Supporting Strong Reliability for Distributed Complex Event Processing Systems

Marco Völz Boris Koldehofe Kurt Rothermel
Universität Stuttgart
Institute of Parallel and Distributed Systems
Universitätsstr. 38, 70569 Stuttgart, Germany
Email: ⟨firstname.lastname⟩@ipvs.uni-stuttgart.de

***Abstract*—Many application classes such as monitoring applications, involve processing a massive amount of data from a possibly huge number of data sources. Complex Event Processing (CEP) has evolved as the paradigm of choice to determine meaningful situations (complex events) by performing stepwise correlation over event streams. To keep up with the high scalability demands of growing input streams, recent approaches distribute event correlation over several correlation nodes. However, already a failure of a single correlation node impacts the correctness of the final correlation result. In this paper, we illustrate the importance of a strong reliability semantics for CEP in the context of a monitoring application in a distributed production environment. Strong reliability ensures each complex event is detected and delivered exactly once to each application entity, and cannot be guaranteed by the naive application of established replication principles. We present a replication scheme which ensures strong reliability in an asynchronous system model and can be applied to an arbitrary distributed CEP system. The algorithm tolerates f simultaneous failures by introducing f additional replicas for each correlation node. We prove correctness as well as evaluate the overhead introduced by the algorithm. Results show, that the overhead scales linearly with the number of deployed replicas and the node failure rate.**

***Keywords*-failure recovery; replication; reliability; complex event processing; monitoring**

I. Introduction

Monitoring applications, e. g. in the domain of real-time production monitoring, have to process huge amounts of information and send alerts or trigger some action if something noteworthy happens. Usually, information is provided in a steady stream of separate events which are detected by several sensors. One part of a monitoring application's task is to send alerts if some important information is *filtered* from the event stream, as for example when the production rate for an assembled part changes significantly. Additionally, the application can *detect and infer* more complex *situations* from the information provided by different sensors, for example how a limited storage space should be utilized in order to keep production cost low. This way, alerts can be sent to the facility's management and appropriate actions can be taken.

Using a CEP system, detected situations can again be combined to derive even more complex situations, introducing new levels of abstraction. Therefore, larger streams of information can be used by different applications with different information needs. Distributed CEP systems [1]–[4] promise significant improvement over the traditional centralized approaches (e. g. [5]–[7]) in terms of scalability. Nevertheless, in the field of monitoring systems not only scalable but also reliable systems are needed.

However, up to now providing reliability has not received a lot of attention in the area of distributed CEP. In detail, applications are usually required to be available and provide correct results even if network nodes fail, as they may be responsible for huge amounts of money within a company or an extremely valuable asset in the field of surveillance. Even a single event that was not received by the user but did occur (*false negative*) or that was received but did not occur (*false positive*) can result in huge unnecessary cost or damage. If duplicate events are not filtered during intermediate processing, false positives can arise. Therefore, the context of monitoring applications needs a distributed complex event processing system that offers not only high availability but also prevents false positives, false negatives, and processing of duplicate events.

While replication is a well-established concept to increase availability of individual functionality, it does alone not meet the requirements for strong reliable distributed CEP. Using a standard replication scheme in this context *creates* false positives, false negatives and duplicates. Existing approaches from other stream processing systems are not readily applicable, as they require using special operators or adding sophisticated methods to extract local state.

In this paper, we present an approach that uses active replication to ensure good responsiveness for reacting to failures in the event correlation network. As a main contribution, we propose an algorithm that coordinates the replicas for each processing node. We prove that the algorithm ensures a strong reliability semantics in the presence of node failures. The algorithm avoids the large overhead for the synchronization of the replicas of traditional replication schemes which grow quadratically in the number of disseminated event messages. By adding only a small additional delay, our performance evaluation show that the algorithm introduces only a linear overhead. The event processing model used

Published in Proceedings of 13th IEEE International Conference on High Performance Computing and Communications (HPCC-2011), pp. 477-486. Banff, Alberta, Canada. Sept 2011.
© IEEE 2011
http://dx.doi.org/10.1109/HPCC.2011.69

in this paper matches the requirement to many existing distributed CEP systems and can therefore be transferred easily to enable monitoring applications with high scalability and strong reliability requirements.

The remainder of this paper is structured as follows: Section II stresses the need for a strong reliability semantics and shows a motivating example. Section VII covers related work. A system model and formalization of a CEP system is given in Section III. We define the problem of a strong reliability semantics formally in Section IV. We describe our approach, show the algorithm and prove its correctness in Section V. Results of performance evaluation are presented in Section VI. Section VIII concludes this paper and deals with future work.

II. Motivation

Consider a distributed manufacturing process for a company that produces a complex product which is assembled from several parts, as illustrated in Figure 1. Each of these parts is constructed at a separate facility. For each of the production facilities exists a local management. A global management oversees the whole production process and issued the policy, that the storage space at each production facility should be kept to a minimum in order to reduce overall production cost. A complex event processing system is used to support the local management of each facility by deciding, how the available warehouse storage space should be divided among all parts necessary for the local assembly and all parts that are produced.

To this end, the following situations will be detected at each facility:

Situation A: Production rate for part of type x: Use the event notification that a part assembly has been completed to derive the current production rate for each part type over a given amount of time. Whenever this rate changes, a situation of type A is sent to all downstream facilities. This situation is correlated using the following event:

- Event A: Part of type x produced (i. e. construction finished)

Situation B: Part of type x should be accepted for storage in the local warehouse at rate y: Use information about the local production process and the incoming and outgoing rates of all parts in order to determine, how the local storage space should be divided among all part types. This situation is correlated using the following events:

- Event A: Part of type x produced (i. e. construction finished)
- Event B: Part of type x consumed (i. e. construction started)
- Event C: Part of type x stored (i. e. new part arrived from other facility)
- Event D: Part of type x delivered (i. e. part shipped to other facility)
- Event E: Current price for additional storage per m^2
- Event F: Current ratio of events of type B and C for part x (i. e. how much faster are parts needed at this facility than shipped from other facilities?)

It can be seen that events of type A and C reduce the available storage space at a facility, while events B and D increase the available space. Event E determines the cost for renting additional storage space and events of type F indicate for each part type, how many parts need to be stored. Whenever the production rate at one of the upstream facilities or at the local facility changes, a recalculation of the optimal distribution of the local storage space among all part types is triggered. The new situation of type B is sent to all upstream facilities for the respective part types.

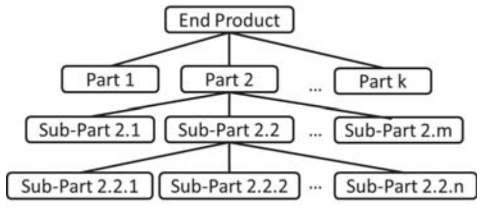


Figure 1. Production hierarchy: Schematic overview of a distributed manufacturing process. The end product is manufactured from parts which itself are based on several sub-parts. Each manufacturing process is located at another facility. Other edges are left out for clarity.

From this scenario, we can see that the information provided by a situation of type B should be always correct: When a part is shipped from one facility to another while there is no storage space available, new space has to be rented for a possibly high cost. Even higher cost can result from a contract penalty, if too few parts of a certain type are shipped because a change in storage space distribution was *not* indicated and therefore a customer's order cannot be fulfilled on time. Therefore, distributed CEP can only be used in this scenario, if the complex event correlation system produces correct information at all times.

By simply using replication for each node participating in the CEP system, this cannot be guaranteed. Consider for example a failure of the node detecting Situation B which had already received a certain amount of events of type A, indicating new available storage. By switching to the replica, this information is lost, and the event that a part of type x can be stored, is not sent. Thus, a false negative occurs. Similarly, if the same node had only received event of type C, indicating additional usage of storage, switching to the replica might result in a false positive event that a part of type x can be stored. Therefore, either the state of all replicas of a CEP node has to be synchronized, by using active or passive replication, or a complete log of all incoming events has to be available that can be re-processed by a replica.

Regardless of which strategy is used to maintain a synchronized state for the replicas, duplicate events will be created. First, consider the above example detecting Situation B if duplicate events are processed. If an event of type A is processed more than once, a false positive might be created. If an event of type C is processed more than once, a false negative might be created.

Next, we show how duplicates arise when using active replication, passive replication or logging. With active replication, duplicates are created all the time, as all replicas process and create the same events. With passive replication, a replica might incorrectly assume the original has died and start processing, as we are operating in an asynchronous system. Using logging, regular acknowledgments have to be sent to prune the logs. If a node crashed before sending an acknowledgment for an event, its replica will receive the same event, which might result in creating a duplicate. Therefore, we need an algorithm to detect and filter duplicates before processing them.

III. System Model

We assume a network of n nodes with unique identifiers of which at most f nodes fail simultaneously. Each node fails with the probability p_{fail}; node failures are not correlated. The considered failure model is *fail-stop*. Furthermore, we use a failure detection at each node such that eventually, no crashed node is considered to be correct by any other correct node.

The nodes are connected via a communication network of reliable FIFO point-to-point links (e. g. via TCP), i. e., links can only fail temporarily, *eventually* deliver in FIFO order and do not create messages. The correlation rules are distributed over the nodes. The data sources are connected reliably to the CEP system. All rules that together define a situation form a tree, called *query tree*. A query containing this tree is sent to the system by an application entity. The rules are deployed on nodes selected by the system's placement algorithm, such as [8], [9].

A logical timestamp is attached to all events by their senders. When referring to a certain timestamp, we denote an event e_k and its timestamp $t_k \in T$ as a tuple (e_k, t_k). If timestamps are not important, we use the short notation e_k. The underlying CEP engine is either correct or not working at all; i. e., even failed nodes make no incorrect correlation. Each node has only *one* source per event type. This does not imply that no other sources for this event type can exist in the system. We make no other assumptions for time and synchronization of nodes. Thus, the event correlation is *asynchronous*.

An asynchronous system model does not imply high communication delays. It simply states that we do not assume an upper bound for message delay. This is useful, if there actually is no such upper bound, or—and this is a reason very relevant for practitioners—if the upper bound would be significantly high. As in synchronous systems all reliability mechanisms—e. g. detecting a failed node and re-sending certain events—depend on the upper bound for message delay, algorithms for failure recovery might suffer from a severe performance penalty compared to an algorithm that does not need an upper bound [10]. Note, however, that the algorithm presented in this paper will work in synchronous systems, too.

To ensure the order of correlated events is deterministic, a well-defined selection function is used to pick an event for correlation, if more than one would be available for correlation. This can be enforced in most of the prevalent languages, e. g. by using consumption policies. Thus, the order of outgoing events depends directly on the order of the incoming events.

The order in which a node sends events of a certain type x corresponds to the order in which it received and processed the events needed for the correlation of the event type x.

Formally, a *CEP system* $S = (\widehat{\Gamma_S}, \widehat{\Sigma_S}, \tau_S, \Omega_S)$ provides a function $\tau_S : \widehat{\Gamma_S} \rightarrow \widehat{\Sigma_S}$ between the set of primary event types $\widehat{\Gamma_S}$ and a set of types of correlated situations $\widehat{\Sigma_S}$. The function τ_S is determined by the set of correlation rules Ω_S.

An individual correlation rule $\omega \in \Omega_S$ operates on a subset of the set of all events—primary or complex. A corresponding output event is created, if predefined conditions are met. All events provide timestamps; the outgoing event's timestamp is derived solely from the incoming events' timestamps. As an example, on detecting a sequence of events of type A and B within 10 seconds, rule ω_1 deployed at node h_1 should create an event of type C. Events of type A be primary, events of type B complex events. Assume the event sequence $(\gamma_A, 10), (\gamma_A, 25), (\sigma_B, 30)$ of events with their corresponding timestamps as input on h_1. After receiving the third event, an event $(\sigma_C, 30)$ is created; the function $g()$ for computing the new timestamp just copies the last event's stamp: $\omega_1(\gamma_A, 25), (\sigma_B, 30)) = (\sigma_C, g(25, 30) = 30)$

Formally, a rule $\omega \in \Omega_S$, correlates a given set of primary or complex events $e_{k_i} \in 2^{(\widehat{\Gamma_S \cup \Sigma_S})}$ with corresponding timestamps $t_{k_i} \in T$ to a complex event $\sigma_j \in \widehat{\Sigma_S}$, also attached with a timestamp:

$$\omega : 2^{(\widehat{\Gamma_S} \cup \widehat{\Sigma_S})} \times T \rightarrow \widehat{\Sigma_S} \times T, \text{ and}$$

$$\omega(\{e_{k_i}, t_{k_i}\}) = (\sigma_j, t_j), \text{ with } t_j = g(t_{k_i}) \qquad (1)$$

Note, that the time *when* the events were actually correlated is not relevant for the resulting event's timestamp. Consequently, timestamp computation on node h is independent of the node's local clock. Therefore, no clock synchronization between correlation nodes is necessary; any two nodes that apply the same rule to the same set of input events will create the same complex event with the same timestamp.

IV. Strong Reliable CEP

Duplicates, false positives, and false negatives arise in a CEP system at *runtime*. Thus, formalizing these occurrences requires to look at one individual run, called an *instance* of a CEP system S. An instance of S comprises additionally the set of actually processed input events, each containing an event type and additionally a timestamp $t \in T$. Therefore, $\Gamma_S \subseteq \widehat{\Gamma_S} \times T$. Similarly, the set of actually correlated situations is constructed: $\Sigma_S \subseteq \widehat{\Sigma_S} \times T$. Both sets are partially ordered by the relation $\prec_S$ according to the timestamps of their elements. Duplicates can be identified using timestamps of Σ_S: We denote Σ_S^t as the set of all events that were created in the system S with the timestamp t; therefore $\bigcup_t \Sigma_S^t = \Sigma_S$. A complex event σ_k is a duplicate, if two occurrences are in the same set of correlated events at time t, Σ_S^t. Note that Σ_S is actually a multiset as it might contain more than one occurrence of the same event with an identical timestamp.

False positives and false negatives in the set of situations are defined with respect to a perfect set of situations. This set is constructed using a *reference system*. A reference system R creates a perfect set of correlated situations Σ_R based on the input data of a CEP system S. In other words, R creates the output set of a failure-free execution of S.

The perfect system R takes the input (Γ_S, τ_S) and simulates $\tau_S(\Gamma_S) \rightarrow \Sigma_R$, ordered according to their timestamps by $\prec_R$. Observe that R is well-defined as it uses the same selection function as S, i. e. it allows no ambiguity in terms of correlation order. Afterwards, false positives and false negatives can be identified by comparing the situation set of the actual execution Σ_S and the reference set Σ_R: All events that are in the actual situation set, but not in the reference set are false positives. Similarly, all events that are in the reference situation set, but not in the actual set, are false negatives.

Using the formalisms, we can now define the semantic for our algorithm using safety and liveness properties:

Definition 1: Strong Reliable CEP
A CEP System $(\widehat{\Gamma_S}, \widehat{\Sigma_S}, \tau_S, \Omega_S)$ comprising the set of incoming primary event types Γ_S, the set of complex event types Σ_S, the correlation function τ_S defined on these sets by the set of correlation rules Ω_S provides strong reliable CEP with respect to its reference system R, if the following properties hold for *all nodes* and *any arbitrary instance*.

1) No false positives: If $e \in \Sigma_S \Rightarrow e \in \Sigma_R$
2) No false negatives: If $e \notin \Sigma_S \Rightarrow e \notin \Sigma_R$
3) No duplication: $\forall t \in T : s\,|\{e|(e,t) \in \Sigma_S^t\}| \leq 1$
4) No false ordering: $\forall e, e' \in \Sigma_S : e \prec_S e' \Leftrightarrow e \prec_R e'$
5) Completeness: $\exists t : \bigcup_{t'<t} \Sigma_S^{t'} = \Sigma_R$

In the following, we present an algorithm that provides strong reliable CEP for any CEP system by using replication. For practical relevance of the solution it is important to prevent quadratic runtime overhead. Therefore, coordinating the replicas efficiently to keep the cost low in terms of additional needed event messages and space requirements is the main objective.

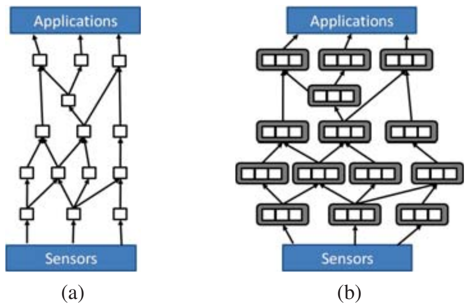


Figure 2. (a) Normal distributed correlation: each node deploys a set of correlation rules. (b) Replication: 3 replicas for each set of correlation rules.

V. Approach for Replica Coordination

First, we give an overview of the algorithm, showing its objectives and basic tasks. Second, we present stepwise, how each task is fulfilled. Third, we give boundaries for the algorithm's overhead and prove its correctness by showing that the properties of Definition 1 hold.

A. Algorithm Overview

The algorithm operates on a network of replicated correlation nodes: As seen in Section III, all nodes deploying correlation rules form a graph. At first, this graph of correlation nodes has to be augmented with f additional replicas for each node, so that each set of rules is replicated $f+1$ times. Figure 2 shows an example for $f = 2$. The algorithm coordinates the replicas to achieve strong reliability semantics. At the same time, it tries to keep the cost in terms of additional needed event messages and additional space requirements at each node low.

We adapt the approach of active standby, in which all nodes are assigned f additional replicas hosting the same functionality. As these replicas are *active*, they receive, process and produce the same events as the original node. If one node fails, f other nodes remain. Therefore, in total f simultaneous failures can be tolerated. Maintenance of an accurate history of events is implicit in active standby, as long as no messages are being lost during execution.

However, active standby induces quadratic message overhead during runtime. Therefore, we introduce two node operation modes: *normal* and *leader* of which only leaders send their outgoing events. All other replicas store the events in their outgoing queue. The queues have to be pruned regularly to reduce storage costs and prevent overflows. We use a leader election algorithm that guarantees that eventually *at least one* leader will be elected out of all correct replicas. Note, that more than one replica might be elected as leader.

To cope with this, we provide a strategy to filter out duplicate events in a node's incoming queue, before they are processed. This mechanism is needed, as events could be delayed arbitrarily long and saving all incoming events and checking for duplicates is not a feasible solution.

B. Algorithm Steps

The main algorithm is shown in Figure 3. It provides methods for executing regular tasks and handling incoming messages. We will explain the algorithm by describing the three individual steps of outgoing queue pruning, duplicate detection, and leader election.

Pruning the outgoing queue: All replicas but the leader buffer the outgoing events instead of sending them. A leader collects acknowledgments for all events it sends to other nodes. If an event has been acknowledged by all receivers it can safely be removed from the replicas' queues. Therefore, a leader sends a list of all completely acknowledged events to all replicas. This is done in regular intervals rather than in a per-event basis to reduce cost.

The Base Algorithm in Figure 3 shows the methods to coordinate this: Lines 7–18 show the processing of an incoming primary or complex event. At first, an acknowledgment is sent to the event's sender (line 8). If the event is no duplicate, the event's timestamp is archived and correlations are executed (lines 12 and 13). All newly correlated complex events are handled by the procedure PROCESSOUTGOINGEVENT in lines 19–27. There, an event is first saved to the outgoing queue (l. 22) and send to all receivers interested in this event *only if* the node acts as leader (l. 23). Lines 28–35 show the processing of acknowledgments. The acknowledgment is archived (l. 30) and if the corresponding event has been acknowledged by all receivers, it is removed from the outgoing queue and added to a temporary list of all received acknowledgments (ll. 31–34). On a regular basis, all completely acknowledged events are sent to all replicas (l. 37). This is shown in the method SENDCOMPLETELYACKEDEVENTS (ll. 40–45). Lines 46–50 show how a replica receives an incoming list of completely acknowledged events and removes them from its incoming queue.

Leader election: We use a leader election that provides eventual correctness in an asynchronous system model. Eventual agreement is *not* necessary.[1] We use latency measurements to the previous node as a leader election criteria, trying to minimize end to end latency in a greedy one-hop manner. As a tie-breaker we select the lowest unique node id. All other sorts of criteria could be used for electing a leader, as long as eventual correctness is guaranteed. Leader election is started, when a leader is no longer available due to node or link failure. The replica detecting the failure determines all other replicas eligible for leader voting and starts the voting procedure. If multiple voting procedures are started, we use the procedures' starter node id as a tie-breaker to decide which procedure will prevail and which will be canceled.

[1]This means that eventually at least one leader will be elected. However, the replicas do not need to agree on one leader, therefore there may be multiple of them.

The Base Algorithm in Figure 3 shows in line 38 that each node checks, whether the leader is still alive. If not, a LEADERDEADEVENT is triggered. If a leader l_1 detects a failed link to another leader l_2, it sends a LEADERDEADEVENT to the remaining replicas of l_2. Additionally, if l_2 detects that it cannot receive events anymore, it sends a LEADERDEADEVENT itself. When such an event is received, a new leader election is started (ll. 51–53). As a result, at least one other replica will start sending outgoing messages.

Duplicate detection: Duplicates can be identified in the incoming queue of each node using timestamps. Following the construction principles of timestamps described in Section IV and the *no false ordering* property of Definition 1, the timestamps for each event type have to be strictly monotonically increasing. In the system model, we restricted our incoming connections to *one* per event type. Therefore, if an event is received with a timestamp that is not greater than the already archived timestamp for this event type, a duplicate is detected and can be discarded before processing it.

The Base Algorithm in Figure 3 shows in line 9 that only event that are not identified as duplicates are processed. A duplicate is detected if the timestamp of an incoming event is not newer than the one already archived for the corresponding event type.

C. Analysis: Bounds and Proof

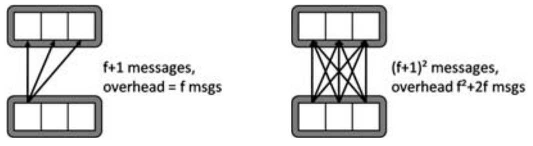
After presenting the algorithm used for providing strong reliability, we analyze it theoretically. In a performance analysis we consider best case and worst case overhead of our algorithm. Afterwards, we prove that our algorithm provides strong reliable CEP according to Definition 1.

The overhead created by maintaining f additional replicas compared to using no replication at all can be measured in terms of additional sent events. We show the additional messages sent in the best case and in the worst case for our approach (cf. Figure 4). By sending with all replicas, the worst case overhead is in $\mathcal{O}(f^2)$ for f replicas. An unadapted active standby approach would deliver exactly this performance. Whereas in the best case only *one* replica per node is sending. This creates overhead which is in $\mathcal{O}(f)$ for f replicas. The best case will be achieved, during periods when synchrony could be reached in the system. If this is never the case, the performance will most likely be worse, as more than one leader will be elected.

The additional space requirements for each node for storing the timestamp array are in $\mathcal{O}(e)$ for e different types of events. Additionally, replicas need to buffer the outgoing events. As events remain in the buffer until they have been acknowledged by all receiving replicas, the queue can grow infinitely. Thus, if the only missing acknowledgments are

```
Use FailureDetector
Boolean $isLeader$
List $completelyAckedEvents$
List $replicaList$                    ▷ The set of all replicas
List $receiverList$    ▷ Set of all receivers of outgoing events
Queue $outgoing$                          ▷ The outgoing queue

upon incomingEvent($iE$)
  SENDACK(iE)              ▷ Acknowledge the incoming event
  if (ISDUPLICATE($iE$)) then
      skip
  else
      ARCHIVETIMESTAMP(iE.$type$,iE.$ts$)
      newComplexEvents ← EXECUTECORRELATIONS($iE$)
      for all (event ∈ newComplexEvents) do
          PROCESSOUTGOINGEVENT(event)
      end for
  end if
end

procedure PROCESSOUTGOINGEVENT($event$)
    receivers ← receiverList.GET(event.type)
    for all ($receiver$ ∈ receivers) do
        outgoing.ENQUEUE(event, $receiver$)
        if ($isLeader$) then
            SEND(event, $receiver$)
        end if
    end for
end procedure

upon Acknowledgment($ack$, $sender$)
  ackedEvent ← ack.$event$
  ADDACKNOWLEDGMENT(ackedEvent, sender)
  if (ISCOMPLETELYACKED(ackedEvent)) then
      outgoing.REMOVE($ackedEvent$)
      completelyAckedEvents.ADD($ackedEvent$)
  end if
end

upon regularBasis()
  SENDCOMPLETELYACKEDEVENTS()
  CHECKLEADERALIVE()
end

procedure SENDCOMPLETELYACKEDEVENTS()
    for all ($rep \in replicaList$) do
        SEND($completelyAckedEvents, rep$)
    end for
    completelyAckedEvents.REMOVEALL()
end procedure

upon ReceiveAckedEventList($listOfAckedEvents$)
  for all ($event$ ∈ listOfAckedEvents) do
      outgoing.REMOVE($event$)
  end for
end

upon LeaderDeadEvent()
  STARTLEADERELECTION()
end
```

Figure 3. Base Algorithm: Behavior of a node h



(a) Best Case (b) Worst Case

Figure 4. Overhead in terms of events for maintaining f additional replicas per node, shown for $f = 2$: (a) In the best case, only one leader sends events to all replicas of another node. (b) In the worst case, each replica assumes itself to be a leader and thus all replicas send events to all replicas of another node.

from nodes which are detected to have failed permanently, the messages are removed from the queue.

Claim 1: Strong Reliable CEP
The Base Algorithm in Figure 3 provides strong reliable distributed CEP if at least one correct replica for each correlation node exists.

Correctness: We will prove that our algorithm provides strong reliability, by showing that the properties of Definition 1 hold: *No false positives*, *no false negatives*, *no duplication*, *no false ordering*, and *completeness*. We prove each of the properties separately in a lemma.

Lemma 5.1: No false ordering
Every rule ω with $\omega(e_i, t_i) = (\sigma_k, t_k)$ creates events σ_{k_j} for $j \in \mathbb{N}$ with monotonically increasing timestamps t_{k_j}.

Proof: We assume h to be the first node to produce out-of-order events. As timestamp creation only depends on the incoming event's timestamps and we can assume that correlation itself is correct, the sequence of timestamps for resulting situations σ_{k_j} is directly derived from the timestamps of incoming events (cf. Equation 1). As event correlation is order preserving (cf. Section III), h did already receive out-of-order events. This contradicts with our initial assumption. ■

Lemma 5.2: No duplication
No node executing the Base Algorithm (cf. Figure 3) processes duplicate events.

Proof: The duplicate detection algorithm marks two events σ_k and σ'_k as duplicates, if they are correlated by the same rule ω_k using the exact same input events e_i, i. e. $\omega(e_i, t_i) = (\sigma_k, t_k) \wedge \omega(e_i, t_i) = (\sigma'_k, t'_k)$. As for timestamp computation the same function $g()$ is applied to the timestamps t_i of events involved in correlation, σ_k and σ'_k must have the same timestamps, i. e. $t_k = t'_k$. Thus, they are identified as duplicates by the duplicate detection algorithm of Section V. ■

Lemma 5.3: No false negatives
With algorithm 3 a CEP System S produces no false negatives w. r. t. a reference system; i. e. if $e \notin \Sigma_S \Rightarrow e \notin \Sigma_R$.

Proof: We need to show the following properties:

- With acknowledgments, no events will get lost due to node or link failures

- No messages will get lost due to outgoing queue pruning
- No messages will get lost due to duplicate filtering

No loss due to acknowledgments: A sender removes an outgoing event if and only if *all* receivers have acknowledged the event or if the only pending acknowledgments are from nodes that are permanently considered faulty. As there are $f+1$ receivers for each subscription of which only f can fail simultaneously, at least one correct receiver remains. If the sender itself fails *before* all receivers have received acknowledged the event, f sender replicas remain, of which only $f-1$ can additionally fail. Eventually accurate leader election ensures that correct replicas will forward all remaining events.

No loss due to pruning of outgoing queue: As only events that have been acknowledged by $f+1$ receivers can be pruned from the outgoing queue. In any case there will at least remain one correct receiver that will continue processing.

No loss due to duplicate filtering: With Lemma 5.1, we have guaranteed that timestamps of events of the same type are strictly monotonically increasing at the receiver side. Events are not received in another order than they are sent. Thus, events with a newer timestamp have been created later and also sent at a later point. Therefore, we do not filter out non-duplicates by checking for timestamps. ■

Lemma 5.4: No false positives
With algorithm 3 a CEP System S produces no false positives w. r. t. a reference system R; i. e. if $e \in \Sigma_S \Rightarrow e \in \Sigma_R$.

Proof: With the no creation property for links (cf. Section IV) and duplicates and false negatives already proven impossible (cf. Lemma 5.2 and Lemma 5.3), false positives are impossible. ■

Lemma 5.5: Completeness
All events that can be correlated from the incoming streams will be correlated eventually.

Proof: Since leader election is eventually correct and at most f of the overall $f+1$ replicas available for each set of correlation rules can fail, correlation itself will not stop. With Lemma 5.3 and Lemma 5.4, all events will reach their destination eventually, thus $\exists t : \bigcup_{t'<t} \Sigma_S^{t'} = \Sigma_R$. ■

VI. Evaluation

In the evaluations we aimed to understand the performance characteristics of the replication scheme independent of specific operator implementations or other system dependent details. Therefore, we implemented the scenario presented in Section II in the PeerSim simulation framework [11].

Our setup consists of a correlation network that comprises 70 nodes deploying. During regular execution, more than 40 000 primary event messages served as input for the system, resulting in about 13 000 situations delivered to the global management. Overall, more than 170 000 event messages were created in the system.

The overhead introduced by our algorithm compared to a system using no replication is shown for both cases, normal behavior with no failures and in the case of node and link failure. For normal behavior, we analyze the overhead depending on the number of additional replicas per node. In the case of failures, we present a more detailed analysis of the overhead depending on failure probability and number of additional replicas per node.

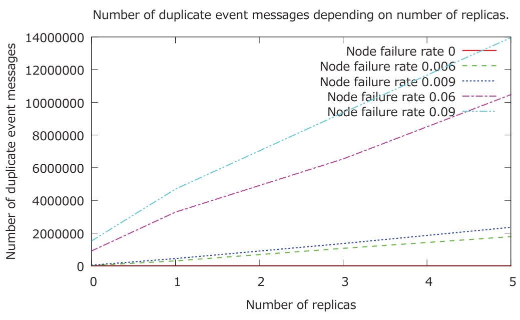
Additionally, we show the average amount of simultaneously active leaders providing the same functionality. Given the theoretical analysis of best and worst case in the previous section, the actual system behavior is of great interest.

In each experiment, f additional replicas for each node are deployed. Nodes are assumed to fail with probability p_{fail} at each simulation step. We show experimental results for $0 < f < 5$ and $0\% < p_{fail} < 0.1\%$. The results for each measurement were determined in 500 experiment runs.

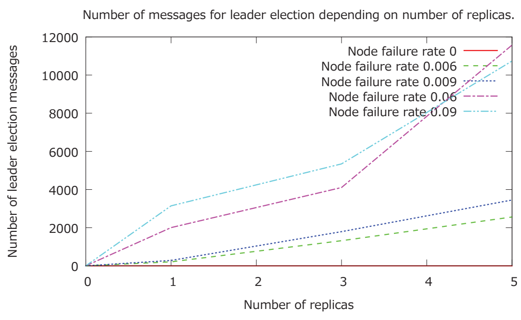
Overhead: Figure 5b shows the overhead for different node failure rates depending on the number of deployed replicas per node. As it can be seen, our approach incurs linear runtime overhead in terms of event messages. During recovery it happens that messages which had been received already but not acknowledged are transferred a second time. It also can be seen that the amount of duplicates scales linearly with the amount of replicas deployed.

Figure 6 shows the overhead for different numbers of replicas depending on the failure probability for each node. We can see that the amount of duplicates scales linearly with the node failure rate. The scaling factor is higher, the more replicas are deployed. In principle, the same development can be seen for the leader election messages. However, the actual transferred amount is highly depending on link characteristics and which node makes the first leader proposal. Therefore, the more replicas are deployed, the more fluctuations are visible in these measurements.

Average leader count: The theoretical analysis of the algorithm's performance already provided boundaries: Linear message overhead in the best case and quadratic message overhead in the worst case. Decisive for the actual performance is the amount of leaders that are active simultaneously for each functionality. The closer the average leader count is to 1, the better performs the algorithm; i. e. fewer duplicate messages are created. Figure VI shows that the measured average leader count is very close to the optimum of 1. At the same time it can be seen that the average count is decreasing, if the nodes are very unreliable, because fewer correct nodes exist within a replica group at a time which could compete for the leader position. Multiple active leaders occur when a replica makes the incorrect assumption that the current leader has failed and starts a new election process. Values below 1 indicate that at certain times no correct replica for an operator exists; the selected failure rate

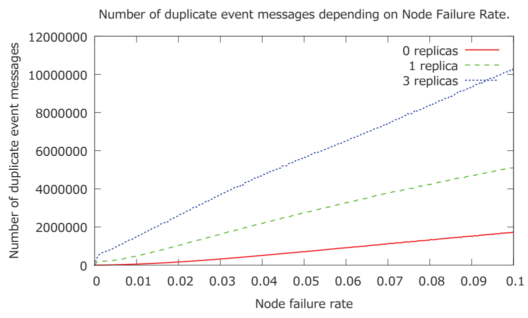


(a) Duplicate event messages

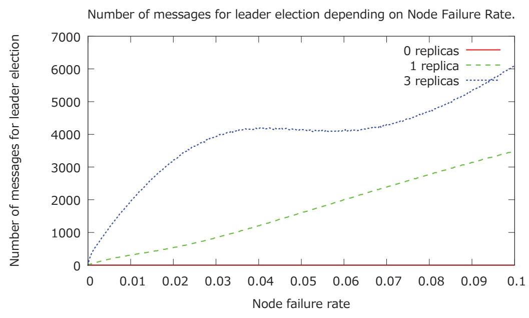


(b) Leader messages

Figure 5. Overhead of replica placement in terms of duplicate event messages and messages for leader election dependent on the number of replicas. Each node deployed $0 < f < 5$ additional replicas. Up to f nodes fail simultaneously. (a) Shows the amount of duplicate event messages. (b) Shows the amount of messages for leader election.



(a) Duplicate event messages



(b) Messages for leader election

Figure 6. Overhead of replica placement in terms of duplicate events and messages for leader election dependent on node failure probability. Each node deployed $0 < f < 3$ additional replicas. Up to f nodes fail simultaneously; the failure probability is $0\% < p_{fail} < 0.1\%$. (a) Shows the amount of duplicate event messages. (b) Shows the amount of messages sent for leader election.

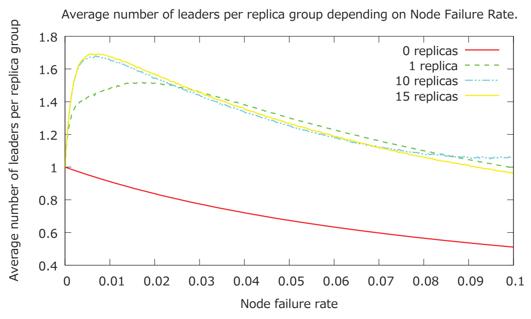


Figure 7. Average count of simultaneously active leaders.

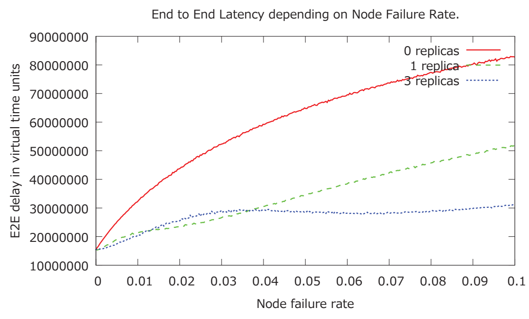


Figure 8. Average end to end latency using the node's latencies to the previous hop as decision criterion.

is too high compared to the number of operators available.

Results show that the mechanism of leader election is suitable to keep the amount of event messages at an acceptable level of linear growth. Despite the fact that replicas do not need to agree on a common leader, most of the time only one leader is active per functionality. At the same time, the additional overhead introduced through leader election is marginal compared to the total number of event messages.

Leader election strategy: Figure VI shows the development of the end to end latency for all detected situations depending on the node failure rate. It can be seen that using only three replicas for each node, the latency for a very unreliable system with nodes with a failure rate of $p_{fail} = 0.1$ can be kept at almost the same level as with no replicas but nodes with a failure rate of $p_{fail} = 0.005$.

We can also see that a higher failure rate introduces an additional delay. The delay is higher, the less replicas are used in the system. This is due to the fact that with no or only few replicas it happens more often, that there is no correct replica for an operator. Thus, processing is delayed until a failed node recovers.

VII. Related Work

Recently, an approach for dealing with unreliable communication channels when delivering events to a CEP system has been proposed [12]. With this algorithm, the impact of events not received by the CEP system can be limited by omitting certain correlations. However, failures of correlation nodes are not considered in this approach.

To the best of our knowledge, no distributed CEP system features algorithms to address reliable event delivery to consuming applications in the face of failures of correlation nodes. In this section, we analyze reliability approaches of other fields and judge their applicability for a distributed CEP system.

Methods for group communication which ensure ordered exactly-once delivery such as reliable multicast, atomic broadcast or consensus require a synchronous system model or are not guaranteed to terminate even in the face of a single node failure [13], [14]. Therefore, they cannot be applied in our case.

Standard principles for replication of nodes include state machine replication, or active standby, and checkpointing, or passive standby [15], [16]. These concepts propose to either add active replicas for redundancy in the complete system and have multiple executions at the same time or to regularly synchronize an original with several passive replicas. However, active standby increases the event messages quadratically with the number of deployed replicas whereas passive standby requires a checkpoint at each state change if a lossless history of incoming events is to be provided. In both cases, the additionally consumed bandwidth is unacceptable for systems with high event rates. Furthermore, applying one of these strategies will lead to duplicate events. Therefore, duplicate filtering is required to guarantee strong reliability semantics.

For Distributed Stream Processing (DSP), which is also a paradigm for processing data in streams of potentially high rates, a few reliability approaches were proposed. These systems, however, typically provide operators less expressive than many CEP operators. The existing approaches can be divided in three categories: The first category adopts the characterization of DSP as "partial fault-tolerant" [17], [18]. 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 [19]–[22]. 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 (still) possible at all. In the scenarios we are examining, decisions might already been made based on incorrect information that are either very costly or impossible to correct. Therefore, correct information is needed at all times.

Solutions of the third category prevent the delivery and processing of incorrect information. They either implement active standby [23], checkpointing [24] or use a logging mechanism to record all messages missed during a node's downtime. On recovery of a node, these messages are resent and can be processed [25]. However, the active approach still increases the message load quadratically. Checkpointing requires frequent execution of sophisticated state-extraction algorithms that need either to be specified individually for each operator or require taking a full memory snapshot. State extraction either restricts the user to using a certain system or requires 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 slows down processing. Approaches using logging at upstream neighbors can only tolerate one failed node at a time; an extension for supporting multiple simultaneous node failures cannot be implemented easily. The same difficulties arise for employing combinations of checkpointing and logging.

Within many event-based systems, publish/subscribe middleware is used to mediate events between the producers (publishers) and their consumers (subscribers). In this field, reliability has been an active research topic. However, approaches only consider event routing. The state of a node—which is necessary for correct event correlation—is not considered.

None of the above strategies can be applied to an existing distributed complex event processing system for providing reliability while at the same time preventing false positives, false negatives and processing of duplicates with acceptable overhead.

VIII. Conclusion

Although reliability is the key for being able to use the benefits of CEP in application fields which are sensitive to incorrect information such as monitoring, it has hardly been addressed in the literature. This paper presented an algorithm that ensures a strong reliability semantics for an arbitrary distributed CEP system. The algorithm uses replication to increase the availability and additionally provides mechanisms to prevent false positives, false negatives and processing of duplicates. Communication between replicas is organized such that the induced overhead is kept low.

We proved correctness and provided practical performance evaluation that analyzed the induced overhead during normal execution and in the case of failures. Although the worst case behavior yields an overhead that is quadratic in the order of replicas per node, the evaluation results show that even under high failure rates the message overhead stays linear. The amount of additional messages for leader election and the duplicates created because of replication are only marginal compared to the total amount of transferred event messages. Therefore, the presented algorithm is a first step to enable the use of distributed CEP systems for scenarios with a high demand for scalability as well as reliability.

Although in relative terms a linear growth of sent event messages is acceptable, the absolute amount of sent messages is still significantly high due to replication. Thus, the bandwidth requirements are pretty high, even if no failures are experienced. However, this is the case with any approach using a proactive replication scheme.

For future work, we see two relevant contributions: While this work is an initial step to provide reliability for CEP, we want to determine and support other relevant semantics in highly distributed settings. Additionally, we want to investigate reducing the runtime overhead and bandwidth consumption e. g. by employing reactive replication.

Acknowledgment

This work has been supported by contract research "Internationale Spitzenforschung II" of the Baden-Württemberg Stiftung. The authors would like to thank Andreas Benzing, Stefan Föll, Adnan Tariq, and all anonymous reviewers for their helpful comments.

References

[1] P. R. Pietzuch, B. Shand, and J. Bacon, "A framework for event composition in distributed systems," in *Proc. of Middleware '03*, 2003.

[2] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski, "The PADRES distributed publish/subscribe system," in *FIW*, 2005.

[3] G. G. Koch, B. Koldehofe, and K. Rothermel, "CORDIES: Expressive Event Correlation in Distributed Systems," in *Proc. of DEBS '10*, 2010.

[4] B. Schilling, B. Koldehofe, U. Pletat, and K. Rothermel, "Distributed Heterogeneous Event Processing: Enhancing Scalability and Interoperability of CEP in an Industrial Context," in *Proc. of DEBS '10*, 2010.

[5] S. Gatziu and K. R. Dittrich, "Samos: an active object-oriented database system," *IEEE Data Eng. Bull.*, vol. 15, no. 1-4, pp. 23–26, 1992.

[6] S. Chakravarthy and D. Mishra, "Snoop: an expressive event specification language for active databases," *Data Knowl. Eng.*, vol. 14, no. 1, pp. 1–26, 1994.

[7] A. Adi and O. Etzion, "Amit - The Situation Manager," *The VLDB Journal*, vol. 13, no. 2, pp. 177–203, 2004.

[8] S. Rizou, F. Dürr, and K. Rothermel, "Solving the Multi-operator Placement Problem in Large-Scale Operator Networks," in *Proc. of ICCCN '10*, 2010.

[9] B. Schilling, B. Koldehofe, and K. Rothermel, "Efficient and Distributed Rule Placement in Heavy Constraint-Driven Event Systems," in *Proc. of HPCC '11*, 2011.

[10] M. Aguilera, "Stumbling over consensus research: Misunderstandings and issues," in *Replication*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. Volume 5959/2010, ch. 4, pp. 59–72.

[11] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris, "The Peersim simulator," 2004, `http://peersim.sf.net`.

[12] D. O'Keeffe and J. Bacon, "Reliable complex event detection for pervasive computing," in *Proc. of DEBS '10*, 2010.

[13] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[15] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[17] 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 INFOCOM '08*, 2008.

[18] G. Jacques-Silva, B. Gedik, H. Andrade, and K.-L. Wu, "Language level checkpointing support for stream processing applications," in *Proc. of DSN '09*, 2009.

[19] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, "Speculative out-of-order event processing with software transaction memory," in *Proc. of DEBS '08*, 2008.

[20] A. Brito, C. Fetzer, and P. Felber, "Minimizing latency in fault-tolerant distributed stream processing systems," in *Proc. of ICDCS '09*, 2009.

[21] 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*, 2008.

[22] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," in *Proc. of SIGMOD '05*, 2005.

[23] J.-H. Hwang, U. Cetintemel, and S. Zdonik, "Fast and highly-available stream processing over wide area networks," in *Proc. of ICDE '08*, 2008.

[24] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. of VLDB Endow.*, 2008.

[25] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proc. of ICDE '05*, 2005.