Active replication at (almost) no cost

André Martin and Christof Fetzer  
Technische Universität Dresden  
Dresden, Germany  
Email: {firstname.lastname}@se.inf.tu-dresden.de

Andrey Brito  
Universidade Federal de Campina Grande  
Campina Grande, Brazil  
Email: andrey@lsd.ufcg.edu.br

Abstract—MapReduce has become a popular programming paradigm in the domain of batch processing systems. Its simplicity allows applications to be highly scalable and to be easily deployed on large clusters. More recently, the MapReduce approach has been also applied to Event Stream Processing (ESP) systems. This approach, which we call StreamMapReduce, enabled many novel applications that require both scalability and low latency.

Another recent trend is to move distributed applications to public clouds such as Amazon EC2 rather than running and maintaining private data centers. Most cloud providers charge their customers on an hourly basis rather than on CPU cycles consumed. However, many applications, especially those that process online data, need to limit their CPU utilization to conservative levels (often as low as 50%) to be able to accommodate natural and sudden load variations without causing unacceptable deterioration in responsiveness.

In this paper, we present a new fault tolerance approach based on active replication for StreamMapReduce systems. This approach is cost effective for cloud consumers as well as cloud providers. Cost effectiveness is achieved by fully utilizing the acquired computational resources without performance degradation and by reducing the need for additional nodes dedicated to fault tolerance.

Keywords—active replication, fault tolerance, energy efficiency, mapreduce

I. INTRODUCTION

In the last years, we have witnessed the continuous and increasing growth of live data sources. Examples of such data sources can be seen in several different application domains: fraud detection, where data such as credit card transactions or call-detail records are analyzed in real time to detect frauds; traffic monitoring, where data coming from traffic lights, cameras and road sensors provide information about traffic conditions; disaster detection, where a large network of environment sensors help to detect natural (e.g., earthquakes) or man-made disasters (e.g., terrorist attacks); and social networks, where location data produced by mobile phones is used to notify users of nearby opportunities or even the proximity of friends. In all these systems, the goal of the application is to derive useful information or to detect situations of interest from a massive amount of raw data.

Despite the processing of massive amounts of data being a classical problem in computer science, the applications above pose new requirements. First, data items are useful for a very short period. Therefore, they need to be collected, processed and the results delivered within relatively short periods of time. Second, the raw data is too massive to be stored. In fact, storing the raw data may not only be unfeasible, but also unnecessary. If data can be processed in real time, only the relevant information, at a much lower rate, needs to be stored. Third, the amount of computational resources needed for the processing may be very large and vary with time.

Processing the data is not trivial, though. Historically, applications that dealt with massive amounts of data have been implemented either through databases or batch processing systems. Databases offer a good programming interface that allows users to quickly develop applications that analyze the data. Batch systems are less intuitive, but scale much more. Some batch systems, such as MapReduce [1], can easily scale to hundreds, or even thousands of nodes. However, batch systems are inefficient in processing data that changes continuously as they often require the complete dataset to be reprocessed. In addition, batch systems focus on throughput, minimizing the role of responsiveness. Databases are also inefficient for such applications. Databases require the storage and indexing of the data before its processing, but because historic data has a low value, these unnecessary steps make it inefficient. In a summary, neither batch nor database systems are adequate platforms for the applications exemplified above. As a consequence, a new class of systems have been proposed to enable such applications that require scalable low-latency processing of online data: event stream processing (ESP) systems [2], [3].

Cloud computing, another recent and promising technology, offers application developers the option of allocating computational resources on demand. This approach is attractive because it relieves developers from having to build and maintain a large number of computational resources that stay idle most of the time. Consequently, it saves costs both in equipment and electricity. Nevertheless, on-demand allocation of resources is still far from ideal. For applications that require low latency processing, cloud-level load balancing is not enough. Cloud-level load balancing requires new nodes to be contracted and deployed. Deployment requires instantiating the processing infrastructure and, most likely, transferring state from the nodes that will be offloaded to
the new nodes. Contracting nodes not only takes time, but is also subject to node availability.

Because of the issues above, when using cloud computing application developers have to consider conservative load levels in the nodes to be able to accommodate eventual short term changes in the load (i.e., spikes). Suitable load levels can be determined for each application, but rules-of-thumb are normally the guidance for application developers [4]. Maximum utilization levels for CPU (but also network and disk, if these can become a bottleneck) can be set to values as low as 50% even for applications with no explicit responsiveness requirements. At the same time, as we detail later, even with only 30% load, nodes consume as much as 90% of their full energy. Therefore, from the provider’s point of view, the energy costs are almost the same and the customer is charged by the hour, even when not using the resources fully.

Finally, once the basic application is designed, the application developer still needs to consider fault tolerance. Because of the responsiveness requirements, approaches based on active replication [5] are best suited to ESP applications. Nevertheless, active replication imposes a huge resource overhead as it requires all nodes to be replicated.

In this paper, we propose an approach that enables active replication to be used without the huge cost in additional resources. The key aspects of our solution are the following: (i) we explore the MapReduce-like state partitioning to distribute the replication burden among different nodes; (ii) we use the processing cycles that are most of the time idle with redundant processing for replication, but when load spikes occur, we prioritize the primary processing in detriment of redundant processing; (iii) we use state transfers and regular load balancing to avoid that replicas become inconsistent. Our fault tolerance approach enables active replication to be used in a cloud environment without the need to contract additional nodes (or, at least, requiring less additional nodes) and making the best usage of the energy consumed.

The rest of this paper is divided as following: in Section II we provide some background information on ESP, on fault tolerance in general, and on our system; in Section III, we discuss our active replication approach; Section IV presents an evaluation of our approach; and, Section VI concludes the paper.

II. BACKGROUND AND SYSTEM MODEL

In this section we provide some important background information. This includes an overview of StreamMine, our MapReduce-like event processing platform and a brief discussion on ESP systems and fault tolerance techniques.

A. System architecture

StreamMine applications consist of a sequence of stages. A stage is essentially an operator that is partitioned across multiple nodes to achieve scalability. Events are flowing from a source stage (stage\(_{0}\)) to a sink stage (stage\(_{n}\)) traversing an arbitrary number of stages in-between as depicted in Figure 1. In the following, we will denote stage\(_{i-1}\) and stage\(_{i+1}\) as the stages upstream and downstream of stage\(_{i}\), respectively.

Each partition of an operator processes only a subset of events. These subsets are defined by ranges in the hash values of a key attribute. An event in StreamMine consists of a key-value pair where the key is an attribute of the event that is used for data partitioning as well as routing events to the appropriate partition of the operator of the next stage. The value portion of an event can be either a single value, an application-specific string of data, or a set of secondary attribute names and attribute values. Events have also system attributes, such as an unique id (composed from a node id and an event id that is unique in that node) and a timestamp.

![StreamMine architecture](image)

Figure 1. StreamMine architecture

StreamMine is a event processing platform that supports stateful operators, hence, an operator can create, access, and modify a state. Keeping state is necessary for implementing operators that execute some form of aggregation. For example, frequency estimation (top-k), pattern detection, pattern matching, and moving averages are very common stateful operations in ESP applications. Because the state is partitioned according to a key, events that map to different pieces of the state can be processed in parallel. However, the processing of events that accesses the same portion of the state is serialized. This is necessary to ensure consistency. Furthermore, as we will discuss in the next section, fault-tolerance requires the serialization order to be deterministic.

As in the MapReduce paradigm, the user implements a function, either a mapper or a reducer, for each stage. On the one hand, reducers in the regular MapReduce paradigm are implemented as stateful operators in StreamMapReduce. However, note that the use of state allow us to break the strict phasing of MapReduce\(^1\), making it suitable for ESP systems. On the other hand, the role of mappers in the traditional MapReduce does not require the use of state, which makes its implementation simpler. For more details on the StreamMapReduce programming paradigm we refer the reader to [6].

\(^1\)Strict phasing forces that a stage is required to finish processing the whole dataset before the next stage can start processing.
B. Event processing systems and fault tolerance

ESP applications are typically represented as a graph of operators. The data from the live stream flows through this operator graph. Common operations are filtering (e.g., selection), aggregation (e.g., moving averages, pattern matching), transformation (e.g., data conversion) or even data mining operations modified to work with event streams (e.g., classification, forecast, frequency estimation) [2]. Some of these operations consider only the last piece of data, i.e., the last event. One such operation is a filter that discards or forwards an event based on some predicate computed over this event. However, many operators require some state to be kept. For example, while computing an aggregation like a moving average, detecting a pattern or applying some data mining algorithm, the operator needs to keep information across the processing of different events. If failures may occur, this state needs to be kept safe.

In addition, ESP systems aim at the low-latency processing of continuous streams of data. Therefore, fault tolerance techniques need to consider latency: both the latency overhead introduced in the failure-free case and the time to recover after a failure need to be considered.

If failures are rare and applications are not time critical, approaches based on rollback-recovery [7] are very common. Basically, they consist in keeping periodic checkpoints of the state of the operator, a log with the data items processed by the operator, or a combination of both. Combining logging and checkpointing is the most common approach as the checkpoints can be used to limit the size of the log (after a checkpoint, the log can be pruned) and the logs can be used to recover the system to a state that occurred between two checkpoints.

Nevertheless, if failures become more frequent, such as in a large cluster or cloud, as we address here, and also if responsiveness is of critical importance, rollback-recovery may not be appropriate. The reason is that even if a recent checkpoint is available, reading a checkpoint from disk back to the memory may take too long. For example, with fast disks connected via Gigabit networks it would take at least 40 seconds to read 4 GB of data back to memory. Then, after the checkpoint is restored, the events in the log must be replayed and reprocessed. As a consequence, if failures are common or recovery must be fast, techniques based on active replication [5] need to be applied.

Active replication requires that an ESP operator is replicated so that a replica masks the failure of the original operator. With active replication, both the original operator and its replicas need to process the same sets of inputs and in the same order to guarantee that both will reach the same state. Therefore, as long as the two replicas are kept consistent, the outputs from both will be equivalent and a failure of the original operator can be transparently masked by the replica.

In our system, we exploit three important facts we have observed in ESP systems. First, in order to achieve scalability, recent systems (e.g., S4 [8], StreamMine [6], among others) apply a MapReduce-like approach to partition the state of the operator. This partitioning enables us to easily split the state of an operator running in a node. Thus, when replicating this operator, we do not need another node with enough resources to hold the complete state of the first node, but instead, we can split the state and replicate different pieces at different locations.

Second, in order to handle natural load fluctuations, node usage levels need to be kept well under 100% (and sometimes even under 50%). Nevertheless, note that even at a load level as low as 30%, the power consumption of such a node may be above 90% of its maximum power (as we detail in Section IV). We then exploit the ease of partitioning discussed above to use the spare cycles (and power already being consumed) for the replica processing (events being redundantly processed as part of the replication of a portion of another node’s state). Finally, the extra cycles will not be always available. While experimenting with different ESP applications as part of large cooperation projects (such as EU FP7 STREAM and SRT-15), we have observed that memory size is generally not a problem for operators running in modern nodes. Our third observation is that if spare cycles in the machine are not available, it is not a burden to buffer the events that are input to the replica processing. Even with a high input rate, a processing node will be able to buffer the data until the load spike has passed or until regular load balancing mechanisms enter in effect.

III. Active replication in StreamMine

In this section, we present the design and implementation of the active replication approach in StreamMine. We first cover how we ensure deterministic execution of events using the deterministic processor component, then we introduce the concept of slices and discuss the techniques used to achieve active replication at (almost) no additional cost.

A. Deterministic processor

In order to use active replication, events must be deterministically processed in all replicas. Therefore, StreamMine uses a deterministic processor component to order both incoming and outgoing events according to a total ordering. The two main subcomponents of the deterministic processor are the sequencer and the finalizer. The sequencer is responsible for ordering events prior to invoking the user-provided function that implements the operator logic. In contrast, the finalizer orders outgoing events prior to emitting them to the next stage. In other words, the sequencer guarantees that the history of changes in the operator state is deterministic, whereas the finalizer guarantees that the sequence of events outputted by an operator is deterministic.
The sequencer as well as the finalizer are implemented as heap-based priority queues with a pointer that advances only if the event’s total order matches the currently expected total order number. To order the events deterministically there are three basic approaches. First, one can use physical timestamps. Physical timestamps assume that nodes are well synchronized (e.g., through a service such as NTP). Second, logical timestamps can be used. Logical timestamps assume that nodes produce events at about the same speed. Third, a round-robin scheme can be used to go through the channels deterministically when receiving messages.

In practice, all three approaches require adaptation. For timestamps, either physical or logical, an approach such as the bias algorithm can be directly used [9]. This approach periodically computes adjustments to be applied to the timestamps. For example, if a node is too slow or too fast, the periodic adjustments will avoid that events from this node block events from other nodes or are blocked from events from other nodes. The adjustments also compensate for consistent differences in communication delays for different channels. Lastly, in the round-robin scheme, a similar adjustment would change the order events are retrieved from the channels, so that in each cycle, more events are picked from faster channels. Our current implementation considers only logical timestamps to order the events.

B. Slices

A slice is a portion of the operator state and is the base unity for balancing load and for replication. Because the domain of possible keys can be arbitrarily large, a slice can contain a single key or a set of keys (a range in the hash value of the keys). In our current implementation, the number of keys in a slice is decided before the execution starts and stays constant through an execution. Note however that the mapping between values of the key attribute and the key itself is pseudo-random (the key used internally is a hash of the key attribute specified by the user). Therefore, if an integer id, for example, is used as a key attribute, two neighbor ids will map to keys that are probably in slices very far away from each other. This pseudo-random mapping (as provided by a good hash function) is the first step in providing an balanced load among the nodes.

When using active replication, we are pairing each slice with a peer that is responsible for the same key set. To distinguish each of the two slices, we will denote them in the following as primary and secondary slices. Secondary slices serve as backup for primary slices and, in case of contention for resources, events of primary slices are processed with a higher priority as we will describe in more detail in the next sections.

The introduction of secondary slices comes with a price: an increase in network traffic as secondary slices will naturally emit the same events to downstream nodes as their primary counterparts. The detection of those redundant events is easy as events, originating from slices that process the same set of keys, received the same unique timestamps.

Alternatively, event emission at secondary slices can be disabled so that no redundant events are flowing through the system. However, the previous approach allows the handling of stragglers, e.g., slow primary slices.

C. Slice distribution

Each StreamMine node hosts an arbitrary number of slices. The number of slices in a node may change depending on the processing power of a node. Because slices use different pieces of the operator state, they are independent and can be migrated from one node to the other. Nevertheless, migrating a slice may take time if its associated partition of the state is large. In this paper, we focus on short-term load changes. We then assume that, initially, load is balanced among the nodes and a regular load balancer is available to handle gradual, medium and long-term load changes.

Figure 2 illustrates an example of slice distribution in a small cluster setup. Because the pseudo-random distribution of the keys, slices have approximately the same resource requirement (e.g., 16%). In addition, in this example, we assume that nodes were setup to be have a 50% load in order to cope with sudden load variations that cannot be handled by the long-term load balancers. Therefore, each node has 3 primary slices (i.e., around 48% loaded on average).

Once a node has, on average, free cycles, StreamMine allocates secondary slices to that nodes. In the example in Figure 2, the nodes have around 50% idle capacity and can handle 3 secondary slices each. There are two basic rules for distributing slices among nodes. The first rule is that two slices associated with the same portion of the state (i.e., the same range of key hashes) cannot be hosted by the same node. This rule will avoid that the primary and secondary copies fail at the same time and that, if a higher...
replication level is used (more than one secondary slice for each primary), that multiple secondary copies of a slice also fail with a single node failure.

The second rule is that two secondary slices should avoid being located at the same node if their corresponding primary slices are also in the same node. This rule aims at minimizing the number of slices that a node may have to take over after some other node fails. In other words, the goal is to distribute the load of a recovery among the highest number of nodes possible.

In the example above, node 2 is hosting the primary slices \( s, t \) and \( u \) whereas nodes 1, 3 and 4 are hosting the corresponding secondary slices \( s', t' \) and \( u' \) fulfilling the first rule. The second rule is also satisfied as no two of the secondary slices \( s', t' \) and \( u' \) share the same physical node. With this distribution scheme, a single node failure will result only in a marginal load increase per surviving node as all secondary slices for a set of primary slices are spread across multiple nodes.

The previously described slice distribution scheme, i.e., the partitioning of state and the placement of its replicas is similar to interleaved partitioning as used in storage and database systems [10]. Interleaved partition splits the state to be replicated among several disks so that after a failure of the primary copy the backup can be read in parallel from multiple disks. Therefore, the backup partitioned copies of the state of a node cannot be kept in the same node.

The distribution of tasks among nodes is a NP-hard problem, but approximations for our case are simple. For example, an application is unlikely to change the amount of needed nodes in more than two orders of magnitude. Therefore, a few tens of slices per node are enough to provide fine-grained load balancing. In addition, due to the hash randomization, each slice will have approximately the same resource demand. We can then use a first fit algorithm that tries to allocate all the primaries and secondary slices in a certain number of nodes while still satisfying our basic rules. If this is not possible, then we allocate an extra node and continue the process. This algorithm can also be easily modified to consider additional rules. For example, in a private cloud, the application developer may want to avoid that primary and secondary slices are deployed in nodes that share the same rack.

Lastly, once slices have been placed across cluster nodes, each slice sends a subscription to all upstream nodes to express its interest for events of its set of keys as depicted in Figure 2. In our example, primary slice \( s \) and secondary \( s' \) are sending the same subscription message to all upstream nodes since both of them are interested in the same set of events. StreamMine is intended to run on dynamic environments where node joins and leaves occur rather often (e.g., as nodes fail and new nodes are added). Therefore, slices can subscribe and unsubscribe at any point of time. Unsubscriptions are triggered implicitly, i.e., if a downstream node becomes unavailable, subscriptions for all slices that the specific node was hosting are automatically removed from the subscription tables of upstream nodes.

D. Active replication versus passive replication/state synchronization

As discussed previously, when the system is in steady state there will be a reasonable amount of computing cycles that are idle. This idleness is necessary in order to allow the system to handle sudden load variations. Our system maximizes the efficiency of a computational resource by allocating secondary slices to use the idle cycles. As a consequence, the system will work on a high level load consistently. Note, the total maximum CPU utilization for a node can also be bounded (through a processing threshold) to lower levels such as 98% as the system may expose an unstable behaviour when constantly running at 100%.

Despite this constantly high load level, the system still must be able to handle sudden load increases in a way that does not excessively compromises its responsiveness. Therefore, when the amount of events that are target at a primary slice increases, the amount of cycles dedicated to secondary slices is immediately decreased. This prioritization is done by processing events targeted at a secondary slice only if there are no events directed at a primary slice that can be processed.

If load spikes are short lasting, the prioritization of primary load (i.e., load caused by the primary slices) will enable the node to keep up with the primary events. As primary load decreases and returns to steady state levels, secondary events, which had been buffered, can be processed again. If no other load spikes occur, the secondary events will be processed and the secondary slices will also be brought to an up-to-date state.

However, if load spikes are frequent or last for too long, secondary slices will get increasingly farther behind. To overcome this problem, StreamMine performs passive replication, i.e., a state synchronization between a secondary slice and its corresponding primary slice. State synchronization actions are triggered when a primary slice gets too far ahead of its secondary slice. The distance between the primary and secondary slices is monitored through periodic heartbeat messages. Heartbeat messages are exchanged at a fixed interval and update peer nodes about the current status of primary slices and their secondary counterparts. If a primary node stops sending its heartbeat messages, for example, due to a failure, its secondary counterpart will be automatically upgraded to primary and a new secondary slice will be created using the state synchronization procedure. Similarly, if a secondary stops sending its heartbeat, a new secondary node will be created.

Note that due to the way the deterministic processor works, failure detection does not need to be perfect. If a primary node is incorrectly suspected (e.g., a temporary
network slowdown causes heartbeats to be late), the only consequence is that a state synchronization process may have already started and needs to be aborted.

Finally, if a secondary slice does not fail, but simply got too far behind because its node is prioritizing the primary slices, then a state synchronization process will copy the state from the primary to the secondary. Once the state of the secondary slice has been updated, events in the sequencer queues can be pruned. Thus, if the node hosting the secondary gets repeatedly overloaded, the active replication deteriorates to passive replication until the long-term load balancers enter in action. Note that it is also possible that the primary gets behind in relation to a secondary, in this case, the secondary will be promoted to primary and its state transferred to the, now demoted, primary.

IV. Evaluation

In this section, we present the results of various experiments we performed to evaluate the scalability and effectiveness of our proposed solution and its implementation.

A. Experiment setup

In total, we have implemented four applications in our system: (1) SLA (service level agreement) conformance monitoring, where the application monitors if a pattern of events that indicate a service being completed occurs within a predefined time interval; (2) telephone fraud detection, in this case the system builds a network of interests for each mobile phone user and triggers an alarm when his behavior changes abruptly; (3) credit card fraud detection, where transactions are considered suspicious if they are executed with the supposed physical presence of the credit card owner, but take place too far away from each other for the time between them; and (4) a canonical word count application as used in the related work. All four applications map well to the MapReduce programming paradigm. The applications have been implemented in the context of the EU FP7 STREAM project and consider real workloads. Details on how these applications perform (with respect to throughput, responsiveness, and scalability) in the StreamMapReduce programming model can be found in [6].

Here, we want to evaluate how the system behaves with sudden load variations that are found in forthcoming applications that monitor highly dynamic systems such as social networks or target at disaster detection. Therefore, we have implemented a synthetic workload generator which matches to previously described real world applications in terms of amount of CPU cycles used per operation, size of state and event distribution. The usage of the load generator allows us to study and visualize interesting cases, such as the behaviour of the system while gradually increasing the frequency of load spikes, which do not occur naturally in real world traces. Therefore, the experiments in this section use this load generator.

The experiments were performed on a 50-node cluster where each node is equipped with 2 Intel Xeon E5405 (quad core) CPUs and 8 GB of RAM. All nodes are connected via Gigabit Ethernet (1000BaseT full duplex). StreamMine applications are written in C++, both the framework and the user-provided map and reduce functions. The nodes run a Debian Linux 5.0 operating system with kernel 2.6.32.

B. Power consumption

In the first experiment, we measure the power consumption against CPU utilization as depicted in Figure 3. For this experiment, we increased the load on the node while measuring the power consumption at the power distribution unit (PDU). As shown in the graph of Figure 3, the power consumption does not increase linearly with the utilization: at around 30% of CPU utilization, the power consumption has already almost reached its maximum of 167 Watt.

This experiment confirms that a large amount of power is wasted when application developers need to keep usage levels considerably low in order to cope with sudden load variation. Therefore, our approach considerably increases the efficiency of the system as it produces more useful work for the same power consumption.

C. Scalability and overhead

In the next set of experiments, we investigated the performance of our system in terms of (1) horizontal scalability (adding nodes), (2) vertical scalability (adding cores to a node), and (3) overhead introduced by our approach. For the experiments, we used the workload generator in an application that consists of three stages: a source, a worker, and a sink stage. Each stage uses 16 nodes. The source stage constantly generates events and send those events downstream to the worker stage, where operators perform some computations to simulate real work. The sink stage receives the results from the previous stage, but does not perform any computation (it can be seen as a gateway that exposes results back to the external world). The throughput measurements detailed below were executed at the worker stage.

To compare the overhead introduced by deterministic execution and active replication, experiments were executed using three different versions: no order, deterministic execution and active replication. Each version executes the same operator code, hence performs the exact same computations. In no-order, events are processed in arbitrary order, as soon as they arrive at the nodes. In deterministic execution events are first enqueued and ordered using the sequencer of the deterministic processor component and output events ordered by the finalizer. This ordering introduces noticeable overhead as events from a node may have to wait for events from another node before they can be processed. Finally, in active replication, events are processed in order as in deterministic execution with an additional overhead as subscription tables
must be inspected for each single event sent downstream. The overhead introduced by the deterministic and the active replication versions in comparison to the no-order version can be seen in Figures 4, 5 and 8. Note that the overhead of active replication is negligible in comparison to the deterministic execution.

In addition to the introduced overhead, Figure 4 depicts the horizontal scalability of the system: with increasing number of nodes, the aggregated throughput increases. Figure 5 shows the per node throughput, which only slightly decreases with the addition of new nodes. This decrease in throughput is expected and is due to the increase in contention on downstream channels as well as the increase in the size of subscription tables that need to be inspected for each event.

Vertical scalability, i.e., scaling with the number of threads, is shown in Figure 8. As expected, this experiment shows that StreamMine can fully utilize multicore processors.

D. Load peaks and active replication

In the following set of experiments, we investigated the system behavior in the event of load peaks. To simulate spikes, we use the load generator to emit events at different rates for predefined periods of time. Figure 6 depicts the aggregated throughput for a single node and the status of the input queues of a single secondary slice on that node over time. In this experiment, the load generator nodes introduced load spikes every 20 seconds for 2 seconds.

During a load peak, no events for secondary slices on that node are being processed, hence queues grow quickly. Once the load decreases, secondary slices resume event processing, thus sequencer queues of secondary slices shrink. Note that aggregate throughput of the node remains high until the shrinking process has been fully completed. During the spike, the aggregated throughput was higher due to the increase in load on the primary slices, after the spike, the throughput is higher due to the accumulated load on the secondary slices.

Next, load spikes were induced more frequently, every 15 seconds and for 4 seconds. This is shown in Figure 7. More frequent and longer spikes result in shorter recovery periods. Consequently, queues of secondary slices grow with each new load peak. Once the total order difference of a primary and its corresponding secondary slice exceeds a certain threshold, a state synchronization action is triggered. In Figure 7, state synchronization occurs at around 43 seconds. State synchronization allows the deterministic processor to prune events queues.

In our current implementation, state synchronization is performed on a rolling checkpoint basis, i.e., only one (fixed size) portion of state is transferred at a time with the advantage of reduced contention on locks while the state is being (de-)serialized. This approach leads to a constant overhead regardless of the size of the state. Note, a new state synchronization is only triggered if the previous transfer has been fully completed or aborted which avoids the interleaving of synchronization actions.

Finally, Figure 9 depicts the average state synchronization interval with respect to the length of load peaks: state synchronization actions are performed more often with increasing length of peaks. In the worst case, if the load increases and remains high, StreamMine will operate in passive-replication mode until a regular load balancer re-stabilizes the system (e.g., by contracting more nodes from the cloud) and reduces the average loads to the original, safe levels.

E. Node failures

In the last set of experiments, we investigated the impact of failures on the throughput. The setup for this experiment is similar to the slice distribution example shown in Figure 2. However, we used three nodes instead of four and only two primary and two secondary slices run on each individual node. Each group with four graphs in Figure 10 and 11 represents one physical node. Inside a group, the two graphs on the left column depict the throughput over time of primary slices, while the two on the right depict secondary slices.

Figure 10 illustrates the evolution of throughput under moderate load. At around 26 seconds, the second node becomes permanently unavailable due to a hardware or software fault, hence, slices c, d, b' and e' are not functional anymore. The unavailability of the crashed node is detected by peer nodes, thus, backup slices c' and d' (running on node 1 and 3, respectively) are transparently promoted to primary slices. Nevertheless, note that the upgrade does not incur any additional load on peer nodes as sufficient CPU cycles were already available for the processing of both primary and secondary slices prior to the crash.

In the final experiment, shown in Figure 11, the system is running constantly under high load. Therefore, secondary slices a', b', c', d', e' and f' have not processed events until the failure of the second node after 26 seconds. At this point, as secondary slices c' and d' become primary, the promotion forces additional load to the node. This additional load lowers the throughput for the primary slices a, b and e, f in nodes 1 and 3, respectively, as seen in the figure.

V. RELATED WORK

MapReduce [1] has become popular in batch processing systems due to its simplicity and scalability. Although not all problems map well to this paradigm, once a solution has been implemented as map and reduce functions, deploying the solution in a large number of nodes is simple with open-source MapReduce implementations such as Hadoop [11]. ESP applications commonly require operations that keep state between events. Nevertheless, parallelizing an stateful
operator is not trivial. In fact, state in a ESP operator is a major challenge to achieve scalability. As applications that need processing of huge amounts of online data became more common, the MapReduce approach has been adapted to ESP systems. These systems (e.g., S4 [8] and the StreamMapReduce approach implemented in StreamMine [6]) use key attributes in the events to define which portion of the operator state they will access.

The idea of interleaved partitioning of operator states in an online processing system was first used in Flux [12]. Flux also considers moving pieces of state to different nodes in order to handle long term load imbalances and handles short term load imbalances by buffering events. However, Flux considers database-like operations and focus on operators with group-by clauses. In contrast, we assume the state partitioning scheme as a feature of the programming model, forcing the user to consider the MapReduce programming model and considerably improving scalability and fault-tolerance potential. We also consider a highly-parallel cloud execution environment, which enables a different load-managing goal: when some partitions are overloaded, Flux tries to allow non-overloaded partitions to proceed, we, in contrast, focus on temporarily freeing more resources for the overloaded partitions.

Another challenge that needs to be solved for ESP applications that require stateful operators is fault tolerance. If nodes rely on state, this state needs to keep safe for when failures occur. In a previous work, we have detailed an approach for passive replication for ESP systems that apply the MapReduce approach [13]. However, as detailed in Section II, traditional passive replication has a low resource overhead, but requires a long recovery phase after a failure as both the checkpoint needs to be restored and events from the log need to be replayed and reprocessed.

Because active replication provides a virtually instantaneous recovery (i.e., a replica can actively produce redundant outputs that are used when a primary fails), it has been commonly used in ESP systems (e.g., in Flux [14] and Borealis [15]). Nevertheless, state is, again, a problem. Because operator replicas must be consistent, operations need to be deterministic. Some works, such as the one by Brito et al. [16] for ESP systems and by Jimnez-Peris et al. [17] for general distributed systems, address a scenario where multithreading is used but there is no static partitioning of the state. In this case, scalability of an operator is limited to a single node. The MapReduce approach requires the operator
state to be partitionable, but enables a single operator to scale to a large number of nodes.

An approach related, but somehow opposed, to ours is proposed by Zhang et al. [18]. In their work passive replication is used when the system is in steady state. The system then switches to active replication if signs of a failure are detected (i.e., they speculatively activate the replica). Their goal is to reduce recovery time when using passive replication while still avoiding the usage of active replication at all times. In contrast, our goal is to exploit cycles that are already available to implement active replication and switch to passive replication, when the system is under a temporary load peak that consumes the normally available extra cycles.

VI. CONCLUSION

In this paper, we presented a new fault tolerance approach based on active replication that scales with the number of nodes and cores within a node. Our approach improves energy efficiency of the system as it makes better use of power that is already consumed. In addition, it implements active replication without incurring (or at least, considerably reducing the amount of) extra costs for cloud consumers.

The key idea in our approach is to exploit the natural fine-grained partitioning of an operator state in ESP applications that use a MapReduce-like paradigm to do fine-grained replication of the state of a node among a set of nodes. This replication then uses spare cycles that would be normally left unused. In case of load peaks, the system automatically stops the processing of replicas and, if necessary, may transparently switch from active replication to passive replication until input load comes back to normal values or cloud-level load balancers enter in action. Finally, because of the fine-grained distribution of replicated slices, recovery after a failure introduces only a minor load increase in surviving nodes.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement number 257843 (SRT-15 project).

REFERENCES


