme cases, e.g., very large state size, increasing fan-out will share the pressure on bandwidth and significantly reduce latency.

Figure 9b shows the state recovery time by varying the path length in the SR3 line-structured recovery. Results show that the state recovery time increases as the path length increases. This is because the longer the path, the more stages of the computation required, and the higher the latency. However, when the state is too large to be finished within one or two stages, we need a longer path that has many stages to distribute the computation.

Figure 9c shows the state recovery time by varying the branch length in SR3 tree-structured recovery. Similar to Figure 9b, given the same state size, the state recovery time increases as the branch length increases. This is because the longer the branch, the more stages of the computation required, and the higher the latency.

Figure 9d shows the state recovery time by varying the tree fan-out in SR3 tree-structured recovery. Note that the tree fan-out n determines the fan-out of each node with 2^n. Given the same state size, when the tree has larger fan-out bit, the depth of the tree will be less and the recovery involves fewer layers, which introduces lower latency for recovering the original state. In addition, larger fan-out trees can tolerate more concurrent node failures or shard loss.



(a) The CPU overhead. (b) The memory overhead. (c) The network traffic overhead per node.

Figure 12. The CPU, memory, and network overhead of SR3.

Therefore, we should choose different tree structures for different applications based on their latency requirements and fault tolerance requirements.

Failure tolerance is evaluated with methods that use human intervention. Multiple simultaneous failures can be easily affected by the location of devices (i.e., in different geo-distributed datacenters) and the type of cloud services [7]. To cause simultaneous failures, we deliberately remove some shards of application's state in some nodes to evaluate how fast SR3 can recover the state. Figure 10a, 10b, and 10c show the average recovery time for different number of simultaneous failures in the star-, line-, and tree-structured recovery with 64MB state, where the replication factors are 2 and 3. In these figures, their two curves show that the recovery time slightly increase with increasing number of shards failures. This is because, when a shard fails, they can quickly retrieve the relevant shards from other nodes which store replicas and rebuild the failed shard. Note that the recovery time with large replication factor (i.e., 3) is lightly less than the small replication factor (i.e., 2). This is because, when one shard fails, a larger replication factor can facilitate retrieval of related shards from other nodes that store the replicated shard. In this way, a larger replication factor can reduce the retrieval time of failed shards. Moreover, in the tree-structured mechanism, when a shard fails, it can quickly retrieve the relevant shards from its leaf set and rebuild the failed shard, and the tree architecture can evenly distribute the recovery overhead for recovering multiple simultaneous failures.

5.3 Load Balance

SR3 has attractive load balance property because it distributes state across all nodes in the overlay, which is especially beneficial when deploying a large number of concurrent applications. We use benchmarks in Table 3 to generate real-world application's state and topology. We evaluate SR3's load balance by deploying 500 stream processing applications and 1,000 stream processing applications on 5,000 Pastry nodes, respectively. We use a mix of benchmark applications' topologies (in Table 3) to replicate 500 to 1000 concurrently running applications. The replication factor is set to be two. The state for each application is 32 MB, and the size for each shard is 512KB. As shown in Figure 11a, each node has around 25 shards (red dash line) when deploying 500 applications. As shown in Figure 11b, each node has around 40 shards (red dash line) when deploying 1,000 applications. This is because the P2P model of SR3's star- line-, and tree-structured recovery ensures that all peers can participate in the state saving process and the state recovery process.

Figure 11c shows the normal probability of the number of shards stored per node. Results show that when deploying 500 applications, around 95% nodes store less than 50 shards (25MB), and around 95% nodes store less than 100 shards (50MB) when deploying 1,000 applications. This demonstrates that the large volume of states from concurrently running applications are almost evenly distributed in the overlay with no centralized bottleneck. This demonstrates that SR3 achieves good load balance when recovering state for large numbers of concurrently running applications.

5.4 Overhead Analysis

We evaluate SR3 runtime overhead and compare them with Storm's checkpointing approach.

CPU overhead. Figure 12a shows the per-node CPU runtime overhead comparison of SR3's three state recovery mechanisms with Storm's checkpointing approach. The CPU overhead of SR3 is around 26.8% ∼ 44.3% less than the checkpointing recovery. This is because SR3 evenly distributes the recovery load across many peer nodes which reduces the per-node overhead, while the checkpointing recovery only relies on one or several centralized nodes for recovery.

Memory overhead. Figure 12b shows the per-node memory runtime overhead comparison of SR3's three state recovery mechanisms with Storm's checkpointing approach. The memory overhead of SR3 is around 30.9% ∼ 35.6% less than the checkpointing recovery. This is because checkpointing recovery involves a coordination service such as Zookeeper that needs to continuously maintain connections with all other nodes while SR3 avoids it.

Network overhead. Figure 12c shows the additional network traffic imposed by SR3 with varying the number of nodes without managing any state (showing purely the maintenance overhead). Results show that the number of bytes sent per node increase only linearly, with an exponential increase in the number of nodes. This is because most network traffics are ping-pong messages used for maintaining the overlay and routing (e.g., initialization and keep alive). So in most cases, each node pings to a limited set of nodes in the leaf set.

6 Conclusion

In this paper we have described and evaluated SR3, a state recovery framework that provides fast and scalable failure recovery mechanisms for protecting large distributed states in stream processing systems. Unlike existing failure recovery approaches in modern stream processing systems, which rely on the central master to perform replication recovery, checkpointing recovery, or DStream-based lineage recovery, SR3 introduces a distributed state recovery framework by leveraging DHT-based consistent ring overlay and routings. SR3 provides three different mechanisms to cater the needs for different stream applications that have diverse computation models and sizes of state.

An interesting question for future work is how to recovery from stragglers. Stragglers are slow nodes. Stragglers are inevitable in large clusters. The root causes for stragglers can be disk failures, CPU contention, memory pressure, network congestion, or other internal factors such as unfair input partitioning. Left unchecked, stragglers will cause serious problems such as state inconsistency. We plan to explore speculation approach to address this challenge, in which speculative backup copies of slow tasks could be run in DHT's leaf set nodes.

We will release SR3 as open source, together with all code and data used to produce the results in this paper.

Acknowledgment

We sincerely thank our shepherd Prof. Vincenzo Gulisano and the anonymous reviewers for their insightful suggestions and comments that greatly improved this paper. This work is supported by the National Science Foundation (NSF-SPX-1919126, NSF-SPX-1919181, NSF-CAREER-1943071, NSF-CCF-1934904).

References

[1] Shrinath Perera. *A Gentle Introduction to Stream Processing*. https://medium.com/stream-processing/what-is-stream-processing-1eadfca11b97. 2018.
[2] Tingmin Wu et al. "Twitter spam detection: Survey of new approaches and comparative study". In: *Computers & Security* 76 (2018), pp. 265–284.
[3] Roberto Mora Cortez and Wesley J Johnston. "The future of B2B marketing theory: A historical and prospective analysis". In: *Industrial Marketing Management* 66 (2017), pp. 90–102.
[4] Amy McGovern et al. "Using artificial intelligence to improve real-time decision-making for high-impact weather". In: *Bulletin of the American Meteorological Society* (2017).
[5] *Bloom filter*. https://en.wikipedia.org/wiki/Bloom_filter.
[6] Haoyu Wang, Haiying Shen, and Zhuozhao Li. "Approaches for resilience against cascading failures in cloud datacenters". In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2018, pp. 706–717.
[7] Erci Xu et al. "Lessons and actions: What we learned from 10k ssd-related storage system failures". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 961–976.
[8] *Apache Flink*. http://flink.apache.org/.
[9] Paris Carbone et al. "State management in Apache Flink®: consistent stateful distributed stream processing". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1718–1729.
[10] Matei Zaharia et al. "Discretized streams: Fault-tolerant streaming computation at scale". In: *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM. 2013, pp. 423–438.
[11] Prateek Sharma et al. "Flint: Batch-interactive data-intensive processing on transient servers". In: *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 6.
[12] *Apache Storm*. http://storm.apache.org/.
[13] *Apache Trident*. http://storm.apache.org/releases/current/Trident-tutorial.html.
[14] Sanjeev Kulkarni et al. "Twitter heron: Stream processing at scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 239–250.
[15] *Spark Streaming*. https://spark.apache.org/streaming/.
[16] Daniel J Abadi et al. "The design of the borealis stream processing engine." In: *Cidr*. Vol. 5. 2005, pp. 277–289.
[17] Zhengping Qian et al. "Timestream: Reliable stream computation in the cloud". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 1–14.
[18] Leonardo Neumeyer et al. "S4: Distributed stream computing platform". In: *2010 IEEE International Conference on Data Mining Workshops*. IEEE. 2010, pp. 170–177.
[19] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. "Highly available, fault-tolerant, parallel dataflows". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM. 2004, pp. 827–838.
[20] Magdalena Balazinska et al. "Fault-tolerance in the Borealis distributed stream processing system". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, pp. 13–24.
[21] Derek G Murray et al. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 439–455.
[22] Arvind Arasu et al. "Stream: The stanford data stream management system". In: *Data Stream Management*. Springer, 2016, pp. 317–336.
[23] Daniel Peng and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications". In: (2010).
[24] Mayank Pundir et al. "Zorro: Zero-cost reactive failure recovery in distributed graph processing". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM. 2015, pp. 195–208.
[25] Joseph E Gonzalez et al. "Graphx: Graph processing in a distributed dataflow framework". In: *11th Symposium on Operating Systems Design and Implementation*. 2014, pp. 599–613.
[26] Wang Lam et al. "Muppet: MapReduce-style processing of fast data". In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1814–1825.
[27] Tyler Akidau et al. "MillWheel: fault-tolerant stream processing at internet scale". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.
[28] Tyler Akidau et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded,

out-of-order data processing". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803.
[29] *Apache Kafka*. http://kafka.apache.org/.
[30] Shadi A Noghabi et al. "Samza: stateful scalable stream processing at LinkedIn". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1634–1645.
[31] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.
[32] Sirish Chandrasekaran et al. "Telegraphcq: Continuous dataflow processing for an Uncertain world." In: *Cidr*. Vol. 2. 2003, p. 4.
[33] Daniel J Abadi et al. "Aurora: a new model and architecture for data stream management". In: *the VLDB Journal* 12.2 (2003), pp. 120–139.
[34] *Apache Samza*. http://samza.apache.org/.
[35] *RocksDB*. http://rocksdb.org/.
[36] *LevelDB*. https://github.com/google/leveldb/.
[37] Siying Dong et al. "Optimizing Space Amplification in RocksDB." In: *CIDR*. Vol. 3. 2017, p. 3.
[38] Shivaram Venkataraman et al. "Drizzle: Fast and adaptable stream processing at scale". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 374–389.
[39] Adam Moody et al. "Design, modeling, and evaluation of a scalable multi-level checkpointing system". In: *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–11.
[40] Pinchao Liu et al. "FP4S: Fragment-based Parallel State Recovery for Stateful Stream Applications". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 1102–1111.
[41] Irving S Reed and Gustave Solomon. "Polynomial codes over certain finite fields". In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pp. 300–304.
[42] Tyler Akidau et al. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1792–1803. ISSN: 2150-8097.
[43] Paris Carbone et al. "Apache Flink: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.
[44] Wei Lin et al. "STREAMSCOPE: Continuous Reliable Distributed Processing of Big Data Streams". In: *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 439–453. ISBN: 978-1-931971-29-4.
[45] Michael Isard et al. "Dryad: distributed data-parallel programs from sequential building blocks". In: *ACM SIGOPS operating systems review*. Vol. 41. ACM. 2007, pp. 59–72.
[46] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350.
[47] Ion Stoica et al. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.
[48] Bram Cohen. "Incentives build robustness in BitTorrent". In: *Workshop on Economics of Peer-to-Peer systems*. Vol. 6. 2003, pp. 68–72.
[49] Miguel Castro et al. "SCRIBE: A large-scale and decentralized application-level multicast infrastructure". In: *IEEE Journal on Selected Areas in communications* 20.8 (2002), pp. 1489–1499.
[50] *Apache Storm 2.0.0*. https://storm.apache.org/2019/05/30/storm200-released.html.
[51] *Pastry*. https://www.freepastry.org/FreePastry/.
[52] Yunhao Zhang, Rong Chen, and Haibo Chen. "Sub-millisecond stateful stream querying over fast-evolving linked data". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 614–630.
[53] *cassandra*. https://cassandra.apache.org.
[54] *Twitter streaming APIs*. https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data.
[55] *Google Finance Data API*. http://finance.google.com/finance/feeds/.
[56] *Wikimedia Dumps*. https://dumps.wikimedia.org/.
[57] *Dublin Bus GPS sample data from Dublin City Council*. https://data.gov.ie/dataset/.