Low-overhead fault tolerance for high-throughput data processing systems

André Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, Christof Fetzer
Technische Universität Dresden
Dresden, Germany
Email: firstname.lastname@inf.tu-dresden.de

Andrey Brito¹
Universidade Federal de Campina Grande
Campina Grande, Brazil
Email: andrey@lsd.ufcg.edu.br

Abstract—The MapReduce programming paradigm proved to be a useful approach for building highly scalable data processing systems. One important reason for its success is simplicity, including the fault tolerance mechanisms. However, this simplicity comes at a price: efficiency. MapReduce’s fault tolerance scheme stores too much intermediate information on disk. This inefficiency negatively affects job completion time. Furthermore, this inefficiency in particular forbids the application of MapReduce in near real-time scenarios where jobs need to produce results quickly. In this paper, we discuss an alternative fault tolerance scheme that is inspired by virtual synchrony. The key feature of our approach is a low-overhead deterministic execution. Deterministic execution reduces the amount of persistently stored information. In addition, because persisting intermediate results are no longer required for fault tolerance, we use more efficient communication techniques that considerably improve job completion time and throughput. Our contribution is twofold: (i) we enable the use of MapReduce for jobs ranging from seconds to a few tens of seconds, satisfying these deadlines even in the case of failures; (ii) we considerably reduce the fault tolerance overhead and as such the overhead of MapReduce in general. Our modifications are transparent to the application.

I. INTRODUCTION

The MapReduce approach [1] proved to be very useful in practice. It allows users to quickly implement simple, but scalable solutions to certain classes of problems. MapReduce uses a simple programming model where users express their programs as map and reduce functions. Map functions transform one input into one or more outputs. Reduce functions aggregate multiple inputs into a single output. A MapReduce application is therefore a sequence of map and reduce processing stages. Once the program is expressed in such a way, an underlying MapReduce implementation such as Hadoop [2] can automatically handle the distributed processing of the program.

One limitation of MapReduce is that it assumes jobs have spans of many minutes. Even relatively simple computations that include many nodes can easily need several minutes to process. There are, however, applications that need to process batches of data within seconds or few tens of seconds. For example, consider an application that monitors service level agreements (SLA, e.g., through quality of service metrics like response time). Application sensors emit data continuously and, periodically (say, every 10 seconds), a job is started to recompute the metrics. If metrics start to deteriorate, the monitoring system can trigger compensating actions (e.g., request nodes from the cloud and deploy more servers). After the metrics are computed, the original data can be discarded and the metrics, with reduced volume, can be stored for long-term planning. A short response time will enable the system to adapt itself before deterioration is perceived by users. Note also that the volume of data can be large, requiring tens (or even hundreds) of nodes in order to process the data within the desired time frames, making trivial centralized solutions unfeasible.

MapReduce is of course not the only approach to solve problems similar to the SLA monitoring example above. Alternatively, one could, for example, use a database or a stream processing system. However, MapReduce has excellent scalability and hence, one can typically build very cost-effective systems to solve problems that require to process large amount of data.

Processing data on large clusters, one has to expect that some nodes crash even during short computations. For applications that continuously process data, node failures must be tolerated. Enabling a distributed system to tolerate failures is not a trivial problem. In a distributed system, fault tolerance is typically provided in two ways: (i) replicating components or (ii) by using logs and checkpoints in order to save any state that needs to be recovered after a failure. Active replication [3] is not suitable for the large-cluster scenarios we address here. Active replication provides fast recovery and only a marginal increase in latency at runtime, but is extremely costly in the amount of extra resources needed. Instead of active replication, MapReduce uses a logging-based approach: Outputs of mappers and reducers need to be logged to disk.

In this paper, we use a variant of MapReduce, which we call StreamMapReduce that uses stateful reducers. This permits us to start reducing tuples before the mappers have done finishing processing. In MapReduce all mappers (of a stage) have to be done processing, before the reducers (of the next stage) can start processing. Using stateful reducers, a

¹Most of this work has been carried out while at the Technische Universität Dresden.
 reducer can start processing as soon as tuples from the mappers are available because a reducer can store intermediate results. In particular, we can perform incremental updates that propagate immediately through the system. In this way, we can achieve excellent response times.

Using stateful reducers introduces the problem of how to deal with crashes of reducers and mappers: The original logging approach of MapReduce is not sufficient anymore because logs could now become unbounded. The standard approach is to use rollback-recovery [4] which uses a combination of logging and checkpointing. Rollback-recovery normally imposes runtime overheads and in comparison to checkpointing and active replication, a longer recovery time.

We introduce a fault tolerance approach that uses a combination of uncoordinated checkpointing and in-memory logging to minimize the runtime overhead and the recovery time. The in-memory log is only used to speed up the recovery and is not needed to ensure recovery. Instead, we use deterministic execution to ensure that we can recreate the same outputs. If due to a crash, an in-memory log is lost, we can recreate the log using the deterministic execution property.

Ensuring deterministic execution of a multithreaded applications is possible but, as we show in our evaluation, introduces some synchronization overhead. Our system reduces the overhead of deterministic execution as inspired by virtual synchrony [5]: Instead of making sure that all events are processed in a deterministic order, events are divided deterministically in a sequence of sets: Each such set we call an epoch. The events in an epoch are processed in the order they arrive. Hence, there is little synchronization overhead within an epoch. As long as the execution of the events within an epoch are commutative, we achieve a "deterministic" state at the end of each epoch. In MapReduce most reducers are already commutative and hence, the requirement of commutative execution of events in an epoch has not been a practical problem so far. Our approach is only inspired by virtual synchrony: Virtual synchrony reduces the overhead of replicated state machines in a distributed system. In this work, we use one idea of virtual synchrony to reduce the overhead of deterministic replayability of multi-threaded MapReduce jobs.

The rest of the paper is organized as follows. In Section II, we give some background information on MapReduce and fault-tolerance, including application examples. Next, our fault-tolerance approach is detailed in Section III. In Section IV, we evaluate our approach and in Section V, we discuss the relation with similar works. Section VI concludes the paper.

II. BACKGROUND AND SYSTEM MODEL

In this section, we discuss the MapReduce programming model and present a simple extension that allows exploiting MapReduce scalability in other application areas. After that we provide some background information on fault tolerance and give some examples of applications that can be used with our system.

A. The (Stream)MapReduce programming model

The popular MapReduce programming model revolves around two basic abstractions. First, a map function. The map function takes as input a key/value pair and produces as output a (possibly empty) list of key/value pairs. The output key/value pairs of all map functions are sorted according to their keys and are the inputs to the second abstraction: The reduce function. One instance of the reduce function receives all key/value pairs for a certain key and typically produces a simple value aggregating all the inputs for this key. The reduce function’s output (the key common to all inputs and the aggregated value) can be the input for another MapReduce stage. Many data processing tasks can be expressed using this simple abstraction. Parallelization is easy: The entire input data is split into disjoint sets of key/value pairs. Every mapper can work independently on its input split. Synchronization happens only under the hood, between the execution of map and reduce functions, when the output key/value pairs are sorted according to their keys.

The traditional example for MapReduce is shown in Listing 1, counting the number of occurrences of a word in a large corpus of text. For each word in document doc the mapper will emit an event (word, 1). Eventually, a reducer will receive a list of all such events for a word and will add them up, emitting a final counter value for this word.

```java
map(Key k, Document doc) {
    for each word in doc:
        emit(word, 1);
}
reduce(Key word, List counts) {
    int total = 0, count;
    for each count in counts:
        total = total + count;
    emit(word, total);
}
```

Listing 1. The word frequency count algorithm for MapReduce.

The last component of MapReduce we will discuss is used to make jobs more efficient. A combiner locally aggregates the outputs of a mapper, sending them periodically to the reducers. The reducers work then as above, doing another aggregation that also considers data with the same key and that come from other map nodes. For instance, instead of having messages like (word, 1) in the word count example a combiner would aggregate many such messages, emitting a (word, X) events, where X ≥ 1, after a certain volume of data had been processed by the mapper. In many cases the combine function is identical to the reduce function.

Fault-tolerance in MapReduce is achieved by writing the result of the map and reduce functions to stable storage, e.g. local disk or distributed file system. If a map function fails, it is restarted with the same input split as the failed invocation.
For reducer failures, the key/value pairs assigned to the failed reducer need to be reprocessed. Fault-tolerance in MapReduce is helped by the fact that both components, map and reduce function, are stateless. Map and reduce process self-contained sets of inputs instead of incremental and dependent pieces of information. If such a data dependency exists in a map/reduce application, multiple iterations of map/reduce cycles must be used. The data for multiple map/reduce cycles is then kept in stable storage, adding to MapReduce’s overhead.

Using state is the main difference to our extended programming model, StreamMapReduce (SMR): StreamMapReduce supports stateful reducers. That is, reducers receive messages as inputs and retain state across different inputs. In StreamMapReduce the user may specify windows, either in time or data volume, for the data that reducers should use to execute an aggregation. For example, if a 10-second window is specified, reducers will automatically aggregate and emit results every 10 seconds.

To reduce the amount of data being sent from mappers to reducers, users can optionally provide a combine function to aggregate results prior emitting them to reducers like in original MapReduce. The combine function is applied to user specified windows of data in the same fashion as reducers.

The introduction of state breaks the strictly staged model of MapReduce, which required mappers to have processed the whole input for a key before a reducer could be started. This change enables using the familiar MapReduce paradigm for applications it was previously not suited, e.g., processing of continuous data streams such as financial stock data and network/equipment monitoring.

With the exception of the optional parameters (i.e., window parameters and reducer state), the basic programming model has not changed. The user can ignore the extensions and use StreamMapReduce with the same interface of MapReduce and benefit only from the responsiveness, throughput and recovery speed. In this paper, we will focus on the basic MapReduce model as only minor details differ for using the fault-tolerance mechanisms in the two modes.

B. Application Requirements

In order to benefit from our proposed optimization, the potential target application must meet the following requirements:

(1) Monotonically increasing timestamps. Events/key-value pairs must be equipped with (per-source) monotonical increasing timestamps in order to be assignable to the specific processing epochs. Timestamps can be defined by either real time or synthetic ones like file offsets in case the incoming key-value pairs are considered as events coming from an infinite file or network stream.

(2) Commutativity of Combiner and Reducers. Commutativity of the combine/reduce function is a mandarory requirement as it allows out-of-order processing of events and still provides low-overhead precise recovery. Furthermore, as the combiner will not see all values associated with a certain key, combiners can only be used if commutative combine functions are provided. This implies that currently existing MapReduce application using reduce functions as combine functions implicitly fulfill this requirement. As an example: For the word count application, it does not matter in which order the word counts for individual documents are accumulated to form a word count comprising all documents.

Although we will only demonstrate this approach for the canonical word count example, the requirements that enable us to perform our optimizations, are present in real world MapReduce applications. For example, we have developed (1) SLA (service level agreement) conformance monitoring, and (2) telephone fraud detection application and both satisfy these requirements. In the first application, reduce operations need to explicitly consider application-level timestamps in the business logic (for pattern matching). Then, because messages from different distributed sensors are not guaranteed to arrive according real-time order, algorithms need to handle potential disorder. Consequently, message delivery ordering is not relevant. For the second application, frauds are detected based on a user’s community of interests. These communities are not affected by short-term changes in the order of the input events and, therefore, reducers are also commutative.

III. LOW-LATENCY FAULT-TOLERANCE FOR STREAMMAPREDUCE

In this section, we discuss the details of our approach for data processing systems based on StreamMapReduce. We start by explaining complete determinism, virtual-synchrony-inspired deterministic execution, and no determinism. Next, we discuss how communication from the mapper/combiner to the reducer can be implemented once saving the messages to disk is no longer necessary. Finally, we discuss the process of recovering from a failure.

A. Deterministic execution inspired by Virtual Synchrony

With complete determinism, the exact processing order of the messages is enforced for each message. Therefore, the state of the application can be recovered for each point in time. For recovery, a checkpoint and the last processed event must be known. Given this knowledge, the application can be restored precisely to any point of the execution by (a) restoring the most recent checkpoint, and (b) replaying and reprocessing the events until the last recorded event.

Unfortunately, complete determinism limits performance as processing is stalled until the next in-order event arrives. This not only introduces a delay but also a synchronization overhead: we have to check if an event can be processed and if the event can be processed in parallel with other events. The state is partitioned according to the reduce keys. For
each key, we need to make sure that the events updating this state are processed in the given deterministic order. In this way, events can be processed in parallel as long as the events have different (reduce) keys. However, the execution of an event $e_1$ has to be delayed until it is known that there exists no other event $e_0$ that has the same key as $e_1$ and has a lower timestamp than $e_1$. This introduces some overhead because we potentially need to enqueue events and need to determine when enqueued events are ready to be processed.

Now compare the completely deterministic execution ordering with a no-order execution: events can be processed without delay as soon as they arrive. However, absence of ordering has other problems. In order to restore the application state precisely, we have to track exactly which events were already processed, which could be done by logging all inputs and outputs to stable storage. This is possible but incurs typically a high runtime overhead for the “normal” failure-free case. The benefit of course is that events need not wait for events from other sources before they can be processed.

Note that the no-order approach above assumes the events can be processed in arbitrary order since it does not enforce any particular ordering. In other words, the ordering of the events is not important to the aggregation. Note also that commutative aggregation operations are common in MapReduce jobs (see discussion in Section II-B). This commutativity can be exploited to achieve better parallelism.

Our deterministic execution inspired in virtual synchrony exploits the fact that the reduce functions are typically commutative. This permits the out-of-order execution, yet it restricts the out-of-order execution to predefined intervals, limiting the overhead of synchronizing the execution of events. Within the intervals, which we call epochs (originally called views), events can be processed in any order. This approach decreases the time spent for synchronization and this benefits throughput. Only at the end of an epoch we introduce additional overhead: we need to ensure that all events falling into this epoch have been processed before execution can progress to events that belong to the next epoch.

Table I illustrates the differences of deterministic, no-order, and virtual-synchronous execution. Assume that two sources (1 and 2) send events to a stateful component of our system, i.e., a combiner or a reducer. The first column depicts the arrival time of these events. The second and third column indicate which events arrived and from which of two possible sources (i.e., mappers) they come from. The subscript in the events indicate their timestamp, which is assigned by the source. The goal is to calculate the imposed latency overhead from each approach, i.e., virtual synchronous, (completely) deterministic, and no-order execution. These overheads are given by the three rightmost columns.

Complete determinism limits performance as processing needs to be stalled until the next in-order event arrives. In our example, $e_3$ is received from source 2 but cannot be processed until it is certain, that no event with a timestamp less than 3 will arrive from source 1. This information only becomes available when $e_5$ finally arrives from source 1. By then, $e_3$ and $e_4$ already waited 4 and 3 turns, respectively. In our example execution, the events had to wait an accumulated 8 turns when ordered deterministically, as indicated in the last row of the table. This wait time is not important only when the system is (at least slightly) overloaded and the next events from the sources are already enqueued. In this case, the code ensuring that events are processed in the correct order introduces the main overhead (in comparison to no-order execution).

For virtual synchrony, only one synchronization point is used in Table I (represented by the horizontal line between row 3 and 4). At this synchronization point, all events $e_n$ with $n \leq 3$ must have been processed. For this reason $e_4$, which belongs to the second epoch, has to wait three turns for $e_5$ to arrive. Only when $e_5$ arrives, the first epoch, containing $e_1$, $e_2$, and $e_3$, can be considered as completed. The exact order in which events were processed inside the epoch is irrelevant. This is why $e_3$ does not have to wait for $e_5$.

In the virtual synchronous execution, we checkpoint the state only at the end of an epoch and enforce the ordering only during epoch changes. Note that in the no-ordering approach, we would need to log at each event and in the (completely) deterministic approach we would need to enforce the order of each event. The virtual synchronous execution permits us to recover the computational state to known points in the case of failures, while keeping the overhead for fault-tolerance manageable. Note that the duration of an epoch can be expressed either in terms of timestamps or data volume.

Finally, if the synchronization of timestamps (or relative speeds of nodes) is a problem even when needed only in epoch changes, an approach like the bias algorithms [6] can be used. If messages from two input channels differ by more than a certain threshold, messages can be assigned new timestamps to make them appear synchronized again.
The decision to assign new timestamps is logged to stable storage. The approach also works for epochs defined in terms of processed data.

B. Communication by state transfer

As detailed in Section II, the MapReduce approach uses combiners to optimize communication. These combiners do a pre-reduce, combining the emissions of multiple map executions in a single message. The combiner is therefore a stateful component. It keeps a state that is partitioned by keys.

Periodically, the combiner will send the piece of the state associated with each key. In the conventional MapReduce, each piece of state is transparently sent to one of the downstream nodes. The destination node is selected based on a hash of the key, which produces a static, but randomized, load distribution.

Note that in the time between two sendings of the combiner, the state is kept locally. The most meaningful way to store this state, which is already partitioned by key, is a hashmap. Also note that the mapping between the key and the bucket in the hashmap is already randomized. In the case of StreamMapReduce, instead of extracting each bucket in the hashmap (which can be as small as a word and an integer, like in word count) and sending it downstream, we divide the hashmap itself and send complete subsets of buckets to a single downstream reducer. As mentioned above, because of the randomization in the hashmap, sending a continuous block of keys does not break the load balancing mechanism, but at the same time, significantly improves the communication efficiency.

Another important optimization in the combiner is to keep two versions of the hashmap. When portions of a hashmap are being transmitted or checkpointed they need to be protected against modifications. In order to avoid the blocking of the mappers and/or the unbounded growth of internal queues while a hashmap is being protected, a temporary hashmap is created to continue with the aggregation. When the checkpoint and sending of the state is finished, the hashmaps are merged in order to preserve a consistent state in the combiner.

C. Recovery

Recovery needs to consider three different component failures, i.e. mapper, combiner, and reducer. We assume crash failures. The system can handle transient and permanent node failures. In case of a permanent node failure, the afflicted component is started on an identical node (we currently assume a homogeneous cluster). For stateful components, the state must be transferred to or reconstructed on the new node.

The mapper is a stateless component. In case it crashes, no state needs to be recovered. The process is just started afresh and fed with input as follows: Input data, regardless of coming from network or file streams is partitioned by hashing information in the event itself, for example, if a document is the input event to the mapper, the partitioning will be done based on the hash of the document id and will be always sent to the same node id. Therefore, when a node fails permanently, the new node assuming the role of the failed node assumes the same id. Thus, as the event sources try to reconnect to the failed node, they will reach the new node and then, if a replay is requested, feed it with the same inputs. Note, if the stream is a network stream, inbound logging (e.g., a proxy that logs events) is necessary if sources are not able to guarantee re-playability of events.

Mappers work independently of other components and have only little coordination with other components for optimization purposes such as skipping unnecessary data (see below).

As discussed in Section II, the combiner is a local optimization that aggregates results of multiple map invocations in the same way the reducer does. In our case, this similarity between combiner and reducer is preserved and the combiner also becomes stateful. This aggregated state must then be preserved in case of a crash failure. The state of the combiner is checkpointed at the end of each epoch, for example, after processing an aggregated volume of 500 MB in events. Alternative definitions for epoch intervals are possible, for example, based on timestamps of the events to process.

For transient failures, the combiner recovers by reading the most recent checkpoint from its local disk. For permanent node failures, the recovery protocol is as follows: the combiner sends a message to all reducers, asking the reducers for their most recent update from the failed combiner. The combiner reconstructs its state using the information sent to it by the reducers. Because each reducer will have only part of the state, the recovery overhead will be distributed among all reducers.

In addition to the application specific state, the state that is checkpointed or that is transferred back from the reducers to the combiner contains some metadata. The most important metadata is an indication of how much data was processed to reach such a state. Based on this information, after the recovery (either from a transient or permanent failure), the system can skip data that is already included in the checkpoint. The amount of data that can be skipped is also forwarded upstream to the mapper nodes. The system then skips the specified amount of data, avoiding unnecessary recomputations. When reconstruction finishes, the combiner can continue processing mapper provided key-value pairs.

Reducer failures, permanent or transient, can be handled in two ways. If reducers and mappers are not expected to fail simultaneously (e.g., in a small cluster with mappers and reducers deployed in non-overlapping nodes), reducers can reconstruct their states simply by waiting for one message from each upstream combiner. Otherwise, if simultaneous failures cannot be excluded, the same checkpoint approach
employed for the combiner is used for the reducer.

D. Virtual Synchrony meets MapReduce

For our word count example, the complete input data is partitioned into disjoint input sets. The partitioning is based on a hash computed over the document’s ID. Partitioning is done to allow for parallel computation of independent subtasks, i.e. counting the words in each document, before summarizing the individual counts to a global word frequency.

The mapper receives a single document as its input, and produces counts for each word in that document. Documents are read from local disk (although they could also be directly retrieved from the network). A physical node executes multiple mappers in parallel, as illustrated in Figure 1. Before sending the result of the map phase to the reducer, multiple map results are combined. This is done to avoid sending many messages with little information to the reducers. Aggregation of individual map results is done according to partitions. Word counts for documents from different partitions can be merged/aggregated in parallel, also illustrated in Figure 1 by multiple combiners per host node. Our experiments showed that not partitioning the combined state leads to performance degradation as combiners may become the bottleneck. Partitioning is transparent because the user cannot make assumptions about which mappers run on which nodes. At the end of the epoch, the partitioned combiner state is merge to form a final result for the epoch. The result of this final combine operation is sent downstream.

![Figure 1. Data movements in the word count example.](image)

Combining the results of multiple mappers is where coordination is required to guarantee that states can be reconstructed if needed. Referring back to the example in Section III-A, events are word counts for a single document. Epochs comprise a certain subset of document data. Events are timestamped with the file offset the originating data had on disk. At the end of an epoch, the combiner must have processed all documents with a file offset less than or equal to the next epoch change value. The state is then checkpointed and sent to the reducer. The intervals are defined by the duration of the epoch.

![Figure 1. Data movements in the word count example.](image)

All our measurements were performed on a 50 node cluster. The machines are connected via Gigabit Ethernet (1000BaseT full duplex). Each machine has 2 Intel Xeon E5405 CPUs (quad core) and 8 GB of RAM. Hard disks are attached via SATA-2. The operating system is Debian Linux 5.0 with kernel 2.6.26.

For the StreamMapReduce experiments, one mapper and one reducer are running co-located on each node. Co-location follows the approach of the original MapReduce where map and reduce tasks are deployed on the same physical node to benefit from reduced network bandwidth usage and more efficient use of computational resources.

A 25 GB pre-partitioned Wikipedia dump stored on the local disk of each node served as input for the map stage in the experiments. According to [7], the rate for data-local-mappers in real world Hadoop clusters is around 70%-95%. To keep the comparison with Hadoop fair, i.e., to avoid network streaming of file chunks due to HadoopDFS usage, we modified Hadoop’s word count implementation to read file chunks solely from local disks as in StreamMapReduce. Note that StreamMapReduce is not limited to input data provided through file streams. Arbitrary streams can serve as input sources for StreamMapReduce such as network or file streams. However, for simulations purposes and a fair comparison with Hadoop, we are using local file streams in our measurements as mentioned above.

A. Failure-free experiments

Our work was motivated by the fact that MapReduce, although being very popular, is not the best tool for parallel data processing in all circumstances. Especially if the job processing time is in the range of a few tens of seconds, job setup and task deployment across the cluster dominate the overall job completion time, making it unfeasible to use MapReduce for jobs that need to finish within small time frames.

Figure 2 shows our measurements for the canonical word count example. The open source Hadoop implementation of MapReduce always fares worse for the selected problem sizes. Job completion time for small problem sizes is dominated by job setup time which is in the 30-second range.

Several modifications to the original MapReduce approach have been proposed recently. Hadoop Online (HOP) [8] is probably the best known modification focusing on reduced overall job completion time, pipelining and incremental updates. HOP benefits from the fact that reducers can start processing as soon as a first mapper output is available resulting in a decrease of overall job completion time compared to traditional Hadoop. However, HOP’s job completion time is also dominated by setup time for small problem sizes. StreamMapReduce does not have this limitation and shows the fastest job completion time
for all problem sizes. All three approaches run with their fault-tolerance mechanisms enabled.

Next, we evaluate how the three proposed execution strategies, i.e., no order, deterministic, and virtual synchrony, fare against each other. Figure 3 shows the three strategies and their scalability for a single node. No-order execution events is always faster, independent of the number of threads, which was expected. Note that no logging was performed for no-order execution. The measurements thus represent the best case scenario for no-order execution. Ideally, logging will have no effect on throughput, but only add to latency and stable storage demand. In addition, performance of deterministic execution deteriorates with more threads as more synchronization becomes necessary. The benefit of virtual synchrony becomes visible when using two or more threads. The throughput for virtual synchrony is close (within 2%, or 1 MB/s difference) to no ordering and up to 20% (4 to 5 MB/s) better than deterministic execution.

Scaling across nodes is investigated in the experiments shown in Figure 4 and Figure 5. These figures depict aggregated throughput and per-node throughput, respectively. With an increasing number of nodes, the benefit of virtual synchrony over determinism becomes more pronounced, reaching 15%. This is expected as the potential for out-of-order execution increases with the number of nodes.

The fault-tolerance overhead impact on the latency is directly affected by the length of the epochs. In the case of virtual synchrony, all output is deferred until the end of the interval. For example, for a epoch/checkpoint interval of 500 MB and a processing speed of around 60 MB/s (see Figure 3), the checkpointing interval and hence, maximum latency should be around 8 seconds. This can be confirmed visually by looking at Figure 8 where latency is plotted over time for the same checkpoint interval size. Note that around 15 seconds into the experiment, a failure was inserted (this case is discussed below).

### B. Failure experiments

We now evaluate the behavior of the system when failures occur. As mentioned above, Figure 8 depicts also the impact of a mapper failure on latency. Around 15 seconds into the experiment the (single, for this experiment) mapper fails. Subsequently, the latency increases to around 18 seconds, 10 seconds more than the previous maximum. This increase can be explained by the time it takes to restart the failed process, load the most recent checkpoint from disk, and (re-)execute events for the current interval. Also note that no updates were received between seconds 23 and 33. This happens because of the previous failure that caused processing to be stopped for 10 seconds.

Next we observed the impact of crash failures on the throughput of the whole system. In this setup, we consider that failures are transient. This means that components are restarted in the same machines they were before the failures. This is equivalent to consider software crashes or system updates (updating the operators or infrastructure software). This will also be the case if two nodes, the primary and a passive backup, share a disk via eSATA and the passive assumes when the primary fails. In this last case too, no
Figure 5. Per node throughput in MB/s when varying the number of compute nodes. (Higher is better.) Virtual synchronous execution performs significantly better than deterministic execution. No order is only slightly better than virtual synchrony.

Figure 6. Per node throughput and latency depending on the epoch size. The “computed” latency is half the checkpointing interval. For 100MB epoch size, the system already approaches the maximum throughput.

Figure 7. Completion time as a function of the number of transient crash failures during program execution. (Lower is better.) Hadoop needs more time to recover after a crash than StreamMapReduce.

Figure 8. Impact of mapper/combiner crash on latency for a epoch duration of about 8 seconds.

state needs to be transferred. Later, we will consider failures where recovery must take place in a remote node that has no copy of the state of the failed node.

For each subplot in Figure 9, we crashed successively more components. Components were made to crash around 18 seconds into the experiment. With failures in 10 machines (second plot from the left), there is a drop to 2400 MB/s from the usual 2750 MB/s. As more and more machines crash, the drop in throughput becomes more pronounced and takes longer to recover.

Figure 10 depicts the impact on throughput and recovery time for permanent crash failures. Permanent crash failures result in an increased recovery time as seen when comparing graphs of Figure 10 and Figure 9.

We show the impact of transient failures in the completion time of jobs in Figure 7. The word count application is executed using Hadoop and StreamMapReduce. The number of failures is varied and the impact on total job completion time observed. Unfortunately, we cannot provide numbers for HOP, as a (confirmed) bug in the software prevented crashed nodes to recover. Job completion time is almost unaffected in the case of StreamMapReduce. Hadoop’s completion time increases visibly with an increasing number of node crashes.

V. RELATED WORK

Related work can broadly be categorized into three topics: (i) fault-tolerance, (ii) determinism and virtual synchrony, and (iii) processing model. We will cover each in turn.

A. Fault-tolerance

Many distributed systems achieve fault tolerance by using logs and checkpoints, which is also known as the rollback-recovery approach [9]. In this case, nodes of the system rely on a stable storage that can be used after a crash to restore the state. In this way, the system does not lose the work done so far. For stateful systems processing continuous data this is specially important as past inputs are usually not kept (for example, because of their huge volume).

Checkpoint-based protocols perform checkpoints periodically. Upon failure the state of the system is rolled back
Figure 9. Impact of transient crash failures on throughput. Successively more components were crashed for each experiment.

Figure 10. Impact of permanent crash failures on throughput. Successively more components were crashed for each experiment.

to some globally consistent checkpoint. Checkpoints can be uncoordinated or coordinated. Uncoordinated checkpointing is easy to implement because nodes can checkpoint their states at any point in time. Once a rollback is initiated, however, the nodes have to coordinate to find a consistent checkpoint across the system. In the search of consistency, the system can suffer from what is called the *domino effect*, which can force the system to roll back to the initial state. To avoid the domino effect, logging is used in combination with checkpoint. In this case the gap between checkpoints can be filled by replaying some messages.

Coordinated checkpointing forms a consistent global state by orchestrating the checkpointing among all nodes in the system. The price of neither suffering from a domino effect nor performing any superfluous checkpoint is the introduction of orchestration overhead and complexity. In our case, determinism provides implicit coordination. Therefore, we need neither to log messages nor to explicitly force nodes to checkpoint.

Finally, if multiple failures can happen simultaneously, nodes need to replicate their saved state. If only checkpoints are used, as in our case, this replication takes place only when there is a new checkpoint. However, if logging is also used, the log also needs to be replicated. Replicating the log imposes a continuous burden to the system as all messages received by a node need to be forwarded to other nodes.

Authors in [10] propose a different approach for stream processing systems based on the speculative execution of events to overlap and minimize logging and checkpointing overhead. However, contrary to our approach, speculative execution requires to rollback state using software transactional memory in case of crash failures.

### B. Determinism and virtual synchrony

By assuming completely deterministic executions [11], [12], the coordination problem can be avoided as mentioned above. Deterministic execution requires coordinating when each message is processed. However, complete determinism is not always necessary and, as we have showed, it imposes an additional overhead to the system. We explore the fact that reducers are normally commutative in MapReduce and then force only the minimum determinism necessary to guarantee periodically consistent checkpoints.

Virtual synchrony was originally used to provide consistency in active replication [5]. Here we extend this idea to rollback recovery by limiting event ordering to moments where it is necessary to provide determinism.

Regarding determinism in multithreaded applications, several approaches have being proposed. Jiménez-Peris et al.
[13] proposed a deterministic scheduler that guarantees that multithreaded replicas will execute deterministically. Nevertheless, only one thread can be active at a time and hence, no parallelism is exploited. More recent approaches enable parallelism by enforcing order in the acquisition of locks (e.g., [14]) or by forcing a total order in code sections processed by a transactional memory (e.g., [15]). In our case, multithreading is not an issue as MapReduce’s approach to parallelism guarantees that independent data is mapped to independent reducers and that related data is serialized and processed in the same reducer.

StreamCloud [16] is a stream processing system which addresses the parallel execution of operators. To preserve the ordering of events, deterministic execution based on real time stamps is used. As shown in our evaluation, deterministic execution induces more overhead than our proposed approach.

With the proliferation of many-core systems, dealing with non-determinism attracts more and more attention. Deterministic applications are easier to debug, make fault-tolerance less complicated, and are more easily made secure. Determinism can be tackled at different levels, e.g., at the operating system level [17], [18], [19]. We focus on determinism at the application level. Whether or not OS-level mechanisms for deterministic execution could be of help in our case makes for interesting future work.

C. Processing models

The MapReduce approach [1] has become very popular. The main reason is that the infrastructure can take care of scaling the processing to a large number of nodes. The interface for the user is also very simple, consisting of two functions, namely, map and reduce. Several implementations are available, Hadoop [2] being the most popular open-source implementation.

Recently, several modifications to the original MapReduce approach have been proposed. Hadoop Online (HOP) [8] is probably the best known modification. HOP allows mappers to forward results directly to the reducers and enable the reducers to compute partial aggregations earlier. This approach reduces the completion time of jobs in two ways. First, because the data can be sent directly from mappers to reducers, reducers do not need to always read it from disk. Second, partial aggregates can be useful to final consumers (e.g., by having enough information to trigger compensating actions in the case of the SLA monitoring). Our main contribution is the low-overhead fault tolerance approach, which is more efficient in both the additional resources used and in the runtime overhead. In contrast, HOP is based on the same fault-tolerance mechanism as MapReduce and suffers from the same inefficiencies, specially regarding the time to recover from a failure.

Regarding the StreamMapReduce model, HOP also supports periodic aggregations in the reducers and we initially tried to use it for problems such as the SLA monitoring problem defined in Section I. However, HOP’s performance for such cases (i.e., throughput, latency, and responsiveness for periodic jobs) was not adequate for the problems we focus.

The Percolator [20] system from Google, addresses incremental processing of continuously changing data. Percolator, however, provides a different programming interface from MapReduce. Users coordinate updates to the underlying data repository using distributed transactions.

The open source system S4 [21] closes the gap between HOP and Percolar: It supports incremental processing of continuous data streams with an interface similar to MapReduce. However, the system is only partial fault tolerant as operator state is not preserved in case of permanent node failures.

VI. Conclusion

We have introduced a new fault-tolerance mechanism for a streaming version of MapReduce which we call StreamMapReduce. This framework overcomes the strict phasing of MapReduce’s map and reduce stages by introducing stateful reducers. The state permits reducers to process key/value pairs as soon as they are emitted by a mapper. In this way, we can use StreamMapReduce in combination with incremental processing for near real-time processing of massive amounts of data. So far, we built a SLA conformance monitoring, a community-of-interest mining application, and a credit card fraud-detection application. With the help of StreamMapReduce we are able to process tens of millions of events per seconds on our 50 node cluster.

The disadvantage of in-memory state is that it is lost when a node or a process crashes. Hence, we need to apply a fault-tolerance mechanism that permits to recover the volatile state. The traditional approach is to log ahead. An event is logged to stable storage and only then processed. In StreamMapReduce we use a different approach: each stateful component is performing a periodic (uncoordinated) checkpointing. During recovery, the latest checkpoint is read and all events since then replayed. These events are kept in memory by the sender until they are not needed anymore. In-memory events can be pruned once all downstream nodes have already processed them and checkpointed their state. For our approach to work, a component needs to be able to reproduce the same sequence of events between the last checkpoint and the crash of the component. We achieve this by dividing time into epochs. The end-of-epoch state is deterministic, i.e., it is reconstructed during replay.

Our approach is inspired by virtual synchrony. Virtual synchrony has been proposed and intensively investigated in the context of state machine replication. Adapting ideas from virtual synchrony, we achieve almost the same throughput as processing events without ordering. This is remarkable
insofar as our no-order implementation is a best-case implementation that does not actually implement event logging (i.e., it provides no real fault tolerance).

In the process of reducing the overhead of deterministic in the context of (Stream)MapReduce, we have evaluated many optimizations with the intention to approach the throughput close to no-order execution. Only by switching to our virtual synchronous inspired execution, we were able to achieve throughputs that almost match the processing of no-order execution. We would like to point out, that we scale with the number of threads, i.e., we permit the parallel execution of non-conflicting events not only for the no-order execution but also for the deterministic and the virtual synchronous execution.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreements number 216181 (STREAM project) and 216852 (VELOX project).

REFERENCES


