Skipping Unused Events to Speed Up Rollback-Recovery in Distributed Data-Parallel CEP

Guilherme F. Lima and Markus Endler
Department of Informatics
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
{glima,endler}@inf.puc-rio.br

Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel
Institute of Parallel and Distributed Systems
University of Stuttgart
Stuttgart, Germany
{ahmad.slo,sukanya.bhowmik,kurt.rothermel}@ipvs.uni-stuttgart.de

***Abstract*—We propose two extensions for a state-of-the-art method of rollback-recovery in distributed CEP (complex event processing). In CEP, an operator network is used to search for patterns in events streams. Sometimes these operators fail and lose their state. Rollback-recovery is a method for dealing with such state losses. The type of rollback-recovery we consider is upstream backup, where the state of a failed operator is recovered by replaying to it the input events that led it to that state. These events are kept in upstream operators' memory buffers, which are trimmed continuously as the downstream operator progresses. The first extension we propose saves memory and speeds up recovery by avoiding to store and retransmit unnecessary events. The second extension makes the base method of upstream backup compatible with data-parallel CEP, allowing that the windows into which operators partition their input be processed in parallel. We evaluated the proposed extensions through experiments that showed a significant reduction in memory usage and recovery time at the expense of a negligible processing overhead during normal operation.**

***Keywords*-complex event processing; fault-tolerance; reliability; recovery;**

I. Introduction

Complex event processing (CEP) is an efficient and scalable method for searching for patterns in event streams. In CEP, events are processed while they traverse an operator graph. Each operator (node in the graph) takes primitive events as input, looks for a given pattern, and, if any matches are found, produces complex events as output. By interconnecting operators, one can specify the detection of complex patterns in a structured manner. Moreover, as operators do not share state, the operator graph can be easily distributed across multiple machines in an arrangement called distributed CEP.

Distributed CEP is adopted by applications that need to process large volumes of events as fast as possible, with events originating from geographically dispersed sources. Examples of such applications include financial stream analysis, intrusion detection, and sensor-network monitoring [1]. Recently, distributed CEP has also been applied to IoT (Internet of Things) [2] and cyber-physical systems [3].

In many applications of distributed CEP, a critical requirement is that the operator graph behave *reliably* even in the presence of operator failures. That is, the outputs produced by the operator graph as a whole must be consistent and correct at all times, even if during the processing some operators of the graph lose their state and need to be recovered.

There are two basic design options for constructing reliable distributed CEP systems: replication and rollback-recovery. In *replication*, reliability is obtained by running replicas of the original operators in background. The replicas are fed with the same inputs as the original operators and updates are coordinated so that if an operator fails one of its replicas takes over [4], [5]. Although this design minimizes recovery time, it imposes a high resource utilization and message overhead—every operator must be replicated and the events send to an operator must also be streamed to the replicas.

The alternative, less resource intensive design option for reliable distributed CEP is rollback-recovery. In *rollback-recovery*, there are no replicas; if an operator fails, its state is recovered from a previous backup. The nature of the backups gives rise to two strategies for rollback-recovery:

1) In *classical rollback-recovery*, checkpoints of the whole state of the operator are saved to persistent storage at regular intervals. If the operator crashes, its state is retrieved from the latest checkpoint [6].
2) In *upstream backup*, the input events that led the operator to its current state are kept in the operators that precede it in the graph (its upstream neighbors). If the operator crashes, its state is reconstructed by replaying to it the events saved in the upstream neighbors [7].

The main problem of classical rollback-recovery concerns the computation of state checkpoints. The operator state can be large and writing it to persistent storage is slow. Moreover, it is often necessary to interrupt the operator to capture its state, which introduces latency during normal processing.

In upstream backup, in contrast, there are no state captures; only events and lightweight recovery data are saved, and these are are kept in primary memory in the upstream

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Accepted for publication in Proceedings of 2018 IEEE/ACM 5th International Conference on Big Data Computing Applications and Technologies (BDCAT), Zurich, Switzerland, December 17, 2018.

neighbors. A drawback of upstream backup, however, is that the saved events and recovery data must be retransmitted during recovery, which introduces latency (but only during recovery). Besides, the technique assumes that the operator state can be reconstructed from its input, which is often (but not always) the case.

In this paper, we aim at reducing the memory usage and recovery time of the upstream backup strategy in distributed CEP. We do so by avoiding to save and retransmit events that play no role in reconstructing the operator state during recovery, namely, those events that were ignored by the operator during normal processing. Such ignored, or *skipped*, events are common in CEP, especially when some of the incoming events are noise to the pattern searched.

We adopt as a base the method for upstream backup introduced in [8], and extend it to collect and distribute information about skipped events. We have chosen this particular method for two reasons. First, because it has a general operator model which is expressive enough to accommodate most features of modern CEP engines. Second, because it guarantees the correct and complete recovery of failed operators while minimizing the amount of recovery data that is transmitted during normal processing.

One drawback of the method described in [8], however, is that it does not support data-parallel CEP—a common approach for parallelizing the internal processing of CEP operators [9]. In [8], each operator splits its input into overlapping segments, called windows, which are then processed in sequence, one after another. As an additional contribution, we update the base method of [8] by allowing windows to be processed in parallel, which effectively extends the method to support data-parallel CEP. Although this extension has little impact on the operator model, it affects the recovery process in nontrivial ways, as we shall detail.

To evaluate our proposal—the extension of the base method of [8] with support for the collection and distribution of information about skipped events and for parallel processing of windows—we wrote a C library that implements the base method and the proposed extensions from the point of view of a single operator. Using this library, we run experiments that compared the recovery cost of executions that do not use the information about skipped events with the recovery cost of equivalent executions that use it. The experiments indicate that collecting and using the information about skipped events reduces the cost of recovery (i.e., decreases the number of events that need to be stored and replayed during recovery and, consequently, the total recovery time) at the expense of a negligible overhead during normal operation (less than 1% in extra time while recovery data is computed).

The rest of this paper is organized as follows. In Section II, we describe the operator model and the base method of upstream backup we are using. In Section III, we present the proposed extensions to the base method. In Section IV, we describe the implementation and the experiments. In Section V, we discuss some related work. Finally, in Section VI, we present our conclusions and future work.

II. Background

A distributed CEP system can viewed as a directed acyclic graph where the nodes stand for operators and the edges stand for interconnections between operators, through which events flow (see Figure 1). Nodes with no incoming edges are called *sources* and only produce events; nodes with no outgoing edges are called *sinks* and only consume events. We say that events flow from the *upstream* portion of the graph (sources) to the *downstream* portion of the graph (sinks).

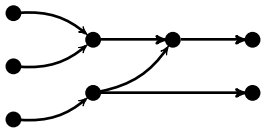

Figure 1: A typical CEP graph.

The intermediate nodes of the graph are event processors. They search for a given pattern in the stream of events arriving on their incoming edges and, if a match is found, produce a corresponding event on their outgoing edges. When we talk about an operator in the sense of event processor, we mean an intermediate node of the graph. From the point of view of such intermediate node, an arriving (input) event is a *primitive event* and an outgoing (output) event is a *complex event*. A complex event embodies a match of the search pattern and usually carries in its payload the primitive events that caused the match.

Besides the payload, every event has a header. The header contains metadata that is used by the system in general, while the payload contains data that is used only by the internal logic of operators. The operator model we adopt (detailed next) assumes as little as possible about this internal logic and only access it through a well-defined interface.

A. The Operator Model

The internal structure of an operator of the CEP graph is depicted in Figure 2. The operator consists of (i) a set of input queues I_1, I_2, …, I_n, one per incoming edge, (ii) a *Splitter* component, (iii) a set of pattern-matching engine instances P_1, P_2, …, P_k, which are associated to windows w_1, w_2, …, w_k, (iv) a *Sequencer* component, and (v) a set of output queues O_1, O_2, …, O_m, one per outgoing edge.

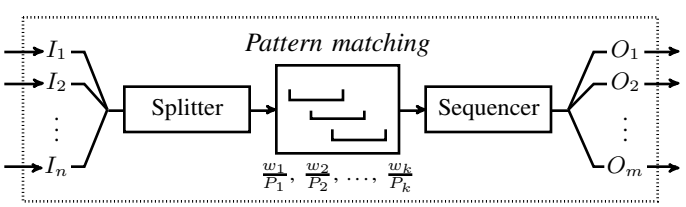

Figure 2: The internal structure of an operator.

The primitive events that arrive from upstream through the incoming edges are stored in input queues I_1, I_2, ..., I_n. The input queues feed the *Splitter*, which partitions their contents into overlapping windows w_1, w_2, ..., w_k. Each window w_i is associated with an instance P_i of the pattern-matching engine, which searches for the same predetermined pattern in w_i and, whenever a match is found, collects the matched (primitive) events into a new complex event and sends it to the *Sequencer*. The *Sequencer* accumulates the complex events it receives and pushes them into output queues O_1, O_2, ..., O_m. Finally, the complex events stored in the output queues are sent downstream through the outgoing edges.

The previous description of the inner workings of an operator was intentionally abstract. The operator model we are using is generic enough to accommodate different operational policies [8]. It has some important requirements though. First, the method used by the *Splitter* to merge the input queues into an event stream and then to split this event stream into windows must be deterministic. That is, upon receiving the same history of events, the *Splitter* must partition them into the exactly same windows.

In practice, common strategies for window partitioning include time-based windows and count-based windows [10]. In time-based windows, the *Splitter* opens and closes windows depending on the time of the incoming events, which is stamped on their header. In count-based windows, the decision to open or close a window depends on the number of incoming events. Both strategies lead to deterministic behavior. Other strategies, such as pattern-sensitive windows [9], are also possible—the only requirement is that the window partitioning process as a whole be deterministic.

A further requirement of the operator model is that the behavior of the pattern-matching instances P_i also be deterministic and depend solely on the content of its associated window w_i. Thus, it must be possible to describe each P_i as a function that given window w_i produces a sequence of complex events.

The last requirement of the operator model concerns the *Sequencer*. Every event in the system carries in its header a sequence number (SN), which is a nonnegative integer that identifies it globally. The *Sequencer* is responsible for assigning sequence numbers to events. While doing this, the *Sequencer* must ensure that sequence numbers are assigned deterministically and in a way that establishes a global (total) order among the events flowing through the system. Any global order is allowed. The only restriction is that the sequence number of events that traverse a same edge of the graph must have increasing values.

B. Acks and Savepoints

We have discussed how the input queues and output queues of an operator are populated—the former with incoming primitive events and the latter with complex events produced internally. We now turn to the discussion of how events are deleted from these queues and of how this process relates to the creation and maintenance of upstream backups.

Primitive events are deleted from the input queues of an operator by the *Splitter* when old windows are dropped. Complex events, in contrast, are deleted from the output queues only after they are explicitly released—or *acknowledged*—by the downstream neighbors of the operator. That is, only after an *ack* message (containing a sequence number greater than or equal to that of the event) is received from all nodes connected to the outgoing edges of the operator.

This deletion-on-ack policy implies that, until being ack'ed, the incoming events of an operator are preserved in the output queues of its predecessors. In other words, the output queues of an operator function as in memory backups for the inputs of its successors in the graph. Thus, as operators behave deterministically, if a successor B of operators A_1, A_2, ..., A_n loses its state, this state can be recovered by replaying to B the events in the output queues of each A_i. This situation is illustrated below.

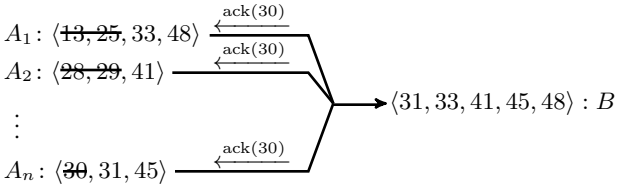

Figure 3: Operator B sends an $\mathrm{ack}(30)$ to A_1, A_2, ..., A_n.

In Figure 3, operator B has just dropped from its input queues all events with sequence number ≤ 30; the only events left are those with sequence numbers 31, 33, 41, 45, and 48. For the sake of the example, we will consider this sequence of events to be the *consolidated state* of operator B. After reaching this state, B sends to A_1, A_2, ..., A_n an $\mathrm{ack}(30)$ message indicating that events with sequence numbers ≤ 30 are no longer needed to restore B to its last consolidated state. These events can thus be safely deleted from the output queues of B's predecessors.

Now, suppose that just after the $\mathrm{ack}(30)$ messages are processed by the A_i's operator B crashes and loses its state. Then, to restore B to its last consolidated state, all we need to do is to replay to it the events that are saved in the output queues of each A_i (the upstream backups). This would put events 31, 33, 41, 45, and 48 back in the input queues of B and, from this point on, the assumption that B's *Splitter* behaves deterministically is sufficient to ensure that the same layout of windows that existed before the crash is recovered.

In general, the consolidated state of an operator consists of four things:

1) The contents of its input queues.
2) The layout and contents of its windows.
3) The state of the associated pattern-matching instances.
4) The contents of its output queues.

The contents of the input queues can be recovered from the upstream backups. The requirement of determinism, ensures that the layout and the contents of the windows can be recovered by rerunning the *Splitter*. Similarly, by rerunning the pattern-matching instances and the *Sequencer*, the complex events which make up the contents of the output queues can also be recovered.

What cannot be recovered, however, is the sequence numbers originally assigned by the *Sequencer* to the complex events. This information, which is effectively the *Sequencer* state, must be saved upstream and retransmitted back to the operator during recovery. One way to reduce the *Sequencer* state is to generate these numbers deterministically from a given initial number. This way the *Sequencer* state becomes a single number: the sequence number of the first complex event to be generated during recovery or of the next complex event (in case no events are generated during recovery). This initial sequence number, call it $\mathrm{SN}_{\text{seed}}$, is sent upstream in the ack message together with the sequence number of the latest event to be deleted from the predecessors' output queues, call it $\mathrm{SN}_{\text{trim}}$. The payload of a complete ack message consists thus of a pair $\langle \mathrm{SN}_{\text{seed}}, \mathrm{SN}_{\text{trim}} \rangle$, which is called a *savepoint*.

We defer the description of the algorithm for computing savepoints to Section III. We anticipate, however, that such computation is triggered by the reception of an ack from downstream. That is, when the operator receives an $\mathrm{ack}(\mathrm{SN}_{\text{seed}}, \mathrm{SN}_{\text{trim}})$ from downstream, it saves the received $\mathrm{SN}_{\text{seed}}$, trims its output buffers using the received $\mathrm{SN}_{\text{trim}}$, and checks if any of the primitive events that comprise the ack'ed complex event ($\mathrm{SN}_{\text{trim}}$) can be dropped from its input queues. If so, the operator computes a new savepoint and sends it upstream in a new ack message. This means that ack messages propagate through the system from sinks to sources—each ack may cause the generation of lower-order acks, which generate further acks, and so on, until the sources are reached.

C. Additional Features

In Sections II-A and II-B, we described the core features of the method for upstream backup of [8]. Although these core features are sufficient for explaining our proposal, the complete method has some other features which are required to ensure correct and complete recovery.

We will not detail these other features, as that would be outside the scope of this paper, but we summarize the main ones next. So, in addition to what we have discussed, in [8]:

- A system model establishes assumptions about the behavior of the underlying hardware and network. Among other things, the system model postulates that the communication channels guarantee eventual in-order delivery of events and that a monitoring component is responsible for identifying and recovering failed nodes.
- The *Splitter* component does not need to be stateless. In [8], the authors discuss the possibility of extracting and saving the state of the *Splitter* upstream together with the state of the *Sequencer*.
- The $\mathrm{SN}_{\text{seed}}$ of the last savepoint sent upstream is replicated in *all* ancestors of the operator and not only in its immediate predecessors. This is necessary to recover from simultaneous failures on adjacent operators.

III. The Proposal

We proceed to describe the proposed extensions. For simplicity, but without loss of generality, we will do so from the point of view of a hypothetical operator, say A. Operator A is a node of the graph with exactly one incoming edge and one outgoing edge. The upstream neighbor of A (the predecessor) feeds A with events and receives from A acks concerning these events. Similarly, A feeds its downstream neighbor (the successor) with events and receives acks from it.

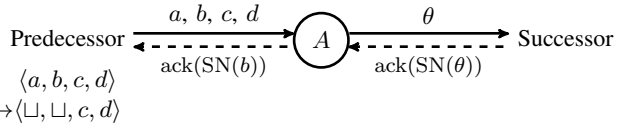

Figure 4: An exchange between A and its neighbors.

A typical exchange between operator A and its neighbors is depicted in Figure 4. The exchange consists of five stages:

1) The predecessor sends to A events a, b, c, and d. These are primitive events from A's perspective which are backed up in the predecessor's output queue.
2) After receiving and processing events a, b, c, and d, operator A finds a match and produces a new complex event θ. Operator A then pushes event θ into its output queue and sends θ downstream to the successor.
3) After some time, the successor acknowledges event θ, i.e., sends to A an ack message in which $\mathrm{SN}_{\text{trim}} = \mathrm{SN}(\theta)$. (In Figure 4, only the $\mathrm{SN}_{\text{trim}}$ component of ack messages is shown; the $\mathrm{SN}_{\text{seed}}$ component is omitted.)
4) Upon receiving this ack, operator A saves the $\mathrm{SN}_{\text{seed}}$ component, deletes θ and any previous event from the output queue, and computes a new savepoint. That is, A looks into the events that make up θ for the latest primitive event that is no longer necessary, which becomes $\mathrm{SN}_{\text{trim}}$; then it looks into the output queue for the first complex event produced after $\mathrm{SN}_{\text{trim}}$, which becomes $\mathrm{SN}_{\text{seed}}$. Operator A then puts the computed savepoint into a new ack message and sends this message upstream to the predecessor.
5) Finally, after the predecessor receives from A the ack, in which $\mathrm{SN}_{\text{trim}} = \mathrm{SN}(b)$, the predecessor deletes from its output queue any events with sequence numbers less than or equal to $\mathrm{SN}(b)$, namely, events a and b. Thus the predecessor's output queue, which at stage (1) was $\langle a, b, c, d \rangle$, is trimmed down and becomes $\langle c, d \rangle$.

The first extension we propose makes the trimming of output queues more aggressive. The idea is to delete not

only a prefix of the output queue, as described in stage (5) above, but to go beyond the prefix determined by $\mathrm{SN}_{\mathrm{trim}}$ and delete also events that were skipped by operator A in stage (2), while A was searching for a match. We describe this extension in detail Section III-A.

The second extension we propose concerns the internal operation of A during stage (2). It is not exactly an extension but a fix—a modification in the structure and attribution process of sequence numbers. This fix allows that the windows into which the operator splits the incoming events be processed in parallel. Without this fix, these windows must be processed in series, one after another. That is, first the events that comprise a window are accumulated until the window is complete; then these events are fed into the associated pattern-matching engine, which also runs to completion; and only after that the next window is processed.

The problem here is that while it waits for the current window to complete the operator does nothing but introduce delay. The more efficient alternative, called data-parallel CEP [9], is to feed each incoming event to all open windows and corresponding pattern-matching instances at once, in parallel. Without the proposed fix, however, such parallel processing of windows can lead to inconsistencies during recovery. We discuss these inconsistencies, the proposed fix, and its implications in Section III-B.

A. Deleting Skipped Events

To collect information about skipped events we need first to extend the pattern-matching engine interface to inform the operator, for each event fed, whether or not the event was used. In CEP, unused (or *skipped*) events are common. For instance, suppose that the operator is searching for pattern "abc" in the incoming events, i.e., an event of type a followed by an event of type b followed by an event of type c, possibly with events of other types in between. Clearly, any events that are not of types a, b, or c, will be skipped. This strategy of search, in which events that do not contribute to the search are ignored, is called *skip-till-next-match* and is frequently adopted in CEP [11].

One way to search for pattern "abc" following a skip-till-next-match strategy is to use an automaton, such as the one depicted in Figure 5. The automaton starts at state 1 and, for each event fed, either collects the event and advances its state, or skips the event and remains in the same state. If state 4 (the final state) is reached, the automaton completes the match: it gathers all events it has collected into a new complex event and returns this complex event to the user (i.e., the operator).

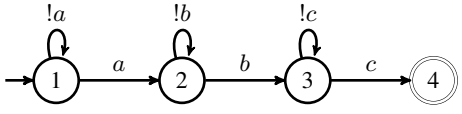

Figure 5: An automaton that detects the pattern "abc".

In Figure 5, an input event is collected by the automaton, and hence used, if the event triggers one of the transitions labeled a, b, or c; otherwise, if it triggers one of the transitions labeled $!a$, $!b$, or $!c$, the event is skipped. Thus, for example, if we feed the automaton with the sequence of events

$$a_1,\ a_2,\ a_3,\ b_4,\ a_5,\ b_6,\ d_7,\ d_8,\ b_9,\ c_{10}, \qquad (\dagger)$$

events a_1, b_4, and c_{10} will be collected into a match and events a_2, a_3, a_5, b_6, d_7, d_8, and b_9 will be skipped. To inform the operator about the skipped events, all the automaton has to do is check which transition each event has triggered.

Besides detecting events that were skipped, the operator also needs store their sequence numbers. We can do this efficiently by storing not the individual numbers but ranges (intervals) of skipped sequence numbers. These *skip ranges* can be represented by an array whose elements are pairs of numbers, where the first and second members of the pair denote the start and end of the range. Using this encoding, after feeding the sequence (†) to the automaton of Figure 5, the operator will have collected as skip ranges the array $\langle\langle 2, 3\rangle, \langle 5, 9\rangle\rangle$, which indicates that events with sequence numbers in the intervals 2–3 and 5–9 were skipped.

The skip ranges collected by the operator are sent upstream in the ack message as part of its payload. This means that, with the skip extension, the savepoint becomes a triple $\langle \mathrm{SN}_{\mathrm{seed}}, \mathrm{SN}_{\mathrm{trim}}, v_{\mathrm{skip}}\rangle$, where $\mathrm{SN}_{\mathrm{seed}}$ and $\mathrm{SN}_{\mathrm{trim}}$ retain their original meaning and v_{skip} is the value of the skip-ranges array at the moment the savepoint was computed. Thus, the operator is assumed to keep a skip-ranges array at all times, which it updates continuously, with each event skipped, and which it clears whenever a new savepoint is computed.

Finally, when an upstream operator receives an ack with a savepoint of the form $\langle \mathrm{SN}_{\mathrm{seed}}, \mathrm{SN}_{\mathrm{trim}}, v_{\mathrm{skip}}\rangle$, it saves $\mathrm{SN}_{\mathrm{seed}}$, uses $\mathrm{SN}_{\mathrm{trim}}$ to delete a prefix from its output queues, and uses v_{skip} to delete further events from the output queues. Note that it is completely safe to delete these further events from the output queues. In an eventual recovery, as these events were skipped in the original run, and as operators behave deterministically, if retransmitted, these events would be skipped once again, and would not affect the state of the downstream operator. Therefore, by not storing nor retransmitting them, we save space and time.

That said, there are cases in which an event must be preserved even if it was skipped by the pattern-matching engine. This is the case, for example, of an event that causes windows to open or close inside the operator. Even when such event does not contribute to the pattern searched, it must be preserved to ensure that during recovery the *Splitter* produces the same windows it has produced originally.

B. Processing Windows in Parallel

In this section, we describe a modification in the structure and attribution process of sequences numbers that makes the method of upstream backup of [8] compatible with

data-parallel CEP while preserving its properties (correct and complete recovery). Before describing the modification, however, we first detail the inner workings of an operator, including the computation of savepoints, and discuss the problem itself, i.e., why recovery does not work with the original method when windows are processed in parallel.

1) Inside the operator: When an operator receives an event, the first thing it does is to determine how this event affects its window layout—which old windows it will close (if any) and whether it will open any new windows. After that, the operator feeds the received event to the pattern-matching instance associated with each open window. Then it (i) collects any complex events produced, (ii) stamps them with a new sequence number, (iii) pushes them into the output queue—or output queues, in case there is more than one—and finally (iv) sends the complex events downstream.

To make matters concrete, consider the example depicted in Figure 6 below. Operator A in the figure is searching for the pattern "ef" in the incoming stream using count-based, overlapping windows. The windows have a fixed size (five events) and a fixed slide (two events). The size determines the number of events each window will contain and the slide determines the distance between consecutive windows. The contents of the windows of A are depicted below the circle that represents the operator. (Although in the figure the windows look like event containers, this is not the case; windows are just logical sections of the input queue.)

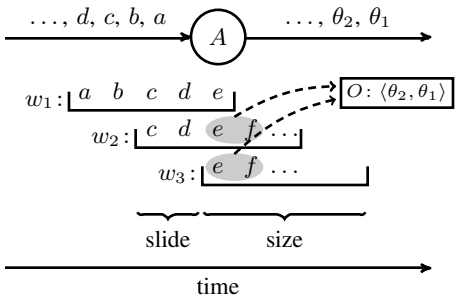

Figure 6: The inner workings of operator A.

At the start of its execution, operator A of Figure 6 has no windows and its output queue (O) is empty. When A receives the first incoming event, a, it opens a new window w_1 and feeds a to it, i.e., to the associated pattern-matching instance (not shown in the figure), which skips the event. The second incoming event, b, causes no windows to open and is fed to all open windows (only w_1 at this point) which skips it. The third incoming event, c, causes window w_2 to open and is fed to w_1 and w_2, both of which skip it. This process goes on until event e, the fifth incoming event, is received.

Upon receiving event e, operator A opens a new window, w_3, and feeds e to w_1, w_2, and w_3, which are all open at this point. This time e is collected by the three windows, but no complex event is produced as the underlying matches are incomplete. The next incoming event, f, will change that.

Upon receiving event f, operator A closes window w_1 (which means that w_1 will be kept in memory but will no longer be fed with events) and feeds event f to the only windows open at this point, w_2 and w_3. Both windows, w_2 and w_3, collect event f, complete the underlying matches, and generate the complex events θ_1 and θ_2. Operator A then stamps these complex events with newly generated sequence numbers, pushes them onto the output queue O, and sends them downstream. Finally, after that, operator A is ready to process the next incoming event.

The crucial moment in the previous example is the generation and attribution of sequence numbers to complex events θ_1 and θ_2. Although not explicitly mentioned, in the description of the example we already considered that windows were being processed in parallel—when given an event, operator A feeds it to all open windows, i.e., it does not wait for the earlier open window to complete before processing later windows. As long as the result is deterministic, there is no problem with that. Processing windows in parallel is completely safe during normal operation. The problem occurs when a savepoint is computed, sent upstream, and then restored during an eventual recovery, as we shall see next.

2) Computing a savepoint: Suppose that during its execution, an operator A (not the same A of Section III-B1) receives an ack acknowledging one of its output events, θ, and that at this point A has a window layout such as the one depicted in Figure 7. After receiving the $\mathrm{ack}(\mathrm{SN}(\theta))$ message, operator A removes θ from the output queue together with any events preceding it, and then computes a new savepoint.

To compute a savepoint, A has to determine $\mathrm{SN}_{\mathrm{seed}}$, $\mathrm{SN}_{\mathrm{trim}}$, and v_{skip}. Let's ignore $\mathrm{SN}_{\mathrm{seed}}$ and v_{skip} for now and concentrate on $\mathrm{SN}_{\mathrm{trim}}$—the sequence number of the event that defines the prefix to be trimmed of the upstream backups.

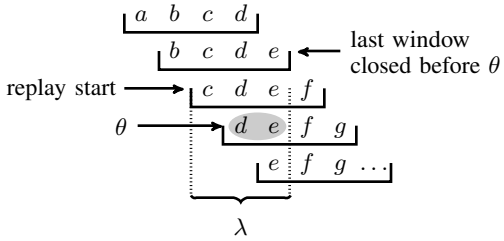

Figure 7: Window layout of operator A when θ is ack'ed.

To determine $\mathrm{SN}_{\mathrm{trim}}$, operator A has to find the first event that should be replayed to restore A to the point it was immediately after event θ was generated. We call this event the *replay start*. Let w be the last window closed before θ was generated, i.e., the last window closed before the last primitive event that comprises θ was matched. As closed windows cannot generate complex events, everything that happened in w, by definition, happened before θ, as w was

closed when θ was generated. So, the replay-start event is the first event of the first window after w. And, consequently, $\mathrm{SN}_{\text{trim}}$ is the sequence number of the event that precedes the replay-start event.

In Figure 7, window w (the last window closed before θ) is the second window from the top and the replay-start event (the first event in the first window after w) is event c. So, $\mathrm{SN}_{\text{trim}}$ (the sequence number of the event that precedes c) is $\mathrm{SN}(b)$.

We call the sequence of events between the replay start event and the last event in w the *replay interval*, denoted λ. If operator A loses its state and we replay to it all events in λ, we will get *all* complex events that were generated immediately after θ and possibly some complex events that were generated before θ, including θ itself. Events of the first kind are those effectively recovered by the replay, as they have not been ack'ed by A's successors. Events of the second kind are artifacts of the recovery process—they are duplicates that already have been ack'ed and must be discarded by the successors. (To do this the successors need only to drop events with old sequence numbers.)

Back to the savepoint, once the replay start is established, $\mathrm{SN}_{\text{seed}}$ and v_{skip} follow naturally. $\mathrm{SN}_{\text{seed}}$ is the sequence number of the first complex event that will be generated during the replay, i.e., the sequence number of the earlier complex event in λ occurring in the windows after w. (Note that $\mathrm{SN}_{\text{seed}}$ is always defined; in the worst case, the case in which only θ happens to be in λ, $\mathrm{SN}_{\text{seed}} = \mathrm{SN}(\theta)$.) And v_{skip} is the current skip-ranges array trimmed so that no range in the array starts earlier than $\mathrm{SN}_{\text{trim}}$.

Finally, after computing the savepoint, operator A sends it upstream in a new ack message and then drops (deletes from memory) window w and any earlier windows. Any information in these windows is no longer needed—w and any preceding windows are not going to be regenerated in an eventual recovery (e.g., in Figure 7, a replay would start with event c, which opens the third original window) so there is no point in keeping them. It is also at this moment that events are deleted from the input queue, in case they do not belong to any of the windows kept.

3) The problem: The fact that old closed windows are dropped while computing the savepoint implies that any complex events produced by these windows will not be produced during recovery. As a consequence, if the operator uses simple increments to generate sequence numbers for complex events (or any other method that depends on the number of events produced), it may happen that during recovery some complex events get stamped with the wrong numbers. This is better explained by example.

Consider the window layout depicted in Figure 8a. In this layout, every primitive event is matched as soon as it is feed to any window, producing a corresponding complex event. The sequence numbers of the complex events produced are show in superscripts. So, in the figure, event a is fed to the first window, is collected, and produces complex event 1. Event b is fed to the first and second windows, is collected by both, and produces complex events 2 and 3, and so on. Note that in this example windows are being processed in parallel and the sequence number of complex evens is generated incrementally, by adding one to the sequence number of the last complex event.

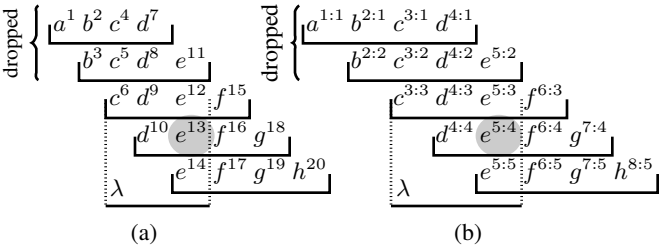

Figure 8: An extreme case.

Suppose that at some point the operator whose windows are depicted in Figure 8a, call it A, receives an ack message for the complex event 13, whose match is highlighted in the figure. After receiving this ack, operator A computes a new savepoint S, where $\mathrm{SN}_{\text{trim}} = \mathrm{SN}(b)$ and $\mathrm{SN}_{\text{seed}} = 6$, drops the first two windows, and sends S upstream.

Now suppose that A loses its state and is restored from S. Then A's *Sequencer* will be initialized with $\mathrm{SN}_{\text{seed}}$ ($=6$) and the primitive events c, d, e, f, and g, which were backed up upstream, will be replayed to A. Event c, when received, causes the first window to open and produces complex event 6. The next event, d, causes a second window to open. When d is fed to the first window it produces complex event 7 ($=\mathrm{SN}_{\text{seed}}+1$) and when it is fed to the second window it produces complex event 8 ($=\mathrm{SN}_{\text{seed}}+2$). Although the contents of these events (i.e, their matches) are correct, their sequence numbers (7 and 8) are wrong. Both of the events were stamped with sequence numbers that are different from those they had before A lost its state (9 and 10).

The problem is that the complex events 7 and 8 that existed in the first two windows in the original window layout (Figure 8a) are not produced in the replay, as their windows are not reconstructed. Thus, the *Sequencer* misses some of the $\mathrm{SN}_{\text{seed}}$ increments, and the original events 9 and 10 and others, get stamped with a different (wrong) sequence number after recovery. The problem is subtle but its solution is simple.

4) The fix: Instead of using nonnegative integers as sequence numbers, we will use pairs of nonnegative integers of the form $x{:}y$, where x is the number of the complex event (incremented per parallel iteration) and y is the number of the window in which the complex event was produced. For instance, Figure 8b depicts the same window layout of Figure 8a but with the original sequence numbers replaced by $x{:}y$ pairs.

In Figure 8b, the process of attribution of sequence numbers can be described as follows. Event a is fed to the first window and produces complex event 1:1. In this case, $x = 1$ as this is the first iteration in which an event was produced, and $y = 1$ as this is the first window. Next, event b is fed to the first and second windows which produce complex events 2:1 and 2:2. In both cases, $x = 2$, as this is the second iteration in which an event was produced, but $y = 1$ in the first event, as it was produced in the first window, and $y = 2$ in the second event, as it was produced in the second window. This process is repeated for the next incoming events until the layout of Figure 8b is reached.

As before, if we suppose that the same complex event (now with sequence number 5:4 and also highlighted in Figure 8b) is ack'ed, the operator will compute exactly the same savepoint S but with $\mathrm{SN}_{\text{seed}} = 3{:}3$. Next, once more, the operator will drop the first and second windows and will send S upstream in the payload of an $\mathrm{ack}(\mathrm{SN}(b))$ message. What is different now is that if the operator loses its state and savepoint S is restored, the *Sequencer* will be initialized with $\mathrm{SN}_{\text{seed}} = 3{:}3$ and during the replay will stamp the complex events produced with the correct pairs.

In order to see this, note that once again the replay will consist of events c, d, e, f, and g. The first event, c, when received by the operator, will cause the first window to open, but this will be window number 3 as $\mathrm{SN}_{\text{seed}} = 3{:}\underline{3}$. Accordingly, event c's match will be stamped with the pair 3:3, i.e., the third (restored) iteration in which a complex event is produced and the third window.

The next event in the replay, event d, will cause second window (number 4) to open and will be fed to the first and second windows, producing a complex event in both. Notice that these two matches were numbered incorrectly in the version of the example discussed in Section III-B4. This time, however, as this is now the fourth iteration in which a complex event is produced, event d's match in the first window will be numbered 4:3 and its match in the second window will be numbered 4:4. Both pairs are correct—they agree with the original pairs attributed to these matches, and the same applies to the matches of subsequent events.

5) Pros and cons: The main advantage of the proposed fix is that it is simple and has little impact in the operator model of [8]. We only need to replace the scalar sequence numbers by pairs of numbers, and to update the *Splitter* and *Sequencer* components to work with these pairs.

One drawback of the pairs, however, is that although they induce a total order, there is no way to tell the distance (in number of events) between two of them. That is, from the lexicographical order of pairs, we know that event 4:3 was produced before event 4:4 and that 4:4 was produced before 5:3. But just by looking at the pairs it is impossible to know how many events were produced between the pairs 4:3 and 5:3. With scalar, incremental sequence numbers this was easy (e.g., there are four integers between 7 and 10 inclusive).

The fact that such arithmetic is meaningless for pairs makes them unsuitable for delimiting count-based windows. (Note that this a problem only during recovery—during normal operation the incoming events can simply be counted.) One way to cope with this problem is to store in the savepoint, and consequently in upstream backups, information about which events caused windows to open downstream, and then use this information to split windows during recovery. This idea can also be used to deal with more esoteric window splitting policies.

IV. Evaluation

To evaluate our proposal we wrote a C library, called libgem[1], which implements the operator model and the v_{skip} extension discussed in the previous sections. The library is only concerned with the internal processing of a single operator and, as such, it offers no support for networking or coordination of distributed operators.

Libgem's operator API consists of the following functions:

1) $new\,(p, w_{\text{size}}, w_{\text{slide}}, F) \rightarrow A$. Creates a new operator A with pattern-matching engine p, window size w_{size}, window slide w_{slide}, and flags F. The library uses state machines for pattern-matching and time-based windows with fixed size and slide; one of the flags is the *skip flag*, which enables or disables the detection of skipped events.
2) $push\,(A, e)$. Pushes primitive event e to operator A, i.e., pushes e into A's input queue and process it, storing any complex event produced in A's output queue.
3) $pop\,(A) \rightarrow \theta$. Pops a complex event θ from operator A. This function simply returns next complex event in A's output queue (or null in case the queue is empty).
4) $save\,(A, \theta) \rightarrow S$. Computes a savepoint S from a previously popped event θ. This call is the effectuation of an ack for θ. It causes A to trim its output queue, drop any old windows, and run the algorithm for computing savepoints of Section III-B2. The function returns the resulting savepoint S containing a $\mathrm{SN}_{\text{trim}}$, $\mathrm{SN}_{\text{seed}}$, and v_{skip} (if the skip flag is on).
5) $restore\,(A, S)$. Restores A with savepoint S, i.e., resets the operator and re-initializes it with the $\mathrm{SN}_{\text{seed}}$ of S.

For the experiments, detailed next, we wrote an execution environment that wraps a libgem operator A and simulates exchanges between A and its neighbors, analogous to those of Figure 6. The execution environment collects statistics about the operator and is responsible for (i) generating random primitive events, saving them in a simulated upstream backup, and pushing them into operator A; (ii) popping the complex events produced by A and simulating their acknowledgment, i.e., triggering the generation of savepoints; and (iii) from the savepoints, simulating the acks that are sent upstream, i.e., updating the contents of the simulated upstream backup.

[1] https://gitlab.com/gflima/gem

Experiments

In each experiment we compared two runs of the same operator executing the same failure-recovery cycle. First with the skip flag off (not collecting/using information about skipped events) and then with the skip flag on. By the *same* failure-recovery cycle, we mean that in both runs the same primitive events were consumed and, consequently, the same complex events were produced. By a *failure-recovery cycle*, we mean that one failure is introduced in the middle of the processing, followed by a complete recovery.

Thus, in each run the operator would execute normally until 50% of the input was processed. Next, the execution environment would reset the operator and restore the last savepoint sent upstream. Finally, the execution environment would replay the events in the upstream backup, recovering the operator and resuming its normal processing.

In all runs, one million input events were processed and the pattern searched was "$abcde$" (skip-till-next-match) with the possible event types varying from a to j (a total of 10 types). The windows had a fixed size of $w_{\text{size}} = 1000$ time units and we considered two variations for the window slide, $w_{\text{slide}} = 50$ and $w_{\text{slide}} = 800$ time units.

We considered five variations for the delay between the production of a complex event and its ack: 8, 16, 32, 64, and 128. This number, the *save interval* (S_i), determines the number of acks the execution environment accumulates before triggering the computation of a savepoint. For instance, with a save interval of 8, the execution environment waits for the acknowledgment of eight complex events before calling $save\,(A, \theta)$ where θ is the first of these complex events. A lower save interval value implies more frequent *save* calls.

The results of the experiments are shown in Figure 9. In the figure, each pair of bars in one of the graphics compares a run with the skip flag off (white bar) and the same run with the skip flag on (crossed bar). The graphics on the left-hand side of the figure, (a), (c), (e), and (g), show the experiments where $w_{\text{slide}} = 50$, and those on the right-hand side, (b), (d), (f), and (h), show the experiments where $w_{\text{slide}} = 800$. In all experiments, the values for pairs of runs are an average of 50 executions with different random seeds.

As shown in (a) and (b), when saves are more frequent, there is a significant reduction in recovery time when the skip flag is on, regardless of the value of w_{slide}. For instance, with the save interval $S_i = 8$, (a) and (b) show a reduction of 57% and 43% in recovery time. The gain is smaller for bigger S_i's because it takes more events and, consequently, more time to recover from an old savepoint. The comparison in number of events replayed is shown in (e) and (f). Again, with the skip flag on, fewer events were replayed in general, and even fewer when saves are more frequent.

If fewer events were replayed with the skip flag on, then it should be the case that fewer events were kept in the upstream backup. Indeed, this is confirmed by (g) and (h) which show a reduction in the maximum size (in number of events) of the upstream backup with the skip flag on—fewer events were kept, which means less memory was used.

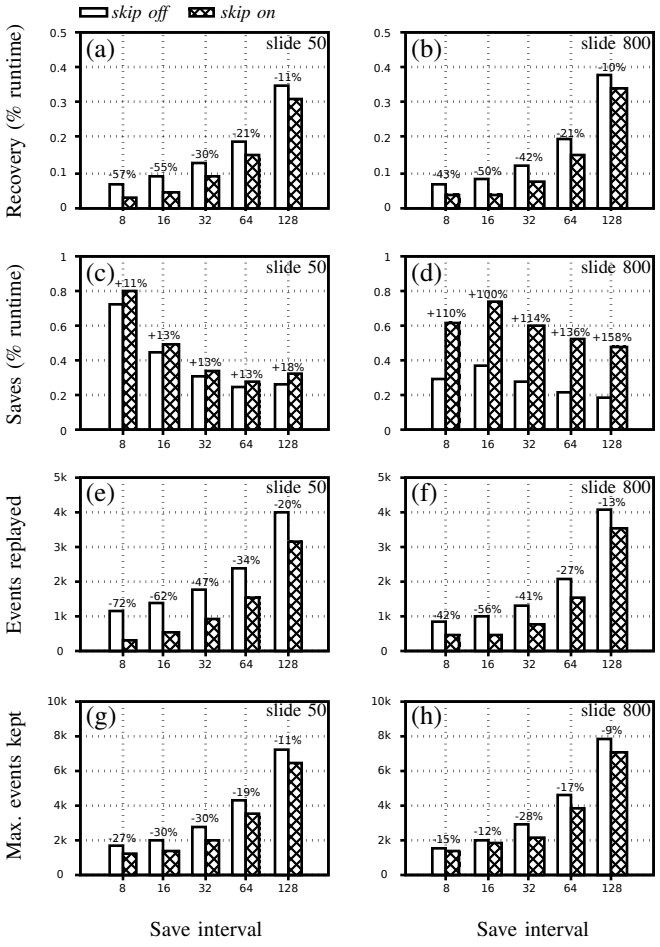

Figure 9: Experimental results.

The costs associated with this economy of space and time are shown in (c) and (d), which depict the time the operator spent computing savepoints in relation to the total time. As expected, with the skip flag on, the computation of savepoints gets more expensive, as the skip-ranges array v_{skip} now needs to be allocated and filled. This computation seems to get even more expensive in (d) where $w_{\text{slide}} = 800$. In this case, the time spent computing savepoints practically doubles. This difference, however, is just an artifact.

On average, the experiments with $w_{\text{slide}} = 50$ take about 14 times more to run than those with $w_{\text{slide}} = 800$; in the experiments with $w_{\text{slide}} = 50$ there are 21 windows open simultaneously (which means 21 active pattern-matching instances) while in those with $w_{\text{slide}} = 50$ there are only 2 windows open at any given time. Hence, any time comparison between these experiments is meaningless. The important thing about the numbers in (c) and (d) is that they are all below 1% of the total running time. The overhead of saves is thus negligible independently of the value of the skip flag.

We conclude this section with an observation that might guide future optimizations. Consider the graphics (a) and (b) and the graphics (e) and (f). Although the difference between the bars get smaller when S_i increases, their actual difference is almost constant, which means that when the operator fails and is restored it skips almost the same number of events, regardless of S_i. This happens because the only events skipped during recovery are those in v_{skip} when the savepoint was computed. Events that arrived after this savepoint will not be trimmed, even though the operator might have skipped them and is waiting for an ack to send this information upstream. One way to improve this is to send the information about the skipped events upstream more frequently, in a new type of message, and not only when an ack is received from downstream and a savepoint is computed.

V. Related Work

The idea of upstream backup for recovery can be traced back to [12] and [7]. These works use the term "repeating recovery" to refer to approaches where the outputs of a restored operator are the same as those of the original operator. The base method of [8] is a method for repeating recovery.

Less strict approaches to recovery also exist. For instance, some applications can tolerate inaccuracies in the output of restored operators, which characterizes them as requiring only partial fault-tolerance [13]. Methods for partial fault-tolerance usually permit that inaccuracies be corrected later through revoke messages [14]. However, in the scenarios we consider, such as real-time financial analysis and intrusion detection, it is often too costly to accommodate late corrections.

Regarding the recovery technique, an alternative to upstream backup is replication, which can be active or passive. In active replication [15], the execution operators is replicated together with the messages they exchange. This speeds up recovery but imposes high processing and network overheads. Passive replication [16] sacrifices recovery time to avoid the runtime overhead, but the general problems remain the same.

The alternative to replication is checkpoint-based rollback-recovery. Upstream backup is in fact a lightweight form of rollback-recovery; it reduces the size of checkpoints to a minimum, and thus avoids the costs of heavyweight state-extraction algorithms of classical rollback-recovery [6].

Some systems support several fault-tolerance schemes, mixing replication with rollback-recovery [17]. Any system that uses event logs to restore operators can benefit from the idea of skipping unused events discussed in this paper.

VI. Conclusion

In this paper, we discussed two extensions for a state-of-the-art method for rollback-recovery via upstream backup in distributed CEP. The first extension speeds up recovery and saves memory by avoiding to store and retransmit unnecessary events, and its cost is negligible (at least from the point of view of the operator). The second extension adds support for data-parallel CEP to the base method adopted in the paper, while preserving its properties (correct & complete recovery).

We are currently investigating further ways to improve the base method and the proposed extensions. One possible improvement is the decoupling of the notifications for acknowledging events from those for trimming the upstream buffer, as discussed at the end of Section IV.

Acknowledgment

We thank Dr. Ruben Mayer of Technical University of Munich for his feedback on the proposed extensions during the development of this work in IPVS/University of Stuttgart.

References

[1] P. Carbone, G. E. Gévay, G. Hermann, A. Katsfodimos, J. Soto, V. Markl, and S. Haridi, "Large-scale data stream processing systems," in *Handbook of Big Data Technologies*, 2017.

[2] A. Akbar, F. Carrez, K. Moessner, J. Sancho, and J. Rico, "Context-aware stream processing for distributed IoT applications," in *IEEE 2nd World Forum on Internet of Things*, 2015.

[3] N. Mao and J. Tan, "Probabilistic event processing with negation operators in cyber physical systems," in *IEEE Int. Conf. Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2015.

[4] M. Volz, B. Koldehofe, and K. Rothermel, "Supporting strong reliability for distributed complex event processing systems," in *13th IEEE Int. Conf. High Performance Computing and Communications*, 2011.

[5] A. Brito, C. Fetzer, and P. Felber, "Multithreading-enabled active replication for event stream processing operators," in *28th IEEE Int. Symp. Reliable Distributed Systems*, 2009.

[6] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, 2002.

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

[8] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz, "Rollback-recovery without checkpoints in distributed event processing systems," in *Proc. 7th Int. Conf. Distributed Event-based Systems*. ACM, 2013.

[9] R. Mayer, B. Koldehofe, and K. Rothermel, "Predictable low-latency event detection with parallel complex event processing," *IEEE Internet of Things Journal*, vol. 2, no. 4, 2015.

[10] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, 2006.

[11] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proc. Int. Conf. Management of Data*. ACM, 2008.

[12] J.-H. Hwang, M. Balazinska, A. Rasin, M. Stonebraker, and S. Zdonik, "A comparison of stream-oriented high availability algorithms," Brown University, Tech. Rep. CS-03-17, 2003.

[13] N. Bansal, R. Bhagwan, N. Jain, Y. Park, D. Turaga, and C. Venkatramani, "Towards optimal resource allocation in partial-fault tolerant applications," in *IEEE 27th Conf. Computer Communications*, 2008.

[14] J.-H. Hwang, S. Cha, U. Cetintemel, and S. Zdonik, "Borealis-r: A replication-transparent stream processing system for wide-area monitoring applications," in *Proc. Int. Conf. Management of Data*. ACM, 2008.

[15] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, 1990.

[16] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems*, 2nd ed. ACM Press, 1993.

[17] 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 IEEE/IFIP Int. Conf. Dependable Systems and Networks*, 2015.