Minimizing Overhead for Fault Tolerance in Event Stream Processing Systems

Dissertation (Kurzfassung)
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. André Martin
geboren am 25. September 1981 in Dresden

Gutachter: **Prof. Dr. (PhD) Christof W. Fetzer**
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Systems Engineering
01062 Dresden

Prof. Dr. (PhD) Peter R. Pietzuch
Imperial College London
Department of Computing
Large-scale Distributed Systems group
London SW7 2AZ, United Kingdom

Datum der Einreichung: 31. Juli 2015

1 Introduction

During the past decade, we have been witnessing a massive growth of data. In particular the advent of new mobile devices such as smart phones and tablets, and online services like facebook and twitter created a complete new era for data processing. Although there exist already well established approaches such as MapReduce [DG08] and its open source implementation Hadoop [Had15] in order to cope with this sheer amount of data, there is a recent trend of moving away from batch processing to low latency online processing using Event Stream Processing (ESP) systems. Inspired by the simplicity of the MapReduce programming paradigm, a number of open source as well as commercial ESP systems have evolved over the past years such as Apache S4 [NRNK10, S4215], Storm [Sto15] and Samza [Sam15], addressing the strong need for data processing in near real time.

Since the amount of data being processed often exceeds the processing power of a single machine, ESP systems are often carried out as *distributed systems* where multiple nodes perform data processing in a cooperative manner. However, with an increasing number of nodes used, the probability for a fault increases which can lead either to partial or even full system outages. Although several well established approaches to cope with system failures for distributed systems are known in literature, providing fault tolerance for ESP systems is challenging as those systems operate on constantly flowing data where the input stream cannot be simply stopped and restarted during system recovery.

One possibility for providing fault tolerance in ESP systems is the usage of checkpointing and logging, commonly referred as *rollback recovery/passive replication* in literature where the state of an operator is periodically checkpointed to some fault tolerant stable storage and in-flight events are kept in in-memory logs at upstream nodes (upstream backup) [HBR+05, GZY+09]. During system recovery, the most recent checkpoint is being loaded and previously in-flight events are replayed. An alternative to rollback recovery is *active replication* [Sch90, MFB11] where two identical copies of an operator are deployed on different nodes performing redundant processing. If one of the two copies crashes, the system continues to operate without having to initiate a long recovery procedure as in passive replication.

Although active replication provides the quickest recovery, it requires almost twice the resources while passive replication consumes only little resources, however, suffers from long recovery times. Despite the fact that both fault tolerance approaches have different characteristics with regards to recovery times and resource overhead, both require a *deterministic* processing of events in order to (i) reliable filter out duplicated events when using active replication and (ii) to provide replay-ability of events needed in order to recover precisely using passive replication. However, the use of deterministic execution imposes a non-negligible overhead as it increases processing latency and lowers throughput at the same time due to the cost of event ordering.

1.1 Goal and Contributions

In this thesis, we explore mechanisms to reduce the overhead imposed by fault tolerance in ESP systems. The contributions of this thesis are as follows: We present

1. the architecture and implementation of StreamMine3G, an ESP system we built entirely from scratch to study and evaluate novel fault tolerance and elasticity mechanisms,

2. an algorithm to reduce the overhead imposed by deterministic execution targeting com-

mutative tumbling windowed operators and improving the throughput by several orders of magnitude when using with passive and active replication,

3. an approach to improve the overall system availability by utilizing spare but paid cloud resources, and

4. an adaption based approach that minimizes the operational costs by selecting the least expensive fault tolerance scheme at runtime based on user-provided constraints.

Roadmap In Section 2, we will first introduce the reader to the architecture and implementation of StreamMine3G, the ESP system we implemented for evaluating the proposed approaches. In Section 3, we present an approach to lower the overhead of event ordering by introducing the notion of an epoch and evaluated it for the use with passive replication. We then extended this approach in Section 4 for use with active replication by proposing an epoch-based merge algorithm and a light-weight consensus protocol. Next, in Section 5, we improve system availability by using a hybrid approach of passive standby and active replication by utilizing spare but paid cloud resources, while in Section 6, we present an adaptation approach for fault tolerance lowering the overall resource consumption while still satisfying user-specified constraints such as recovery time and recovery semantics. In Section 7, we showcase the applicability of our approach using real world applications originating from the smart grid domain as well as geo-spatial data analysis, and finally conclude the dissertation in Section 8 with a summary of contributions and potential future work.

2 StreamMine3G Approach

In the following section, we will provide a brief overview about StreamMine3G, the ESP system we implemented in order to evaluate the proposed approaches.

StreamMine3G is a highly scalable ESP system targeting low latency data processing of streaming data. In order to analyze data, users can either opt for writing their own custom operators using the provided MapReduce-like interface and implementing a user-defined-function (UDF), or choose from an existing set of standard Complex Event Processing (CEP) operators such as filter, join, aggregation, and others. In addition to the operators, users must specify the order events are supposed to traverse the previously selected operators using a topology. A topology in StreamMine3G is represented by an acyclic directed graph (DAG) where each vertex, i.e., an operator, can have multiple upstream and downstream operators. In order to achieve scalability, operators in StreamMine3G are partitioned. Each partition processes only a subset of events from the incoming data stream. For data partitioning, users can either implement their own custom partitioner similar as in MapReduce, or use the provided hash-based or key-range based partitioner.

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

In order to prevent the node hosting the manager component being the single point of failure, the state of the component is stored in zookeeper upon each reconfiguration of the

system. In the event of a crash of the node, another node can transparently take over the role of the manager by simply recovering with the previously persisted state.

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

3 Lowering Runtime Overhead for Passive Replication

In the following section, we present an approach for lowering the overhead for passive replication by introducing the notion of an epoch.

3.1 Introduction & Motivation

ESP applications are often long running operations that continuously analyze data in order to carry out some form of service. In order to identify patterns and correlations between consecutive events, operators are often implemented as stateful components. One way for providing fault tolerance for such components is periodic checkpointing in combination with event logging for a later replay. However, since events may arrive in different orders at an operator during a recovery than they would have arrived originally due to small delays in network packet delivery, the immediate processing of such events would lead to possibly incorrect results. One way of preventing such situations is to *order events* prior to the processing in order to ensure a deterministic input to the operator code at all times. However, the ordering of events is costly as it introduces latency and lowers throughput as we will show in the evaluation of this section.

Many ESP operators used in ESP applications share the property of *commutativity* and operate on jumping windows where the order of processing within such windows is irrelevant for the computation of the correct result. Examples for such operators are joins and aggregations. However, processing the same event twice or even missing an event may still distort the result of those operators. Hence, determinism is still needed in order to provide *exactly once semantics*.

3.2 Approach

Based on our observation, we introduce the notion of an *epoch* that comprises a set of events based on their timestamps and matches the length of the window an operator is working on. In order assign events correctly to those epochs, i.e., the time-based windows, we require that events are equipped with monotonically increasing timestamps. The key idea of our approach is to process events within such epochs, i.e., windows in *arbitrary order*, however, *processing epochs itself in order*. Exactly once semantics can now be provided by solely performing checkpointing and recovery at *epoch-boundaries* as at those points in time the system still provides a deterministic snapshot.

3.3 Evaluation & Results

We evaluated our approach using a canonical streaming word count application that consists of two operators: A stateless map operator that splits lines read from a Wikipedia corpus file into individual words which are then accumulated by a stateful reduce operator. The stateful operator summarizes the word frequencies using a jumping window, i.e., an epoch where the length of the epoch is defined in terms of file position the word originated from in the source file. We evaluated in our experiments the performance of our epoch-based approach and compared it with no-ordering and strict event ordering approach. In strict ordering, every single event is ordered rather than applying the weak ordering scheme based on epochs. The results of the measurements are shown in Figure 1 and 2.

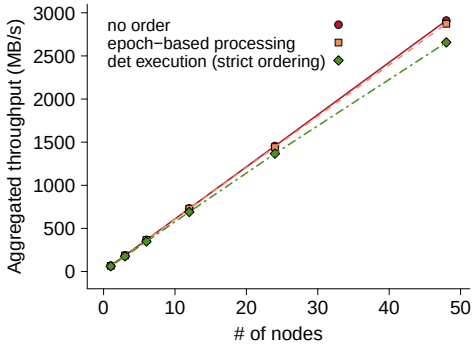

Figure 1: Aggregated throughput with increasing number of nodes.

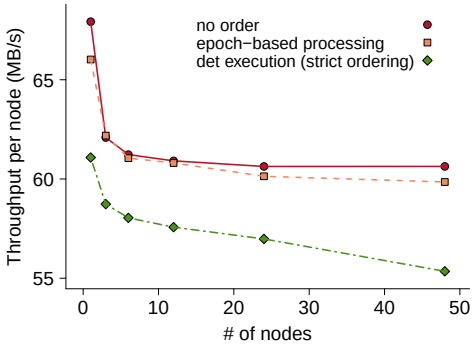

Figure 2: Per node throughput with varying number of compute nodes. The epoch-based approach significantly outperforms deterministic execution.

Figure 1 shows the accumulated throughput for the experiment running on a 50 nodes cluster while Figure 2 depicts the per node throughput. The experiments reveal that using our weak ordering scheme we can almost achieve the same throughput as when not applying any ordering, however, provide precise recovery as when using strict determinism.

4 Lowering Runtime Overhead for Active Replication

In the previous section, we have seen an approach for reducing the overhead of event ordering that provides *exactly once semantics* and *precise recovery* when used with passive replication based on checkpointing and in-memory event logging. In this section, we extend the previous approach to be used with active replication.

4.1 Introduction & Motivation

Active replication is a useful approach to recover applications that accumulate large portions of state as the secondary instance is holding the state already in memory rather than reading it from disk during a recovery. However, in order to use active replication, a costly atomic broadcast or deterministic execution (i.e., strict event ordering) must be used in order to ensure consistent processing across all replicas. However, when using commutative and windowed

operators, event ordering solely serves the purpose of maintaining consistency across replicas but does not have any impact on correctness due to the commutativity.

4.2 Approach

Inspired by the previous epoch-based processing approach, we will now present an approach that performs an *epoch-based deterministic merge* that ensures correctness for active replication, however, at a much lower cost than a strict event order/merge. The key idea of our approach is to merge epochs rather than individual events which is far less costly than a strict per event merge as we will show in the evaluation.

In order to perform an epoch-based deterministic merge, we enqueue events arriving from different upstream channels on a per epoch basis in separate so called *epoch-bags* first. Once an epoch completes, i.e., all channels have passed the epoch boundaries, the content of the epoch-bags is merged by processing the bags in the order of predefined channel identifiers. Since the channel identifiers are globally defined and events from upstream operators are delivered in FIFO order through TCP, the final sequence of events is identical and deterministic for all replicas.

In case upstream operator replicas or sources deliver identical but differently ordered sequences of events, and in order to reduce latency caused by stragglers, we furthermore propose a *light-weight consensus protocol* that selects for the available upstream replicas the set of bags to merge so that all downstream peers process the same sets of events. The protocol stops furthermore the propagation of non-determinism while decreasing latency at the same time.

4.3 Evaluation & Results

For the experimental evaluation of our approach we implemented a canonical application operating on jumping windows that consists of three operators, a partitioned source operator, an equi-join that combines the output from the source operator and a sink. In order to assess the performance of our approach, we compared our approach with no event ordering, strict ordering, the epoch-based merge and the consensus-based protocol. The outcome of the experiments is depicted in Figure 3 and 4.

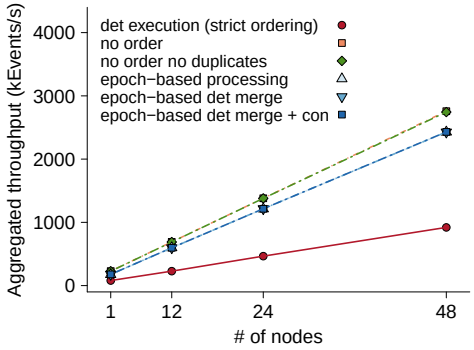

Figure 3: Horizontal scaling – aggregated throughput with increasing number of nodes.

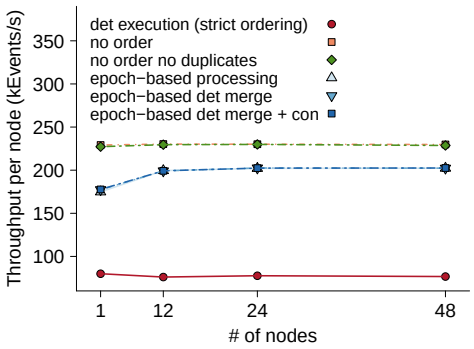

Figure 4: Horizontal scaling – per node throughput with increasing number of nodes.

Figure 3 depicts the accumulated throughput for the experiment running on a 50 nodes cluster while Figure 4 depicts the per node throughput. As shown in the figures, the epoch-based deterministic merge has a noticeable higher throughput than strict determinism (ordering) while there is only a marginal difference for the consensus-based variant in comparison to the epoch-based deterministic merge without agreement.

5 Improving Resource Utilization and Availability through Active Replication

While the objective of the previously presented approaches was to minimize the runtime overhead for fault tolerance by introducing a weak ordering scheme based on the notion of epochs, we will now present an approach that improves system available by efficiently using spare but paid resources in cloud environments.

5.1 Introduction & Motivation

ESP systems are naturally highly dynamic systems as they process data often originating from live data sources such as twitter fire hose or facebook where the data rate highly fluctuates and may rise or decrease by several orders of magnitude within relative short amounts of time. In order to cope with those peak loads and to prevent unresponsiveness of the system, the systems are usually run at conservative utilization levels often as low as 50%. Although the cloud computing model enables customers to acquire resources quite easily, migration and rebalancing mechanisms are still not fast enough to accommodate sudden load spikes forcing the service providers to run their applications at low utilization levels. However, cloud users are nevertheless charged by full hours regardless of the actual resource consumption of their virtual machines.

5.2 Approach

In order to fully utilize all available resources a virtual machine offers, we use a *hybrid approach* for fault tolerance where we transition between *active replication* and *passive standby* based on the utilization of the system. In order to transition between the two states, we use a *priority scheduler* that prioritizes the processing of the primary and transparently pauses secondaries once resources are exhausted. Hence, the system transparently transitions between active replication and passive standby where the secondary is paused until sufficient resources become available again. In order to keep the secondary up-to-date during high system utilization, the state of the primary is periodically checkpointed to the memory of the secondary which allows the secondary to prune enqueued but not yet processed events from queues. Using an interleaved partitioning scheme for the placement of primary and secondaries across the cluster, we can furthermore decrease the overhead imposed on nodes during system recovery.

5.3 Evaluation & Results

In the following experiment, we investigated the system behavior of our proposed solution in the event of load peaks. To simulate spikes, we use a load generator to emit events at different

rates for predefined periods of time. Figure 5 depicts the aggregated throughput for a single node and the status of the input queues of secondary slices on that node over time. In this experiment, the load generator introduced load spikes every 20 seconds for two seconds.

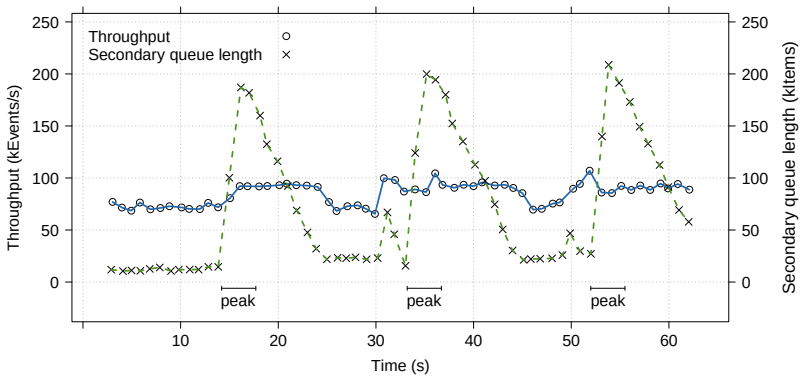

Figure 5: Event throughput and queue length behavior of secondary slice queues with induced load spikes.

During a load peak, no events at the secondary slices on that node are being processed, hence queues grow quickly. Once the load decreases, the secondaries resume the processing of events, hence, the amount of events in the queues of the secondary slices shrink. Note that the aggregated 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.

6 Adaptive and Low Cost Fault Tolerance for Cloud Environments

In the previous section, we presented an approach to utilize spare but already paid resources in order to improve system availability by transitioning between active replication and passive standby during runtime. In this section, we present an approach that lowers the overall resource consumption by selecting the fault tolerance scheme at runtime that consumes the least amount of resources while still guaranteeing user-defined constraints such as *recovery time* and *recovery semantics*.

6.1 Introduction & Motivation

Fault tolerance in ESP systems can be carried out through various mechanism and replication schemes. For example, in *active replication*, redundant processing is used as a mechanism to carry out fault tolerance where as in *passive replication*, a combination of periodic checkpointing and event logging is used in order to mitigate system crashes. Although active replication provides a quick recovery, it comes with a high price as it consumes almost twice of the resources while passive replication consumes only little resources, however, suffers from a long recovery time. Besides active and passive replication, there exist several more schemes to carry

out fault tolerance such as *active* and *passive standby* where recovery time is traded by resource usage.

Choosing the right fault tolerance scheme is not easy as all those schemes have different resource footprints and recovery times. Moreover, the footprint and recovery time for those schemes are not static as they strongly depend on system parameters that can greatly vary during the course of processing. For example, the recovery time for passive replication strongly depends on the size of the checkpoint that must be read during a recovery. However, the size of a checkpoint strongly depends on the incoming data rate when considering a stateful sliding window operator.

6.2 Approach

In order to free the user from the burden of choosing an appropriate fault tolerance scheme for his application, we propose a *fault tolerance controller* that takes decisions on behalf of the user at runtime. The controller takes into account user-provided constraints such as the desired *recovery time* and *recovery semantics*, i.e., precise or gap recovery.

We therefore extended StreamMine3G to support six different fault tolerance schemes the controller can choose from as shown in Figure 6: ① *active replication*, ② *active standby*, ③ *passive standby hot* and ④ *cold*, ⑤ *deployed*, and ⑥ *passive replication*. The schemes have different characteristics with regards to resource consumption of CPU, memory, network bandwidth, the amount of nodes used, and recovery time. In order to choose the correct scheme, the controller uses an estimation-based approach where historically collected measurements are continuously evaluated for an estimation of the expected recovery time for each of the availably schemes.

6.3 Evaluation & Results

We evaluated our approach with regards to the amount of resources that can be saved in comparison to a conservative use of full active replication. Figure 6 shows the runtime behavior of our system.

At the top graph, the throughput and how it varies over time is shown. Since the operator used for evaluated is a sliding window operator that accumulates events of the past 20 seconds, the size of the state follows the pattern of the throughput curve. At the bottom graphs, the chosen fault tolerance scheme is shown for each time slice. As shown in the plot, the system starts with active replication, as it is the safe choice. Once enough measurements have been collected, the controller quickly switches to the deployed state replication scheme as the state size and the throughput are quite low and, hence, recovery from disk and replay from upstream nodes can be easily accomplished within the user's specified recovery time threshold. However, as spikes occur which let the state and upstream queues grow, the controller switches between passive replication and deployed replication scheme. A cool down time of five seconds prevents the system from oscillating due to sudden load spikes which are common in workloads originating from live data sources such as Twitter streams. In summary, the controller chose a combination of passive replication and deployed during the first half of the experiment, whereas the second half was dominated by passive hot standby.

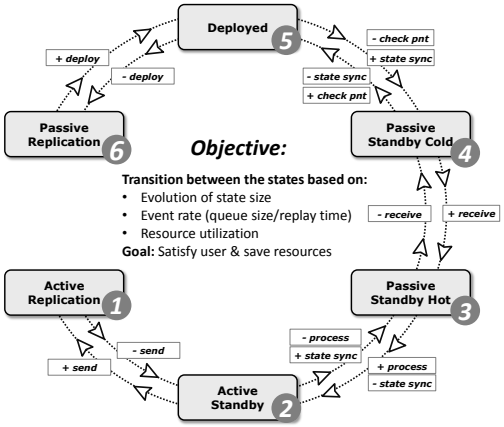

Figure 6: Fault tolerance schemes state transition wheel: Active replication, active standby, passive standby hot and cold, deployed, and passive replication.

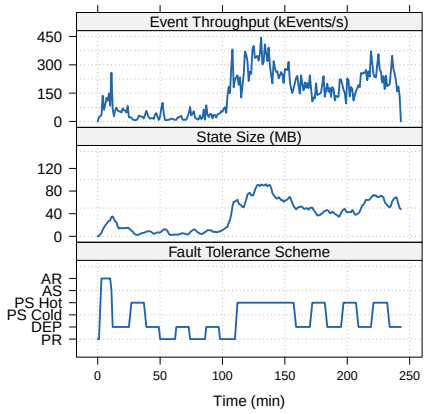

Figure 7: Throughput, state size and fault tolerance scheme evolution over time using Twitter workload with a recovery threshold set to 5.5 seconds.

7 Application Examples

In order to evaluate the applicability of our StreamMine3G approach with regards to the proposed programming model and system architecture, we implemented several real world applications and evaluated them with regards to scalability and their overall performance.

7.1 Energy Consumption Prediction

In the first use case, we used StreamMine3G to provide a short term load prediction and outlier detection for sensor data originating from smart plugs in the context of smart grids. The topology of the application consists of three operators: A *source operator* for converting the data originating from the smart plugs into a StreamMine3G compatible data format. A *prediction operator* to provide an estimation for the energy consumption for a future time slice which is two steps ahead of time. The prediction must be furthermore provided for different time slice slices ranging from 1 to 120 $mins$. And an *outlier detection operator* that has to detect outliers, i.e., smart plugs that consume excessive energy in comparison to the median of all power plugs connected to the grid.

The application has been implemented using StreamMine3G's MapReduce-like interface which manages state for the application developer. The state has been carried out as multi-dimensional hash-maps in order to store historical measurements to drive the prediction and to detect outliers using a 24 $hours$ sliding window. Since the application requires to keep the complete history, the application benefits from the elasticity properties of StreamMine3G where new resources can be easily acquired and the system expanded during runtime without having to pause or shutdown the system. Moreover, the fault tolerance mechanisms implemented in StreamMine3G allows the application to recover in a precise manner without loosing any stored information.

The contribution [MBF15] has been selected as finalist for the annual *DEBS 2014 grand challenge* where the application has been showcased with regards to their implementation and

its properties such as correctness and fault tolerance, while the *scalability* and *elasticity features* of StreamMine3G have been showcased at the UCC 2014 conference [MSBF14] where it was awarded with the *UCC 2014 cloud challenge award*.

7.2 Taxi Rides Analysis

For the second use case, the StreamMine3G approach has been applied in the context of geospatial data analysis of taxi rides originating in the New York city area. The objective of the application is to provide the *top-ten most frequently driven routes* and *most profitable areas* in a continuous fashion.

In order to provide the two output stream, the *top-tens* are computed using two different operators: One which determines the most frequent driven routes while the second the most profitable areas. In order to make the system scalable, the operators have been furthermore split up in a *tracker* operator and a *top-k* computation operator where the first one consumes the incoming data stream in a partitioned fashion. Using StreamMine3G's deterministic execution properties, i.e., strict event ordering, the correctness for the partitioned variant can be guaranteed. The application [MBF15] has been selected for presentation and as finalist for the annual *DEBS 2015 grand challenge* again.

8 Conclusions

In this dissertation, we have presented several approaches for lowering the overhead imposed by fault tolerance mechanisms in ESP systems. We first presented StreamMine3G, our ESP system we implemented to study and evaluate our approaches. We then showed that the overhead of event ordering required to recover applications in a precise manner and to ensure replay-ability can be noticeable reduced when using *epochs* for *commutative* and *tumbling windowed operators* [MKC+11]. In order to apply this concept also for actively replicated operators, we extended our approach by delaying the processing of epochs and performing a *epoch-based deterministic merge*. In conjunction with a *light-weight consensus protocol*, latency as well as the propagation of non-determinism can be reduced and prevented, respectively.

Next we explored an approach to increase system available by efficiently using spare but paid cloud resources [MFB11]. We therefore combined *active replication* and *passive standby* where the system transparently switches between the two states using a *priority scheduler*. Our evaluation shows that the system maintains responsiveness while still providing high availability through active replication at (almost) no cost.

As a last approach, we presented a *fault tolerance controller* [MSD+15] that selects an appropriate fault tolerance scheme on behalf of the user at runtime based on previously provided constraints such as recovery time and recovery semantics. Our evaluation reveals that with our approach a considerable amount of resource can be saved in comparison to the conservative use of active replication.

In order to evaluate the applicability of our StreamMine3G approach including the proposed programming model, system architecture, fault tolerance and elasticity mechanisms, we implemented several real world application and participated successfully twice in the annual DEBS grand challenge[MMBF14, MBF15] and the UCC cloud challenge [MSBF14].

References

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[GZY+09] Yu Gu, Zhe Zhang, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. An empirical study of high availability in stream processing systems. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 23:1–23:9, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[Had15] Hadoop mapreduce open source implementation. http://hadoop.apache.org/, 2015.

[HBR+05] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 779–790, Washington, DC, USA, 2005. IEEE Computer Society.

[MBF15] André Martin, Andrey Brito, and Christof Fetzer. Real time data analysis of taxi rides using streammine3g. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 269–276, New York, NY, USA, 2015. ACM.

[MFB11] André Martin, Christof Fetzer, and Andrey Brito. Active replication at (almost) no cost. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, pages 21–30, Washington, DC, USA, 2011. IEEE Computer Society.

[MKC+11] André Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, Christof Fetzer, and Andrey Brito. Low-overhead fault tolerance for high-throughput data processing systems. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 689–699, Washington, DC, USA, 2011. IEEE Computer Society.

[MMBF14] André Martin, Rodolfo Marinho, Andrey Brito, and Christof Fetzer. Predicting energy consumption with streammine3g. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 270–275, New York, NY, USA, 2014. ACM.

[MSBF14] André Martin, Rodolfo Silva, Andrey Brito, and Christof Fetzer. Low cost energy forecasting for smart grids using stream mine 3g and amazon ec2. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 523–528, Washington, DC, USA, 2014. IEEE Computer Society.

[MSD+15] André Martin, Tiaraju Smaneoto, Tobias Dietze, Andrey Brito, and Christof Fetzer. User-constraint and self-adaptive fault tolerance for event stream processing systems. In *Proceedings of The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*, Los Alamitos, CA, USA, 2015. IEEE Computer Society.

[NRNK10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[S4215] Apache s4 - distributed stream computing platform. https://incubator.apache.org/s4/, 2015.

[Sam15] Apache samza - a distributed stream processing framework. http://samza.incubator.apache.org/, 2015.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[Sto15] Apache storm - distributed and fault-tolerant realtime computation. https://storm.incubator.apache.org/, 2015.