E-Storm: Replication-based State Management in Distributed Stream Processing Systems

Xunyun Liu, Aaron Harwood, Shanika Karunasekera, Benjamin Rubinstein and Rajkumar Buyya
Cloud Computing and **D**istributed **S**ystems (CLOUDS) Lab
School of Computing and Information Systems
The University of Melbourne, Australia
Email: xunyunl@student.unimelb.edu.au, {aharwood, karus, brubinstein, rbuyya}@unimelb.edu.au

***Abstract*—Apache Storm is a fault-tolerant, distributed in-memory computation system for processing large volumes of high-velocity data in real-time. As an integral part of the fault-tolerance mechanism, Storm's state management is achieved by a checkpointing framework, which commits states regularly and recovers lost states from the latest checkpoint. However, this method involves a remote data store for state preservation and access, resulting in significant overheads to the performance of error-free execution.**

In this paper, we propose E-Storm, a replication-based state management system that actively maintains multiple state backups on different worker nodes. We build a prototype on top of Storm by extending it with monitoring and recovery modules to support inter-task state transfer whenever needed. The experiments carried out on synthetic and real-world streaming applications confirm that E-Storm outperforms the existing checkpointing method in terms of the resulting application performance, obtaining as much as 9.44 times throughput improvement while reducing the application latency down to 9.8%.

I. Introduction

Stream processing systems ingest data continuously and concurrently in memory, performing computations on a record-by-record basis. Apache Storm, for example, is a distributed Data Stream Management System (DSMS) designed to process unbounded streams of data in real-time. As a middleware bridging the gap between applications and resources, it provides improved programmability to developers through the abstraction of processing primitives and the simplification of stream routing models. It also enables horizontal scalability on distributed infrastructure, allowing for the adjustment of computation scale at runtime without exposing low-level implementation details to developers. Besides, Storm greatly enhances the system manageability for operations staff: runtime controllability is achieved through an interactive interface, while the reliability and fault-tolerance issues faced by the up-level applications are automatically handled by Storm itself. These remarkable features make Storm an ideal host for running continuous streaming logic, and many consider it as the counterpart of Hadoop in the real-time computation field.

There are three fault-tolerance mechanisms built in Storm that enable reliable stream processing, namely: (1) Supervised and stateless daemon execution, which allows the failed Storm daemons to be restarted, resuming their stateless execution under the supervision of an external process monitoring tool; (2) Message delivery guarantee, which ensures the consistency of processing semantics by using a subtle anchoring and acknowledgement algorithm; and (3) State persistence, which persists the computation states to somewhere in order to mask the loss of states caused by JVM or node crashes. This paper proposes a novel state management framework to better achieve this goal.

Since version 1.0.0, Storm's core has abstractions for stream operators to save and retrieve states against a persistent state store. However, the current state persistence technique introduces significant overhead to error-free execution. From the implementation's perspective, state persistence is now achieved through checkpointing, where a remote data store is constantly involved in all state accesses. Specifically, there is an internal data source that initializes a checkpoint transaction by sending signals across the streaming application. Upon receiving the checkpoint signal, stateful operators prepare and preserve their intermediate states to a Redis[1] store, and then empty the in-memory cache to commit the transaction. The frequency of checkpointing is defaulted to every second, which brings significant state synchronization overhead; while setting the checkpoint interval too large would risk losing state between checkpointing and being unable to replay failed messages. Secondly, the use of any committed state resorts to the remote data store, which imposes non-trivial data access delay for latency-sensitive streaming applications. Lastly, there could be a massive amount of operators accessing the remote data store simultaneously for state retrieval or check-pointing, which makes the store a potential performance bottleneck to application throughput.

In this paper, we propose E-Storm, a light-weight, replication-based state management framework in Storm that eliminates the use of remote data store during error-free execution. To ensure state persistence in the case of failures, our framework automatically maintains live state replicas on different nodes of Storm and transfers state when needed. The number of replicas can be customized with regard to the user's needs, but in general a stateful operator with k replicas is able to tolerate the failure of any $k-1$ worker nodes.

The main **contributions** of this work are as follows:

- We propose a replication-based state management framework for achieving state persistence in the case of fail-

[1] https://redis.io/

ures, which exposes a concise fluent-style interface and works transparently to the upper-level logic.
- We design a failure recovery protocol that guarantees application integrity when failover occurs. The recovery operates at the lowest thread level and is seamlessly integrated to Storm's execution flow. The replication of state is also autonomous and high-performance, which allows multiple transfers to occur concurrently.
- We implement the framework and conduct extensive experiments to demonstrate the superiority of our approach compared to the existing check-pointing method, which reaches as much as 9.44 times throughput boosts and 90.2% latency reduction.

Our implementation of E-Storm is loosely coupled with the existing Storm modules, and externally configurable to provide different levels of state resilience in different use cases. Such implementation design makes it viable to be generalized to other operator-based stream processing systems.

II. Background

In recent years, Apache Storm emerged as a new generation of data stream management system for tackling many real-time use cases such as on-line machine learning, continuous computation and Distributed Remote Procedure Call (DRPC). The scalable, fault-tolerant, and language-agnostic design of Storm offers seamless integration with the mainstream queueing and database technologies, making it much easier to process unbounded fast data on a set of distributed resources.

From the structure point of view, a Storm cluster much resembles Hadoop — its counterpart in batch processing. It also includes a master node and several worker nodes: the master node has a *nimbus* daemon that is responsible for monitoring the cluster and distributing workload; while the worker node hosts the *worker processes* to carry out the streaming logic in JVMs. Additionally, there is a *supervisor* daemon that communicates with the nimbus and constantly governs the worker processes during runtime. The whole Storm cluster relies on *Zookeeper* — a distributed hierarchical key-value store to coordinate and failover.

In Storm's terminology, a *tuple* is an ordered list of key-value pairs (each pair is referred to as a *field*) and a *stream* is an unbounded sequence of tuples. From the logical perspective, the workflow of a streaming application is represented by the *topology* — a Directed Acyclic Graph (DAG) of *operators* standing on streams. Among operators, the sources of streams are called *spouts* that pull stream data to the topology, while the others are referred to as *bolts* that can either generate new streams based on inputs or simply consume data without emission. Different bolts may contain various user-defined processing logics such as functions, filtering, and aggregations.

From the viewpoint of execution, an operator is distributed across the Storm cluster as one or more tasks, the process of which is called *operator parallelization*. Each task is an operator instance that handles a portion of the operator input with the same streaming logic, so Storm makes full use of distributed resources by distributing tasks to different worker nodes. Also, as a consequence of parallelization, each incoming stream is accompanied by a *grouping policy* that determines how tuples are routed among the receipt tasks. When a streaming application is submitted to the cluster, the worker process will spawn *executors* — the minimal schedulable entity of Storm — to wrap the execution of tasks. Note that each executor is a thread that may run one or more tasks for the same component (spout or bolt), Therefore, internally tasks have to run in sequence.

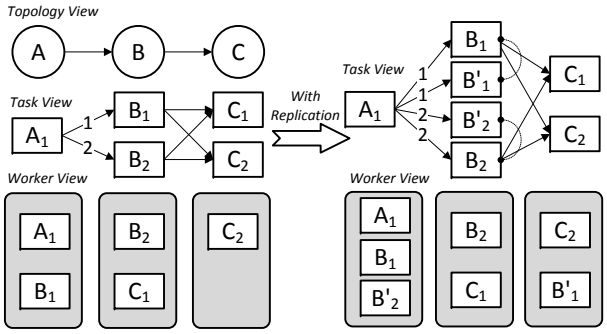


Fig. 1. The execution of an example streaming application on Storm, with or without state replication.

Most of the streaming applications involve stateful operators that accumulate states such as window-grouped tuples or aggregation results during runtime. Therefore, it is crucial to ensure the integrity of operator states in the case of failures. From the execution point of view, the parallel execution of stateful operator requires each task to maintain a unique partition of the internal state. All the internal states are temporally stored in memory and are thus subject to JVM failure. Currently, the only way to achieve state persistence is through regularly checkpointing them to a remote Redis store. Storm has a built-in checkpoint mechanism which implements a three-phase commit protocol on top of the existing message delivery system, ensuring that the states of different tasks would be saved in a consistent and atomic manner. However, as we have explained in Section I, such implementation introduces non-trivial overhead to the error-free execution of streaming applications.

III. Framework Overview

In order to maintain multiple state backups independently, our state management framework duplicates the execution of stateful tasks on different worker nodes. Fig. 1 uses an example application to illustrate the changes we have made to the Storm execution model. There are three linearly connected operators in the example application: Op_A is the spout, Op_B is a stateful operator, and Op_C is a stateless operator. Both Op_B and Op_C are parallelized into two tasks for distributed execution:

1) Having set the number of replicas for Op_B to 2, the framework spawns two *shadow tasks* $T_{B_1'}$ and $T_{B_2'}$ to mirror the execution of the *primary task* T_{B_1} and T_{B_2}, respectively. Tasks sharing the same state make up a *task*

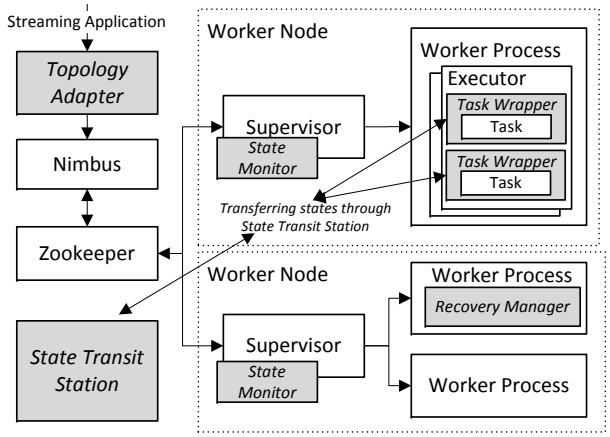


Fig. 2. The extended Storm architecture with the state management framework, where the newly introduced modules are highlighted in grey.

fleet, which are exclusively placed on different worker nodes for independent execution.

2) Any input stream sent to the primary task is copied to its shadow counterparts. However, shadow tasks have no output stream as they only serve as state containers.
3) In the case of failures, the restarted tasks recover their lost states from the alive partners of the same fleet.

We have extended Storm with several modules to implement these changes. These include the Topology Adapter, the State Monitor, the Task Wrapper, the Recovery Manager and the State Transit Station, which are highlighted in grey in Fig. 2.

The *Topology Adapter* is written in Storm core to help alleviate the adaptation effort on the application level. Developers can define the number of replicas using a fluent-style replication API, just like how they specify the number of tasks for operators. The adapter is also in charge of re-grouping streams for stateful operators and initializing other modules for state management. When the application is submitted to Nimbus, this module ensures that the shadow tasks are transparently set up across the Storm cluster.

The *State Monitor*, located alongside the supervisor daemon, is responsible for monitoring the health of states residing in this worker node. Once a state issue is detected, it will send a recovery request to the Recovery Manager through Zookeeper. The State Monitor itself is stateless and fail-fast, with execution placed under constant supervision.

The *Recovery Manager* is an internal operator that initializes, oversees and finalises the recovery process. It implements the Zookeeper watcher interface to monitor recovery requests, then exploiting the Storm's acknowledgement system to ensure the consistency of recovery. Being a stateless operator, its fault-tolerance is guaranteed by Storm to survive from node and JVM crashes.

The *Task Wrapper* encapsulates the task execution with the logic to handle state transfer and recovery. There is also the *State Transit Station* that decouples the senders and receivers during the state transferring process. By directing all the state transfers to the station, task wrappers perform state

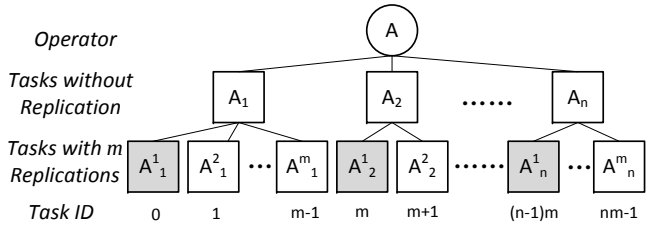


Fig. 3. The role of a task is decided based on the index of its task ID. The primary tasks are coloured in grey.

management without synchronization and leader selection, which would have been necessary in a peer-to-peer style recovery and introduce non-negligible overhead.

The state management framework has two different working modes, namely error-free execution and failure recovery. In the following sections, we discuss in detail how these extended modules are implemented to achieve state persistence against JVM and node crashes.

IV. Error-free Execution

When a streaming application is submitted, the topology adapter is responsible for deciding the task roles in execution (primary or shadow). It is also in charge of rewiring the task communication for message replication and placing the tasks of the same fleet on different machines for failure-independence.

The role of a task is statically decided based on its task ID — the primary task is the one with the lowest ID in a fleet. As shown in Fig. 3, Op_A is a stateful operator that is initially parallelized as n tasks. After users set the number of replicas to m, the topology adapter transparently multiplies the number of tasks to n times m and composes each m tasks as a task fleet. Therefore, $T_{A_1^1}, T_{A_2^1} ... T_{A_n^1}$ are set as primary tasks and each one of them is accompanied by $m-1$ shadow tasks that are adjacent in ID. Note that the role of these tasks will not change throughout the application lifecycle.

In order to replicate states across the task fleet, the contained tasks must receive the same inputs for processing. To this end, the topology adapter replaces the original grouping that connected to the stateful bolt with a custom, replication-aware stream grouping method, which replicates the tuples in transmission transparently at the message channel.

Take the *fields grouping* — the most common grouping type in stateful computation — as an example. It routes a particular tuple $Tuple$ to its target task T_{target} according to the following equation: $T_{target} = hash(Tuple.fields)\%n$, where n is the number of tasks and *hash* is a concatenating function on the hash codes of the selected grouping fields. When the fields grouping is replaced with the replication-aware fields grouping, the message channel computes a list of m target tasks rather than a single one, which is formulated as $T_{targets} = \{hash(Tuple.fields)\%n*m+i | (i = 0, .., m-1)\}$.

Algorithm 1: The replication-aware task placement algorithm

Input: A Storm cluster with n_m nodes and a topology Γ
Input: A task set $\vec{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ to be assigned
Output: A node set $\vec{m} = \{m_1, m_2, \ldots, m_{n_m}\}$ with each node hosting a disjoint subset of $\vec{\tau}$

1 $A_{m_i} \leftarrow \lceil \frac{n}{n_m} \rceil \ (i = 1, 2, ...n_m)$
2 $Q_{op} \leftarrow BFSTraversal(\Gamma, spout)$
3 $\vec{\tau}_{ordered} \leftarrow \varnothing$
4 **while** $\vec{\tau}_{ordered}$ *does not contain all the tasks in* $\vec{\tau}$ **do**
5 **foreach** *Operator* $op \in Q_{op}$ **do**
6 **if** *op has an unvisited shadow task* τ_i **then**
7 $\vec{\tau}_{ordered}.append(\tau_i)$
8 $op.remove(\tau_i)$
9 **foreach** *Task* $\tau_i \in \vec{\tau}_{ordered}$ **do**
10 **foreach** *node* $m_j \in \vec{m}$ *that has* $A_{m_j} > 0$ **do**
11 **if** m_j *has no conflicting tasks to* τ_i **then**
12 $I_{m_j} \leftarrow$ the increase of the intra-node communication pairs if τ_i were put onto m_j
13 Place τ_i to node m_j with the largested I_{m_j}
14 $A_{m_j} \leftarrow A_{m_j} - 1$
15 **return** $\vec{m}$

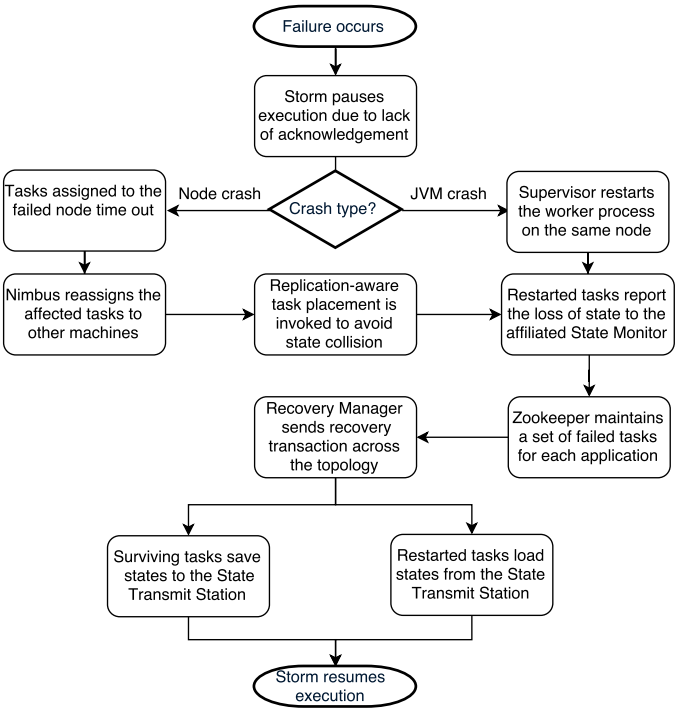


Fig. 4. The flowchart of the recovery process, which is seamlessly integrated with the Storm's error-handling logic and leverages the acknowledgement system to pause and resume the execution flow.

A. Replication-aware Task Placement

Essentially, the task placement problem is a bin-packing variant that takes tasks as items and worker nodes as bins, while the optimization target is to reduce the number of inter-node communication pairs for improving application performance. Besides, our problem has a hard constraint that tasks from the same fleet are not to be put on the same worker node.

The task placement problem itself is NP-Hard since it can be reduced to the PARTITION problem [1]. However, it is feasible to find a sub-optimal solution by using efficient heuristic methods. We therefore propose a replication-aware task placement algorithm based on the greedy heuristic, with the following desirable features in its design:

- It is only responsible for placing shadow tasks to worker nodes; while the placement of other tasks are left for the user-given task scheduling algorithm to decide. Such a design allows for the use of various existing scheduling algorithms that optimize towards different targets, such as throughput, latency, resource-awareness, etc.
- The shadow tasks are spread as far as possible across the cluster, so the overhead of replication is balanced and the effort of state recovery is minimized in the case of failures.
- The algorithm makes use of the topology structure to place communicating tasks as close as possible.

Algorithm 1 depicts the pseudo-code for the replication-aware task placement. It first calculates A_{m_i}, the capacity of each node, by enforcing the shadow tasks to spread out across the cluster. Then the standard Breadth-first traversal procedure is applied to the topology structure, yielding an operator queue Q_{op} which is a partial ordering of operators with the communicating pairs placed in succession.

Lines 3-8 of the algorithm describe the procedure to generate an ordered list of tasks $\vec{\tau}_{ordered}$ based on Q_{op}. For each operator being traversed, the algorithm takes out one shadow task at a time and appends it to the ordering list. This process continues until all the tasks to be placed are ordered, which ensures that, in the later placement phase, the communicating tasks have better chance to be placed in close vicinity.

The rest of the algorithm determines the exact node where a particular task would be placed. As shown in line 13, the greedy heuristic chooses the one that is capable of turning more communications into intra-node message passing. If there is a tie between multiple alternatives, the one with the highest remaining capacity will be selected.

V. Failure Recovery

The recovery phase is triggered when any worker crashes during runtime. Fig. 4 briefly illustrates the work flow of recovery after the failure occurs.

In general, Storm automatically pauses the application execution due to the lack of tuple acknowledgement. Through the heartbeat mechanism between worker processes and Storm daemons, the failed tasks will be transparently restarted with the same task ID, but possibly placed on different worker nodes depending on the type of failure. Therefore, it is required that the replication-aware task placement is invoked to avoid two tasks from the same fleet being collocated. During the

Algorithm 2: The state operation logic encapsulated in the StateManipulator

Input: A recovery transaction tx with ID $tx.id$ and the set of tasks that have lost their state $tx.set$
Input: An initilization flag $isInit$ that indicates if s, the state of the wrapped task, has been initilized
Input: A CuratorFramework client c_f that opeartes on the Zookeeper
Input: The task fleet t_f that the wrapped task belongs to

```
1  if isInit == True then
2      c_f initilizes a shared inter-process lock on t_f
3      if c_f.acqureLock(t_f) and s ∈ tx.set then
4          if s is not on the State Transition Station then
5              Save s to the State Transition Station
6          c_f.releaseLock(t_f)
7  else
8      while s is not on the State Transition Station do
9          Sleep a while, recheck until recovery times out
10     if s exist in the State Transition Station then
11         Read s from the State Transition Station and
             assign it to the wrapped task
12         Process the pending tuples that received before s
             is initilized
13         isInit = True
14     else
15         Return with a recovery failure flag
16 Emit the recovery transaction tx to downstream
17 Acknowledge the recovery transaction tx
18 return
```

preparation process, these restarted tasks report the loss of state to the state monitor, which initializes a recovery transaction on a dedicated Zookeeper node, recording a transaction ID as well as the set of tasks being affected. The recovery manager that constantly monitors the Zookeeper would make sure that all the affected tasks get initialized with its previous states through the failure recovery process.

The recovery manager is implemented as an internal spout, which is automatically added by the topology adapter if there is at least one stateful bolt and the state replication is turned on. The adapter also connects the recovery manager with other operators through a separate internal stream, allowing it to send recovery signal across the topology for starting and supervising the failure recovery process. Once the recovery manager receives acknowledgement from all the downstream operators, the state recovery is complete and the streaming application can resume execution from the point it left off.

As mentioned in Section III, the task wrapper encapsulates the state transfer and recovery logic, making the state management mechanism autonomous and transparent to its wrapped task. There are two different types of wrappers in our framework, encompassing stateless and stateful tasks, respectively. The wrapper for stateless tasks is called *SignalForwarder*, whose only duty is to forward the signal tuple to all its downstream tasks; while the *StateManipulator* for stateful tasks not only handles the state management on receiving the recovery transaction, but also relays the received signal for it to be broadcast across the topology DAG.

Specifically, Algorithm 2 illustrates the pseudo-code of state operations in the StateManipulator. Lines 1-6 of the algorithm are executed by the statful tasks that are not affected by the failure. Considering that there could be multiple tasks alive in the same fleet and they all attempt to preserve states without prior-synchronization, our algorithm takes advantage of the inter-process lock on Zookeeper, ensuring that there is only one task in each crash-affected fleet communicating to the state transmit station, which greatly reduces the network flow during the state transfer process.

Lines 7-15 of the algorithm describe the state recovery logic for restarted tasks. Once the recovery signal is received, tasks that are initialized from scratch start querying the state transmit station for accessing their lost state. However, the corresponding state preservation process may not be complete by the time they restarted, so these tasks need to repeat the retrieval attempts until the recovery times out. Besides, any tuple that is received before the state initialization is added to a pending list to delay its execution.

Due to the limitation of the acknowledgement system in Storm's core, our failure recovery logic cannot eliminate duplicate tuple evaluation and provide only at-least-once message processing guarantee. However, it is possible to achieve exactly-once semantics with the Trident abstraction, where the idea of replication still applies for state persistence.

VI. Performance Evaluation

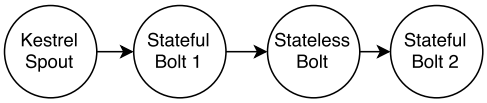
In this section, we explore in detail the performance of our prototype (E-Storm) compared to the existing checkpointing method, by applying them to both synthetic and real-world streaming applications. The design of evaluation answers the following questions:

- What are the runtime overheads for enabling state persistence and how do these overheads vary in different use cases? (Section VI-B)
- How does the resilience level, i.e. the number of state replications, affect the performance of E-Storm? (Section VI-B)
- How long does it take for E-Storm to recover a streaming application from JVM crashes. (Section VI-C)

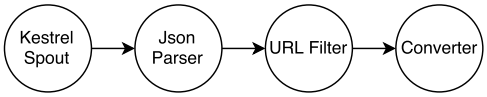
A. Experiment Setup

Our experiments are conducted on Storm v1.0.2 using the Nectar IaaS Cloud[2]. The Storm cluster consists of 10 worker nodes and 2 administrative nodes for Nimbus and Zookeeper, respectively. Apart from that, there is also (1) a Kestrel[3] node that caches inputs for streaming applications when the

[2]https://nectar.org.au/research-cloud/
[3]https://github.com/twitter-archive/kestrel



(a) The topology of the synthetic test application



(b) The topology of the real-world test application

Fig. 5. The illustration of the test application topologies. In Fig. 5a, Stateful Bolt 1 and State Bolt 2 have the same implementation.

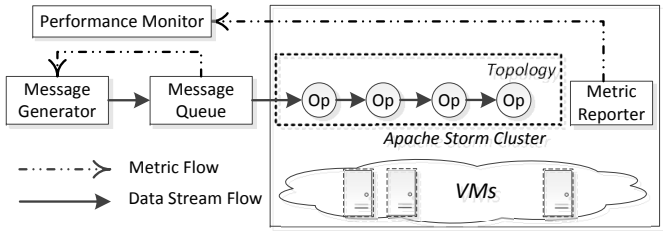


Fig. 6. The profiling environment set for the performance evaluation. Solid connectors represent the generated data stream flow, while the dashed connectors denote the flow of performance metrics.

processing capability of the cluster cannot catch up with the speed of data generation; and (2) a Redis node that works as the remote data store in the checkpointing method and the state transmit station in E-Storm. The above-mentioned nodes are all of "m2.medium" size, provisioned from the same NCI availability zone and equipped with 2 VCPUs, 6GB RAM and 30GB root disk.

1) Test Applications: Our first test application is synthetically designed to mimic different intensities of state usages. As shown in Fig. 5a, it consists of four operators ($Op_1, ..., Op_4$): Op_1 is the KestrelSpout that pulls input data from the Kestrel queue server, while Op_2 and Op_4 are two stateful operators that are connected through the stateless operator Op_3. Table I illustrates the application configurations that adjust the size of internal states and the way of accessing them.

TABLE I
THE CONFIGURATION OF THE SYNTHETIC TEST APPLICATION

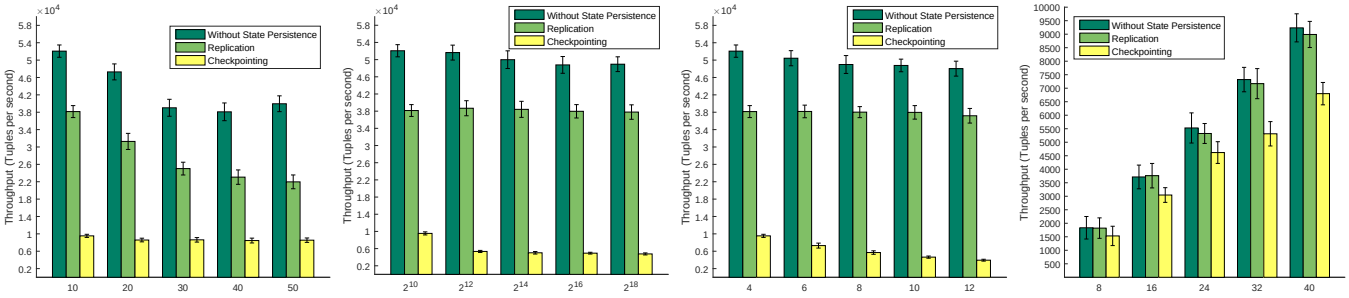
Symbol	Configuration Description
N_s	The number of stateful tasks in this topology
E_s	The size of states being kept in each stateful task
F_s	The number of state access in the execute method

In particular, N_s denotes the number of stateful tasks in total, where Op_2 and Op_4 equally get $N_s/2$ tasks for parallel execution; whereas the parallelism degree of Op_3 is fixed at 10, as it is sufficiently large to ensure the stateless operator will not be the bottleneck of the topology. In terms of the streaming logic, each stateful task maintains a key-value map and continuously fills it with the recently received data. We externally cap the maximum number of map entries at E_s, which essentially determines the size of the internal state to be kept in this particular task. Lastly, F_s denotes the number of state access operations encapsulated in the *execute* method, which effectively determines the frequency of state access for processing a single tuple.

The second application is drawn from a real-world use case — extracting short Uniform Resource Locators (URLs) from incoming tweets and replacing them with complete URL addresses. As depicted in Fig. 5b, the whole application also consists of 4 operators: the KestrelSpout as used in the synthetic application, the JsonParser bolt that parses the tweet string and extracts the main body from the JSON content, the URLFilter that isolates and filters short URLs from the message body, and the Converter that actually performs the conversion. Among them, the Converter is a stateful operator caching the map of short and complete URLs in its memory, so trending pages can be identified from the map statistics and it does not need to check the remote database whenever an input tuple is received. As for configuration, this application has only one parameter to be set, i.e. the number of tasks that are evenly distributed among all these four operators.

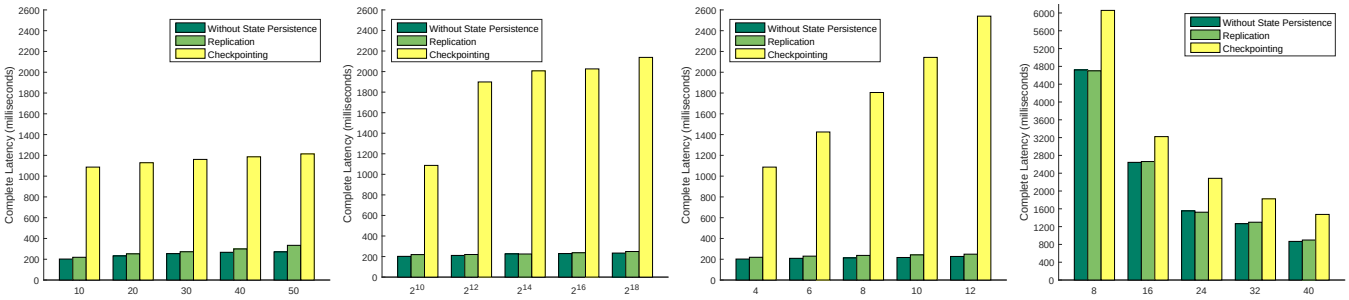
2) Evaluation Methodology: We examine the application performance in two major metrics, namely throughput and complete latency. The application throughput is obtained externally by observing the number of acknowledgements per unit of time, while the complete latency is a built-in Storm metric, which calculates the average time taken by a tuple and all its offspring to be completely processed by the topology.

In order to evaluate the performance overhead brought by different approaches of state persistence, we have set up a profiling environment that feeds the streaming application with sufficient inputs and continuously monitors the resulting performance. The components of the profiling environment are briefly depicted in Fig. 6. The Message Generator is a Java program that reads the workload file on-demand to emit a particular size of profiling stream, and the workload file contains 899,560 tweets in JSON format that collected from 24/03/2014 to 14/04/2014. Running on the kestrel node, the Message Queue module is built with Twitter Kestrel, which exposes a Thrift interface for the message generator to retrieve the length of the message queue and further determine whether the streaming application has been overwhelmed by the profiling data. The application metrics, such as throughput and latency, are externally collected by the performance monitor which is implemented as a RESTful client. With ample profiling inputs, the Storm cluster will be pushed to its performance limit, i.e. exhibiting the highest throughput, after the application is stabilized. For all the test applications, we also set the Storm configuration *MaxSpoutPending* to 10000, which is the maximum number of unacknowledged tuples that can be pending on a spout task at any given time. Therefore, such environment setting makes it possible to compare the performance across different test applications.



(a) Varying N_s (Sythetic App) (b) Varying E_s (Sythetic App) (c) Varying F_s (Sythetic App) (d) Varying number of tasks (Real App)

Fig. 7. The application throughput under different state persistence methods. Each result bar is an average of 10 consecutive throughput readings collected every 60 seconds, with the standard deviation plotted in the error bar.



(a) Varying N_s (Sythetic App) (b) Varying E_s (Sythetic App) (c) Varying F_s (Sythetic App) (d) Varying number of tasks (Real App)

Fig. 8. The application latency under different state persistence methods. Each result is an average of statistics collected in a time window of 10 minutes, and the error bar is omitted as the standard deviation of latency is negligible for stabilized applications

B. Performance of Error-free Execution

1) Overhead of State Persistence: In this section, we first examine the performance overhead brought by state persistence, as well as how it varies under different application behaviours when the configurations of state have been altered. Table III describes the evaluated and default values for each application parameter. When a particular parameter is being examined, the others were set to their default values.

TABLE II
EVALUATED PARAMETERS AND THEIR VALUES (DEFAULT VALUES ARE SHOWED IN BOLD).

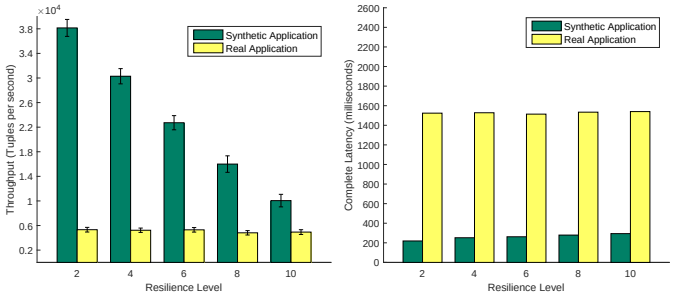
Parameters	Values
N_s (synthetic application)	**10**, 20, 30, 40, 50
E_s (synthetic application)	$\mathbf{2^{10}}$, 2^{12}, 2^{14}, 2^{16}, 2^{18}
F_s (synthetic application)	**4**, 6, 8, 10, 12
Number of tasks (real application)	**8**, 16, 24, 32, 40
Number of state replications	**2**, 4, 6, 8, 10

As shown in Fig. 7 and Fig. 8, the results obtained from the synthetic application clearly demonstrate that enabling checkpoint for state persistence leads to significant performance degradation. Under the default configuration, checkpointing yields 18.3% throughput and 5.38 times complete latency, compared to the baseline case with no state management. As a matter of fact, the acknowledgement of processed tuples have to be delayed until the internal state has been committed to the remote data store, therefore, it is not possible for the checkpointing method to reduce the complete latency below the pre-designated checkpoint interval, which is default to 1 second for performance consideration.

Furthermore, by altering the application configuration, we can identify and measure the factors that contribute to the checkpointing overhead, namely periodic synchronization and state access. When the size of state is increased from 2^{10} to 2^{18}, the application throughput drops to about 49.9% while the complete latency soars to 196.7%, indicating that larger state involves more state updates and thus imposing significant overhead for the remote data store to synchronize. However, this overhead does not increase linearly along with the size of state as the checkpointing method actually adopts the strategy of incremental update for synchronization.

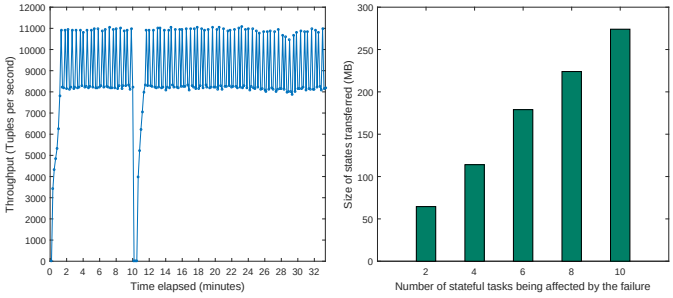
However, such update strategy also brings non-negligible network delay for state access. After F_s varies from 4 to 12, the checkpointing method suffered from 59% throughput loss and 233.7% latency increase, and the performance degradation almost changes linearly in regard to the variation of F_s.

The replication-based state persistence, by contrast, shows promising performance against checkpointing. To start with, the replication method exhibited steady throughput and latency when varying E_s and F_s, i.e. the size of state and the frequency of state access. In the worst case, it accounts for 74.9% of throughput and introduces only 11.5% of latency compared to the non-persistent baseline. The rationale behind these results



(a) Throughput changes (b) Latency changes

Fig. 9. The application performance under different resilience levels, i.e. number of replicas for stateful tasks. The throughput and latency notations have the same meaning as explained in Fig. 7 and Fig. 8.



(a) Monitored application throughput (b) Size of transferred states

Fig. 10. The recovery test performed on the real-world streaming application.

is that our approach manages the internal states in memory resembling the way how baseline works, and the performance is unlikely to be bottlenecked by the memory access speed.

However, as expected, it has been identified that the overhead of state replication climbs as the number of stateful tasks increases. To put it quantitatively, after adjusting N_s from 10 to 50, the throughput of replication reduces from 73.3% to 55% and the complete latency rises from 108.4% to 123.2%, with all figures obtained from the comparison to the baseline in which no state persistence is provided. Our analysis deems such performance degradation as the result of bandwidth contention. As there are more shadow tasks to be spawned at different nodes and their inputs to be replicated at the message channel, our approach causes additional bandwidth consumption and impairs the maximum performance. However, this overhead does not increase super-linearly with the number of stateful tasks, so we argue that the proposed method is still applicable to production-scale applications for state persistence.

2) Overhead of Maintaining More Replicas: To investigate how the number of replicas affects the application performance, we deployed the two test applications to the testbed: the synthetic application uses its default configuration, while the real application sets the number of tasks to 24 for better demonstration. The evaluation results are illustrated in Fig. 9. For the synthetic application whose performance is bounded by the inter-node bandwidth, the resulting throughput dramatically decreases to roughly 26.4% of the highest point as the number of replicas increases to 10, while the complete latency experiences a slight increase from 219 ms to 294 ms during this variation. On the other hand, introducing more state replicas to the real application has not produced noticeable performance degradation, which can be explained by the fact that the whole application is actually bounded by the lack of tasks to process the incoming stream in parallel, rather than the duplication of messages at the communication channel. We also observed that tasks of the real application spend most of their time executing tuples, contrasting to the tasks of the synthetic topology having a relative small capacity with more time spent on waiting I/O operation to complete. Through this experiment, we reach the conclusion that our method is better suited for applications with mild or medium bandwidth consumptions. If an operator is known to be bandwidth bound, E-Storm has the ability to individually set the number of replicas for this operator to a relatively small value, in order to avoid significant performance penalty.

C. Performance of Recovery

To inject failures and invoke the recovery process, we send *SIGKILL* signals to the designated worker processes, forcing them to terminate without proper clean-up. The real-world application (url-mapping) is selected as the test application, with number of tasks set to 40 and each stateful task having one replica running on another worker node. The application outputs and metrics are collected throughout the test to validate the efficacy and efficiency of E-Storm.

1) Recovery from a Single Error: In the first experiment, a JVM crash was injected to the test application on the 10^{th} minute, causing two stateful tasks to lose their states. Fig. 10a depicts the application throughput obtained through the Storm's RESTFul API. Note that they are calculated as an average statistics of a short time period (10 seconds), as the instantaneous throughput can better reflect the consequence of failure on the application performance. The results demonstrated that the application was paused for 30 seconds, and then gradually increased its throughputs for 60 seconds until stabilization. The recovery time includes: (1) the time used for the supervisor daemon to restart the failed worker process (5.6 s); (2) the time taken for the failed tasks to be prepared (1.8 s); (3) the time taken by the alive tasks to write to Redis (6.2 s) and (4) the time taken by the restarted tasks to retrieve their states from Redis (7.9 s). This break-down data is obtained from the analysis of the Storm log file and the status of Redis server. However, it is also worth mentioning that by default, the Kestrel Spout waits for 30 seconds before replaying the failed tuples, so the application did not produce throughputs immediately after completing the recovery process.

2) Recovery from Multiple Errors: The second experiment is to evaluate the performance of recovery triggered by multiple errors. We injected multiple *SIGKILL* signals to the target worker processes at the same time, with care taken not to bring down all the states backups for a stateful task. As shown

in Fig. 10b, the size of states being cached in Redis almost rises proportionally along with the number of tasks being affected. This result can be explained by two factors: firstly, each stateful tasks maintains roughly the same size of states as the hash-code based streaming grouping balances the load well among them; secondly, only one alive task in each affected task fleet got to preserve its state to Redis, while the others were staying idle during the whole recovery process.

TABLE III
THE COMPARISON OF RECOVERY TIME UNDER MULTIPLE ERRORS (UNIT: SECONDS)

No. of stateful tasks affected	$\mathbf{W_r}$	$\mathbf{T_p}$	$\mathbf{W_s}$	$\mathbf{R_s}$	Total
2	5.6	1.8	6.2	7.9	90
4	5.6	1.9	11.2	13.2	120
6	5.6	2	16.7	18.9	120
8	5.6	1.9	20.2	22.5	120
10	5.7	2.1	25.8	28.7	140

In Table III, we also compare the recovery time needed for the application to resume execution under multiple errors. In this table, W_r is the time used for the failed worker processes to be restarted; T_p is the time for restated tasks to be prepared; W_s is the time used for writing states to Redis and R_s is the time for loading states to the restated tasks. Note that each failed node executes the recovery protocol asynchronously, so they may take different times to complete each recovery stage. Therefore, we report the results by averaging the readings collected on the failed nodes, except the $Total$ column which is the time taken for the topology to restore its normal performance (reaching 90% of the average throughput observed before failure).

The comparison results indicate that the recovery time for Storm daemons are independent from the scale of failures, but the time used for writing and reading states increases along with the state sizes, which are shown in Fig. 10b. However, we can reasonably envision that the use of multiple Redis instances can reduce these time, as different task fleets are mapped to their corresponding Redis instance for concurrent state transfer.

VII. RELATED WORK

State management is one of the major research topics in distributed streaming processing. In this section we review the approaches that manage the transient operator states with particular goals in data stream management systems.

Some works manage states for application integrity in the event of failure or operator scaling. Fernandez et al. proposed a set of state management primitives to expose operator states explicitly to the middle-ware system, so that the DSMS is able to periodically checkpoint them to the upstream VMs with partitions in order to enable state recovery and scaling [2]. Similarly, ChronoStream [3] provides elastic support for stateful operators by dividing the application states into a set of computation slices, which are checkpointed to specified nodes exploiting locality-affinity and lineage-free progress tracking to ensure deterministic semantics. StreamScope is a recent effort to provide declarative interface for users to express complex streaming logic, which also offers the *snapshots* abstraction that periodically checkpoints the operator states without user intervention [4]. Also, MillWheel checkpoints its work in progress at fine granularity so that the states are persisted against failure and message senders are relieved from buffering the pending data for a long period [5]. There are even more works that fall into this area [6]–[8]. However, regardless the level of which the checkpoint is performed or the place where the checkpoint data is stored, periodic state manipulation still introduces non-negligible runtime overhead.

Some works, on the other hand, focus on migrating states for dynamical application scaling. Cardellini et al. realize dynamic horizontal scaling for stateful operators in Storm by allowing the states to be migrated between existing and newly added tasks [9]. Gedik et al. explores the profitability of auto-parallelization by providing a state management API, a run-time migration protocol, and compile-time topology optimization techniques [10], [11]. Ding et al. further investigate the trade-off between synchronization overhead and result delay during state migration, so that the selection of migrated tasks can be optimized to lower the latency spike [12]. However, these methods cannot be used to provide state persistence against failures.

The strategy of replication in stream processing has also been discussed in the literature. Stormy uses replications for high availability, so there is no failure recovery mechanism provided to transfer states between different replicas [13]. For fault-tolerance, Balazinska et al. incorporate a replication-based approach in the Borealis system [14], which duplicates the execution of the same query network on multiple worker nodes [15]. To ensure all replicas processing data in the same order, they also introduce a data-serializing operator that sorts the multiple streams as input and produces a single output stream with a deterministic order. By contrast, E-Storm performs replication at the fine-grained operator level with the flexibility to adjust the resilience guarantee individually, and it does not duplicate the execution of stateless operations. Also, we achieve replica consistency through a lightweight acknowledgement mechanism, thus avoiding the heavy messaging sorting overhead. To reduce the burden of active replication, Martin et al. present an approach that first conducts state partitioning and then distributes state slices across the participating worker nodes [16]. However, this method only profits in MapReduce-like event processing systems as the state partitioning is a side effect of execution during runtime, whereas in the state-of-the-art data stream systems, this method would incur significant state transfer cost when the execution is error-free.

With the similar goal of reducing the replication overhead, Henize et al. combine active replication with upstream backup, allowing for the adaptive selection of replication mechanism for individual operators based on the characteristics of the current workload [17]. However, the placement of operator and replicas to hosts is not discussed in the paper. Flux is

an opaque operator implemented in TelegraphCQ [18] that composes duplicated dataflows to enable online-recovery and mask load imbalances [19]. Nevertheless, replicating the whole data flow and adding an Exchange layer [20] between each Producer-Consumer pair would incur more overhead than our approach, which requires only replicating the stateful operations.

VIII. Conclusions and Future Work

In this paper, we designed and implemented E-Storm, a replication-based state management system that masks the loss of operator states in the case of JVM and node crashes. By using E-Storm, the stateful tasks are replicated on different worker nodes under a replication-aware placement strategy, and the restarted tasks are able to retrieve their previous states from the alive partners through an asynchronous recovery protocol. During the state transfer process, the implementation of E-Storm takes advantage of Redis to decouple the coordination of state senders and receivers, and makes use of Zookeeper to reduce the size of states being transmitted. Therefore, it achieves concurrent and high performance system recovery in the presence of failures.

Through a comprehensive performance evaluation, the results confirm that our approach greatly outperforms the existing checkpointing method in terms of throughput and latency overhead. Specifically, E-Storm can bring up to 9.44 times throughput improvement while reducing the application latency down to 9.8% compared to that of the checkpointing method (witnessed in the synthetic test application when F_s, the number of state access in the *execute* method, is set to 12). We also identified that the overhead of checkpointing is attributable to the frequent state access and remote synchronization, which cannot be mitigated by enlarging the checkpointing interval as it would incur unacceptable latency penalty for real-time streaming applications.

As for future work, we plan to investigate adaptive replication schemes with intelligent replica placement strategies. We also aim to revise the recovery protocol to make it location-aware so that states are transferred between nodes in vicinity. In the long term, we would like to explore the possibility of integrating different fault-tolerance techniques, including active and passive replicas, to provide a sophisticated state-persistence solution that caters for both the characteristics of the streaming application and its workload.

Acknowledgment

The authors would like to thank Chenhao Qu and other members of the CLOUDS Lab at the University of Melbourne for their valuable comments towards improving the quality of the paper. This work is supported by Australian Research Council Future Fellowship and DP150103710.

References

[1] E. G. Coffman Jr., J. Csirik, G. Galambos, S. Martello, and D. Vigo, *Bin Packing Approximation Algorithms: Survey and Classification*. Springer New York, 2013, pp. 455–531.

[2] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '13. New York, USA: ACM Press, 2013, pp. 725–736.

[3] Y. Wu and K.-L. Tan, "ChronoStream: Elastic stateful stream computation in the cloud," in *Proceedings of 2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 723–734.

[4] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "StreamScope: Continuous Reliable Distributed Processing of Big Data Streams," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, 2016, pp. 439–453.

[5] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," in *Proceedings of the Very Large Data Bases*, no. 11, 2013, pp. 734–746.

[6] R. C. Fernandez, M. Weidlich, P. Pietzuch, and A. Gal, "Scalable stateful stream processing for smart grids," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. ACM Press, 2014, pp. 276–281.

[7] B. Lohrmann, D. Warneke, and O. Kao, "Nephele streaming: stream processing under QoS constraints at scale," *Cluster Computing*, vol. 17, no. 1, pp. 61–78, 2014.

[8] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "TimeStream: Reliable Stream Computation in the Cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM Press, apr 2013, pp. 1–14.

[9] V. Cardellini, M. Nardelli, and D. Luzi, "Elastic stateful stream processing in storm," in *Proceedings of 2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2016, pp. 583–590.

[10] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-parallelizing stateful distributed streaming applications," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12. ACM Press, 2012, pp. 53–63.

[11] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic Scaling for Data Stream Processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.

[12] J. Ding, T. Z. J. Fu, R. T. B. Ma, M. Winslett, Y. Yang, Z. Zhang, and H. Chao, "Optimal Operator State Migration for Elastic Data Stream Processing," *arxiv:1501.03619 [cs]*, jan 2015.

[13] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, "Stormy: An Elastic and Highly Available Streaming Service in the Cloud," in *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, ser. EDBT-ICDT '12. ACM Press, 2012, pp. 55–60.

[14] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, and Others, "The Design of the Borealis Stream Processing Engine." in *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, 2005, pp. 277–289.

[15] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. ACM, 2005, pp. 13–24.

[16] A. Martin, C. Fetzer, and A. Brito, "Active replication at (almost) no cost," in *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, 2011, pp. 21–30.

[17] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and C. Fetzer, "An adaptive replication scheme for elastic data stream processing systems," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. ACM Press, 2015, pp. 150–161.

[18] S. Chandrasekaran, M. A. Shah, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, and F. Reiss, "TelegraphCQ: Continuous Dataflow Processing," in *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*. ACM Press, 2003, pp. 668–668.

[19] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of Data*, ser. SIGMOD '04. ACM Press, 2004, pp. 827–838.

[20] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. ACM, 1990, pp. 102–111.