# Rollback-Recovery without Checkpoints in Distributed Event Processing Systems

Boris Koldehofe
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
boris.koldehofe@ipvs.uni-stuttgart.de

Ruben Mayer
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
ruben.mayer@ipvs.uni-stuttgart.de

Umakishore Ramachandran
College of Computing
Georgia Tech, USA
rama@cc.gatech.edu

Kurt Rothermel
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
kurt.rothermel@ipvs.uni-stuttgart.de

Marco Völz
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
voelzmo@gmail.com

## ABSTRACT

Reliability is of critical importance to many applications involving distributed event processing systems. Especially the use of stateful operators makes it challenging to provide efficient recovery from failures and to ensure consistent event streams. Even during failure-free execution, state-of-the-art methods for achieving reliability incur significant overhead at run-time concerning computational resources, event traffic, and event detection time. This paper proposes a novel method for rollback-recovery that allows for recovery from multiple simultaneous operator failures, but eliminates the need for persistent checkpoints. Thereby, the operator state is preserved in *savepoints* at points in time when its execution solely depends on the state of incoming event streams which are reproducible by predecessor operators. We propose an expressive event processing model to determine savepoints and algorithms for their coordination in a distributed operator network. Evaluations show that very low overhead at failure-free execution in comparison to other approaches is achieved.

## Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; C.4 [**Performance of Systems**]: Fault tolerance

## Keywords

Reliability, Recovery, Complex Event Processing

## 1. INTRODUCTION

*Event processing systems*, also commonly referred to as stream processing or complex event processing (CEP) systems, are nowadays deployed in many business applications including logistic chains, manufacturing, or stock exchange. They allow to integrate and analyze streams of events that stem from many distributed data sources such as sensors. Consumers are provided with event streams that capture correlations of the incoming event streams and this way provide feedback and even trigger interactions with physical processes.

With the increasing scale and inherent distributed deployment of data sources, the paradigm of distributed event processing systems has gained increasing importance. In a distributed event processing system, operators hosted at potentially many different nodes of the network are taking a share in analyzing input streams and producing streams of outgoing events. Since many physical processes, e.g., the control of a manufacturing process, depend on the output of event processing systems, their correctness and performance characteristics are of critical importance. For event processing systems, this imposes strong requirements with respect to availability and consistency of their outgoing streams. In particular, the event streams provided to consumers of event processing systems should be indistinguishable from an execution in which the hosts of some operators fail or event streams are not available during a temporary partitioning of the network.

The efficiency of reliable event processing can be measured with respect to its *runtime overhead* in a failure-free execution as well as its *recovery overhead* in the presence of failures. Currently, dealing with reliability leaves two basic options for event processing systems, known as replication and rollback-recovery. While active replication [18] minimizes the time to deal with host and communication failures, it imposes high processor utilization on the hosts at run-time since the execution of every operator needs to be replicated. Replication also raises significantly the message overhead since event streams targeted to an operator must also be streamed to all of its replicas. Passive replication [5] has slightly different properties, sacrificing recovery-time in order to avoid run-time overhead, but the general problems remain the same. Rollback-recovery [8], on the other hand, requires in its classical form to store checkpoints at regular times to persistent storage. This adds additional run-time overhead regarding bandwidth that is needed to transfer (incremental) state information, which is a burden especially for high bandwidth streams. Furthermore, to ensure the atomic capturing of (incremental) checkpoints, the processing of operators needs to be interrupted inducing event detection latency. Given that an operator state even for simple processing such as aggregation may comprise several Gigabytes [19], minimizing the state for performing a recovery is one of the important research questions in providing large-scale event processing systems. A promising way towards avoiding the need for persistent checkpoints is to recover the state of an operator by replaying logs of incoming event streams [10, 11] (known as “upstream backup”). Yet, the approach is very restrictive regarding operators for which a consistent state can be guaranteed after a recovery.

In this paper, we propose a method for recovery which avoids any interruption of the event processing system and minimizes the amount of state to be transferred in a failure-free execution. The proposed approach relies on the observation that at certain points in time, the execution of an event processing operator solely depends on a distinct selection of events from the incoming streams. So, the operator state only comprises necessary parts of the incoming streams and information about the current event selection on them. Events from incoming streams can be reproduced from predecessor operators, so that only event sources need to provide outgoing streams in a reliable way. However, information about the current event selection is not reproducible and therefore is stored in a *savepoint* and replicated at other operators for failure tolerance.

In this context, our contributions are to propose an expressive, general execution model that enables for any operator to signal on its incoming streams the selection of events that it is currently processing. Such a model can be applied to stream processing as well as complex event processing systems. To illustrate its expressiveness, a comparison to the event specification language *snoop* is drawn. Based on this operator execution model, we propose a *savepoint recovery system* that i) provides the basis in identifying an empty operator processing state, ii) manages the capturing and replication of savepoints and ensures the reproducibility of corresponding events, iii) implements a recovery in which also simultaneous failures of multiple sequential operators can be tolerated.

The paper is structured as follows: Section 2 introduces a general event processing and system model suitable to describe event processing systems. In Section 3, we define the requirements with regard to the reliability of the system. Section 4 sketches our novel approach for rollback-recovery. In Section 5, an expressive operator execution model is introduced that works with *selections* of events. Based on that model, Section 6 shows how operator state is minimized to replicated *savepoints*. The recovery algorithms are described and their correctness is proved in Section 7. Afterwards, evaluations are provided in Section 8. In Section 9, possible extensions on the proposed recovery method are discussed. Related work is discussed in Section 10. Finally, Section 11 concludes this paper and briefly discusses future work.

## 2. EVENT PROCESSING & SYSTEM MODEL

### 2.1 Event Processing Model

The operation of a distributed event processing system can be modeled by an operator graph $G(\Omega \cup S \cup C, L)$ interconnecting sources in $S$, operators in $\Omega$, and consumers in $C$ in form of event streams in $L \subset (S \cup \Omega) \times (\Omega \cup C)$. In this model, the sources act as producers of basic events like sensor streams, operators perform correlations on their incoming event streams to produce new outgoing events, and consumers define event streams that require reliable delivery. We will later formalize what reliable delivery means.

An *event* $e$ is a tuple of attribute-value pairs, i.e we use $e = (a_1 : v_1, ..., a_m : v_m)$ to refer to the content of an event that comprises $m$ attribute-value pairs. An *event stream* $(p, d) \in L$ of the operator graph is directed from a producer to a destination and ensures that events are delivered to the destination in the order they are produced. Events in a stream are of a distinct *event type*. We call $p$ the predecessor of $d$ and $d$ the successor of $p$. Accordingly, $(p, d)$ is called an *outgoing stream* of $p$ and an *incoming stream* of $d$. For an event $e \in (p, d)$ we refer to $SN(e)$ the sequence number of $e$, which is deterministically assigned and independent of the physical event production time. Events from different incoming streams have a well-defined, global ordering that is independent of the physical time of their arrival at the operator. Any ordering is possible, as long as the local ordering by sequence numbers on single streams is not violated.

Each operator $\omega$ performs processing w.r.t. the set of all incoming streams $(in, \omega) \in L$, denoted by $I_\omega$. During its execution, an operator $\omega \in \Omega$ performs a sequence of correlation steps on $I_\omega$. In each correlation step, the operator determines a *selection* $\sigma$ which is a finite subset of events in each stream of $I_\omega$. A correlation function $f_\omega : \sigma \to (e_1, \ldots, e_m)$ specifies a mapping from a selection to a finite, possibly empty set of events produced by the operator. The produced events are written in order of occurrence to its outgoing event streams. For each outgoing stream a different set of events may be written.

### 2.2 System Model

The operators of the operator graph $G$ are hosted by a set of $n$ nodes, each node hosting possibly multiple operators. At any time, each node can fail according to the crash recovery model, where at most $k < n$ nodes are assumed to permanently fail or crash and recover an unbounded number of times. The nodes communicate via communication channels that are established for each event stream in $G$ and guarantee eventual in-order delivery of streamed events. In addition, we will consider event sources and event consumers to be reliable. This means that each event produced by a source will be accessible until all dependent operators have signaled that it can be discarded. Furthermore, event sources must be able to reliably store savepoints from their successors. Similarly, for events streamed to consumers we will use a fault tolerant delivery mechanism so that eventually a consumer receives all events sent to it in the right order.

Note that we do not make any assumptions on timeliness for links connecting sources, operators and consumers, nor do we demand any synchronization of their clocks. The system can be realized as a highly distributed correlation network that involves communication over an Internet-like topology. We will use a monitoring component to suspect faulty processes and trigger reconfigurations on the placement of operators on nodes. The accurateness of this component, however, will only affect the performance, but not the correctness of the proposed method.

## 3. RELIABLE EVENT PROCESSING

For event processing systems, it is important that detected events capture the status of the monitored real world in a reliable way, i. e., no events of interest are disregarded (*false-negative event detection*) as well as no "wrong" events that did not really occur are delivered to consumers (*false-positive*). Even if the operators of a system are well-defined and sources of an event processing system reliably capture all basic events of the system, the loss of intermediate events in the event processing system can lead to the occurrence of false-negatives or false-positives. For example, consider a business monitoring system for which an operator $\omega$ monitors whether customer requests were successfully answered. In this case, $\omega$ is required to produce an alarm or confirmation dependent on whether or not a customer request was answered within 10 minutes. The correct detection depends on the successful detection of a customer request, say $e_r$ as well as detecting successful answers, say $e_a$. In particular, not detecting a successful answer could trigger both a false positive and a false negative, namely an alarm instead of a confirmation.

The major cause for the loss of intermediate events is the failure of nodes. We require a CEP system for a set of given primary event streams and a set of consumers to guarantee the following properties despite of the simultaneous failure of an arbitrary number of nodes:

1. Completeness. Each event that can be detected by the event processing system from a given set of available primary streams will eventually be delivered to every correct consumer interested in the event. Depending on the application, a false-negative event detection can cause that, e.g., decisions are made that base on wrong data in business monitoring systems, or even severe dangers in cyber-physical systems occur.

2. Consistency. The streams the event processing system provides to consumers in an execution with operator failures are indistinguishable from an execution without failures (in particular regarding order and attributes of the comprised events). False-positive events can have similar negative effects as false-negative events, leading to wrong decisions based on faulty information. Also the order of produced events can play a role for event consumers, e.g., if decision-relevant out-of-order events arrive when an irrevocable decision has already been made.

Besides these correctness requirements, we aim with our approach for a low run-time overhead as the main efficiency goal in order to provide high scalability of the system.

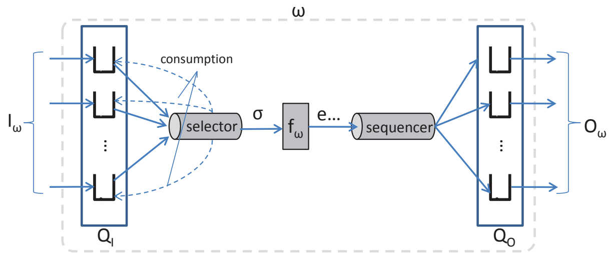

**Figure 1: Model of an operator.**

## 4. OVERVIEW OF THE APPROACH

Figure 1 shows a model of the components of an operator. Incoming events from $I_\omega$ are cached in queues $Q_I$ from which the *selector* determines selections of events to be mapped to outgoing events by the correlation function $f_\omega$. Further, the selector can exclude events from any further correlations, i. e., events are consumed and removed from $Q_I$. The produced events are augmented with a sequence number by the sequencer and put into queues $Q_O$ from which they will be transferred to $\omega$'s successors. Events are discarded from $Q_O$ only if all successors have acknowledged them.

Note that any approach that allows for the recovery of the state of failed operators requires the replication of state. The difference between distinct recovery methods is how and where state information is replicated, e.g., at standby operators or at a persistent storage for classical rollback-recovery. One important observation from the rollback-recovery approach is that state $\Lambda(T)$ at a point in time $T$ can be seen as state at a previous point in time $T_{sp}$ plus a deviation $\Delta(\Lambda(T_{sp}), \Lambda(T))$ that happened on the state between $T_{sp}$ and $T$. We are looking for the optimal $T_{sp}$, when the state of an operator is minimal, so that its replication requires only a minimum of resources.

The state $\Lambda(T)$ of $\omega$ comprises the states of $Q_I$, the selector, $f_\omega$, the sequencer and $Q_O$. Observe that $f_\omega$ implements a mapping in its mathematical sense from selections $\sigma$ to sets of produced events, or, more precisely, attribute-value-pairs of events from $\sigma$ are mapped to attribute-value-pairs of produced events, each mapping denoted as a *correlation step*. Although $f_\omega$ builds up internal state, in this model there are no dependencies in between two subsequent correlation steps. So, at a point in time between two correlation steps denoted $T_{sp}$, $f_\omega$ is *stateless*. The state of the sequencer just comprises one parameter which is the next $SN$ to be assigned to the next processed event. The state of the selector is harder to determine: In an arbitrary operator implementation, the state may cover manifold relations; e.g., selections could depend on other selections, on intermediate consumptions, even on the results of previous correlation steps. In the general case, taking a snapshot of the processing stack of the selector or implementing a sophisticated state extraction method would be inevitable. However, the selector state can be drastically reduced when a specific operator execution model constrains the scope of possible selections. Finally, the state of $Q_I$ and $Q_O$ comprises all events contained in the queues.

In our *savepoint recovery system*, in order to be able to recover $\omega$ once it has failed at a point in time $T$, the recovery procedure determines an earlier point in time $T_{sp} < T$ with the following properties: (i) $f_\omega$ is stateless, (ii) all events

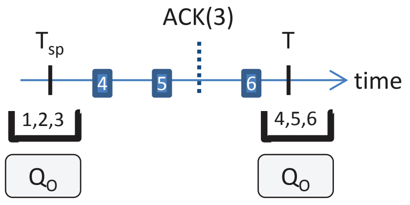

**Figure 2: Between $T_{sp}$ and $T$, all events up to event 3 in $Q_O$ have been acknowledged, so they are not part of $\Lambda(T_{sp})$ w.r.t. the proposed recovery approach.**

that $Q_O$ contained at $T_{sp}$ have been *acknowledged* in the meantime, and (iii) events in $Q_I$ at $T_{sp}$ are eventually available in $\omega$'s predecessors. In this process, stateless times of $f_\omega$ are indicated to an execution environment by a hook that is installed in the correlation logic. The second property is achieved by a distributed acknowledgment algorithm, which is described in Section 6.2. It synchronizes the points in time $T_{sp}$ for recovery between adjacent operators. Thereby, an *acknowledgment* indicates that an event is not needed anymore in order to achieve consistent event streams at the consumers and therefore can be discarded from $Q_O$. The algorithm ensures that the latest recovery point $T_{sp}$ at an arbitrary point in time $T$ is always chosen with regard to this property. See Fig. 2 for an example. It shows that despite $Q_O$ is not empty at $T_{sp}$, it is not necessary to restore it in order to recover $\Lambda(T)$, because all events have been acknowledged in the meantime. Thus, in the following we will not count $Q_O$ as part of $\Lambda(T_{sp})$ with regard to the proposed recovery approach. The third property is also achieved by the acknowledgment algorithm, which only acknowledges events at a predecessor if they are not part of $Q_I$ at $T_{sp}$ anymore.

Now, the relevant, *non-reproducible* state of $\omega$ at $T_{sp}$ comprises only the state of the selector and the sequencer, which is captured in a *savepoint* and replicated at $\omega$'s predecessors. If $\omega$ fails at time $T$, $\Lambda(T_{sp})$ is restored from the replicated savepoint and replayed events of $Q_I$ from the predecessors. From this point on, the re-execution of $\omega$, i. e., performing a sequence of correlation steps, will allow the operator to fully restore $\Lambda(T)$.

## 5. EXECUTION MODEL

The following execution model refines the operator model introduced in Section 4 (cf. Fig. 1). It describes the implementation of the selector and the sequencer, and defines the interface of an arbitrary operator's correlation logic to these components. Thereby, we aim to keep the interface simple, so that existing implementations of $f_\omega$ can easily be embedded into the proposed system.

Let a window $w\langle\langle SN_i^{start}, SN_i^{end}\rangle, \cdots\rangle$ on $I_\omega$ comprise for each incoming stream $i \in (in, \omega)$ all events between a start event with $SN_i^{start}$ and an end event with $SN_i^{end}$. Then a selection $\sigma$ can be defined as the set of all events within $w$ and contains all incoming events on which $f_\omega$ executes one correlation step. To implement the mapping $f_\omega$, each the selector and the sequencer, which we together name the *execution environment (EE)*, provide an interface. The interface of the selector is defined as C(*consumption*), with *consumption* being a map of event types and the corresponding number of consumed events. The consumption of $x$ events from a stream $i \in (in, \omega)$ results for the next

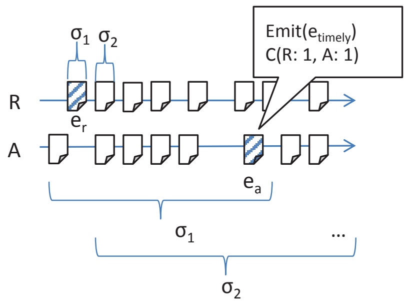

**Figure 3: Interface calls to the EE.**

selection $\sigma$' in: $SN_i^{start'} = SN_i^{start} + x$, i. e., the start event moves further w.r.t. the consumption. Each correlation step has to result in at least one consumption in at least one of the incoming streams to ensure progress. That way, the selector keeps track of the start events of the selections, and just streams events from $Q_I$ in their deterministic order to $f_\omega$ until its interface is called and the next selection starts. Events that are marked as consumed by $f_\omega$ get deleted from $Q_I$. The interface of the sequencer EMIT(*event*) takes a produced event from $f_\omega$, assigns it a SN and puts it into $Q_O$.

*Example* (Figure 3): Consider the example given in Section 3. In a correlation step, $f_\omega$ takes one event of type R (requests), say $e_r$, and then checks events of type A (answers) e.g. for a matching request ID attribute. The step ends when an event $e_a$ of type A is reached that is either corresponding to $e_r$ or has a timestamp that is more than 10 minutes older than the one of $e_r$, so that $f_\omega$ is sure that the 10 minutes timespan has been violated (given events are ordered by timestamps in streams). In the first case, $f_\omega$ would emit an event $e_{timely}$ and would consume $e_r$ and events of type A that have a smaller timestamp than $e_r$ as answers are not expected to appear before the next request. In the second case, an event $e_{alarm}$ is emitted and e.g. delivered to a special agent who will work on the request with a high priority. $e_r$ and events of type A that have a smaller timestamp than $e_r$ are consumed in this case, too.

### 5.1 Properties

PROPERTY 5.1. (STATE OF AN OPERATOR AT $T_{sp}$.) *Let $T_{sp}$ be a point in time when an operator $\omega$ starts processing a new selection $\sigma_{sp}$. Then, $\Lambda(T_{sp})$ of $\omega$ comprises:*

- *Events in $Q_I$.*
- *The state of the selector: For each incoming stream $i \in (in, \omega)$: $SN_i^{start}$ of $\sigma_{sp}$.*
- *The state of the sequencer: The SN of the first event to be produced in $\sigma_{sp}$.*

Events in $Q_I$ are replayed from predecessors. In order to restore the selector, the SNs of the start events of $\sigma_{sp}$ have to be restored. Then, the selector can provide to $f_\omega$ exactly the same selection that had been provided in the primary execution of the correlation step, which leads to the production of exactly the same events. The subsequent selection $\sigma_{sp+1}$ depends only on $\sigma_{sp}$, and so on, so that all subsequent selections are indistinguishable from a failure-free execution. To restore the sequencer, it is initialized with the $SN$ of the next emitted event.

PROPERTY 5.2. (START EVENTS OF CONSECUTIVE SELECTIONS.) *For a selection $\sigma_s$, each start event has a higher or equal sequence number compared to the selection $\sigma_p$ of a preceding correlation step.*

Selection windows are moved when events are consumed. A consumption always leads to a higher sequence number of the next start event. This property limits the dependencies between event streams. For two consecutive correlation steps (and thereby for the production of events in such steps), the selection start events from a stream in $I_\omega$ are never descending.

## 5.2 Expressiveness

After we have defined how the execution model of our event processing system works, one might ask whether this really reflects the requirements of real-world CEP systems with regard to the detection of event patterns. Does it provide enough expressiveness so that any realistic situation detection can be implemented?

As a reference, we will take the event specification language *snoop* [6] and check whether all event patterns that are formalized in snoop can be implemented in our execution model. This has several reasons: First, snoop has been motivated by real-world event processing scenarios and not just on some academic theoretical models. Such scenarios comprise, amongst others, sensor applications (e.g., hospital monitoring and global position tracking), applications that exhibit causal dependency (e.g., between aborts and rollbacks, bug reports and releases) and trend analysis and forecasting applications (e.g., security trading, stock market analysis). Furthermore, snoop is well-established in the scientific event processing community. But above all, it provides a high expressiveness in comparison to other languages, as Margara and Cugola show in their article [7].

In snoop, complex events can be correlated by *event operators*, which are the following ones: Disjunction, sequence, conjunction, aperiodic and periodic operators. Further, *parameter contexts* in snoop describe which incoming events are considered in the computation of the parameters of a produced event in a correlation step, when there are several possible events to choose from. The following parameter contexts are defined: (i) *Recent*, where only the most recent occurrences are used for correlation. (ii) *Chronicle*, where incoming events are correlated in the chronological order in which they occur. (iii) *Continuous*, where continuously each possible event starts a correlation execution. (iv) *Cumulative*, where all events between a possible start and end event are correlated. For more detailed explanations and examples, please refer to the original snoop paper [6].

PROPOSITION 5.3. (EXPRESSIVENESS OF THE EXECUTION MODEL.) *All event operators and parameter contexts of snoop can be implemented using the execution model that is proposed in this work.*

PROOF. In snoop, *event expressions* define a finite time interval in which one or more atomic happenings, or events, can occur. Thus, they correspond to finite sets of sequential events. So they are equivalent to event selections as they are defined in the execution model. All snoop event operators work solely with event expressions as operands. Thus, the sequence of execution iterations of snoop event operators can be seen as an execution with a sequence of event selections $(\sigma_1, \sigma_2, \sigma_3, ...)$ as operands.

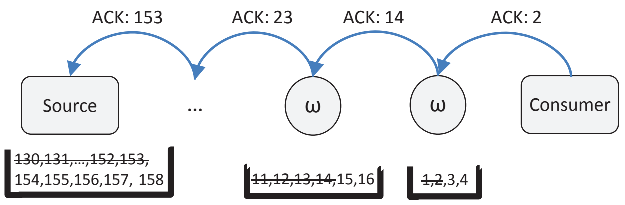

**Figure 4: ACK flow and pruning of $Q_O$.**

LEMMA 5.4. *For the operands $(\sigma_1, \sigma_2, \sigma_3, ...)$ of a snoop event operator, the following properties are satisfied:*
*(i) $SN_j^{start} \in \sigma_{k+1} \geq SN_j^{start} \in \sigma_k \; \forall j \in (in, \omega)$,*
*(ii) For all event operators and parameter contexts exists an implementation of $f_\omega$ so that when it is executed on $\sigma_1$, $\sigma_2$, $\sigma_3$..., the events produced by $f_\omega$ satisfy the semantics of snoop.*

PROOF. (i) Proof by contradiction. If $\exists e$ with $e \in \sigma_{k+1} \wedge e \in j : SN(e) < SN_j^{start} \in \sigma_k; j \in (in, \omega)$, then $e$ would be in a later time interval but have a lower SN than the start event of $\sigma_k$. This contradicts to the policy that sequence numbers are assigned sequentially.

(ii) is satisfied because given the event selections, $f_\omega$ can implement any operations on a selection, especially any functionality of a snoop event operator can be implemented. □

LEMMA 5.5. *None of the parameter contexts demands for the consumption of an intermediate event that is located within the window bounds of the selection of the next event operator execution.*

PROOF. In *Recent*, when an event is detected, all events which cannot become the initiator of an event detection and are ordered between the detection initiator and the event that triggered the detection are consumed. This cannot affect intermediate events, because if an earlier event of a type can potentially start an event detection and is not consumed, then a later event of the same type can potentially start an event detection, too.

In *Chronicle*, events are used for parameter computation in the order they occur and then they are consumed, so no intermediate consumption can occur.

In *Continuous*, no explicit consumptions happen at all.

In *Cumulative*, all events of a selection are consumed. □

As a conclusion of Lemma 1 and Lemma 2, $f_\omega$ works on sequential selections that do not demand intermediate consumptions and so it can implement the interface to the EE defined by the execution model. So, our execution model is at least as expressive as the snoop event specification language.

# 6. CAPTURING AND REPLICATING SAVEPOINTS

## 6.1 Log and Savepoint Management

### 6.1.1 Logs of Outgoing Events

Events in $Q_I$ of $\omega$ are preserved in the $Q_O$ of its upstream neighbors (a.k.a. predecessors), so that *no additional events* *need to be transferred over the network at failure-free runtime.* If $\omega$ fails, $Q_I$ can be restored when its predecessors re-send their $Q_O$. $Q_O$ must always contain enough events to restore the successor to its latest acknowledged state $\Lambda(T_{sp})$ which depends on the coordination of savepoints described in Section 6.2. Note, however, that outgoing events are reproduced when an operator recovers, so that events in $Q_O$ are *reproducible* and do not need to be replicated.

### 6.1.2 Savepoints and Savepoint Trees

Savepoints contain the non-reproducible part of $\Lambda(T_{sp})$, which comprises the state of the selector and the sequencer. They are stored together with $Q_O$ in the volatile memory of the predecessors of $\omega$. If $\omega$'s predecessor is an event source, savepoints and events can be stored there in a reliable way, as event sources are implemented fault-tolerant.

So, when $\omega$ crashes at a point in time $T$, its predecessors hold all state information that is necessary to restore $\Lambda(T)$: $\Lambda(T_{sp})$ is restored from the savepoints and events from $Q_O$ at predecessors, and $\Delta(T_{sp}, T)$ is restored by re-running $\omega$ from $T_{sp}$ until $T$. We will determine later the points in time when an operator has to update and distribute its savepoint. To deal with asynchrony, it is necessary that all predecessors store a *complete* savepoint, so that at the recovery of $\omega$ a self-consistent savepoint is available, i.e., the information about the start events of $\sigma$ belong to the same $\sigma$, and the next $SN$ fits with that $\sigma$. In contrast to that, the re-sent events from incoming streams in $I_\omega$ do not have to be consistent w.r.t. the same savepoint, because it is easily possible for an operator that is getting restored to ignore events that stem from older selections and therefore are not part of $Q_I$.

By now, only one failed operator can be restored at a time, but not several adjacent operators that fail at the same time. To make that possible, the non-reproducible part of $\Lambda(T_{sp})$, namely the savepoint, is replicated at all operators of the transitive closure of the predecessor relation in the operator graph, so that each operator preserves a *tree of savepoints* in its volatile memory.

## 6.2 Coordination of Savepoints

In order to restore $\Lambda(T)$ of $\omega$, one has to determine the point in time $T_{sp}$ to which it has to be restored in order to allow for the restoring of $Q_O$ at $T$. This depends on the events that are part of $Q_O$, more precisely, the earlier the events of $Q_O$ at $T$ had been produced, the earlier is $T_{sp}$. A trivial implementation would never prune $Q_O$ so that it contains all events an operator has ever produced. In case of a recovery, $T_{sp}$ would be "zero", i. e., the operator would be restored to the point in time when it initially started its work. To avoid that, it is necessary that an operator *prunes* $Q_O$ from time to time, i. e., excludes events from $\Lambda(T)$ and thereby increases $T_{sp}$. Events can be pruned when they are not necessary for the consistency of the event streams delivered to the consumers anymore. The necessity is defined as follows:

Definition 6.1 (Necessity of Events). *An event e is a necessary event if a consumer is interested in it and has not yet acknowledged its receiving.*

If all consumers interested in $e$ have acknowledged its receiving at a point in time $T_{ACK}$, a predecessor operator $\omega$ can be sure that $e$ is not necessary anymore and delete it (and all earlier events) from its $Q_O$ (see Figure 4). That way, $e$ and all earlier events are not part of $\Lambda(T)$ for any $T \geq T_{ACK}$, so that $T_{sp}$ and thereby the savepoint can be adjusted to the correlation step in which the first event following $e$ had initially been produced. This is done by means of the *inverse correlation function* $f^{-1} : e \rightarrow \sigma_e$, which maps $SN(e)$ to the start events of the selection $\sigma_e$ in which $e$ had been produced. The most efficient implementation of $f^{-1}$ is for each selection to store $SN_i^{start}$ for each stream $i \in (in, \omega)$ together with the produced events in $Q_O$. For each incoming stream, events that are placed before the corresponding start event of $\sigma_e$ are discarded from $Q_I$ at $T_{sp}$ and their $SNs$ are acknowledged at $\omega$'s predecessors. They proceed in the same way that $\omega$ did, prune their $Q_O$ and adapt their $T_{sp}$ to the production of events in $Q_O$, update their savepoint accordingly in the savepoint tree, and acknowledge $SNs$ from $f^{-1}$ at their predecessors as well as send them the updated savepoint tree for further replication.

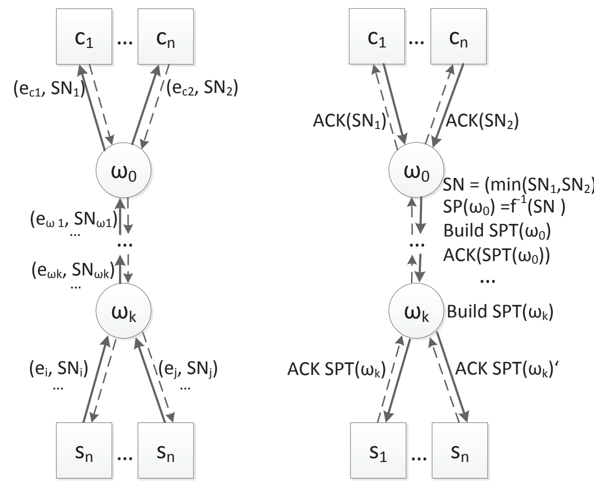

**Figure 5: Event and ACK flow in a CEP system.**

To coordinate savepoints, we make use of acknowledgment messages, say ACKs, which contain both the $SN$ of the acknowledged event and the updated savepoint tree. When receiving an ACK, an operator replaces the obsolete part in its savepoint tree, prunes $Q_O$ and checks whether $T_{sp}$ can be updated. If this is the case, the operator sends an ACK to each of its predecessors. That way the ACKs flow *upstream*, i.e., against the flow direction of events, until they will finally reach the event sources signaling that replicated events have become unnecessary and therefore can be discarded. The algorithms for log and savepoint maintenance of an operator are given in Figure 6.

*Example:* Figure 5 shows the event and ACK flow in the system. On the left graph, events are flowing downstream, starting at the event sources and getting correlated with each other until some events are delivered to consumers. On the right graph, the consumers acknowledge different $SN$s of received events. The minimal $SN$ acknowledged by all connected consumers at $\omega_0$ signals the latest unnecessary event, $Q_O$ is pruned, the savepoint tree is updated accordingly and sent with an ACK to all predecessors. They store the new savepoint tree, update their own savepoint if applicable, send ACKs to their predecessors, and so on, until finally the ACKs reach the event sources, where the save-

```
Map<successor, ACK> $latestRecACKs$    ▷ Contains latest received ACK from each successor
Savepoint $ownSP$    ▷ Contains current own savepoint
List<Event> $Q_I$    ▷ Queues of incoming events
List<Event> $Q_O$    ▷ Queues of outgoing events

upon ⟨RECEIVEACK⟩($inACK$)
  $latestRecACKs$.INSERT($inACK$.producer, $inACK$)
  if $latestRecACKs$.GETOLDESTACKEDSEQ() has changed then    ▷ update own savepoint
      $sn = inACK$.GETACKEDSEQNO()
      $e\_acked = Q_O$.GETEVENT($sn$)
      $map < instream, SeqNo > = f^{-1}(e\_acked)$
      $Q_I$.PRUNE    ▷ prune $Q_I$
      $newSavepoint$ =
      NEW SAVEPOINT($map < instream, SeqNo >$, $sn$)
      $Q_O$.PRUNE($sn$)    ▷ prune $Q_O$
      $SavepointTree$ = NEW SAVEPOINTTREE()
      $SavepointTree$.SETROOT($newSavepoint$)
      for all ACK in $latestRecAcks$ do
          $SavepointTree$.ADDCHILD(ACK.savepointTree)
      end for
      $newACK$ = NEW ACK($SavepointTree$)
      for all predecessors do
          SEND($newACK$)
      end for
  end if
end
```

**Figure 6: Algorithms for log and savepoint maintenance at an operator $\omega$.**

```
upon ⟨RECEIVEINIT⟩($predecessors, successors$)
  for all $op$ in $predecessors$ do
      SEND($op$, RECOVERYREQUEST)
  end for
  while not received all RECOVERYINFORMATION do
      upon ⟨RECEIVERECOVERYNOTIFICATION⟩($predecessor$)
        SEND(predecessor, RECOVERYREQUEST)
      end
      upon ⟨RECEIVERECOVERYINFORMATION⟩()
        list<RecoveryInformation>.ADD(RecoveryInformation)
      end
  end while
  RESTORESTATE(latestRecoveryInformation)
end

upon ⟨RECEIVERECOVERYREQUEST⟩($successor$)
  $SavepointTree$ = $latestRecACKs$.GET($successor$).GETSAVEPOINTTREE()
  $RecoveryInformation$ = NEW RECOVERYINFORMATION($Q_O$, latestRecACKs)
  SEND(successor, RecoveryInformation)
end
```

**Figure 7: Algorithms for recovery of an operator $\omega$.**

point trees are replicated and all acknowledged events are discarded.

## 7. ALGORITHMS FOR OPERATOR RECOVERY

### 7.1 Recovery of the State of Failed Operators

For the description of the recovery algorithm, we will at first assume that operator failures are detected immediately and that failed operators are restarted automatically. Also, we assume that an operator knows his direct predecessors, even after it has crashed and recovered. From this point, we describe how an operator will be able to restore its state w.r.t. the latest available savepoint, so that event streams that were lost through the failure get reproduced. Later, we will describe how the failure detection and operator topology management can be solved in an asynchronous system.

#### *7.1.1 Recovery Procedure*

After its restart, a failed operator $\omega$ sends to its predecessors a message called RECOVERYREQUEST. When an operator receives such a RECOVERYREQUEST, it answers by sending the *recovery information* necessary for restoring the state of $\omega$, which comprises $Q_O$ (replay of the outgoing stream) and the savepoint tree of $\omega$. $\omega$ waits until it has received all recovery information. Then it identifies the answer containing its latest savepoint SP, which is the answer with the highest value for the $SN$ of the next event to be produced. $\omega$ restores $\Lambda(T_{sp})$ by initializing the selector with the selection defined in the SP, restores $Q_I$ with the replayed events from the predecessors, and initializes the sequencer with the next $SN$ to be assigned to a produced event.

To cope with multiple simultaneous failures of adjacent operators, $\omega$ sends a RECOVERYNOTIFICATION to its successors after its recovery. So, if one of those operators is awaiting recovery information from $\omega$, it can detect that the RECOVERYREQUEST might have been lost because of the failing of $\omega$ and resend it. This way, a failed predecessor does not lead to an infinite waiting time for receiving all recovery information. From bottom up, failed operators can recover, each sending the necessary recovery information to its successor, until all operators are restored to their latest ACKed state again. The algorithms for the recovery of an operator are listed in Figure 7.

### 7.2 Control and Adjustment of the Operator Topology

By now, we have assumed an error-free, immediate detection and restart of failed operators. However, in an asynchronous system, a perfect failure detector cannot be implemented to solve that problem. Instead, we have to work with a weaker failure detector abstraction that suspects operators to have failed, but the suspicions might be wrong.

#### *7.2.1 Coordination of Operator Recovery*

To cope with that problem, we use a central component called *coordinator*, which has global knowledge about the operator topology and is eventually always up and running, i.e., there might be times when the coordinator is not available, but it will always come back online. The coordinator uses a failure detector with strong completeness (each failed operator will eventually be detected) and eventual weak accuracy (there is a time after which some correct process is never suspected), i.e., an *eventually strong failure detector*. Such a failure detector checks for heartbeat messages that correct operators send in a certain frequency. If a heartbeat message from $\omega$ did not arrive at the coordinator within a *time bound* $\tau$, it will be *suspected* to have failed. As we work with an asynchronous system model, the coordinator can never be sure whether the operator has really failed, but it is sure that a failed and not yet fully recovered operator will not send heartbeat messages anymore, so eventually every failure will be detected. The possibility to implement

```
list<operator> operators                          ▷ list of all operators
list<operator> suspected                          ▷ suspected operators
list<operator> progressed                         ▷ operators that have
    participated in the overall computational progress
map<string, list<operator> > replacements         ▷
    replacements of an operator type

procedure MonitoringProcedure()
    while true do                                 ▷ infinite loop
        nextCheck ← currentTime() + checkFrequency
        for all operators do
            CheckLiveness(operator)
        end for
        wait until nextCheck
    end while
end procedure

procedure checkLiveness(operator)
    if (currentTime - lastReceivedHeartbeat.time ) > $\tau_{operator}$ then
        if operator ∉ suspected then
            suspected.add(operator)
            startReplacement(operator)
        end if
    else                                          ▷ op is alive
        if operator ∈ suspected
            and operator ∈ progressed then
            recallReplacements(operator)
            adaptTau(operator, higher)   ▷ increase $\tau_{operator}$
        end if
    end if
end procedure
```

**Figure 8: Algorithms for monitoring and management of operator topology at the coordinator.**

the coordinator in a distributed manner is discussed in Section 9.

If $\omega$ is suspected to have failed, a replacement operator $\omega'$, i. e., an operator that implements the same correlation function as $\omega$, is installed on a free system resource. When $\omega'$ initializes, it starts the recovery procedure described in Section 7.1. Now, it might be the case that the coordinator suspects $\omega'$ to have failed, too, so that $\omega''$ is initialized, and so on. That is why suspected operators are not terminated immediately, but rather have the ability to run in parallel with their replacements. When the first of these parallel operators makes some real *progress* in event processing, the coordinator decides on that operator to remain in the topology and terminates all other replacement operators, i.e., they are shut down and their direct successors and predecessors are notified not to send messages to them any longer. The notion of progress is defined as follows:

Definition 7.1. (Processing Progress of an Operator.) *An operator $\omega$ has made progress after the restoration of its state when it updates its own savepoint for the first time.*

A savepoint update moves forward the point in time to which an operator gets recovered after its failure. That way, liveness of the system is guaranteed and the topology will finally stabilize. Note, that it is no problem for successors and predecessors of $\omega$ to cope with multiple replacements running in parallel: As they produce exactly the same events, the duplicates can easily be filtered, and ACKs are sent only to operators from which the ACKed events have been received.

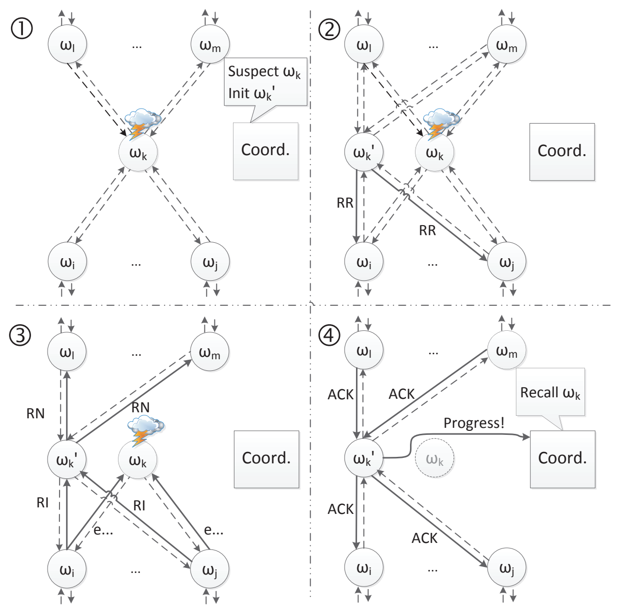

**Figure 9: Recovery from an operator failure.**

When it turns out that $\omega$ had been suspected by mistake, i.e., when the coordinator had suspected $\omega$ and received a heartbeat after that, the time bound $\tau$ can be adjusted to avoid such false suspicions in the future. We will not go further into detail with the time bound adjustment here as there exist already established algorithms for such problems in the field of eventually strong failure detectors. The algorithms at the coordinator for monitoring and control of the operators are listed in Figure 8.

*Example:* Figure 9 shows how an operator $\omega_k$ fails and is replaced by $\omega'_k$. The coordinator suspects $\omega_k$ and starts $\omega'_k$. $\omega'_k$ initializes by sending RecoveryRequests to its predecessors, receives recovery information as responses, restores its state and sends RecoveryNotifications to its successors. Now, $\omega'_k$ is fully incorporated into event production, and when enough events have been produced so that ACKs are received that make $\omega'_k$ update its savepoint, a progress notification is sent to the coordinator. Then, $\omega_k$ is deleted from the system, i.e., its successors and predecessors are notified to stop trying to communicate with $\omega_k$.

## 7.3 Correctness Analysis

For proving correctness and liveness, i. e., completeness and consistency of event streams at consumers (see Section 3), we prove that there are no false-negatives or false-positives and that the overall system makes processing progress despite an arbitrary number of simultaneous operator failures.

Proposition 7.2. (No Event Loss.) *In spite of the failure and recovery of an arbitrary number of operators at the same time, no necessary event in the sense of Definition 6.1 gets lost in an unrecoverable way.*

Proof. Let $C$ be a consumer who has not yet received and acknowledged an event $e_c$. Let $\omega_{p1}$ denote a direct predecessor of $C$, $\omega_{p2}$ a direct predecessor of $\omega_{p1}$, and so on. Then the latest savepoint of $\omega_{p1}$ is captured with respect to a point in time $T_{sp_1}$ before $e_c$ has been produced. So, $e_c$ is reproducible by a recovered operator $\omega_{p1}$. Further, the latest savepoint of $\omega_{p2}$ is captured with respect to a point in time when events that are part of $\Lambda(T_{sp_1})$ of $\omega_{p1}$ are reproducible, and so on, so that all necessary events are reproducible. □

Proposition 7.3. (No False-Positive Events.) *In spite of the failure and recovery of an arbitrary number of operators at the same time, there are not delivered any events to the consumers that would not have been delivered in the failure-free execution of all operators.*

Proof. Property 5.1 shows that the state of $\omega$ at $T_{sp}$ comprises exactly the information that is kept in a savepoint plus events from $Q_I$. As the savepoint is captured and replicated, it cannot deviate from the original savepoint after the recovery of $\omega$. Further, necessary events from $Q_I$ are either replayed from a preceding $Q_O$ or recursively reproduced, whereas the recursion stops at a point where events from a $Q_O$ are replayed (at the latest from the event sources). Events in $Q_O$ are exactly the same events as sent in outgoing streams. As the recovered $\omega$ starts to process the same $\sigma$ on indistinguishable copies of the events on which it would have been processing in a failure-free operator execution, the produced events are indistinguishable, too. □

Proposition 7.4. (Liveness of the System.) *Events are delivered to the consumers after a finite time interval from their physical occurrence, i.e., the event processing system makes progress in spite of the failure and recovery of an arbitrary number of operators.*

Proof. As only correct operators send heartbeat messages to the coordinator, failed operators will eventually be suspected and replacements are started. The topology becomes stabilized when an operator signaled processing progress w.r.t. Definition 7.1. So, the liveness of the system is ensured, given that only a finite number of hosts is failing and there are enough correct hosts to run all operators. □

## 8. EVALUATION

In the evaluation, first of all we want to analyze the overhead of our approach induced at failure-free run-time: We measure the communication overhead and compare it with the overhead that would be induced by an active replication approach. Further, we analyze how the frequency of acknowledgments, the induced communication overhead and the size of $Q_O$ are connected. In doing so, we have implemented the algorithms in an event-based simulation without considering incidental influences like underlying hardware topologies and communication protocols in order to emphasize the *inherent overhead* that would be caused in any implementation on any underlying infrastructure.

As a second aspect, we address the delay that the recovery of operators induces. Thereby, we identify significant parameters and develop a mathematical model of the recovery time of a failed operator.

### 8.1 Run-time Overhead

Our approach mainly induces run-time overhead with respect to two different aspects: The transmission of ACKs induces communication overhead, and the volatile storage of $Q_O$ and savepoint trees impacts the memory footprint.

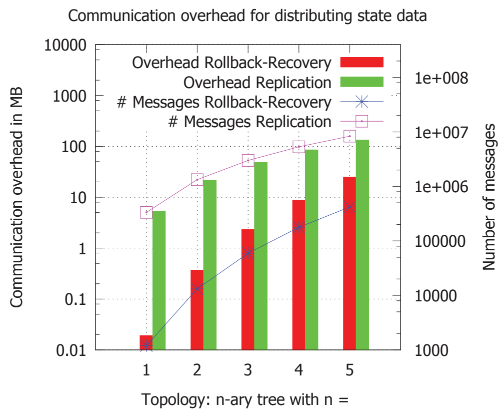

**Figure 10: Run-time overhead comparison between rollback-recovery and replication.**

#### 8.1.1 Communication Overhead

The only data sent over the communication links at failure-free run-time are ACKs. We compute the size of an ACK as: $S(SimpleACK) + (\#Savepoints \times S(Savepoint))$. $S(SimpleACK)$ is 4 bytes (for the ACKed $SN$), $S(Savepoint)$ is 4 bytes for the $SN$ of the next produced event, and $n \times 4bytes$ in a $n$-ary tree for the $SN$s of the start events in $I_\omega$. An event is considered to be of a size of 16 bytes, 4 bytes for its $SN$, 4 bytes timestamp and 8 bytes payload. As the simulated operator topology, we chose $n$-ary trees with a depth of 3 for $n = 1$ to 5, with the root operator connected to 1 consumer and each leave operator connected to 1 source. Event sources produce events with a frequency of 1 event / ms. We compare the overhead with the messaging overhead that would have been caused by duplicate events in the CEP-optimized *active replication* approach [20] developed by Völz et al. We assume a low replication factor of 2 and the best case scenario for the leader election (only one leader at a time), leading to an overhead that approximately equates to the number of events sent through the network regularly, and neglect the overhead that the leader election would cause. Figure 10 shows how much additional data is sent over the network within 5 minutes. As one can see, our rollback-recovery approach induces less communication overhead than the compared active replication approach. Conclusions on this are drawn in Section 8.3.

#### 8.1.2 Memory Consumption

The consumption of main memory at a host that an operator induces can be divided into two different parts: One part is the memory that is used for intermediate results in event processing, containing $Q_I$ and the memory stack of $f_\omega$. This part contains no specific overhead of the rollback-recovery approach, but rather the normal memory footprint of any event processing operator, so that we do not consider this in our evaluations. The other part is the memory used by all data stored solely for the purpose of enabling efficient rollback-recovery. This part is determined by two aspects: The size of the stored savepoint trees of the successors and the size of $Q_O$. The size of the savepoint trees depends solely on the operator topology and basically consists of one savepoint for each member of the transitive closure of the successor relation and is static (for a static operator topology).

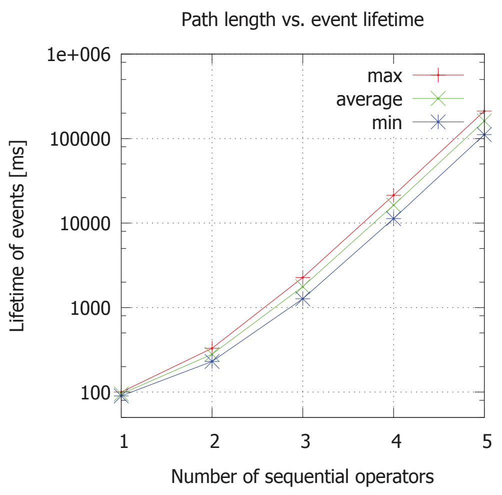

**Figure 11: Lifetime of events in the sources against the number of sequential operators between sources and consumers.**

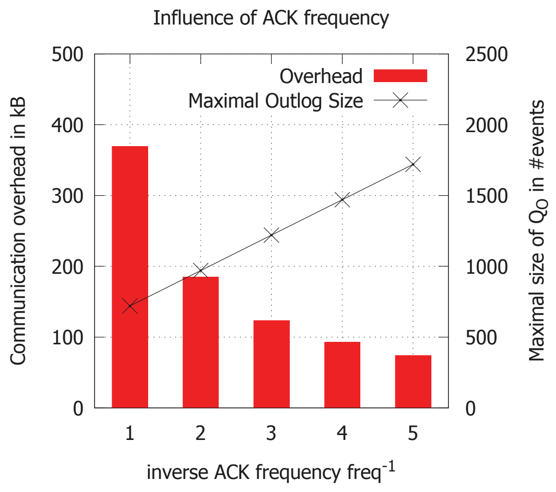

**Figure 12: Influence of the ACK frequency at the Consumer on the overall communication overhead and the maximal size of $Q_O$ at the source.**

The size of $Q_O$, however, is dynamic and depends on the amount of produced events of an operator that are (possible through intermediate steps) correlated into an event that is delivered to event consumers and on the point in time between two consecutive ACKs that lead to its pruning.

To determine the maximal memory footprint, we analyze the event sources, as they have to store the maximal savepoint trees and events produced by sources have the maximal lifetime, i. e., the time between their production and storage in $Q_O$ and the receiving of their ACK which triggers their deletion from $Q_O$. We have measured the influence of the complexity of the events delivered to the consumers, i. e., the number of simple source events that are aggregated to a complex event, on the size of $Q_O$ in the event sources. To do so, we built a simple topology containing one event source producing events in a frequency of 1 event / ms, a variable number of sequential operators and one consumer. In each correlation step, an operator takes 10 new events from its incoming stream and produces 1 outgoing event. Figure 11 shows the results: With an increasing complexity of the events delivered to the consumers, the lifetime of events in the outlog of the event sources increases. Conclusions on this are drawn in Section 8.3.

### 8.1.3 *Influence of ACK Frequency*

We have further evaluated how the frequency of ACKs influences the maximal size of $Q_O$ at sources and the run-time communication overhead. In doing so, we programmed the Consumer to only acknowledge each $freq$-th event that it receives. The underlaying topology is a binary tree with a depth of 3, the rest of the parameters is as in the preceding scenarios. Figure 12 shows the results: When the ACK frequency decreases, the outlog size increases, but the communication overhead decreases. Conclusions on this are drawn in Section 8.3.

## 8.2 Recovery Overhead

As the rollback procedures take some time until an operator state is restored, it takes longer for the system to recover from failures in comparison to active replication (where a replicated operator can take over processing with almost no latency). The recovery time of an operator $\omega$ is
$recoverytime(\omega) = T_{fd} + T_{deploy} + T_{rec} + T_{pred}$
with the parameters: (i) $T_{fd} = T_{channel\_delay} + T_{hb\_freq}$: Failure detection latency, depends on the communication delay between operators and the coordinator and on the frequency of heartbeat checks. (ii) $T_{deploy}$: Allocation of resources and deployment of the replacement operator, mainly depends on the underlying technology such as communication channels and the availability of resources. For example, when using an elastic compute cloud such as Amazon EC2, it can take some minutes until a new node is allocated (this highly depends on the deployed system, users report from about 1 minute up to 15 minutes). The time can be reduced when pre-deployed operators are provided. (iii) $T_{rec} = max(T_{channel\_delay}) + max(size(rec\_inf) \times channel\_rate)$: Recovery of the deployed replacement, depends on the slowest connection to a predecessor and on the size of the recovery information. (iv) $T_i$: Recovery of predecessor operators in the case of $i$ adjacent failures: $T_{pred} = \sum_{j=1}^{i-1} recoverytime(\omega_j)$. This is the summation of the recovery time of failed predecessors, which need to be recovered successively first in order to recover $\omega$.

## 8.3 Conclusions on the Evaluation

### 8.3.1 *Run-time Overhead*

We see that in comparison to active replication the network load can be reduced drastically by applying the proposed approach. An increasing node degree n causes that more sources participate in the production of simple events that get eventually aggregated to a complex event delivered to consumers. That way, the number of simple events per source aggregated in such a complex event decreases and the frequency of ACKs increases. Therefore, the communication overhead increases faster with rollback-recovery than with active replication. However, this behavior can be controlled by the consumers: If they decrease their frequency of ACKs, the overall network load decreases exponentially, but the size of $Q_O$ at sources increases linearly. Besides the low network load, we do not need to preserve redundant resources as we would in active replication. An additional advantage is that we are able to recover from multiple arbitrary operator failures with rollback-recovery (in fact, all operators can fail at the same time and be restored), a property which would cause *immense* costs in active replication.

### *8.3.2 Recovery Overhead*

Recovery generally takes longer than in active replication. The main parameters highly depend on the communication channels and the provision of nodes to deploy replacement operators. As the size of recovery information is limited to the size of the savepoint tree plus the size of re-streamed $Q_O$, it can be inferred from the evaluations in Figures 11 and 12 and the calculations given in Section 8.1.1.

## 9. DISCUSSION

This section briefly discusses possible extensions of the proposed recovery method.

*How can intermediate consumptions be handled?* Intermediate consumptions, i.e., consumptions of events in an arbitrary order instead of a sequential consumption, build up dependencies between different correlation steps and therefore are involved into the operator state. Such consumptions need to be replayed when restoring an operator state. To make that possible, they can be stored in the operator savepoint, e.g., in a table that connects correlation steps with the intermediate consumptions. Thus, the size of the savepoints and thereby the run-time overhead would increase.

*How can the execution model by simplified for stream processing?* In stream processing, in contrast to complex event processing, the window sizes of the execution model can be determined without considering the content of the contained events and the correlation function semantics. Therefore, the execution environment can track the selection window movement without needing information from the correlation function $f_\omega$ about the number of consumed events. This simplifies the interface between the EE and $f_\omega$: $f_\omega$ just has to signal the end of each correlation step.

*How can the coordinator be implemented in a distributed manner?* When distributing the coordinator functionality over different nodes, it is intuitive to make a predecessor the coordinator of a successor, as they communicate with each other anyway and so heartbeat messages could be piggybacked. In doing so, it is important that each operator has exactly one coordinator responsible for its failure detection and recovery, so that the operator topology stabilizes. To solve this problem for asynchronous systems, concepts of leader election or group membership are necessary, e.g., as used in [9].

## 10. RELATED WORK

The existing approaches for distributed stream processing systems can be divided in three categories: The first category targets applications characterized as "partial fault-tolerant" [2,14]. In the case of a failure, systems try to produce information which is not perfectly accurate but might still be useful to the receiver. In the second category, information is published tentatively and corrections can be issued at a later point in time that revoke the messages sent before [1,3,4,13]. These solutions are based on two premises: (i) Dependencies of operators on each other's output have to be within a reasonable limit to keep correction cost acceptable and, more important, (ii) the correction of incorrect messages has to be possible at all. In the scenarios we are examining, decisions might have already been made based on incorrect information that are either very costly or even impossible to correct. Therefore, accurate information is needed at all times.

Solutions that provide accurate information at all times involve the replication of functionality in active or passive replication [18] [5], or rollback-recovery [8] using checkpoints in combination with logs. Among others, these three principles have been applied to distributed stream processing systems; however, none of the current approaches provides all of the necessary properties for large-scale monitoring systems. In the following, we will discuss the proposed solutions individually.

Approaches using active replication [12] incur quadratic overhead in terms of messages during failure-free execution. Checkpointing [15] requires the frequent execution of sophisticated state-extraction algorithms that need either to be specified individually for each operator or require taking a full memory snapshot. Therefore, these approaches either restrict the user to using predefined operators only or require additional expertise to implement the extraction function. On the other hand, a memory snapshot can only be taken if the respective pages are write-locked, which incurs significant additional detection delay during failure-free execution. The approach of "upstream backup" [10,11] uses a similar construct of a simple ACK to recover from only one single failure at a time without any state replication. However, the approach is very restrictive regarding operators for which a consistent state can really be guaranteed after a recovery. More sophisticated, asynchronous event processing operators need a different execution model that at least allows for independent windows on different incoming streams and therefore inherently possess non-reproducible state that needs to be replicated.

Reliability for distributed complex event processing systems has not been researched as actively as for stream processing systems. Active replication has been applied to CEP as well, providing a leader election algorithm to reduce the message overhead during failure-free runtime [20]. Still, for tolerating $n$ simultaneous failures, $n+1$ replicas are deployed for each operator, creating at least a linear message overhead. An approach for dealing with unreliable communication channels when delivering events to a CEP system has been proposed [16]. However, operator failures are not considered in this approach.

## 11. CONCLUSION

Although reliability is critical for many applications involving event processing systems, state-of-the-art approaches have clear shortcomings in providing accurate processing and low run-time overhead in a large-scale deployment.

This paper proposed a novel rollback-recovery mechanism for multiple simultaneous operator failures in distributed stream and event processing systems that eliminates the need for checkpoints and does not use persistent memory at operators. It therefore avoids the main drawbacks of previous approaches, which increase processing and network load for creating and maintaining large checkpoints, or burden application developers with defining operator specific mechanisms for checkpointing and recovery.

We defined an event processing model based on the concept of event selections to find points in time when an event processing system has minimal non-reproducible state which is then stored in replicated savepoints. The rest of the operator state can be reproduced from primary event streams, so that only event sources have to maintain events in a reliable way. An algorithm to coordinate savepoint maintenance over multiple levels of operators is provided, allowing to recover from simultaneous operator failures. We proved the algorithm correctness and provided evaluation results demonstrating its behavior in different parameter settings in comparison to active replication. The evaluations have shown that the network load can be reduced drastically, and that the frequency of acknowledgments at event consumers is a design parameter that can be used to balance between memory requirements and network load.

Future work will focus on the further exploration of the proposed recovery scheme. We want to improve the runtime behavior by dynamically adapting the acknowledgment frequency to system properties and by the segmentation of the operator topology by means of reliable persistence layers. Besides, the concept of savepoints can be used for the efficient implementation of operator migration [17]. Furthermore, in order to allow for a better scalability and speedup of the event correlations, we want to explore how the operator execution model can support the parallelization of processing.

## Acknowledgment

This work has been supported by contract research "Internationale Spitzenforschung II" of the Baden-Württemberg Stiftung.

## 12. REFERENCES

[1] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proc. of SIGMOD '05*, pages 13–24.

[2] N. Bansal, R. Bhagwan, N. Jain, Y. Park, D. Turaga, and C. Venkatramani. Towards Optimal Resource Allocation in Partial-Fault Tolerant Applications. In *Proc. of the 27th IEEE Conference on Computer Communications, INFOCOM '08*, pages 1319 –1327.

[3] A. Brito, C. Fetzer, and P. Felber. Minimizing Latency in Fault-Tolerant Distributed Stream Processing Systems. In *Proc. of the 29th IEEE Int'l Conference on Distributed Computing Systems, ICDCS '09*, pages 173 –182.

[4] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative Out-Of-Order Event Processing with Software Transaction Memory. In *Proc. of the 2nd ACM Int'l Conference on Distributed Event-Based Systems, DEBS '08*, pages 265–275.

[5] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems (2nd ed.). chapter "The primary-backup approach", pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[6] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.

[7] G. Cugola and A. Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[8] 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.*, 34:375–408, September 2002.

[9] M. Franceschetti and J. Bruck. A Leader Election Protocol for Fault Recovery in Asynchronous Fully-Connected Networks. Technical report, California Institute of Technology, 1998.

[10] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proc. of the 21st IEEE Int'l Conference on Data Engineering, ICDE '05*, pages 779 – 790.

[11] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. A Comparison of Stream-Oriented High-Availability Algorithms. Technical Report CS-03-17, Brown University, September 2003.

[12] J.-H. Hwang, U. Cetintemel, and S. Zdonik. Fast and Highly-Available Stream Processing over Wide Area Networks. In *Proc. of the IEEE 24th Int'l Conference on Data Engineering, ICDE '08*, pages 804 –813.

[13] 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. of SIGMOD '08*, pages 1303–1306, 2008.

[14] G. Jacques-Silva, B. Gedik, H. Andrade, and K.-L. Wu. Language Level Checkpointing Support for Stream Processing Applications. In *Proc. of the 39th IEEE/IFIP Int'l Conference on Dependable Systems and Networks, DSN '09*, pages 145 –154.

[15] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant Stream Processing using a Distributed, Replicated File System. *Proc. of VLDB Endow.*, pages 574–585, 2008.

[16] D. O'Keeffe and J. Bacon. Reliable Complex Event Detection for Pervasive Computing. In *Proc. of the 4th ACM Int'l Conference on Distributed Event-Based Systems, DEBS '10*, pages 73–84.

[17] B. Ottenwälder, B. Koldehofe, U. Ramachandran, and K. Rothermel. MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing. In *Proc. of the 7th ACM Int'l Conference on Distributed Event-Based Systems, DEBS '13*.

[18] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[19] Z. Sebepou and K. Magoutis. CEC: Continuous Eventual Checkpointing for Data Stream Processing Operators. In *Proc. of the 41st IEEE/IFIP Int'l Conference on Dependable Systems and Networks, DSN '11*, pages 145 –156.

[20] M. Völz, B. Koldehofe, and K. Rothermel. Supporting Strong Reliability for Distributed Complex Event Processing Systems. In *Proc. of the IEEE 13th Int'l Conference on High Performance Computing and Communications, HPPC '11*, pages 477 –486.