User-Constrained and Self-Adaptive Fault Tolerance for Event Stream Processing Systems

André Martin*, Tiaraju Smaneoto†, Tobias Dietze*, Andrey Brito†, Christof Fetzer*

*Technische Universität Dresden, Dresden, Germany
Email: andre.martin@tu-dresden.de / tobias.dietze@mailbox.tu-dresden.de / christof.fetzer@tu-dresden.de
†Universidade Federal de Campina Grande, Campina Grande, Brazil - Email: tiarajums@gmail.com / andrey@dsc.ufcg.edu.br

Abstract—Event Stream Processing (ESP) Systems are currently enabling a renaissance in the data processing area as they provide results at low latency compared to the traditional MapReduce approach. Although the majority of ESP systems offer some form of fault tolerance to their users, the provided fault tolerance scheme is often not tailored to the application at hand. For example, active replication is well suited for critical applications where unresponsiveness due to a background recovery process is not acceptable. However, for other classes of applications without such tight constraints, the use of passive replication, based on checkpoints and logging, is a better choice as it can save a significant amount of resources compared to active replication.

In this paper, we present StreamMine3G, a fault tolerant and elastic ESP system which employs several fault tolerance schemes, such as passive and active replication as well as intermediate alternatives such as active and passive standby. In order to free the user from the burden of choosing the correct scheme for the application at hand, StreamMine3G is equipped with a fault-tolerance controller that transitions between the employed schemes during runtime in response to the evolution of the given workload and the user’s provided constraints (recovery time and semantics, i.e., gap or precise). Our evaluation shows that the overall resource footprint for fault tolerance can be considerably reduced using our adaptive approach without consequences to the recovery time.

Keywords—fault tolerance, active replication, passive replication, active standby, passive standby, adaptation, deterministic execution, precise recovery, gap recovery

I. INTRODUCTION

With the advent of Google’s MapReduce [1] approach and its open source implementation Hadoop [2], large scale data processing has become increasingly popular. The simple and intuitive programming interface, and the available open-source tools enable practically anyone to perform data analysis in large scale. Moreover, through its disk-based approach for keeping intermediate results, fault tolerance in Hadoop comes practically for free. However, such benefits come with a high price as such systems perform data processing in a batching manner, delivering results only after job completion, which can take several minutes to hours. In fact, many current applications require results in second or even sub-second range and make batch processing alternatives unsuitable.

Event Stream Processing (ESP) is a set of techniques specifically crafted for low latency and continuous data processing. Although ESP systems exist already for more than a decade now, they are currently facing a true renaissance as more and more companies need real-time analytics to stay competitive. The usage of such systems enables companies to quickly react on relevant situations and opens up entirely new business opportunities. The examples where ESP systems are currently used range from click stream analysis¹, fraud and intrusion detection, recommender systems, online log file analysis, to high-frequency trading.

Inspired by the simplicity of MapReduce, a number of new and open-source ESP systems have emerged and gained traction over the past three years such as Apache Samza [4] (LinkedIn), Storm [5] (Twitter) and S4 [6] (Yahoo!). Although all of those systems have a simple MapReduce-like interface in common, they have different guarantees when it comes to fault tolerance. For example, Apache S4 can recover from faults by restarting an operator on a new node and loading a previous checkpoint of the operator’s state. However, in-flight events, which were not included in the checkpoint are simply lost, hence, only gap recovery is provided. On the contrary, Apache Storm guarantees no event loss through its transactional topologies, however, lacks appropriate mechanisms for state persistence.

Although the previously mentioned ESP systems offer fault tolerance to their users per se, the provided schemes are often only suitable for certain types of applications: While Apache S4 is a good choice for applications with stateful operators, where state cannot be recreated by simply reprocessing events, applications sensitive to event loss require schemes such as offered in Apache Storm.

Since the majority of ESP systems often employ only a single fault tolerance scheme, users have to choose from a pool of ESP systems rather than a pool of schemes that best matches the requirements of the application at hand. In fact, a wide variety of fault tolerance schemes for ESP systems is known in literature ranging from active replication [7], [8], active or passive standby [9], [10], to passive replication where a combination of checkpoint and logging (i.e., upstream backup [10]) is used.

Choosing the right fault tolerance scheme is often not a trivial task as there is a trade-off between recovery time and resource overhead imposed by each scheme. For example, using active replication, an operator can recover almost instantaneously, however, at the cost of consuming twice of

¹For instance, simple online or mobile advertisement through the participation in services such as the Google’s AdWords [3] platform is a common current usage of ESP tools.
the resources (CPU, memory and network). On the contrary, passive replication consumes only little additional resources for state persistence (disk) and the in-memory log upstream. However, it comes with the price of a long recovery time comprising the time it takes to load the most recent checkpoint from disk and replaying events from the upstream node’s in-memory log.

Fault tolerance schemes such as active or passive standby can be considered as intermediate or hybrid alternatives as they trade recovery time by resource consumption so that they can recover faster than passive replication, however, at a much lower resource usage cost compared to the use of full active replication.

For applications that have very tight constraints such as found in the financial trading sector, the choice of using active replication is clear as those applications do not tolerate downtimes of even a few seconds. However, there exists a wide variety of applications which are less critical and blocking for a few seconds is acceptable. Consider for example a recommendation system: During a recovery, an e-commerce site may not be able to serve its visitors with dynamically-updated recommendations while they are shopping. However, this degraded service will not necessarily lead to high financial losses, as opposed to financial trading or fraud detection applications. Hence, there is a huge potential for a variety of applications to save resources while still tolerating faults.

On the other hand, from the development perspective, application developers and data analysts often lack a comprehensive knowledge about fault tolerance concepts and their implications with regards to recovery times and resource footprint. However, even then, users have clear constraints such as the (i) maximum amount of time an application may stay unresponsive due to recovery and if (ii) events may be lost or not.

Considering those constraints, choosing an appropriate scheme seems to be straightforward. However, ESP systems are highly dynamic systems where the natural fluctuation in throughput originating from online data sources can highly influence the time an operator may need to recover. Consider for example an application that processes tweets using a time-based sliding window. In case the user opted for passive replication, the application may recover quite quickly if the throughput is low, as the state it keeps is relatively small. Nevertheless, with increased throughput, more tuples are accumulated per window, increasing the size of the state, and, consequently, checkpoint sizes and recovery times. If recovery time is a priority, the above example is a good use case for adaptation; while in times of low system load passive replication may be sufficient to satisfy the user’s specified recovery time threshold, schemes providing faster recovery such as active standoff must be used in times of high system load.

In this paper, we present StreamMine3G, a fault-tolerant and elastic ESP system that employs several fault tolerance schemes such as passive and active replication as well as intermediate alternatives such as active and passive standby. In order to free the user from the burden of choosing the right scheme for the application at hand, StreamMine3G is equipped with a self-adaptive fault tolerance controller that transitions between the employed schemes during runtime based on evolution of the given workload and the user’s provided constraints (acceptable recovery time and recovery semantics, i.e., gap or precise recovery). Our evaluation shows that the overall resource footprint for fault tolerance can already be reduced by 50% with a recovery time threshold of 3 seconds using our adaptive scheme compared to a conservative use of active replication.

The rest of the paper is organized as follows. In Section II, we give some background information on ESP systems and fault tolerance schemes. Next, our system model for the self-adaptive controller and its implementation are detailed in Section III. In Section IV, we evaluate our approach and in Section V, we discuss related work. Section VI summarizes the contributions of the paper.

II. BACKGROUND

In the following section, we will first provide a brief overview about StreamMine3G, our elastic and fault-tolerant ESP system. We also overview fault tolerance schemes commonly used in ESP systems.

A. StreamMine3G – System Overview

StreamMine3G is a highly scalable ESP system targeting low latency data processing of streaming data. In order to analyze data, users can either opt for writing their own custom operators using the provided MapReduce-like interface and implementing a user-defined-function (UDF), or choose from an existing set of standard Complex Event Processing (CEP) operators such as filter, join, aggregation, and others.

In addition to the operators, users must specify the order events are supposed to traverse the previously selected operators using a topology. A topology in StreamMine3G is represented by an acyclic directed graph (DAG) where each vertex, i.e., an operator, can have multiple upstream and multiple downstream operators.

In order to achieve scalability, operators in StreamMine3G are partitioned. Each partition processes only a subset of events from the incoming data stream. For data partitioning, users can either implement their own custom partitioner similar to MapReduce, or use the provided hash-based or key-range based partitioner.

A typical StreamMine3G cluster consists of several nodes where each node runs a single StreamMine3G process hosting an arbitrary number of operator partitions, named slices. One of such nodes takes up the role of a manager which is responsible for placing operator partitions across the set of available nodes as well as moving partitions (through a migration mechanism) to other nodes for load balancing in situations of overload or underutilization. An overload can be detected by the manager node by analyzing the system utilization of each node, which is periodically reported through heartbeat messages exchanged between nodes.

In order to prevent the node hosting the manager component from being the single point of failure, the state of the component is stored in zookeeper upon each reconfiguration of the system. In the event of a crash of the node, another node can transparently take over the role of the manager by simply recovering with the previously persisted state.

Lastly, StreamMine3G supports the implementation of stateless and stateful operators. However, contrary to other ESP systems such as Apache S4 and Storm that have either no, or only limited, state support StreamMine3G offers an explicit...
state management interface to its users. The interface frees the users from the burden of having to implement their own locking mechanism to ensure consistent state modifications when processing events concurrently (to exploit multiple cores), and provides a full stack of mechanisms for state checkpoints, recovery, and operator migration. Using these mechanisms requires users only to implement appropriate methods for serialization and de-serialization of the state, which can comprise arbitrary data structures.

B. Fault Tolerance

With the steadily increasing amount of data originating from various data sources such as web, mobile applications, or even simple sensors (in the Internet of things context), highly scalable and distributed data processing technologies are needed where additional computational resources can be easily added on the fly. The majority of such systems are running in cloud environments, where additional computational resources can be easily provided by simply spawning new virtual machines. However, with each addition of a virtual or physical node, the probability of a fault increases. In the worst case, a single failure can make the whole application unusable at once, leading to high financial losses. As an example, a critical application that performs fraud detection or some financial trading service need to be prepared to cope with all typical faults that can occur in distributed systems and cloud environments, such as simple node or processes crashes, and network partitions.

Typically, ESP systems target low-latency data processing that needs to be ensured transparently, even when failures occur. Hence, several fault tolerance schemes for ESP systems have been proposed in the literature, offering different trade-offs regarding the recovery time and amount of resources needed to provide such timeliness.

For example, active replication provides the quickest possible recovery, however, at the cost of consuming twice the resources: two identical copies of the same operator are deployed and run on two different nodes, hence, redundant processing and communication is the key mechanism in order to mask a fault. Not only this scheme requires twice the processing nodes and the duplication of the events coming from upstream operators, but also, in order to produce identical results, atomic broadcast [11] and deterministic execution [12] is required so duplicates can be reliably filtered at downstream operators imposing additional overhead onto the ESP system. In summary, active replication requires twice the CPU, network and memory resources, however, provides a nearly instantaneous recovery.

An approach that consumes the least resources, at the cost of a long recovery time, is passive replication. In passive replication, only a single instance of an operator runs on the ESP system. Application robustness is provided through periodic checkpoints to save the state of (stateful) operators either to a local disk or a distributed, fault-tolerant file system. Since ESP systems work on continuous streams of events, an in-memory log at upstream operators is used to buffer events which have been produced since the last taken checkpoint. The events in the buffer can be replayed and hence reprocessed in case a failure occurs, ensuring a gapless recovery. Although the approach consumes only little additional resources for providing fault tolerance, such as disk space for storing checkpoints and memory for the buffered events, its recovery time comprises of the loading of the most recent checkpoint, de-serializing the stored state, and the reprocessing of the events from the upstream in-memory log. Depending on the size of the checkpoint interval, replaying events can take an considerable amount of time. Also, if state is big (e.g., in an application that holds an 24-hour data window), loading and de-serializing the state will further delay the recovery. In summary, passive replication can be considered as the opposite of active replication as it consumes considerably less resources, avoiding replicated communication and processing, at the cost of recovery time.

Besides active and passive replication, there exist several approaches which can be considered as a composition of active and passive replication. For example, in passive standby, an identical copy of the operator is deployed on the system, however, it does not perform any event processing (i.e., resides in standby mode). Instead, the replica in standby mode is used to hold a copy of the state rather than having it stored on a file system. For an overview of major approaches and their properties we refer the reader to Section III, where we describe how our system manages the available fault tolerance schemes.

Finally, for some fault tolerance schemes, different recovery guarantees may apply. In ESP systems, one can distinguish four different guarantees [10]:

1) Precise recovery, the strongest guarantee, completely masks failures. The state is fully recovered and neither events are lost nor duplicated. In order to provide such strong guarantee, deterministic execution is required in order to guarantee replayability. Replayability prevents event loss and enables safely ignoring duplicates at the following operator downstream.

2) Rollback recovery, slightly weaker, recovers the state of the operator and reprocesses events from upstream buffers, but due to non-determinism, duplications and inconsistencies may occur.

3) Gap recovery simply recovers the state. Events from upstream buffers are not reprocessed, causing the so-called gap.

4) Amnesia considers the case where not even the state is being recovered. The operator is restarted with a fresh state.

Depending on the guarantees an application may require in order to operate correctly, resource consumption and recovery time can be reduced. For example, in an application that analyzes frames from a live video stream, gap recovery may be sufficient since the lost frames (events that occurred in the past) may not be relevant for the current computation anymore. Hence, buffering frames upstream as well as replaying those frames can be omitted, saving a considerable amount of resources, as well as time during recovery.

III. ADAPTIVE FAULT TOLERANCE

In the following section, we will describe our approach on providing runtime adaptation for fault tolerance in ESP systems. We will first provide a detailed description about the fault tolerance schemes employed in StreamMine3G including its guarantees, resource consumption and impact on recovery.
time. We will then describe the model employed in our fault tolerance controller that provides runtime adaption based on the user provided constraints such as recovery time, recovery guarantees and cost model.

A. Fault Tolerance Mechanisms

In the remainder of this paper, we only consider the crash-stop failure model. Nevertheless, some Byzantine failures can be transformed into crash-stop failures using techniques such as software encoded processing [13]. Since StreamMine3G uses zookeeper [14] for storing its cluster configuration in a reliable fashion, network partitions are transparently handled through zookeeper’s heartbeat mechanism where unresponsive nodes are taken out of the cluster automatically.

The downtime of an ESP systems comprises of two components: (i) The time it takes to detect the failure, and (ii) the time it takes to execute compensation actions such as state recovery and event replay until normal operation resumes. In our system, we rely on zookeeper’s failure detection mechanism where the detection time can be bounded through the configuration of the session timeouts and the tick time (i.e., heartbeat interval). In the remainder of the paper, we use the term recovery time only for the second component, i.e., the time it takes to execute the recovery steps excluding the detection time.

An operator in StreamMine3G is equipped with several components that contribute to fault tolerance as shown in Figure 1.

First, an outgoing event queue (i.e., upstream buffer/in-memory log) is used to log events for a replay in case of a crash of a downstream operator. Of course, event replay comes only into play if the user opted for a precise recovery where event loss is unacceptable. In order to prevent memory exhaustion, the queue is purged whenever the state of the downstream operator has been included in a checkpoint successfully. StreamMine3G uses an acknowledgment protocol in combination with the sweeping checkpoint algorithm as described in [15].

Second, an incoming event queue installed at each operator instance is used for merging and ordering events coming from different upstream operators to ensure a consistent processing across replicas. Events are merged and ordered using application timestamps and a variant of the Bias algorithm as presented in [12].

In addition to the merging and ordering of events, the incoming queue is also used to detect duplicates. Duplicate detection is performed through a state timestamp vector associated with each queue which keeps track of the last seen event’s timestamp from each of the upstream operator partitions. Events with a timestamp smaller than the last registered one are automatically filtered and not passed to the operator. Duplicates do naturally occur whenever an operator receives events from a replicated upstream operator or during event replay within an ongoing recovery.

While the incoming and outgoing event queues ensure that neither events are lost nor processed twice, operators often accumulate state which must be protected through appropriate mechanisms as well. A well established approach is to make checkpoints, where the state that can comprise potentially any kind of data structure is first serialized in binary form and then either written to some stable storage or sent to a peer node for a take over in case of a system failure.

StreamMine3G employs in total six different fault tolerance approaches the controller can choose from as shown in Figure 2. In each of the subfigures in Figure 2, three operators comprising a small topology are shown: An non-replicated upstream (left) and downstream (right) operator and a replicated one in the center of each subfigure. In order to identify uniquely each instance of the replicated operator, we denote them as primary and secondary. While the purpose of the primary is to process events continuously, the secondary will solely serve as a backup which is going to be switched on or off depending on the chosen fault tolerance schema.

We will first start with the fault tolerance approach which guarantees the quickest recovery time, however, at the cost of consuming twice of the resources. As depicted in subfigure of Figure 2, in active replication replication, an operator partition is replicated where both replicas receive, process and send out events to downstream operators. In order to prevent processing events twice originating from the two replicas, the
downstream operator transparently filters duplicates using the previously mentioned incoming event queue with its associated state timestamp vector. The advantage of active replication is that it can practically recover within zero seconds as a crash of either of the two replicas will not affect the downstream operator in any way.

A first approach towards reducing the resource overhead of active replication is depicted in subfigure 2. In active standby, the secondary replica does not send its processing results to downstream operators. Although this saves network resources and the overhead for filtering duplicates, it increases the time prior the system can resume normal operation compared to active replication. During recovery, first the network links between the secondary and the downstream operators must be established, and second, events buffered in the in-memory log of the secondary must be sent and processed at the downstream operator which introduces additional latency.

Passive standby hot describes the state where the secondary still receives events, however, does not perform any processing saving computational resources in addition to network bandwidth. Since the secondary still receives events which are enqueued in the operator’s incoming event queue, state can be safely recreated by simply reprocessing enqueued events. However, this approach would lead soon to memory

Fig. 2: Fault tolerance schemes and their impact with regards to resource consumption and recovery times.

"Deployed“
exhaustion and an increased recovery time. Hence, the state from the primary is periodically included in checkpoints and these are stored at the operator’s replica. We call this process state synchronization as it updates the secondary’s state as depicted in subfigure 3. During state synchronization, the state timestamp vector is updated so outdated events can be pruned from the incoming event queue. In case the primary crashes, the following steps must be executed before resuming to normal operation: First, the links between the secondary and its downstream operators must be established so that the results the secondary produces will arrive at its interested parties. Second, events enqueued in the incoming queue of the secondary must be processed before accepting any new events coming from upstream operators. Note that since a priority queue is used here, events are always accepted, however, newer events are transparently shifted to the end of the queue until it is their turn.

Passive standby comes in two flavors: While in passive standby hot, the secondary still receives events from the upstream operators even though no events are processed, only enqueued, in passive standby cold, no events are sent to the secondary at all with the benefit of additional network bandwidth savings. However, during a recovery, network links from upstream as well as downstream operators must be established first and a replay of events buffered at the upstream operators performed as shown in subfigure 4.

Keeping a copy of the operator’s state in a peer node (i.e., secondary) rather than on a distributed file system comes with the advantage of a fast recovery as it saves the time (i) to load the checkpoint from disk and (ii) to de-serialize its data structures. However, it comes with the price of memory consumption which can be a problem if memory resources are scarce and if applications accumulate a considerable large amount of state over time. An approach to cope with this problem is to store the state on disk rather than in memory as depicted for the deployed and passive replication approach in subfigure 5 and 6. The two approaches differ in the way that in the deployed case, the internal data structures and binary code to execute the operator are already loaded and present in the system (but the state is uninitialized/virgin) while for passive replication the secondary is simply absent. Hence, the deployed state can be considered as a way of preloading part of an operator’s state as it updates the secondary’s state as depicted in Figure 1: Let’s assume that operator instance A′ is not enabled to send-out any event, neither B nor B′ will receive results produced by A, however, they will still receive results produced by operator instance A.

For state persistence, we use checkpoints, where the operator’s state is either stored on some (distributed and fault tolerant) filesystem or sent to its secondary operator instance to be kept in memory instead. As with the processing mode, taking snapshots (i.e., checkpoints) and choosing the destination can be controlled in a fine granular manner on a per operator instance basis.

In the following, we will describe the interplay of the previously described components in order to transition between the different fault tolerance states as depicted in Figure 3:

First, we define the term of a high availability unit (HA unit). A HA unit is an operator partition that consists of two replicas which we denote as primary and secondary. We use checkpoints2 rather than more than two replicas as it allows us to tolerate an arbitrary number of concurrent node failures while keeping our model simple.

HA units can reside in any fault tolerant state (e.g., active replication, active standby etc.) as depicted in Figure 3. A transition between the states requires several actions to be taken where certain components of the secondary operator instance are enabled or disabled. For example, in order to switch from active replication to active standby, the secondary must be instructed to stop emitting events denoted as [-send] in Figure 3, while going back to active replication requires re-enabling the secondary to emit events [+send]. Although Figure 3 shows all possible states that can be reached, we reduced the number of transitions to a minimum for better readability. Hence, in our system it is also possible to directly switch from active replication to passive standby. In fact, our fault tolerance controller employs a complete graph with all possible transitions and their required actions which allows a quick transition from one state to another.

C. Adaptive Fault Tolerance Controller

The objective of our fault tolerance controller is to choose the best fault tolerance state for each HA unit ensuring that the user defined constraints are met at any point of time. Hence, the controller evaluates constantly the recovery time for each scheme and transitions to another state if needed.

2A checkpoint comprises the operator state as well as its outgoing queue.
In order to prevent oscillation of the system, the controller has a cool down period where no transition to a new state is performed. Choosing the best scheme involves the following four steps the controller has to perform:

1) Compute the recovery time for each fault tolerance scheme.
2) Filter out the candidates that do not satisfy the user specified recovery time threshold.
3) Rank the remaining candidates according to the costs they would incur.
4) Choose the least expensive one and trigger a transition if the currently active state differs from the new one.

The controller starts first with active replication as it guarantees zero recovery time, i.e., we reside on the safe side first. In a user defined interval, by default every 500 ms, the current choice is reevaluated with the goal to switch to a less expensive state if possible.

### D. Recovery Time Computation

We will now describe, how we compute the recovery time for each of the schemes. Since each fault tolerance scheme requires several steps to be taken before the system can resume normal operation after a crash, the overall recovery time comprises of several components as depicted in Table 1. For example, if a HA unit has to recover from a failure using passive replication, first a backup (secondary) operator instance must be deployed in the system which takes time $t_{dep}$. In addition to the deployment of the operator, the most recent checkpoint (in case the operator is stateful) must be loaded and de-serialized which is reflected by $t_{readChkpt}$. Establishing connections to upstream and downstream operator partitions as defined through the given topology can be executed in parallel. Hence, the maximum of $t_{wrec}$ (time it takes to wire downstream operators) and $t_{wsend}$ (time it takes to wire downstream operators) is used. In case the user opted for a precise recovery, the replay time $t_{replay}$ is added to the overall recovery time as the last step contributing to a complete recovery.

For the time it takes to execute a certain recovery step, we use a mixture of historical collected values and an estimation based approach as depicted in Table 2. We first define a function $h(ts)$ to retrieve a historical measurement for some point in time specified by a timestamp $ts$. For example, the function $h_{chkptSize}(ts)$ returns the size of a checkpoint (i.e., the serialized form of an operator’s state) reported at time $ts$. Hence, we define the following functions which are instantiated per operator partition:

- $h_{chkptSize}(ts)$: Checkpoint (state) size at time $ts$.
- $h_{writeChkptTP}(ts)$: Write throughput (checkpoint) at $ts$.
- $h_{rec}(ts)$: Event throughput (receive) at time $ts$.
- $h_{procTP}(ts)$: Event throughput (processing) at $ts$.
- $h_{wsend}(ts)$: Wire (send) time at time $ts$.
- $h_{wrec}(ts)$: Wire (receive) time at time $ts$.
- $h_{dep}(ts)$: Deploy time at time $ts$.

Using the previous definitions, we can get an adequate estimate, e.g., for the time it takes to deploy an operator $t_{dep}$ by taking the maximum of all collected measurements from the past. Using the maximum reflects worst case behavior which we think is an appropriate approximation as the controller guarantees a recovery within the specified threshold. Since StreamMine3G is an elastic system where operator instances can be moved around depending on the systems capacity using operator migration, each migration employs the deployment of a new operator instance which increases accuracy of the collected measurements. Similar as with the deploy time, $t_{dep}$, we use the maximum recorded time it takes to establish connections to upstream and downstream operator partitions to set $t_{wrec}$ and $t_{wsend}$ respectively.

The overall time it takes to recover a stateful operator comprises on several components: First, the time it takes to read the binary form of the state from the stable storage, and second, the time it takes to de-serialize and reconstruct the state from binary form to its original. Since recovery happens far more seldom than taking checkpoints for a potential recovery, we use the historical values gathered from checkpointing the state. In order to estimate the recovery time for a stateful operator, we use the size of the most recent checkpoint divided by the lowest recorded write throughput (which includes the serialization overhead in addition to the disk throughput). Since the writing to disk is usually much lower than reading, we find this approximation appropriate. For an even more accurate estimation of the recovery time for the state, a lookup table can be used which contains recordings as a mapping of state size.

<table>
<thead>
<tr>
<th>Fault Tolerance Schema</th>
<th>Recovery Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Replication</td>
<td>$t_{recover} = 0$</td>
</tr>
<tr>
<td>Active Standby</td>
<td>$(✓)$ $t_{recover} = 0 + t_{deploy}$</td>
</tr>
<tr>
<td>Passive Standby Hot</td>
<td>$(✓)$ $(✓)$ $t_{recover} = t_{wsend} + (t_{replay})$</td>
</tr>
<tr>
<td>Passive Standby Cold</td>
<td>$(✓)$ $(✓)$ $(✓)$ $t_{recover} = \max(t_{wsend}, t_{wrec}) + (t_{replay})$</td>
</tr>
<tr>
<td>Deployed</td>
<td>$(✓)$ $(✓)$ $(✓)$ $(✓)$ $t_{recover} = t_{dep} + t_{readChkpt} + \max(t_{wsend}, t_{wrec}) + (t_{replay})$</td>
</tr>
</tbody>
</table>

Tab. 1: Recovery steps required to perform for each fault tolerance schema and the overall recovery time.

Note: Replay events is only needed for precise recovery.
to recovery time, however, since state size can highly vary over time which could result in a high number of entries, we favor a simple estimation based approach as described previously.

In addition to the recovery of state, events must be replayed in case the user opted for a precise rather than gap recovery. Since the amount of events that must be replayed is strongly influenced by the checkpoint interval, event replay can take a considerable amount of time. Hence, we use the number of received events by the primary since the last checkpoint and divide it by the current processing throughput in order to retrieve an estimate for the replay step. Note that it is also possible to adjust the checkpointing interval in order to reduce the replay and recovery time. However, there is a trade-off in overhead imposed through a more frequent checkpointing and the gain in a decrease of the recovery time as we will show in the evaluation in Section IV.

For a refinement of the parameter estimation, advanced techniques such as Kalman filters [16] or machine learning based approaches can be used as they may result in a more positive estimation of the recovery time compared to our approach. However, in order to keep the system model simple, we left the exploration of such techniques for future work.

### E. Cost Savings Adaption

In order to select the fault tolerance scheme which not only guarantees the user specified recovery threshold but also reduces costs by using as little resources as possible, users can optionally annotate resources with costs which ideally matches the cost model of the environment the application is running in. For example, an application running in the Amazon EC2 cloud environment will incur charges the more virtual machines used but not by the amount of CPU cycles or network bandwidth used unless traffic goes across regional availability zones. However, in a different setup such as a local cluster where several applications or virtual machines share the same host, a user might also be interested in reducing the network traffic that is imposed by fault tolerance rather than only the number of hosts used. Hence, user can provide a **cost weight vector** $\mathbf{v}_{\text{costs}}$, comprising the costs for CPU, memory, network and virtual machines. Using the cost weight vector, a ranking between applicable scheme, i.e. candidates, can be established. For example consider the following situation: Let’s assume a user chose five seconds as a recovery time threshold and the controller identified active replication, active standby and passive standby hot as valid options. While active replication and standby incur almost identical costs with regards to CPU consumption due to processing of events at the two replicas, i.e., primary and secondary, passive standby incurs additional network traffic costs due to state synchronisation. If the user weighted CPU costs higher than network costs, passive standby will be chosen as it consumes the least CPU resources at the cost of additional bandwidth usage while in the counter case active replication will be selected by the controller.

Costs are normalized based on the measurements received from the primary operator instance before applying the user provided cost weight vector and summarizing the components for a ranking. If two approaches have the same relative costs, the approach providing the lowest recovery time is chosen in favor for the user.

### IV. Evaluation

In this section we present the results from various experiments we performed in order to evaluate the benefits regarding resource and cost savings of our proposed solution.

#### A. Experiment setup

For our evaluation, we used two different applications and workloads. The first application performs a sentiment analysis using Twitter streams we collected over a period of a month. The application comprises of two operators where the first one performs a simple filtering based on certain hash-tags or keywords while the second one performs a sentiment analysis and an aggregation using a sliding window of ten seconds length. The workload is depicted in Figure 4.

The second application performs a short term energy consumption prediction for Smart Grids [17]. As with the first application, two operators are used where the first one performs a data conversion of the data tuples coming from smart plugs while the second one performs the short term load prediction using several sliding windows.

Both applications have in common that the query/topology consists of stateless and stateful operators, and the stateful operators use a time-based sliding window. Time-based sliding windows have the property of quickly accumulating state once the throughput rises, hence, the evolution of the state follows the pattern of the throughput as shown in Figure 4.

We implemented the applications in C++ to run on top of StreamMine3G’s native interface. However, application developers can also use Java as their language of choice by using the supplied Java interface wrapper. As for the environment, we performed our experiments on a 50-node cluster where each node is equipped with 2 Intel Xeon E5405 (quad core) CPUs and 8 GB of RAM. The nodes are inter-connected via Gigabit Ethernet (1000BaseT full duplex) and run an Ubuntu Linux 14.04.1 LTS operating system with kernel version 3.13.0.

#### B. Validation

In our first experiment, we performed a sanity check to validate our approach. We first analyzed the given Twitter

<table>
<thead>
<tr>
<th>Recovery Step</th>
<th>Recovery Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>read checkpoint</td>
<td>$t_{\text{readChkpt}} = \frac{h_{\text{chkptSize}}(t_{\text{lastCkpt}})}{\min{h_{\text{writeChkptTP}}(ts_1), \ldots, h_{\text{writeChkptTP}}(ts_n)}}$</td>
</tr>
<tr>
<td>replay events</td>
<td>$t_{\text{replay}} = \sum h_{\text{rec}}(ts_{\text{now}}), h_{\text{rec}}(ts_{\text{now}}-1), \ldots, h_{\text{rec}}(ts_{\text{now}}-\text{lastCkpt}) / h_{\text{procTP}}(ts_{\text{now}})$</td>
</tr>
<tr>
<td>wire to receive events</td>
<td>$t_{\text{wrec}} = \max{h_{\text{wrec}}(ts_1), \ldots, h_{\text{wrec}}(ts_n)}$</td>
</tr>
<tr>
<td>wire to send events</td>
<td>$t_{\text{wsend}} = \max{h_{\text{wsend}}(ts_1), \ldots, h_{\text{wsend}}(ts_n)}$</td>
</tr>
<tr>
<td>deploy operator</td>
<td>$t_{\text{dep}} = \max{h_{\text{dep}}(ts_1), \ldots, h_{\text{dep}}(ts_n)}$</td>
</tr>
</tbody>
</table>

Tab. 2: Recovery time for each recovery step.
workload with regards to the evolution of event throughput and state size. As mentioned previously, there is a correlation between throughput and state size for time based sliding windows where the state size follows the pattern of the throughput, as shown in Figure 4. Since the throughput is quite low during the first 100 seconds (around 100 kEvents/s), the amount of events being kept in the sliding window accumulates to roughly 20 MB, while with the sudden increase in throughput the state size quickly rises to 90 MB. In our experiment, we set the recovery time to 5.5 seconds since application criticality regards only user experience. However, we chose precise recovery rather than gap recovery for two reasons: first, precise recovery provides repeatability, a very useful feature for debugging distributed applications; second, because it is the safest approach it is typically the one selected. Since the state is quite small, we set the checkpoint and state synchronization interval to a rather small value of 3.5 seconds. We did not specify a cost weight vector, hence the default one is used where the approach that consumes the least CPU, network, memory and virtual machine resources is selected.

As shown in the beginning of the bottom plot in Figure 4, the system starts with active replication, as it is the safe choice. Once enough measurements have been collected, the controller quickly switches to the deployed scheme as the state and the throughput are quite low and, thus, recovery from disk and replay from upstream nodes can be easily accomplished within the user’s specified recovery time threshold. However, as spikes occur which let the state and upstream queues grow, the controller switches between passive replication and deployed schemes. The cool down time of five seconds prevents the system from oscillating due to sudden load spikes which are common in workloads originating from live data sources such as Twitter streams. In summary, the controller chose a combination of passive replication and deployed during the first half of the experiment, whereas the second half was dominated by passive hot standby.

C. Resource Overhead and Savings

For the next experiment, we were interested in the evolution in the resource overhead for fault tolerance with an increasing recovery time threshold. For this experiment and the following ones, we used the Twitter workload and 10 nodes of our infrastructure. We ran the experiment several times, each time with a different value for the recovery time threshold. Similar to the previous experiment, we used the default cost weight vector. The results for the experiment are depicted in Figure 5.

The plot in Figure 5 shows the overhead for CPU, memory, network (incoming and outgoing) and infrastructure utilization, where the infrastructure utilization reflects the number of virtual machines used. The resource utilization has been normalized to the the execution of active replication. Hence, when

![Resource Overhead](image)

Fig. 4: Throughput, state size and fault tolerance scheme evolution over time using Twitter workload with a recovery threshold set to 5.5 seconds.

![Trade-off](image)

Fig. 10: Trade-off between saved CPU and additional network traffic for different state synchronization and checkpoint intervals.

![Savings](image)

Fig. 9: Fraction of time spent in each fault tolerance scheme with varying recovery time thresholds for different recovery semantics (gap and precise recovery).
the user chooses zero seconds recovery time, we can witness an
10% overhead for CPU, memory, network and infrastructure utilization as the system runs solely in active replication mode. However, with increasing recovery threshold, resources can be saved. For example, using a recovery threshold of four seconds or more, the CPU overhead already drops to 50%, whereas the network utilization rises up to 300%. This is due to the fact that the system predominately uses hot passive standby where CPU cycles are saved due to the suspended secondary, however, at the cost of network bandwidth due to the periodic state synchronization mechanism. In fact, an overhead of 300% also indicates that the state synchronization mechanism consumes more network bandwidth than event dissemination alone. The overhead can be reduced by lowering the state synchronization frequency as we will show later.

Depending on the nature of the application, such savings in CPU resources can be considerable. For example, for an application that is CPU bound due to some very costly operations, using the adaptive scheme would trade CPU resources for network resources while still providing the same guarantees regarding recovery time and semantics as active replication.

D. Cost Model

We slightly modified the previous experiment by providing a cost weight vector that matches the Amazon EC2 cost model. In Amazon EC2 users are charged based on hours they use a virtual machine rather than actual CPU cycles or network bandwidth. Hence, there are no additional charges if they constantly fully utilize the CPU and the (internal) network. To match the profile, we set the weights for CPU, memory and network to zero so they will not be taken in consideration when ranking the different approaches. In other words, the approaches that use the least number of virtual machine hours are preferred.

The outcome of this modification is shown in Figure 6. If we compare the infrastructure utilization depicted in Figures 5 and 6, we can see that with a recovery time threshold of 5.5 seconds, the overhead (i.e., the number of required nodes) decreases faster than with the default cost weight vector, confirming the emphasis on infrastructure costs rather than on individual resources. On the other hand, the overhead for CPU, memory and network resources stays constant for recovery time thresholds of less than 5.5 seconds. This is due to the fact that the controller favors active replication as it provides a faster recovery compared to approaches that consume a similar amount of resources.

E. Relation between State Size and Resource Savings

As mentioned previously, the size of the state has a strong impact on the recovery time. A larger state requires significantly more amount of time to be loaded from disk and reconstructed in memory. Hence, in the following experiment, we extended our sentiment analysis application with an extra data field attached to each event. We then varied the event size to investigate its impact to resource savings. It is expected that applications with a relatively small state allow more potential resource savings when considering fault tolerance than applications with larger state.

Figure 7 depicts the lower bound, i.e., the lower limit for the recovery threshold a user must choose in order to achieve resource savings.

The results reveal that regardless of the state size a recovery threshold of more than four seconds allows already saving resources since the system can then transparently switch to passive standby mode, reducing the CPU overhead. However, one has to keep in mind that with increasing state size, state synchronization can be performed less frequently which increases the number of events in upstream logs.
In order to save memory resources, users have to provide a large recovery time threshold prior saving resources. This is due to the fact that for the default cost weight vector, CPU hungry schemes such as active replication and active standby are replaced with memory consuming states such as passive standby. For network and infrastructure utilization we can observe similar trends where with increasing state size: higher recovery time thresholds must be provided prior saving resources. This is directly shown in the graphs, but indicated through the plateau.

Since StreamMine3G supports fine grained state partitioning, the state per partition is usually small and would only rarely exceed more than 100 MB if the workload is well balanced. Even then, as with Hadoop stragglers, breaking a stage is often a possible approach to reduce the amount of state in a partition. As a consequence, users can benefit from resources savings even with short recovery time thresholds.

**F. Relation between the Cost Models and the Use of Fault Tolerance Schemes**

In the next experiment, we varied the recovery time threshold for different cost models to get an insight about the time the system spends in each of the schemes. As with the previous experiment, we can identify a clear correlation between the chosen cost model and the used fault tolerance scheme as shown in Figure 8. Using the default cost weight vector, we can see that the system stays in active replication for recovery time thresholds lower than three seconds. With increased thresholds, active standby is used, and, then, schemes such as passive standby. Since the system starts always with active replication, a fraction of the time is always associated to active replication regardless of the specified recovery time threshold.

For the Amazon EC2 cost model, we can see that the system primarily chooses two different states: Active replication and passive replication. Passive replication is preferred as it reduces more costs if less replicas are deployed on the system since nodes can be deallocated. On the other hand, active replication is used if passive replication cannot be used as it provides the quickest recovery time and still consumes considerable amounts of resources which are paid anyway by the Amazon EC2 customer.

**G. Relation between the Recovery Guarantees and the Use of Fault Tolerance Schemes**

Similar to the previous experiment, we varied the recovery time threshold, but now with different recovery guarantees. Requiring only gap recovery leads to lower recovery times in comparison to precise recovery as the event replay can be omitted. As shown in Figure 9, requiring precise recovery keeps the system more time in active standby compared to cases in which the user opted for gap recovery. Moreover, using gap recovery, the system can already remain in passive replication when a recovery threshold of 18 or more seconds was specified.

**H. CPU and Network Consumption Trade-off**

In the last of our experiments, we investigated the trade-off between saved CPU resources and network resources when varying the interval for state synchronization and checkpoints. A smaller interval leads to a more up-to-date state and shorter outgoing and incoming queues, which improves recovery time, however, at the cost of network bandwidth as every state synchronization imposes additional overhead on the network. Figure 10 depicts the overhead for CPU and network utilization with varying state synchronization interval. If state synchronization is performed continuously, without pauses, we can experience up to 600% peak overhead depending on the nature of the application and the state. For example, an application with average state size of 10 MB would exhibit an overhead of up to 200%. However, the savings regarding CPU are only marginal, hence, state synchronization should not occur more often than every two seconds.

**V. Related Work**

In this section we give a brief overview about fault tolerance techniques used in ESP systems.

Inspired by the schemes used in database systems, several approaches based on checkpoints and logging have been proposed, such as [10] and [15]. While Hwang et al. [10] proposes upstream backup where events are logged at upstream nodes for recovery, Gu et al. [15] combines logging with checkpoints by introducing the sweeping checkpoint algorithm. We used an adaption of the second approach, however, improved it so that the checkpoint intervals are adjustable to allow us to directly control the recovery time.

A more recent approach which does not require checkpoints is the work of Koldehofe et al. [18] where safe-points are used and track state modifications through a dependency graph. While this approach can save the overhead of doing and keeping checkpoints, which are not negligible, it is not suitable to our current operator model and API as it requires a notification mechanism to track state modification in relation to the incoming events.

A system similar to StreamMine3G is the one presented by Castro Fernandez et al. [19] where the ESP system comes with explicit state management support, which serves for elasticity and rollback recovery at the same time. Although both systems (SEEP and StreamMine3G) share many similarities, SEEP does only support a single fault tolerance scheme while our system covers a range of well established schemes.

An approach which guarantees a user specified recovery time similar to ours is presented by Balazinska et al. [20]. However, instead of switching between appropriate schemes and keeping consistency guarantees, temporarily inconsistency is introduced by forwarding only partial results due to the unavailability of upstream operator partitions.

Finally, several approaches have been proposed to combine more than a single fault tolerance scheme. However, they do so with different objectives: Authors in [21] and [9] use a combination of active replication and passive standby. However, while Martin et al. [9] runs in active replication during normal operation using spare and already paid cloud resources, Zhang et al. [21] use passive standby and switch only in failure cases to active replication. In our approach, we combine several approaches in a single system and switch between the schemes based on the user specified recovery time. Using the cost weight vector, our approach can also be used to use spare resources at no additional cost for fault tolerance.
as presented by Martin et al. [9].

An approach which is closest to our adaptation mechanism was presented by Upadhyaya et al. [22]. The authors propose an optimization algorithm which is tailored to specific operators rather than a whole query with the goal of guaranteeing a user specified recovery time. However, in contrast to our work, the approach is not adaptive and does not consider resource overheads.

VI. Conclusion

In this paper, we presented StreamMine3G, our elastic ESP system that, to our best knowledge, is the first ESP system to combine several fault tolerance schemes in a single system. In order to free the user from the burden of choosing the most appropriate scheme, the system is equipped with a fault-tolerance controller for which users are only required to specify a recovery time threshold, the recovery guarantees (precise or gap recovery), and, optionally, a cost weight vector used for resource and cost optimization. Using the provided input, the system will adapt during runtime, selecting the fault-tolerance scheme that ensures the user specified recovery time threshold while incurring the lowest resource consumption costs. For adaption, the system uses an estimation approach based on performance metrics collected during the execution of the application.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2012-2015) under grant agreement number 318809 (LEADS), from the CAPES, DAAD and GIZ through the NoPa program (TruEGrid project, 2011-2013) and by the German Excellence Initiative, Center for Advancing Electronics Dresden (cFAED), Resilience Path.

REFERENCES