Active replication for latency-sensitive stream processing in Apache Flink

Guillaume Rosinosky*, Florian Schmidt◇, Oleh Bodunov◇, Christof Fetzer◇, André Martin◇, and Etienne Rivière*

* ICTEAM, UCLouvain, Belgium – first.last@uclouvain.be

◇ TU Dresden, Germany – andre.martin@tu-dresden.de

***Abstract*—Stream processing frameworks allow processing massive amounts of data shortly after it is produced, and enable a fast reaction to events in scenarios such as data center monitoring, smart transportation, or telecommunication networks. Many scenarios depend on the fast and reliable processing of incoming data, requiring low end-to-end latencies from the ingest of a new event to the corresponding output. The occurrence of faults jeopardizes these guarantees: Currently-leading high-availability solutions for stream processing such as Spark Streaming or Apache Flink's implement passive replication through snapshotting, requiring a stop-the-world operation to recover from a failure. Active replication, while incurring higher deployment costs, can overcome these limitations and allow to mask the impact of faults and match stringent end-to-end latency requirements. We present the design, implementation, and evaluation of active replication in the popular Apache Flink platform. Our study explores two alternative designs, a leader-based approach leveraging external services (Kafka and ZooKeeper) and a leaderless implementation leveraging a novel deterministic merging algorithm. Our evaluation using a series of microbenchmarks and a SaaS cloud monitoring scenario on a 37-server cluster show that the actively-replicated Flink can fully mask the impact of faults on end-to-end latency.**

***Index Terms*—Stream Processing, Apache Flink, Fault tolerance, Active replication**

I. Introduction

Stream processing is now established as the leading paradigm for in-line processing of high-velocity data [1]. A stream processing framework allows the programmer to express the processing of incoming data as a directed graph of operators, through which incoming data flows in the form of *events*. These events update the state of stateful operators, generating new events until an output result or trigger is eventually generated towards a data sink.

Stream processing has a large range of applications that include internet of things (IoT) applications [2], smart environments [3], traffic monitoring [4], or fraud detection in online payment systems [5], [6]. A large number of stream processing frameworks have been proposed in the past two decades including, e.g., Spark Streaming [7], Apache Storm [8], LinkedIn's Samza [9], Twitter's Héron [10], and the popular and open-source Apache Flink [11] framework.

Many stream processing applications have stringent service level objectives regarding latency. We mention three out of numerous examples of such applications presented by industry representatives at the major Flink developer's series of conferences [12]. Representatives of a large telecommunication industry report a real-time infrastructure monitoring application processing around 80K events per second and outputting around 40K key performance indicators (KPI) per time window, all under the requirement that end-to-end latency is under 100 ms [13]. BetterCloud, a SaaS cloud operator, uses stream processing to detect misuses and policy violations from hundreds of millions of events daily, with the constraint that latency from ingest to alert remains below 2 seconds [14]. As a final example, financial industries use stream processing for financial transactions to detect potential fraud [6]: in this case, the load can range from 250 thousand to a million transactions per second. Each fraud detection processing is in the critical path of transaction validation and requires latencies bounded by at most a few hundred milliseconds.

In addition to their latency objectives, interactive stream processing applications often require high availability. Stream processing engines typically scale-out the computation over multiple servers to handle increasing loads. A failure of one of the servers must not jeopardize the continuation or the consistency of the processing. Latency-sensitive applications must also guarantee that such failures do not lead to violations of their service level objectives.

The common approach to fault tolerance in stream processing systems is passive replication [7], [15]–[19]. The state of the computation is periodically saved to stable storage (e.g., on disk or to a replicated in-memory database), and restored on a new server after a fault. Flink state management mechanisms enable the creation of consistent distributed snapshots [20] of the processing graph state, enabling rollback recovery [21]. While efficient for throughput-oriented workloads, this mechanism falls short to provide high availability in the presence of failures for latency-sensitive applications: We show in our evaluation that the failure of one server can result in a downtime of more than a minute for a representative application.

Motivation. We posit that *active replication* is necessary to support high availability for stream processing applications with strict latency requirements[1]. Active replication entails running each stream processing element on two different servers, implementing state machine replication (SMR). Both replicas receive the same stream of input events, perform the

This work was partially funded by the Belgian FNRS project DAPOCA (33694591).

[1] In their note listing the 8 requirements for real-time stream processing systems, Stonebracker, Çetintemel and Zdonik [22] also argue that only a "Tandem-style" hot backup and failover can enable high-availability (rule #6).

same computation, and must output the same events. If one of the replicas fails, the other replica remains available. While conceptually simple, active replication requires great care in its implementation, to ensure that the scalability of the replicated system remains unchanged, that unavoidable synchronization overheads are reasonable in the failure-free case, that the impact of failures on latency can effectively be masked, and that its operation is transparent to the programmer and does not impair the computation consistency.

Contributions. We explore the design and implementation of active replication mechanisms for Flink[2]. After presenting the foundational principles of Flink (**§II**), we identify the requirements for enabling active replication of processing *tasks* of the stream processing engine while retaining its consistency, scalability, and performance (**§III**). We design and implement two alternative approaches to active replication. These two approaches differ in how they ensure that the stream of events reaching two replicas of a task from one or multiple up-stream tasks is processed in the same order by those two replicas, and in how they allow the remaining replica to guarantee continuity of service in case of a failure[3]:

- Our first approach, named KaZoo (**§IV**), uses two external services, ZooKeeper [23] and Kafka [24]. These services are used to implement consensus on the order of event consumption, following a leader-based approach. One of the replicas is elected as a leader using ZooKeeper. It publishes to a Kafka queue a chosen order of consumption for incoming events that are used by both replicas.
- Our second approach, named LiveRobin (**§V**), employs a leaderless mechanism to ensure in-order delivery of input events to replicas. We implement a novel protocol implementing the concept of *deterministic merging* [25]. This protocol is adapted to the peculiarities of a replicated execution and enables replicas to independently and deterministically process incoming events.

We discuss the relative merits and constraints of the two models (**§VI**), and we validate these observations experimentally on a cluster of 37 servers (**§VII**). We use both a series of micro-benchmarks and a macro-benchmark reproducing the principles of a Cloud SaaS monitoring application described by BetterCloud [14]. Our results show the effectiveness of both solutions in enabling active replication in Flink, resulting in only slightly larger processing latency in the absence of faults, while enabling very fast recovery (for KaZoo) or the complete masking of faults (for LiveRobin). We also highlight the different tradeoffs between costs and performance: KaZoo requires fewer changes to Flink but has higher costs than LiveRobin, but the latter requires careful tuning of its operational parameters, for which we provide guidelines. We finally discuss related work on active/passive replication in stream processing systems (**§VIII**) and conclude (**§IX**).

[2]We note that, while we target the Flink system, our contributions would apply to systems following similar models, such as Apache Storm [8].

[3]Note that we leave the provisioning of a new active replica after such a failure for future work.

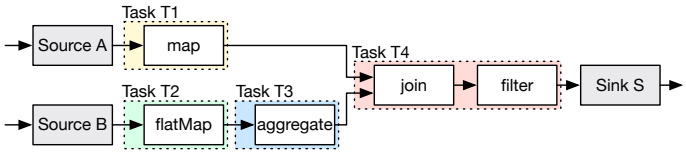

Fig. 1. Example of a processing graph with two data sources, five vertices, and one sink. The five vertices map to four Flink tasks.

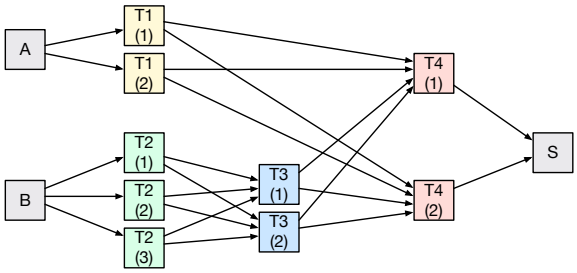

Fig. 2. Physical instance of the processing graph of Figure 1, using parallel processing for all of the four tasks. The example uses 2 partitions for tasks T1, T3 and T4 and three partitions for task T2. Each partition may receive events from every partition of each upstream task, resulting in event subscriptions as represented by directed edges.

II. A primer on Flink

We start by detailing the operational principles and terminologies of the Apache Flink stream processing platform.

Flink allows a programmer to define the processing of incoming stream(s) of data using elementary processing operators or using high-level abstractions provided by libraries (e.g., for machine learning or graph processing). The resulting computation forms a weakly connected graph where each vertex represents a processing step. Incoming data events are injected into this graph by specific *source* vertices, and events may flow out of *sink* vertices. Note that a Flink processing graph does not have to be acyclic. Figure 1 presents an example of a processing graph with five unary vertices and one binary vertex joining two streams of input events. Flink offers support for different state management primitives allowing the programmer to express the level of disjoint-access parallelism for each of the vertices [21].

The client-side library compiles the processing graph and applies optimizations (e.g., merging two vertices operating on the same flow and under the same parallelism constraints). The resulting graph is mapped to a set of *tasks*: Each task combines processing logic and state management, and explicitly subscribes to the output events of its upstream task(s).

The Flink runtime uses a classical organization for a distributed data processing framework: Several worker nodes named *Task Managers* host instances of processing tasks. They are coordinated by a logically centralized *Job Manager* using ZooKeeper [23] as a fault-tolerant coordination kernel. Flink supports elastic scaling by allowing the Job Manager to adapt at runtime the number of Task Managers used to run *partitions* of a given task. Sharding of upstream events

between the partitions is based on the parallelism allowed by the state of this task while preserving its consistency. For instance, for a task using a `Keyed-State` state, partitions are associated with non-overlapping ranges of keys, and partitions can process input events for different key ranges in parallel. Figure 2 presents an example instance of the graph of Figure 1 where tasks have been scaled up to support higher processing throughput through parallel processing of incoming events.

III. Enabling active replication in Flink

Active replication requires running more than one copy of each task using different servers (Task Managers). Each partition of a task must be replicated individually. Our goal is to mask the impact of a Task Manager fault (and that of all of its supported tasks) on end-to-end user-perceived latency. We detail our usage and fault models and analyze the requirements for implementing efficient active replication.

Scope and fault model. We first emphasize that the use of active replication is not antinomic to the use of passive replication. Active and passive replications are, in fact, *complementary* mechanisms. The storage of incrementally built consistent distributed snapshots as integrated with Flink and detailed by Carbone *et al.* [21] does not significantly increase processing latencies during fault-free operation, as our evaluation results indicate. Passive replication allows recovering from catastrophic failures, e.g., the failure of an entire Flink cluster. It also allows seamlessly supporting *planned* reconfiguration events in production systems, such as scale-in and scale-out operations or software updates. A reconfiguration after an *unexpected* loss of a server using rollback recovery from the distributed snapshot is, however, a time-consuming operation. It requires to halt the processing of events until a backup copy is fully bootstrapped. The role of active replication is primarily to support high availability under such an unpredictable node failure, which must not result in a halt of the entire computation and must have minimal impact on observed end-to-end processing latency.

Our fault model targets high availability under a single ($f = 1$) server failure, thereby requiring two copies of each task. If more than one server fails, we can end up in one of the two following situations: either the lost servers hosted replicas for a *disjoint* set of tasks, in which case we maintain high-availability guarantees or both replicas for a specific task are lost simultaneously, in which case we must apply rollback recovery. Using higher values of f would be straightforward with our algorithms, ensuring robustness to more faults before applying rollback recovery but also requiring a significant amount of resources.

We focus in this paper on the impact of a fault on continuity of service. The design and evaluation of recovery mechanisms, which would allow the creation of a new active replica after a failure, is part of our future work. Such mechanisms should work hand-in-hand with passive replication mechanisms such as checkpointing and event replaying.

Requirements. It is necessary that both replicas of a task output the same events and eventually hold the same state.

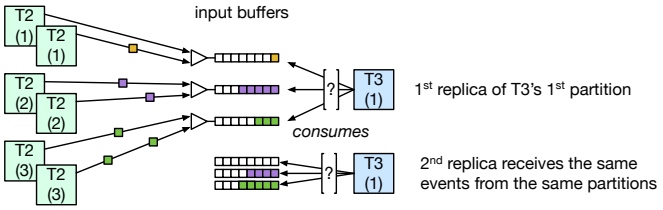

Fig. 3. The first partition of task T3 from Figure 2 is replicated. Each of its replica receives input events from all partitions of the upstream task T2. As these upstream task partition are themselves replicated, each event is received twice. Duplicate events are filtered out (as figured by the triangle filter symbol). The second replica receives the same events, but not necessarily in the same order across multiple input buffers.

This requires that (1) the set and order of reception of input events by the two replicas be strictly the same and that (2) the processing of the same set of input events at both replicas produces the same set of output events. For this second constraint we assume, similarly to related works [26]–[29], that the implementation of tasks is deterministic. Firstly, the code must not use sources of non-determinism in the execution such as calls to an external pseudo-random generator (a pseudo-random generator seeded by an initial event is deterministic and fits into the model). This condition is easy to meet in practice in the data flow model of Flink, which promotes sandboxed operations and does not easily allow external calls. Secondly, multi-threading could be a possible source of non-determinism, as multiple threads processing events on the same node could be scheduled differently on two replicas. We assume, therefore, the use of single-threaded tasks. Parallelization of the computation, including on the same multi-core server, can simply be achieved by using more partitions of each task.

In addition to deterministic processing, we need to ensure that the set of input events for both replicas of each task partition is identical and that these events are consumed in the same order. We illustrate in Figure 3 the case of a replicated partition of task T3 from Figure 2. Each replica receives events from all upstream tasks, individually from each of their partitions. In this example, there is a single upstream task T2 with three partitions. Both replicas of T3's first partition receive events from all three of T2's partitions. As our goal is to mask faults' impact on latency, we choose to send duplicated events from both partitions and filter the duplicates at the destination[4]. Sequence numbers in input events enable this deduplication. As the connection between tasks' replicas uses TCP, delivery of events is in the same order for both pairs of replicated upstream and downstream tasks' replicas. For instance, messages from the two T2 (1) replicas reach T3 (1) and T3 (2) partitions in the same order. Note, however,

[4]Sending a single event would require synchronization between the two upstream replicas on different servers, requiring network traffic and additional latency. We note, in addition, that using duplicate sending can leverage the colocation of one of the upstream with one of the downstream replica for almost instantaneous local message passing.

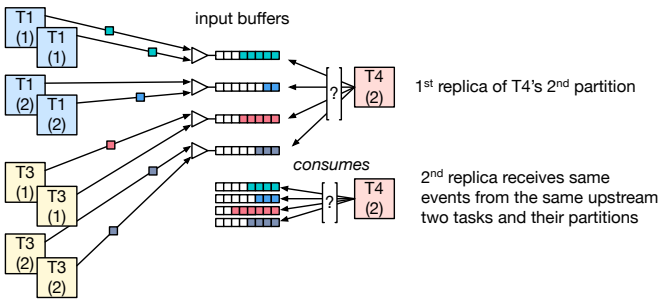

Fig. 4. The second partition of task T4 from Figure 2 is replicated. In contrast with task T3 in Figure 3, T4 receives events from *two* upstream tasks. The same set of events is received by both replicas of T4's first partition, but not necessarily in the same order across different input buffers.

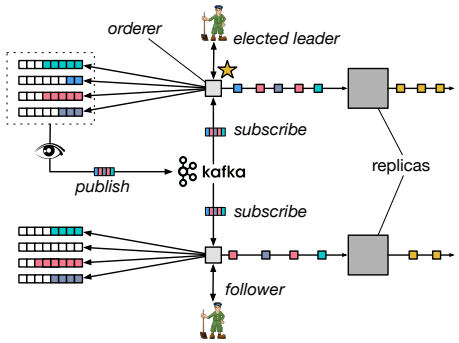

Fig. 5. Leader-based ordering with the KaZoo solution. ZooKeeper is used to elect a leader out of the orderers associated with the two replicas. An in-order publish-subscribe Kafka channel allows the leader to share a unique order of input event consumption that is subscribed by the two orderers.

that the same event consumed by two replicas of the same T3 (1) partition may have been produced by different replicas of T2 (1): The *deduplication* of incoming events simply selects the first event received with a sequence number higher than the last selected event.

While event delivery from replicas of the *same* task partition is ordered identically in the input queues of all downstream tasks, delivery *across* queues does not provide any guarantee about relative reception order. The second T3 (1) replica may have received, for instance, more events from the T2 (3) replicas than from the T2 (2) replicas, while the opposite could hold for the first T3 (1) replica. The same holds for tasks receiving events from several upstream replicas. Figure 4 presents the example of the replicated second partition of task T4. The same set of events are received by both replicas from tasks T1 and T3, but with no ordering guarantees across buffers.

As both replicas receive the same set of events, we need to ensure that both replicas *agree* on the processing order of these events, i.e., when to pick input items from the different input buffers. This choice is illustrated by the question mark boxes in Figures 3 and 4. Imposing a strict order to streams of commands received by different replicas is constitutive of the state-machine-replication approach. We consider two possible solutions to enforce this order, which we present in the next two Sections. Our first solution is leader-based and relies on support from Apache ZooKeeper [23] and Apache Kafka [24]. Our second solution is leaderless: It does not require explicit synchronizations between replicas. It takes advantage of the explicit and fixed communication graph to implement an *economical* and efficient atomic broadcast primitive, using a novel algorithm implementing the principles of deterministic merging but that, in contrast to past implementations of this principle [25], is able to work with replicated sources and provide liveness guarantees. We discuss the relative advantages and disadvantages of these two methods in Section VI.

IV. KaZoo: Leader-based atomic broadcast

Our first approach to ordering is named KaZoo for its use of the popular Apache Kafka [24] and Apache ZooKeeper [23] frameworks. It is illustrated in Figure 5.

The Flink Task Manager runs two additional threads alongside the thread executing the task. A first thread implements a *leader election* protocol. It leverages one of the ZooKeeper's Curator recipes for leader election. This recipe interacts with the coordination kernel to initially establish a leader upon bootstrap and later react to events received from ZooKeeper (in particular, the failure or disconnection of the current leader triggers a new election).

The second thread implements an *orderer* service that plays two roles. For the replica acting as the current leader, the orderer inspects the content of the local input queues and decides arbitrarily on a consumption order (i.e., a list of queues indices). This order is not used immediately to start processing events but rather sent to a pre-established channel with an instance of Apache Kafka running on the cluster. Both replicas' orderers are subscribed to this channel, which guarantees same-order delivery[5]. Upon reception of the chosen order, the orderer pushes them to a queue towards the task processing thread. Note that if an event is not yet available in the follower's input queue (as is the case for the third queue in the follower's example of Figure 5), the orderer halts processing to resume it upon its reception.

The performance of ordering is improved by using batching, whereby ordering operations are sent for groups of events. In the general case of a busy system, input queues will all contain a few unprocessed elements and the reading order can and even should be sent for many events at once: It is reasonable, however, to avoid ordering too many events for the same queue (or very recent events) to avoid the risk of halting computation at the follower for events that are yet to arrive. When the system is mostly idle, a timeout triggers quickly after the last push to the Kafka topic, to avoid increasing end-to-end latency.

[5]Kafka offers different delivery guarantee models. By using the `acks=all` configuration, we obtain *at least once* guarantees for the delivery of orders. We easily avoid risks associated with duplicate reception by making order messages idempotent using incremental version numbers.

V. LiveRobin: Leaderless deterministic execution

Our second approach does not require external services and is leaderless, i.e., it does not require one replica to decide and push an ordering of events to the other replica. The motivation for this approach is the avoidance of latency spikes associated with reconfigurations after faults: If one of the replicas fails then the other replica simply continues processing events.

A. Atomic broadcast through deterministic merging

Both replicas of a task partition receive the same ordered stream of events from each of their upstream tasks. Each stream is buffered into a separate FIFO queue. There is, however, no guarantee that events from different tasks arrive in the same relative order at the two downstream replicas. Our objective forfeits the use of synchronization between these replicas to ensure that events are processed in the same order. The two replicas must, instead, independently take the same decision about the order in which queues must be polled. Such a process is termed a *deterministic merge* [25]. It enables *economical* atomic broadcast when the possible sources of incoming messages are fixed and known by all destinations, as is the case in our model.

An initial, simple approach to enable deterministic merge is for each replica to consume from its input queues in a round-robin (RR) fashion. In the example of Figure 4, T4 (2)'s first replica could consume first from the stream from T1 (1) replicas, then from T1 (2), from T3 (1), and so on. RR applied on the second replica of T4 (2) will consume events in the same order. As the event order for the same input stream is identical, this scheme achieves deterministic execution. RR deterministic merge is subject, however, to arbitrary *merge delays*: if one of the upstream partitions produces more events than another then these events may queue up for arbitrarily long times. In addition, RR does not provide any *liveness* guarantee: if some upstream task stops producing, all downstream tasks will eventually be blocked on attempting to consume from the corresponding queue.

The Bias algorithm [25] is a standard approach to limit merge delay inflation. Each producer stamps outgoing events using its local clock value at the time of their generation. Consumers deterministically determine which queue must next "catch up" based on relative generation time at the event sources, efficiently supporting varying generation rates. To ensure liveness, Bias sources periodically inject empty keep-alive events that ensure that queues are periodically filled with at least one timestamped event, enabling decision-making.

Our context does not allow the use of an upstream-timestamped deterministic merge algorithm such as Bias due to our use of replication. Each task is, indeed, receiving events for every input queue from *two* upstream replicas. Deduplication simply selects the first event with a given sequence number received from *either* of the two upstream replicas. This first-come, first-picked approach is essential to ensure that a failure of one of the replicas does not impact end-to-end latency. As time-stamping operations cannot be synchronized between upstream replicas, the stream of events at both downstream replicas may contain deduplicated events with identical payloads but different timestamps, preventing any deterministic selection of polled queues based on these timestamps. The periodic generation of keep-alive from both upstream replicas also leads to similar problems: Two downstream replicas are not guaranteed to deduplicate a keep-alive in the same position in their two deduplicated streams, breaking determinism.

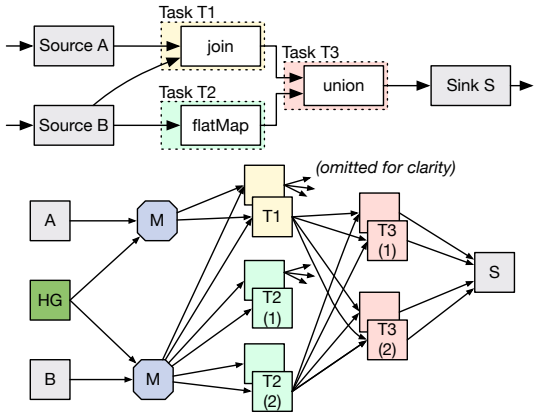

Fig. 6. Example processing graph with two sources and three tasks (top), and an instantiation in Flink with a single partition for T1 and two partitions for T2 and T3 (bottom). The HG and M components are specific to the LiveRobin leaderless approach. All task partitions are replicated twice, and we show only a subset of links for clarity.

B. Deterministic merging for deduplicated flows

We design LiveRobin, a novel deterministic merge algorithm adapted to our context. Our algorithm is an evolution of the round-robin (RR) principle that addresses merge delay inflation. It ensures liveness through the use of periodic keep-alive messages, but these are issued by a single source and propagated deterministically in the task graph.

We introduce a new component in the Flink computation graph called the heartbeat generator (HG), as well as a stateless component, the merger (M). We transform the Flink computation graph as illustrated by Figure 6. Each source S connects directly to an M instance, and this M instance connects to the tasks that were consuming this source in the initial graph. In addition to S's stream of events, a merger M receives periodic heartbeat messages from the HG. These heartbeats contain monotonically increasing sequence numbers. M may consume events from its two input queues (from S and HG) in arbitrary order and merge them to its output. Note that the M role can be implemented by the source task directly and does not require a specific server to run.

Heartbeat messages address merge delay inflation and provide liveness guarantees. They can be seen as a global *pulse* flowing through the graph of tasks, implicitly coordinating cycles of events consumption. We note that our usage of heartbeats can be related to the well-known concepts of punctuation and watermarks. Punctuations are annotations embedded in the datastream specifying the end of a subset

of data [30]. Watermarks are a specific class of punctuation based on the usage of timestamps, guarantying no event future to the timestamp will be processed [31]. The stream processing engines' underlying mechanisms for punctuation or watermark propagation can be employed to ensure the transmission of heartbeats [32], [33].

A task receiving a heartbeat of a given sequence number *for the first time* and from any input queue propagates it on its output, to be consumed by downstream tasks (two replicas will deterministically see the heartbeat for the first time in the same queue and position and, therefore, propagate it in the same position in their output streams). The role of heartbeats is to ensure that all events from the previous implicit cycles are consumed *before* proceeding to the next phase. We use for this purpose an evolution of the simple round-robin principle, illustrated by Algorithm 1.

The LiveRobin algorithm repeatedly dequeues (line 10) from an input queue represented by an index "next". The dequeue is a blocking operation, returning only when an event is present in this queue. For regular events, the local task partition is called to process it (line 19), possibly issuing output events in the "out" output queue. When receiving a heartbeat, the algorithm may send it out (lines 12–14). It then removes the current queue from the active one for this cycle (line 15) and starts a new cycle if this was the last heartbeat missing (lines 16–17). In both cases, the next queue to poll is deterministically selected using round-robin on the queues that are still active (lines 20–23). The algorithm is illustrated for one of the replicas of task T3's first partition in Figure 7. In this example, one of the replicas of an upstream task partition (T2 (2)) has failed, yet the processing of events continues with no reconfiguration delay.

We observe that a halt of the HG can result in a temporary absence of heartbeat messages and, as a result, in halted consumption of events at downstream tasks. This is a fundamental requisite for LiveRobin's liveness. We assume that this problem can be solved using well-known solutions from the dependability literature, and leave the implementation of such schemes for future work.

Algorithm 1 The LiveRobin algorithm

```
1:  constants
2:      in[1..n]                          ▷ array of input FIFO queues
3:      out                               ▷ output FIFO queue
4:      T                                 ▷ local supported task (partition)
5:  variables
6:      next : integer                    ▷ next queue to poll, init. 1
7:      gen : integer                     ▷ highest heartbeat seen, init. 0
8:      A : set(integer)                  ▷ active input queues, init. {1..n}
9:  repeat
10:     event e ← in[next].dequeue()
11:     if e.isHeartbeat() then           ▷ read heartbeat
12:         if e.seq > gen then           ▷ first reception?
13:             out.send(e)               ▷ propagate heartbeat
14:             gen = e.seq
15:         A ← A \{next}                 ▷ ignore this queue until next cycle
16:         if A = ∅ then                 ▷ saw all HBs for this cycle?
17:             A ← {1..n}                ▷ restore set for new cycle
18:     else
19:         T.process(e)                  ▷ regular processing & event output
20:     if next ≥ max(A) then
21:         next ← min(A)                 ▷ implement RR between active queues
22:     else
23:         next ← min({c ∈ A | c > next})
24: end
```

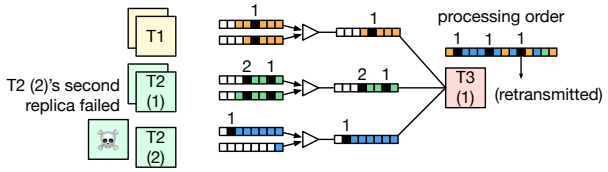

Fig. 7. LiveRobin (Algorithm 1) at the first replica of task T3's first partition. The replica receives duplicate events from each couple of upstream partition replicas, and deduplicates them: The failure of one of T2's second partition replicas does not impact the reception of events from this partition.

VI. Discussion

KaZoo and LiveRobin are two different approaches to active replication in Flink. Before presenting our evaluation results we discuss their relative merits and weaknesses from the design, implementation, and maintainability points of view.

The main expected drawback of KaZoo is that failures of the one replica selected to act as a leader will inevitably delay the processing of incoming events at the surviving replica, as the ZooKeeper fault detection mechanism requires some time to react to the absence of keep-alive from this replica. The election of a new leader following this detection will further incur some latency. We expect, however, that the resulting cumulated latency will be orders of magnitude smaller than the restoration of a full copy of the failed replica from stable storage, as required with passive replication. KaZoo has, on the other hand, the advantage of its simplicity. It requires less profound changes to the Flink runtime than LiveRobin, thereby being simpler to implement and easier to maintain. Finally, the use of a leader brings more flexibility to implement event consumption policies, e.g., when one input stream has more stringent processing latencies than the other, while LiveRobin by its nature dictates a unique, specific processing order for the two replicas of a given task.

LiveRobin is designed to avoid any reconfiguration latencies when one of the replicas fails and targets applications with strict end-to-end (tail) latency requirements. The selection of the HG heartbeat generation frequency is an additional parameter that operation engineers have to set properly: a low frequency will lead to congestion in the processing graph due to empty input buffers[6], while a high frequency leads to higher bandwidth consumption. As we detail in our evaluation, LiveRobin is also sensitive to the buffer timeout parameter of Flink. We can note, also, that in LiveRobin all links in the task graph are subject to increased bandwidth consumption, regardless of the number of data events flowing over these

[6]It is desirable that the latency waiting for a heartbeat to arrive in LiveRobin be of same order of magnitude as the latency imposed by the roundtrip with Kafka in KaZoo.

links: The costs increase linearly with the size of the graph, irrespective of the actual data traffic, while for KaZoo the extra costs (communication with/from Kafka) are proportional to the volume of data events.

As is constitutive of the active replication approach, both solutions require additional resources. The replication of all tasks doubles the number of required physical servers[7]. For both approaches, the use of duplicate sending and deduplication at the destination quadruples (x4) network bandwidth consumption (each replica receiving from a couple of replicas for every upstream task partition) for data events.

VII. Evaluation

Our evaluation answers the following research questions:

1) Are our two active replication solutions able to mask the impact of a task fault on end-to-end processing latency?
2) What is their impact on fault-free performance, and how does this impact evolve with the task graph structure?
3) What are the intrinsic costs of the two solutions and how do they compare to each other?
4) Finally, how does active replication compare to passive replication integrated with Flink [21] in case of a fault?

We answer these questions with micro- and macro-benchmarks. We use the Flink 1.7.2 codebase as a basis for our implementation of KaZoo and LiveRobin. Our macro-benchmark is directly inspired by the description of an industry use case documented by a SaaS cloud provider [14].

Configurations. We consider five Flink configurations:

1) **No replication**: A Flink cluster with neither passive nor active replication. A fault of one TaskManager results in the failure of the entire computation. This is the baseline.
2) **Passive replication**: Flink configured to use periodic snapshotting on HDFS [21]. Upon a fault, failed task(s) are restored from the snapshot.
3) **No ordering**: This configuration replicates all tasks but does not enforce the same order of consumption of deduplicated events by the two replicas of the same task. This configuration is obviously incorrect but it is useful as a comparison point to measure the relative cost of synchronization in 4) **KaZoo** and 5) **LiveRobin**.

Note that we do not enable periodic snapshotting in the three latter actively-replicated configurations, in order to focus on the performance overheads over a non-fault-tolerant Flink.

Experimental setup and reproducibility. All of our experiments run over a Kubernetes 1.19.3 cluster of 37 non-virtualized servers, as illustrated by Figure 8. Each server features two 4-core Intel® Xeon® E5405 CPUs running at 2 GHz and 8 GB of RAM. The network is 1 Gbps Ethernet.

[7]While KaZoo requires the use of external services (ZooKeeper and Kafka), we note that these are generally already deployed alongside Flink to support its internal configuration and the management of input streams–we do not need to use separate instances. For LiveRobin, additional HG and M components are lightweight and can be deployed alongside replicated tasks and also do not require dedicated machines. Our implementation does, in fact, merge the M component with the code of the source tasks.

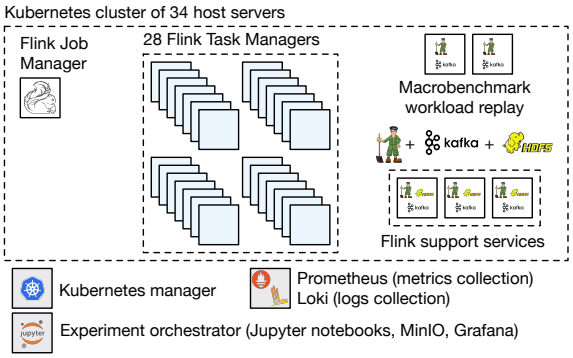

Fig. 8. Experimental setup of 37 servers including 28 Task Managers.

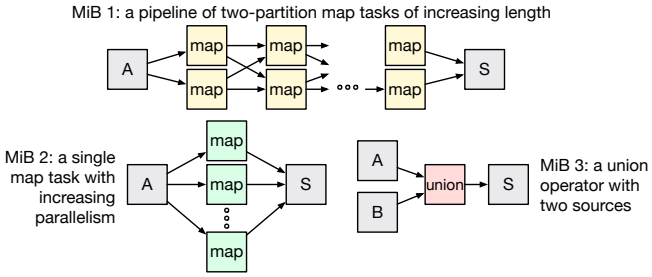

Fig. 9. Micro-benchmarks topologies MiB 1, MiB 2 and MiB 3.

Our experiments are designed for reproducibility. They are defined using replicable Jupyter notebooks. We use MinIO for the storage of experimental results and Prometheus, Loki, and Grafana for the collection and analysis of logs and metrics. All our code and results are available open-source [34].

A. Microbenchmarks

We first consider the three topologies illustrated in Figure 9. In all microbenchmarks, each replica is deployed on a dedicated task manager server (no replica co-location). MiB 1 is an increasing-length pipeline of map tasks implementing the identity function, i.e., forwarding received events. Each task has two partitions. MiB 2 considers a single task with a variable number of partitions. Finally, MiB 3 considers a single non-partitioned task with two sources, merging two incoming streams towards a sink.

In all active replication cases, all task partitions are replicated as two instances. We inject 5,000 events per second. LiveRobin uses heartbeats with a default period of 10 ms. Latency is measured from the generation of the event by the source task and its reception by the sink task on the same server. We collect latency samples for 120 seconds after a bootstrap period of 30 seconds, over 5 consecutive runs, and present the resulting distribution. Box plots present the 25th and 75th percentiles as well as the median of the latencies distribution (boundaries of the box and internal line). Whiskers represent its 1st and 99th percentiles.

MiB 1: Increasing task graph length. Figure 10 presents the end-to-end event processing latency in the MiB 1 configuration, with 1 to 4 tasks in sequence. In all configu-

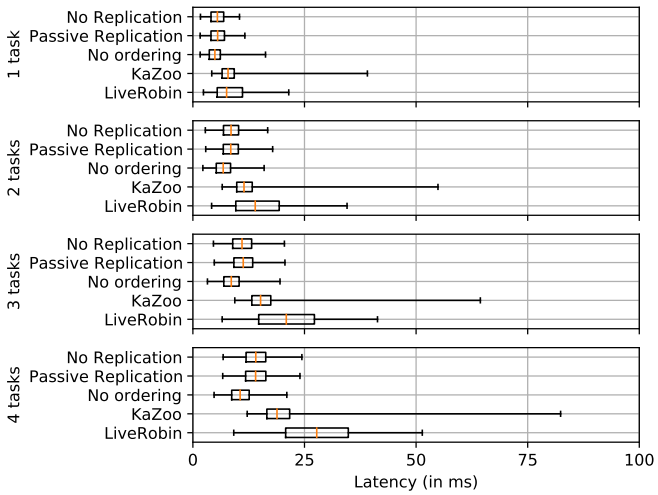

Fig. 10. End-to-end latency distributions for 1 to 4 tasks in MiB 1.

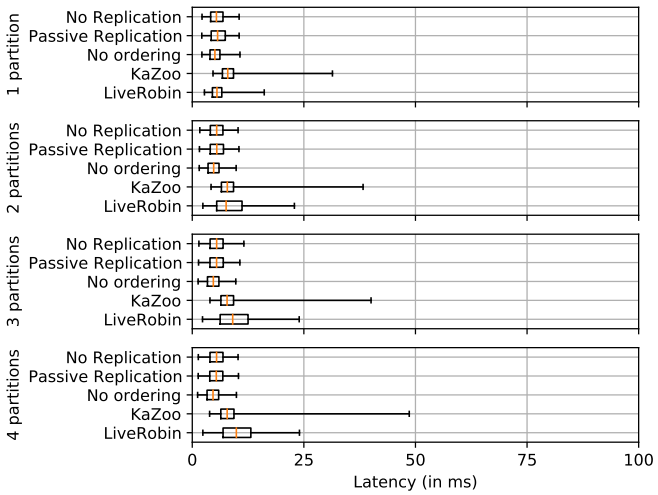

Fig. 12. Impact of increasing task parallelism on end-to-end latency.

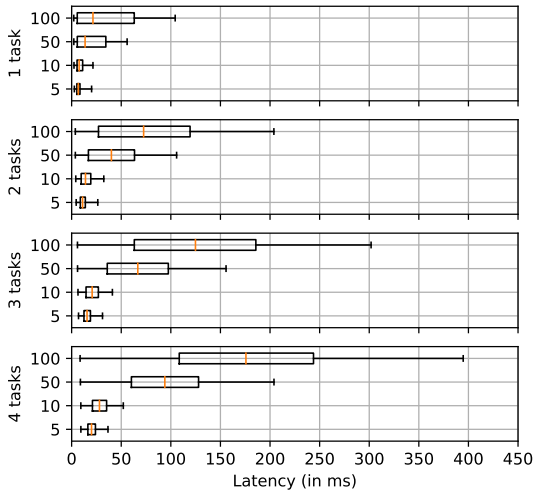

Fig. 11. Impact of heartbeat generation period for LiveRobin in the MiB 1 micro-benchmark. Values on the left are heartbeat periods in ms.

rations, latency increases as expected with the number of consecutive tasks. Passive replication only impacts minimally latency. The No ordering configuration presents the lowest median latencies and the lowest variability with an increasing number of consecutive tasks, as can be expected: On any task replica, any first instance of an event is immediately processed, independently of choices made by the other replica. KaZoo and LiveRobin present higher latencies than the non-replicated case: Deterministic processing introduces a delay, either through Kafka roundtrips in KaZoo or waiting for heartbeats in LiveRobin. KaZoo presents lower median latency than LiveRobin, but also greater variability.

There is a clear correlation in LiveRobin between the heartbeat period and latency, as we show in Figure 11 with heartbeat periods of 5 to 100 ms. Events may, indeed, get "blocked" in an input buffer when they are received after a heartbeat that has not yet been received on all other input channels. The probability of this effect increases with the length of the task graph. A smaller period reduces the duration and odds of such blockades, at the price of higher bandwidth consumption. We select 10 ms as a reasonable default value.

We further identify a relation between the heartbeat period and another important Flink parameter, the *buffer timeout*. To balance latency and throughput, Flink buffers incoming events before processing them in (small) batches. The default timeout for this buffering is 100 ms. In LiveRobin, heartbeats are received and processed alongside application events. A large timeout value can delay the propagation of heartbeats, inflating delays for the processing of all events. Our experiments with varying timeout values (not shown due to space constraints) indicate that setting the timeout to a value lower than the heartbeat generation period has no significant impact on maximal achievable throughput, and reduces the risk of latency inflation. We use a timeout value of 5 ms in all experiments.

MiB 2: Increasing task parallelism. Figure 12 presents latency distributions for a single task with an increasing number of partitions. As before, we inject 5,000 events per second. We observe that task parallelism has little impact on end-to-end latency in almost all configurations. The exception is LiveRobin when using a single partition. In this case, the two replicas of the task unique partition receive events from a single upstream task (the source task) and can consume this single stream of deduplicated events as soon as it is available, exactly as in the No ordering configuration. The median latency of KaZoo is stable with an increasing number of partitions, while it tends to slightly increase for LiveRobin. KaZoo exhibits, however, the highest tail latencies of all configurations.

MIB 3: Multi-source task. Our third micro-benchmark analyzes the impact of unbalanced event reception frequencies from multiple upstream tasks. In MiB 3, a task receives

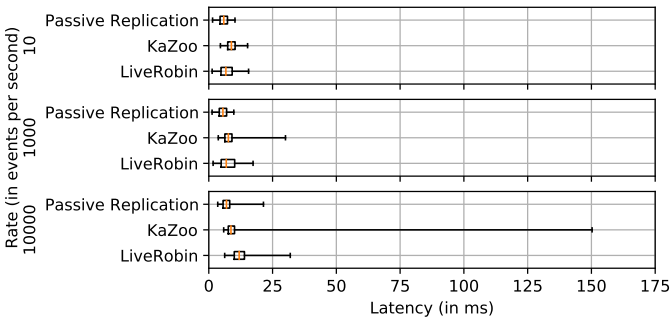

Fig. 13. Impact of relative event rates in the MiB 3 double-sourced task.

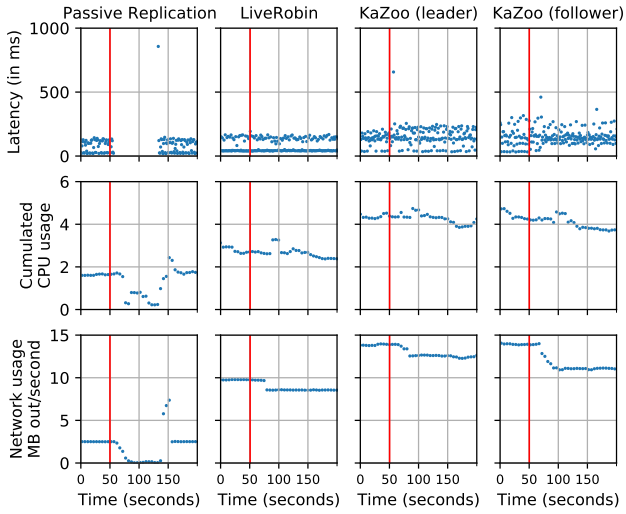

Fig. 14. Tail latency (99th perc.), CPU, and network utilization over 1-second periods for 3 consecutive map (identity) tasks. A fault is injected 50 seconds into the experiment (red vertical line), crashing a random task manager replica. For KaZoo, we distinguish the crash of a leader or of a follower replica.

events from two source tasks and pushes the union of the two streams. Figure 13 presents latencies in three configurations. The first source always generates 5 events per second, and the rate of the second source varies from 10 to 10,000 events per second. We can observe that the median latency remains stable for passive replication and KaZoo. It increases slightly for LiveRobin while remaining well within acceptable margins. For all configurations, imbalance results in higher latency variability, and KaZoo shows the highest tail latency. LiveRobin shows an increase in tail latency for the 10,000-to-5 scenarios but remains close to the passive replication in the already significantly unbalanced 1,000-to-5 case.

B. Resource usage and impact of faults

We now measure resource usage and the impact of a fault in one run of each of the four correct replicated configurations. Figure 14 presents the tail latency, overall CPU usage (cumulated over all task managers), and cumulated network usage. We use a setting similar to MiB 1 with two partitions per

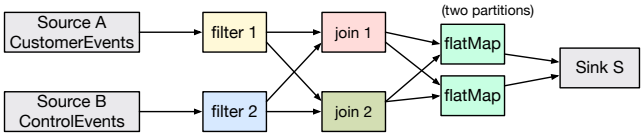

Fig. 15. Task graph of the application inspired by BetterCloud [14].

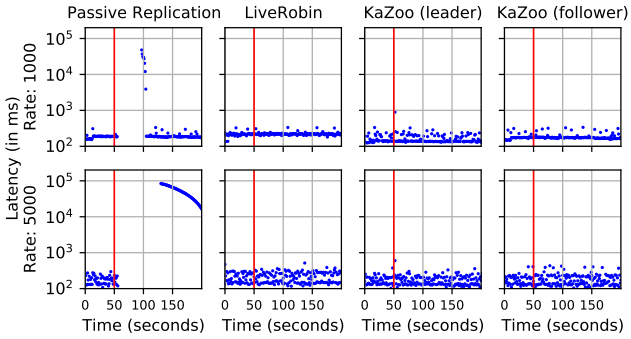

Fig. 16. Per-second tail latencies (99th perc.) in BetterCloud application.

task, and with 3 consecutive tasks. In contrast with previous experiments, our cluster is instrumented with the fine-grained collection of performance metrics using Prometheus. This instrumentation has an impact on performance, but this impact is similar in all configurations. We inject 5,000 input events per second. 50 seconds into the experiment, we randomly crash one task manager hosting a task replica. We first observe that passive replication requires significant time to resume processing after recovery, resulting in latencies of more than 60 seconds. In contrast, both of our actively replicated solutions are able to mask faults and continue processing events after the crash. We observe, however, that the failure of a leader in KaZoo leads to some outlier latencies (in the order of up to a second) due to the time required for the fault detection and leader election process. The failure of a follower replica in KaZoo, or the failure of any replica in LiveRobin, has no visible effect on end-to-end processing latency, as expected. In terms of resource usage, we can observe the increased CPU usage of active replication (requiring twice as many task managers and the same number of support nodes), and the quadrupled network consumption (each event is sent twice by each of the two replicas). The increased CPU and bandwidth consumption in KaZoo is due to the usage of the Kafka service, and the necessary exchanges of ordering messages, weighting more in total than LiveRobin heartbeats.

C. Application benchmark

We finally evaluate the ability of our active replication mechanism to mask the impact of faults in a representative processing task graph, inspired by a use case described by BetterCloud [14]. In this flow monitoring application, events from customers (e.g., reception of an email, login, SaaS applications notifications, etc.) are received and analyzed against a number of rules in order to detect policy violations and trigger specific actions and workflows. Control events allow

to update these rules. Figure 15 present the task graph, with the two sources, each followed by filtering tasks, two parallel joins, and a final task implementing a flatMap. We developed a workload injection tool using an independent Kafka service. All tasks use a single partition except for the final flatMap that uses two partitions.

This use case requires an end-to-end latency of less than 2 seconds between the reception of a trigger customer event and the reception of a notification at the sink, and must not suffer from an unavailability period despite faults, making it an ideal candidate for our active replication solutions. The reported deployment supports 100 million incoming events per day, i.e., on average 1,200 events per second–we consider, therefore, input rates of 1,000 and 5,000 events per second.

Figure 16 presents the tail latency for four configurations, using the same presentation as previously, but this time using a logarithmic scale for latencies. After 50 seconds in the experiment, we kill one of the two partitions of the flatMap task. As we can observe, passive replication is subject to a long period of inactivity (more than a minute). With 5,000 events per second, the system takes a long time to recover and "catch up" events missed during this period. Consistent with our previous results, LiveRobin and KaZoo with a failure of a follower replica see little impact on tail latency (remaining below the objective of 2 seconds, although higher in the failure-free case than for the non-replicated version). The failure of a leader replica in KaZoo imposes higher tail latencies for a limited amount of time but does not exceed the 2-second objective.

In summary, our experiments show that LiveRobin and KaZoo are successful in masking the impact of fault on end-to-end latency for Flink jobs with tight latency and availability requirements. They also require more resources, and due to the cost of synchronization, slightly degrade performance in the fault-free case compared to the sole use of passive replication. In terms of limitations, LiveRobin is sensitive to the heartbeat generation period and buffering settings in Flink, while KaZoo has higher resource requirements and may suffer from short-term latency spikes when the leader replica fails.

VIII. Related Work

Passive replication uses a combination of checkpointing and logging to restore the state of tasks after a failure. Hwang et al. [15] introduce upstream backup, allowing to replay all events when restoring a fresh task after a fault. This mechanism suffers from increasingly long recovery times as events accumulate, hence it is typically combined with periodic checkpointing [16]: Only events stored between checkpoints must be replayed. Restoration can be optimized, e.g., by storing checkpoints across multiple nodes [17] or using data structures avoiding costly (de)serialization [18]. Systems such as Timestream [19] or Spark Streaming [7] discretize streams in batches and apply fault tolerance mechanisms similar to the ones used in a batch processing system. The knowledge of the lineage of the batches enables their parallel reconstruction. Despite these optimizations, the recovery time depends on the state size and can reach several seconds. In Apache Flink, passive replication is implemented as a variant of the Chandy-Lamport distributed snapshot algorithm [20], that avoids buffering in-flight events for replay. Checkpoint markers are introduced in the stream [21], triggering globally consistent snapshots. The propagation of these markers resembles the use of heartbeats in LiveRobin, although they are sent at a much lower frequency. As our evaluation has shown, recovery time in passively-replicated Flink can also reach tens of seconds. While it can be made more efficient, e.g., using faster HDFS servers, it is intrinsically unable to fully *mask* the impact of faults and recovery.

Shah et al. [35] proposed an early approach to active replication introducing new stream operators for replication and deduplication, used explicitly by a programmer to replicate a computation while respecting its semantics. In contrast, our solutions are fully transparent. Borealis [36] supports duplicated operators but trades high availability for consistency: duplicate operators may emit *tentative* events when an upstream operator has failed but this does not guarantee any processing determinism. E-Storm [26] is an active-standby solution for Apache Storm [8]. It duplicates processing operators but only a single one sends out events, thereby requiring no deduplication but requiring a reconfiguration after a fault. Our approaches leverage duplicate sending to avoid this reconfiguration. Martin *et al.* show how to dynamically switch between active replication and active-standby based on resource usage [27] or how to switch between replication schemes for different resource usage and dependability guarantees tradeoffs [28], but these solutions have not been implemented in an industrial system such as Flink or Storm.

The use of deterministic merge in actively-replicated stream processing engines has been previously advocated by Martin *et al.* for the StreamMine3G system [27] and is mentioned in a paper describing Microsoft's StreamScope [29]. The work of Martin *et al.* [27] uses source-assigned timestamps that are kept as events flow through the graph of tasks and generate new downstream events. These timestamps are assigned using the local clock at the source tasks. The clocks of the servers hosting these source tasks must be synchronized using NTP to avoid performance degradation. Tasks replicas consume events in the order of these timestamps, under the condition that at least one event is present in the input queue from *all* upstream tasks. If this condition is not met, events consumption has to be halted. In contrast, our solutions do not suffer from such starvation issues. In KaZoo, the leader replica is always able to enforce an order based on the events present in its *own* input queues. In LiveRobin, the propagation of heartbeats imposes a maximal time before any even can be consumed. Microsoft's StreamScope [29] also mentions the use of active replication. Similarly to our solutions, it uses deduplication based on sequence numbers assigned by two replicas of a deterministic upstream task. The authors do not detail, however, how StreamScope deals with different orders of arrivals for events at two replicas of a downstream task, i.e., how it handles deterministic input event ordering.

IX. Conclusion

We presented the design, implementation and analysis of two active replication techniques for Flink. KaZoo is a leader-based solution based on Kafka and Zookeeper, and LiveRobin is a leaderless solution based on a new deterministic merge algorithm offering liveness guarantees. Our evaluation shows that both methods are able to mask the impact of faults on user-perceived end-to-end latency. Each represents a different tradeoff between dependability, performance, and cost: KaZoo requires fewer modifications to Flink and uses existing support services, but has higher runtime costs and may suffer from short-term latency spikes in case of leader failures, while LiveRobin requires deeper changes to the Flink codebase and careful parameter setting, but is always able to mask failures.

This work opens interesting perspectives. First, we wish to investigate a tighter integration between our active replication mechanisms, passive replication and recovery, in order to allow the transparent recovery of active replicas after a fault. Second, we wish to explore the interplay between replicas co-location for resource consolidation, and the resulting dependability and fault masking capabilities of the system.

Acknowledgments. We thank our shepherd, Leonardo Querzoni, and the anonymous reviewers for their comments.

References

[1] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *21st International Conference on Parallel and Distributed Systems*, ser. ICPADS. IEEE, 2015.

[2] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: A real-time IoT benchmark for distributed stream processing platforms," *arXiv preprint arXiv:1701.08530*, 2017.

[3] H. Nasiri, S. Nasehi, and M. Goudarzi, "Evaluation of distributed stream processing frameworks for IoT applications in smart cities," *Journal of Big Data*, vol. 6, no. 1, 2019.

[4] B. Zhou, J. Li, X. Wang, Y. Gu, L. Xu, Y. Hu, and L. Zhu, "Online internet traffic monitoring system using spark streaming," *Big Data Mining and Analytics*, vol. 1, no. 1, 2018.

[5] M. Visser and B. Geerdink, "Fast data at ING – building a streaming data platform with Flink and Kafka," https://www.youtube.com/watch?v=e-_6gijUGAw, 2017.

[6] A. Pande, "A journey of scaling high-volume low-latency stateful processing (ThoughtWorks India)," https://www.youtube.com/watch?v=FzYx567Rm90, 2020.

[7] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *24th ACM symposium on operating systems principles*, ser. SOSP, 2013.

[8] "Apache Storm," http://storm.apache.org.

[9] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at LinkedIn," *Proc. of the VLDB End.*, vol. 10, no. 12, 2017.

[10] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *ACM SIGMOD*, 2015.

[11] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[12] "FlinkForward," https://www.flink-forward.org.

[13] T. Lamirault and M. A. Abdessemed, "A brief history of time with Apache Flink - real-time monitoring and analysis with Flink, Kafka and HBase," https://www.youtube.com/watch?v=Do7C4UJyWCM, 2016.

[14] D. Hardwick, S. Hester, and D. Brelloch, "Dynamically configured stream processing using Flink and Kafka," https://www.youtube.com/watch?v=_yHds9SvMfE, 2016.

[15] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *21st Intl. Conf. on Data Engineering*, ser. ICDE, 2005.

[16] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "An empirical study of high availability in stream processing systems," in *10th ACM/IFIP/USENIX Intl. Conference on Middleware*, 2009.

[17] Y. Wu and K.-L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *31st IEEE International Conference on Data Engineering*, ser. IDCE. IEEE, 2015.

[18] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. of the VLDB End.*, vol. 1, no. 1, 2008.

[19] 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 *8th ACM European Conf. on Computer Sys.*, ser. EuroSys, 2013.

[20] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, feb 1985.

[21] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink®: consistent stateful distributed stream processing," *Proc. of the VLDB End.*, vol. 10, no. 12, 2017.

[22] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, 2005.

[23] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX annual technical conference*, ser. ATC, 2010.

[24] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with Apache Kafka," *Proc. of the VLDB End.*, vol. 8, no. 12, 2015.

[25] M. K. Aguilera and R. E. Strom, "Efficient atomic broadcast using deterministic merge," in *19th annual ACM symposium on Principles of distributed computing*, ser. PODC, 2000.

[26] X. Liu, A. Harwood, S. Karunasekera, B. Rubinstein, and R. Buyya, "E-Storm: Replication-based state management in distributed stream processing systems," in *46th International Conference on Parallel Processing*, ser. ICPP. IEEE, 2017.

[27] A. Martin, C. Fetzer, and A. Brito, "Active replication at (almost) no cost," in *30th Intl. Symposium on Reliable Distributed Systems*, ser. SRDS. IEEE, 2011.

[28] A. Martin, T. Smaneoto, T. Dietze, A. Brito, and C. Fetzer, "User-constraint and self-adaptive fault tolerance for event stream processing systems," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN, 2015.

[29] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "Streamscope: continuous reliable distributed processing of big data streams," in *13th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI, 2016.

[30] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003.

[31] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "Streambox: Modern stream processing on a multicore machine," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 617–629.

[32] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008.

[33] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2004, pp. 263–274.

[34] "Companion repository with code and experimental data." https://github.com/CloudLargeScale-UCLouvain/flink-active-replication.

[35] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *ACM SIGMOD international conference on Management of data*, 2004.

[36] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 1, 2008.