DISSERTATION

MINIMIZING OVERHEAD FOR FAULT TOLERANCE IN EVENT STREAM PROCESSING SYSTEMS

André Martin
Technische Universität Dresden,
Fakultät Informatik

Minimizing Overhead for Fault Tolerance in Event Stream Processing Systems - Dissertation

Printed in Germany.

Minimizing Overhead for Fault Tolerance in Event Stream Processing Systems

Dissertation
submitted for the degree of
Doktoringenieur (Dr.-Ing.)

at
Technische Universität Dresden
Fakultät Informatik

by
Dipl.-Inf. André Martin
born on September, 25th 1981 in Dresden

Reviewers:

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

Date of defense: December, 17th 2015

Dresden, June 2016

Wings are a constraint that makes
it possible to fly.

— Robert Bringhurst

To my family.

Acknowledgments

First of all, I want to thank my supervisor Prof. Christof Fetzer who provided me with the inspiration, freedom, room for creativity and the opportunity to execute the research leading to this work. Second, I would like to acknowledge Prof. Peter Pietzuch for his curiosity and openness concerning my research and his commitment to serve as a secondary reviewer. Third, I'm deeply indebted to my friend and former colleague Prof. Andrey Brito who taught me all the essential skills to accomplish such work - muito obrigado! And I want to thank my family, especially my parents who equipped me with the right tools for life to start and complete such a big project, and all my friends who have been always accompanying me during all those years, constantly giving me support and the strength needed.

Thanks to my family, all my friends and former or present colleagues that inspired me and left a foot mark in my life in some or another way: Jurik A., Kornelia B. aka Konni, Björn B. aka Bolle, Georg B., Fabian B., Andrey & Esther B., Berenike B. aka Nike, Veronika B., Carolin B. aka Caro, Felicitas E. aka Feli, Pascal F., Judith G., Rebekka G., Eleni G., Sylvia G. aka Sylvi, Anneka G., Annemarie G., Till H., Pete I-C., Judith K., Johanna K. aka Jojo, Anne K., Bernhard K., Paula K., Claudia K. aka Claudi, Marcus K., Christian K., Paula K., Manja K., Christian K., Ellen LV., Robert W. LV. aka Bob, Franka L., Maria M. aka Mia, Julia M., Lauren M., Mary Ellen M., Holger M., Frank M., Simon N., Br. Paolo, Br. András, Marcelo P., Sandra P. aka San, Franziska P., Frank P aka. Cod father, Jacqueline P., Frido P. SJ, Nicole P., Stefan P., Jakob R., Anja R., Marc R., Adam & Jen P.R., Heinz-Wilhelm R-S., Anke R., Sebastian R., Erwin P. S., Josefine S. aka Fine, Diana S. aka Dine., Jan S., David S. aka Dave., Maria S. aka Ria., Sebastian S., Stella S., Josef S., Cathleen T. aka Caddle., Hans-Joachim T. aka. HanJo., Zane V. D., Michael V, Susann W. aka Sus, Mike W. aka Wickert, Stefan W., Jacqueline v. d. L. and many more.

Abstract

Event Stream Processing (ESP) is a well-established approach for low-latency data processing enabling users to quickly react to relevant situations in soft real-time. In order to cope with the sheer amount of data being generated each day and to cope with fluctuating workloads originating from data sources such as Twitter and Facebook, such systems must be highly scalable and elastic. Hence, ESP systems are typically long running applications deployed on several hundreds of nodes in either dedicated data-centers or cloud environments such as Amazon EC2. In such environments, nodes are likely to fail due to software aging, process or hardware errors whereas the unbounded stream of data asks for continuous processing.

In order to cope with node failures, several fault tolerance approaches have been proposed in literature. Active replication and rollback recovery-based on checkpointing and in-memory logging (upstream backup) are two commonly used approaches in order to cope with such failures in the context of ESP systems. However, these approaches suffer either from a high resource footprint, low throughput or unresponsiveness due to long recovery times. Moreover, in order to recover applications in a precise manner using exactly once semantics, the use of deterministic execution is required which adds another layer of complexity and overhead.

The goal of this thesis is to lower the overhead for fault tolerance in ESP systems. We first present StreamMine3G, our ESP system we built entirely from scratch in order to study and evaluate novel approaches for fault tolerance and elasticity. We then present an approach to reduce the overhead of deterministic execution by using a weak, epoch-based rather than strict ordering scheme for commutative and tumbling windowed operators that allows applications to recover precisely using active or passive replication. Since most applications are running in cloud environments nowadays, we furthermore propose an approach to increase the system availability by efficiently utilizing spare but paid resources for fault tolerance. Finally, in order to free users from the burden of choosing the correct fault tolerance scheme for their applications that guarantees the desired recovery time while still saving resources, we present a controller-based approach that adapts fault tolerance at runtime. We furthermore showcase the applicability of our StreamMine3G approach using real world applications and examples.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

During the past decade, we have been witnessing a massive growth of data. In particular, the advent of mobile devices such as tablets and smart phones as well as social web applications like Facebook and Twitter started a complete new era for data processing. As such devices and applications generate constantly new data, new paradigms and systems are needed in order to handle, process and extract useful information from this sheer amount of data. Moreover, as the volume of this data does not fit into a single machine anymore, data processing must be performed using *distributed systems* imposing new challenges on data analysts and system engineers.

In 2004, Google presented a new and revolutionary approach called MapReduce [DG08] for processing data in a distributed fashion. The approach is inspired and derived from its counterpart found in the functional programming paradigm, however, targets large scale data processing in clusters involving thousands of nodes. In particular, the simplicity of the programming model has made MapReduce the de facto standard for data processing of (unstructured and) large amounts of data over the following years.

As the volume of data being generated each day is constantly growing, an increasing number of machines is needed in order to extract meaningful information within an acceptable amount of time. However, using a large number of nodes increases the probability for hardware failures that may occur during execution and then may lead to partial or even full system crashes. Moreover, software bugs in user code or libraries used in the application for information extraction are another source for crashes, hence, *fault tolerance* has become an equally important requirement for such big data systems. Fortunately, Google's MapReduce approach comes already with implicit fault tolerance freeing its users from the burden of having to implement appropriate mechanisms for fault tolerance themselves.

Although MapReduce seems to tackle the majority of challenges data analysts are facing when using large scale data processing systems nowadays, the approach is only suitable for classes of applications that do not have tight requirements with regards to processing latency. As MapReduce follows the *store-and-process* execution pattern, results can only be

obtained after storing the data on a distributed filesystem (DFS) first and second, after the complete execution of a MapReduce job. However, nowadays applications have often tighter requirements such as processing the data at the instant it is generated in order to produce results within a few seconds or even milliseconds. *Event Stream Processing* (ESP) Systems is a class of systems that addresses those application scenarios and which will be the focus in this dissertation.

In contrast to MapReduce-based systems, ESP systems operate on continuous streams of events (data tuples) which traverse a topology of operators where several events are continuously filtered, aggregated or enriched in order to extract useful information. Examples for such applications range from click stream analysis or recommender systems to credit card fraud detection systems.

Despite the existence of ESP Systems for more than a decade, ESP systems are recently facing a true renaissance as more and more enterprises are transitioning from batch to real-time data processing systems. Inspired by the simplicity of the MapReduce programming paradigm [DG08] and its open source implementation Hadoop [Had15], a number of new ESP systems have evolved during the past couple of years addressing the strong need of real-time computation in industry. Examples for such systems range from Apache S4 [NRNK10, S4215] (originally pushed by Yahoo!), Storm [Sto15] (by Twitter) to Samza [Sam15] (by LinkedIn).

Even though all of those systems serve their original purpose, i.e., the processing of large amounts of data in near real-time and in a scalable manner, only little attention has been brought to fault tolerance and elasticity in those systems. For example, the popular open source ESP system Storm [Sto15] provides only partial fault tolerance as it only supports event replay through its transactional topologies, however, it does not preserve accumulated operator state which is simply lost upon node crashes. Moreover, Storm clusters can only scale-out using the re-balance feature during runtime, however, the contraction of a cluster in order to save resources in times of low system utilization is not supported.

1.1 Challenges and Contributions

Providing and implementing fault tolerance in ESP systems is not trivial. First, ESP systems operate on constantly moving rather than static data as in MapReduce where the data stream cannot be simply stopped or paused in the event of a node crash. Second, since ESP applications operate on streaming data, most operators are stateful where several events are aggregated or correlated in order to extract meaningful information. In order to provide fast access to those previously seen events and to identify correlations, such operator state is often solely kept in the node's main memory. Hence, in order to provide fault tolerance when processing streaming data, the ESP system must ensure that (i) the operator state can be fully recovered after a node crash and that (ii) no previously "in-flight" events are lost due to the streaming nature of the system.

A commonly used mechanism to achieve state persistence is *checkpointing* where a copy of the operator state is periodically saved to stable storage such as a local or remote hard disk. In combination with event logging, an ESP system can then fully recover from node failures without losing any events. This mechanism is also commonly referred to as *checkpoint-recovery* or *passive replication* in literature. An alternative to passive replication, is *active replication* where two identical copies of an operator are instantiated on two different physical nodes so that the crash of one instance does not affect the other.

Although fault tolerance improves system availability, it also introduces a non-negligible overhead: For example, in order to provide fault tolerance based on passive replication using *precise recovery* (i.e., neither a single event will be lost nor processed twice after a crash occurred), events must be ordered deterministically prior to processing which consumes additional computational resources as well as introduces latency. Or, when using active replication, at least twice the resources are needed doubling in worst case the operational costs.

The goal of this thesis is to reduce the overhead imposed by fault tolerance mechanisms used 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 commutative tumbling windowed operators and improving the throughput by several orders of magnitude when using with passive or 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 operational costs by selecting the least expensive fault tolerance scheme at runtime based on user-provided constraints.

1.2 Dissertation Outline

The remainder of this dissertation is organized as follows:

Chapter 2 provides an introduction and general background information about ESP such as concepts and models serving as a foundation for the following chapters. The chapter also reviews important concepts in fault tolerance commonly used in distributed ESP systems.

Chapter 3 introduces the reader to our StreamMine3G system. The chapter provides an overview about its programming model and architecture followed by an in-depth description about the algorithms and protocols implemented for carrying out event ordering and fault tolerance. The chapter concludes with an overview about the employed elasticity algorithms in the system sharing several mechanisms used also for fault tolerance.

Chapter 4 presents and evaluates an approach to reduce the overhead of event ordering for commutative tumbling operators by assigning events to epochs matching processing windows. The approach is evaluated in the context of passive replication using checkpointing and replay.

Chapter 5 extends the approach presented in the previous chapter in order to be used with active replication. This requires postponing the processing of epochs until the start of the following epoch. In order to reduce latency and to prevent the propagation of non-determinism originating from replicated sources, a light-weight consensus protocol is proposed in the second part of the chapter.

Chapter 6 describes and evaluates a hybrid approach that combines active replication and passive standby in order to improve system availability by using spare but paid cloud resources. The system transitions between those two states based on the availability of resources.

Chapter 7 presents a fault tolerance controller component that transitions between several fault tolerance schemes based on user-provided constraints such as recovery time and semantics with the goal of reducing the overall resource utilization while still ensuring high availability.

Chapter 8 provides applications examples where our StreamMine3G system has been evaluated using real world applications. Various design challenges as well as their solutions for different types of applications are thoroughly presented.

Chapter 9 summarizes the achievements and contributions of the dissertation and outlines possible future work.

2 Background

In this chapter we introduce the reader to the basic concepts in Event Stream Processing (ESP) and fault tolerance that will be relevant in later chapters. We start with an overview of the fundamentals of ESP covering the data model and concepts such as operators, topologies and queries, and conclude the chapter with an introduction into fault tolerance concepts used in the area of ESP systems.

2.1 Event Stream Processing

2.1.1 Data Model

Event stream processing systems operate on stream of events where an event represents any kind of information. Hence, the information an event is carrying can be either provided in structured or unstructured form. In addition to the information itself which we will refer as payload ρ in the following, events are equipped with some metadata needed to ensure a correct processing by the ESP system at hand. Such metadata consists typically of some timestamp τ, used for ordering events prior to processing and some key κ for partitioning the data. Hence, an event can be represented as the following tuple: $e = (\tau, \kappa, \rho)$, and a sequence of such events is called a *stream* that consists typically of an infinite number of events.

2.1.2 Operator Model

Events in an ESP system are processed by *operators*. An operator o takes an event e from an input stream i, processes the event and produces (depending on the nature of the operator) new events forming an output stream o. The processing of an event is accomplished through the invocation of an *operator function* $f_o : (e_i, \Theta_i) \rightarrow (o, \Theta_o)$ where Θ represents *state* that is updated during the course of processing.

2.1.3 Query Model

ESP systems follow a modular design in which operators solely perform one specific and simple task such as filtering or aggregating etc. events. Even though this design comes with the benefit of reusing the same operator for a wide range of applications, it also implies the usage of more than a single operator in order to process and extract useful information in an application using ESP. Hence, ESP applications are generally specified through *queries*.

A query $q = (\mathcal{O}, S)$ is defined through a directed acyclic graph (DAG) where $\mathcal{O}$ is the set of operators and S the set of streams defining the flow of events. A stream $s \in S$ connects two operators o_u and o_d through a communication channel allowing events to flow from operator o_u to o_d, i.e., $s = (o_u, o_d)$ where $\{o_u, o_d\} \subseteq \mathcal{O}$. Since operator o_d consumes events from operator o_u, it is called the *downstream operator* of operator o_u while operator o_u is the *upstream operator* of operator o_d.

The query graph definition allows the consumption of events originating from multiple upstream operators. Therefore, an input stream s_i of an operator o comprises a set S_o of output streams produced by the set of predecessor operators. The set S_o of output streams is *merged* through a *merge function* $f_m : S_o \rightarrow s_i$ prior to providing it to the operator at hand as a single input stream s_i. The objective of the merge function is to define a *total order* for the set of events originating from previously independent and uncorrelated streams produced by upstream operators. The ordering of events is important as it will have a large influence in the processing semantics when providing fault tolerance and guaranteeing repeatability of events.

2.1.4 Execution Model

Once a query has been defined by the application developer, it is deployed on a set of nodes for execution. A node can host an arbitrary number of operators allowing an efficient usage of the underlying physical resources. However, in order to ensure the *scalability* of the system, *partitioning* of operators becomes necessary. We therefore distinguish in the following between the *logical* and *physical* representation of a query as shown in Figure 2.1.

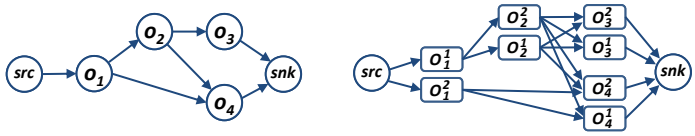

Figure 2.1: Logical (left) and physical (right) representation of a query graph.

In the physical representation of query q, an operator o can be partitioned into π operator partitions, i.e., $o = o^1, \ldots, o^\pi$ where $\pi \in \mathbb{N}^+$ specifies the parallelization level for operator o and where each operator partition o^j shares the semantics of operator o.

An operator partition o^j consumes events from a partitioned input stream s_i^j that carries a

subset of events from the original input stream s_i. The input stream s_i is partitioned using a *partitioner function* $f_p : s_i \rightarrow \{s_i^1, \ldots, s_i^\pi\}$. The objective of the partitioner function is to define how events are distributed across a set of input streams.

Since an operator may consume events from multiple upstream operators, and the upstream and downstream operator may be partitioned as well, a sophisticated interplay of partitioning and merging of streams is required in order to allow a scalable and correct processing of events.

In principle, the merging and partitioning of streams can be executed in different orders as shown in Figure 2.2. In the first scheme, the set of output streams S_o produced by one or multiple upstream operators is merged first into a single stream s prior splitting it up into multiple input stream s_i ready for consumption at the partitioned downstream operator. Note that m in Figure 2.2 represents the previously defined merge function, p the partition function, respectively.

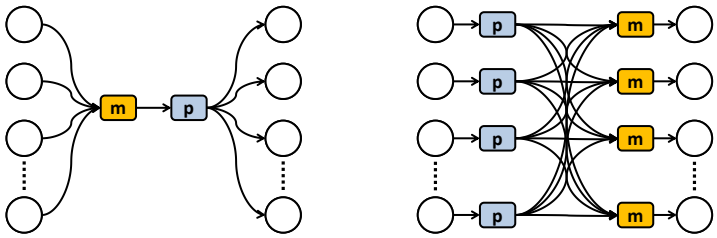

Figure 2.2: Merge-partition (left) and partition-merge (right) scheme

Even though this schemes comes with the advantage of having to instantiate only a single merger and partitioner, and reduces the number of communication channels, it is not scalable as all events produced by the set of upstream operators must be transferred to a single merger instance which becomes quickly the system's bottleneck. An alternative more scalable solution is depicted on the right side in Figure 2.2. In this scheme, output streams of upstream operators are first partitioned according to the level of parallelism of the downstream operators using the provided partitioner function, and then merged downstream individually at each operator partition. While this scheme introduces more complexity as it relies on multiple partitioner and merger instances as well as more communication channels, it is highly scalable as we will show in the later chapters.

2.1.5 Input and Output Operators

Operators are the central unit of processing in an ESP system. They are consumers and producers at the same time, as they consume events provided through an input stream and produce events for consumption by other operators. However, there are two types of operators diverging from this scheme: *sources* and *sinks*.

Source operators do not consume any events, however, they produce events offered as input streams for other operators, hence, acting as a source in a query. Although source operators do not consume event streams coming from other operators, they often act as *input adapters* or *data converters* through the consumption of arbitrary data coming from external world and transforming such data streams for further processing within an ESP system.

On the contrary, *sink operators* do consume event streams specified by the query but do not produce any events streams that can be further processed by any other operator within the query. In fact, sink operators do often produce events, however, in a different format targeting the consumption of results by external systems. Hence, source and sink operators can be considered as interfaces of an ESP system and the external world.

2.1.6 Operator State and Windows Types

Operators in ESP systems can be distinguished in *stateful* and *stateless*. While the output of a stateless operator solely depends on the current event, a stateful operator takes some state and the current event as input in order to produce some output. The state can comprise any kind of data structure such as a collection of events received so far or any other kind of complex data structure needed in order to hold some accumulated information, hence, stateful operators can be used to derive higher level information from input streams such as detecting event patterns or computing aggregates, e.g., sums or averages etc. across some period of time.

ESP systems are often considered as *inverted databases*: While database systems store data items with the objective of executing different queries on a static data set, ESP systems take the opposite approach where the set of queries is fixed[1] and applied on dynamic data, i.e., a stream of events.

Since ESP systems share many similarities with database systems, the majority of operators found in database systems are also available in many ESP systems. However, certain classes of operators such as aggregation require an additional parameter specifying a *window* in order to operate on infinite streams of events as found in ESP systems. Such windows define finite sets of events derived from the infinite input stream though a *boundary condition* marking the beginning and the end of a window. Without windows, many operators inherited from database systems would simply block infinitely as input streams in ESP systems do neither mark a beginning nor an end.

Window Types

Window boundaries can be defined by *predicates*, by *time* or by a *counter*. In the first case, the start and the end of a window are specified through predicates: Once an event satisfies the

[1] Queries in ESP systems can also be (un)installed and replaced over the course of time.

start predicate, the window is opened and new events are added to it until an event satisfies the end/closing predicate. Windows defined by predicates are called *landmark windows* as the start marker is fixed while having a variable size and end marker. Contrary to landmark windows, time and count-based windows are *sliding windows* as their start and end marker is constantly moving during the arrival of new events keeping the size of the window constant. The size for time-based windows (i.e., *time windows*) is either defined through wall clock or logical time whereas for count-based windows (i.e., *count windows*), the number of events is used to define its size. In principle, it is also possible to define sliding windows using predicates where the start and end markers move upon satisfaction of the previously defined predicates.

Sliding windows require one additional parameter: the *sliding step* specifying when new windows are created. For example, a time window with a length of eight seconds and a sliding step of two seconds will result in four concurrent windows (i.e., $8/2 = 4$). Hence, each event will contribute to four coexisting windows. Once a window closes, the events of the closing window will be provided as a set to the operator for consumption. Windows where the sliding step matches the size of the window itself are also known as *tumbling* or *jumping window* as shown in Figure 2.3.

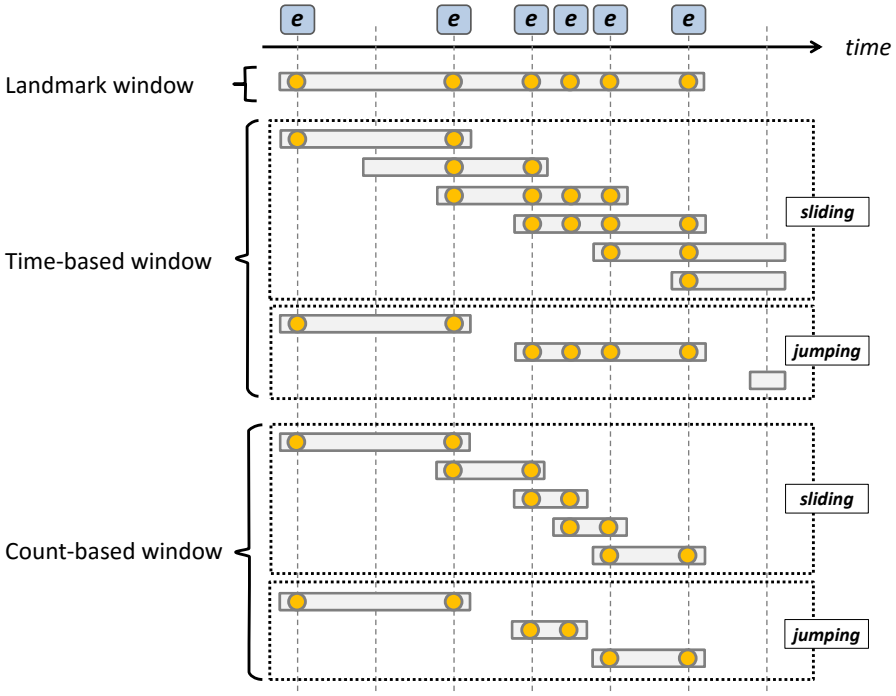

Figure 2.3: Window types: Landmark, Time and count-based.

In addition to the different window types presented previously, there exists various models with respect to providing the content of a window to the operator. The content can either be made available upon *windows creation*, *content change* of the window, or *window closure*. For

simplicity, we consider only the last case during the remainder of this thesis as this is the most frequently used pattern in ESP systems nowadays.

2.1.7 Operator Types

In the following section, we will provide an overview of several standard operators inherited from database systems and commonly used in ESP queries.

Filter

A filter operator is a generalized version of a selection operator and used to forward or discard events based on the evaluation of a predicate P. The operator can be either stateless or stateful. For example, consider temperature data as an input stream where the filter is used to discard measurements above a certain threshold, e.g., 30 degrees Celcius. The predicate in this example is stateless while the filtering of duplicates would require a stateful version of the operator in order to keep track of previously seen events. A filter can be formalized as follows:

$$e_o = \begin{cases} e_i & \text{if } P(e_i) = true \\ \varnothing & \text{otherwise} \end{cases} \tag{2.1}$$

where $e_i \in s_i$ and $e_o \in s_o$.

Map

A map operator is a generalized version of a projection operator and is used to perform format conversions or transformations of the input stream of events. The conversion or transformation is carried out through a user-provided function f_{map} given by:

$$e_o = f_{map}(e_i) \tag{2.2}$$

where e_i is of type T_i,e_o of type T_o , $f_{map} : (T_i \rightarrow T_o)$ and $T_i \neq T_o$.

In a similar way as with filters, map operators can be implemented as stateful operators if the computation of f_{map} depends on previously seen events.

Aggregate

An aggregate operator is used to combine multiple events into a single event. The set of events to be considered in an aggregate is defined by a window w. The aggregate function f_{agg}

specifies how events within such a given window w are summarized upon window closure. Commonly used aggregate functions are average, median, min, max and sum. An aggregate operator can be formalized as follows:

$$e_o = f_{agg}(e_{i1}, e_{i2}, \ldots, e_{ij}) | e_{ij} \in w \tag{2.3}$$

where e_{ij} is of type T_i,e_o of type T_o , $f_{agg} : (T_i \rightarrow T_o)$ and $T_i \neq T_o$.

Join

A join operator combines two events originating from two different input streams forming a new output event carrying information derived from both sources. The input events must *jointly* satisfy a predicate P in order to be selected as join candidates. Similarly as with the aggregate operator, the join operator considers windows of events (w_r and w_l) provided by the two input streams s_{ir} and s_{il}. Hence, a join operator can be formalized as:

$$e_o = f_{join}(e_{ir}, e_{il}) | e_{ir} \in w_r, e_{il} \in w_l \text{ and } P(e_{ir}, e_{il}) = true \tag{2.4}$$

where w_r,w_l comprises events originating from input stream s_{ir} and s_{il}, respectively.

2.2 Fault Tolerance

ESP systems are used for a wide range of applications such as real-time bidding (RTB), fraud detection and click stream analysis. Some of those applications are classified as *critical* systems as a system failure and its temporary unavailability may result in high financial losses. Hence, fault tolerance is essential for those applications. However, applications offering services exclusively designed for leisure can also benefit from fault tolerance as an improved service availability will in turn increase customer satisfaction and financial revenue if for example ads are delivered with the service or optional paid features.

Since the amount of data being processed by an ESP application exceeds the processing power of a single machine, several tens to hundreds of machines are generally needed for carrying out a specific data processing task. However, a high number of machines composing a system increases in turn the probability for a fault that can occur at least at one of those machines compromising the whole task if not properly handled. Hence, fault tolerance is equally important for all applications running on top of distributed systems such as ESP. In the following section, we will review first the failure model and failure detection mechanisms and then provide an overview of commonly used techniques to provide fault tolerance in ESP

systems.

2.2.1 Failure Model and Detection

Prior to the design and implementation of a fault tolerance mechanism in a distributed system, it is important to agree first on the set of failures a system must be able to tolerate and to recover from. In a distributed system, both processes and communication channels may fail, hence, there are three categories of failures that can be found in literature [CD88, HT94]: *Omission failures*, *Arbitrary failures* and *Timing failures*.

Omission failures refer to the case when a process has been stopped or crashed, i.e., does not execute any further or messages are lost which is generally caused by a lack of buffer space in the input buffer of the receiver or broken channels. Other processes may not be able to detect these states correctly as it would require some form of heart beat and timing mechanism where a reply is expected to arrive within a certain interval of time. However, those mechanisms are not applicable to asynchronous systems such as ESP where message delays may be incorrectly interpreted as process halts or crashes.

Contrary to omission failures, **Arbitrary failures or *Byzantine failures*** cover the worst case of failures which can be any type of error that may occur. For example, a process might emit messages with faulty content or might even set wrong values in variables in order to expose an arbitrary behavior.

The last class of failures considers **Timing failures** where processes do not execute or respond to messages within a certain interval of time. Of course this class of failures is only applicable to synchronous systems and therefore will not be considered further here.

The cost of overhead and introduced complexity for handling those various types of failures greatly varies from class to class: For example, handling crash (i.e., omission) failures requires at least $f + 1$ nodes in order to continue a system to operate while an agreement on a single value (i.e., reaching consensus) requires at least $3 \cdot f + 1$ nodes in order to tolerate f failures in the arbitrary failure model. Hence, it is important to choose a failure model that matches the environment and the requirements for the system at hand most.

Fortunately, there exist a set of techniques to transform some types of failures into another one covered by a different failure model with the advantage of broadening the types of failures being handled by one failure class while saving resources at the same time. For example, value failures caused by hardware errors such as bit flips can be detected through encoded processing [WF07] and transformed into process crashes requiring less resources in order to be properly handled rather than in the arbitrary failure model. Moreover, local watchdogs (in hardware or software) can be used in order to crash nodes behaving arbitrarily slow. Those few examples show that the crash-failure model is sufficient for the majority of application scenarios in ESP systems and will be considered throughout the remainder of this work.

Once the failure model has been defined and narrowed down to process crashes, a failure detector is needed in order to trigger recovery and compensation actions in a system. There exist various approaches for the construction of failure detectors such as proposed by [Fet03] in literature, however, in practice simpler solutions such as heart beating are generally used. Examples for state of the art big data systems using simple heart beating (incorporated into Zookeeper [HKJR10]) for failure detection range from Apache Hadoop [Had15] as a representative for MapReduce [DG08] to Apache S4 [NRNK10], Storm [Sto15] and Samza [Sam15] representing open-source ESP systems.

2.2.2 Recovery Guarantees

ESP systems process continuous streams of events where operators are used to filter, extract or accumulate events in order to transform low quality information into high quality information, or to trigger certain actions. During the course of data processing, operators may accumulate state which must be made *resilient* in order to survive system crashes.

One way of making an operator resilient is periodic *checkpointing* of its accumulated state. However, in order to completely mask a failure, "in-flight"-events must be made resilient as well, i.e., may not be lost and must be replay-able in a predefined order in order to generate the exact same output. In-flight events define the set of events which were processed *after* the last checkpoint taken and *prior* to the occurrence of a crash. ESP systems that completely mask failures, i.e., guarantee (i) neither a loss of events nor duplicated processing, (ii) the recovery of operator state and (iii) to reprocess events in the same order regardless of a failure, provide the strongest possible guarantee to its users, i.e., *precise recovery* .

However, not every ESP application requires such strong guarantees in order to continue processing and fulfilling its objective. An alternative, with slightly weaker guarantees provides *rollback recovery* [HBR+05], where neither in-flight events nor state is lost, however, the reprocessing of in-flight events might take a different execution paths after a crash than it would have taken in a failure-free execution, hence, resulting in different conclusions. In contrast to precise and rollback recovery, *gap recovery* does not preserve any events for a replay but operator state, while in *amnesia,* neither the operator state nor in-flight events are preserved so that an operator is always restarted with a fresh virgin state and resumes event processing using the next available input.

2.2.3 Fault Tolerance Approaches

Fault tolerance in ESP systems is mainly achieved through ***replication,*** either of state an operator may maintain or the processing of events itself. Hence, approaches for implementing fault tolerance in ESP systems are either based on one of the two replication mechanisms or a combination of both.

One approach that is solely based on replicated processing is ***active replication***. In active

replication, a second copy of an operator exists (called secondary) processing the same input stream of events as its primary peer as shown in Figure 2.4. Active replication can transparently mask failures as events are being continuously processed by both operator instances producing identical outputs for downstream operators. However, active replication is not trivial to implement and comes with a high resource foot print. First, in order to produce identical results, the computation needs to be strictly deterministic. This can be only achieved if input streams are identical which requires coordination across replicas through an atomic broadcast protocol [CASD95]. However, since coordination among nodes is costly, simpler mechanisms such as a deterministic merge [AS00] can be used, guaranteeing identical input sequences at a much lower overhead. Second, as the computation is replicated, active replication requires at least twice the resources, computational as well as network bandwidth wise which is wasted during the time of a failure-free execution.

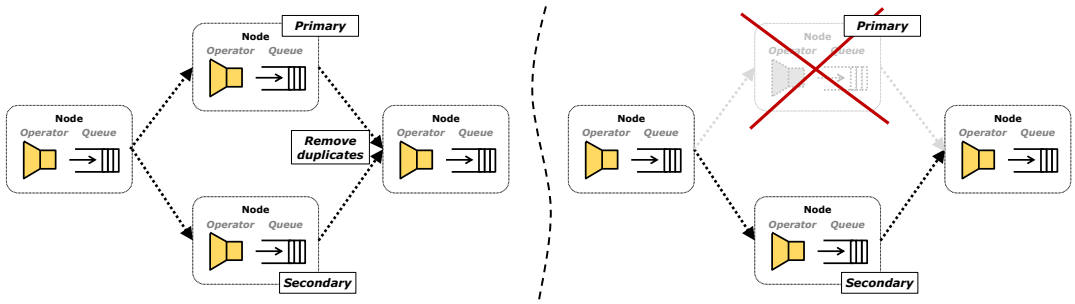

Figure 2.4: Active replication - prior (left) and after (right) a crash.

An alternative to active replication is ***upstream backup*** [HBR+05] where only one operator instance is running and doing event processing at a time. However, in order survive crashes, events must be additionally buffered at upstream operators in order to be replayed to a newly created instance of an operator as depicted in Figure 2.5. Although this approach saves the computational costs for redundant processing, it comes with the disadvantage of long recovery times. During recovery, a new instance of the operator needs to be created first, events buffered at upstream operators must be retransmitted and then reprocessed in order to resume event processing without information loss. Since memory at upstream operators is bounded and reprocessing of events takes a substantial amount of time, this approach is impractical especially for long running operations. Hence, mixed approaches are common where both, replication of state and processing is used.

One of the approaches that makes use of state replication is ***passive replication***. In passive replication, the state of an operator is periodically checkpointed to stable storage that can be either a local disk if the system under consideration only suffers from process crashes where the system only needs to be restarted using a watchdog mechanism after experiencing a crash, or some fault tolerant distributed filesystem (DFS) spanning multiple nodes if hardware failures need to be tolerated as well. In addition to state replication, events are buffered at upstream operators as in upstream backup. However, contrary to upstream backup, the

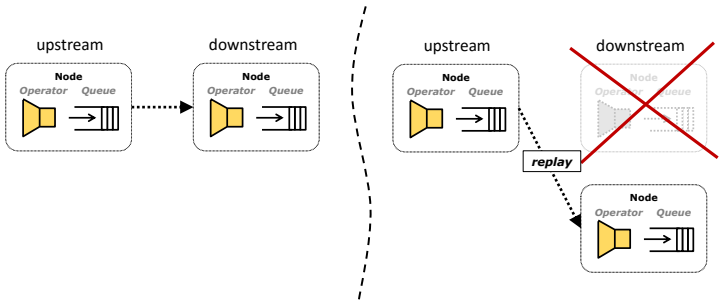

Figure 2.5: Upstream backup - before (left) and after (right) a crash.

buffers at upstream operators are periodically pruned (via an acknowledgment message) with each successful checkpoint in order to keep the buffer small and speeding up the event replay during recovery as shown in Figure 2.6. Although this approach has several advantages over the classical upstream backup, the recovery time can still greatly vary as it depends primarily on the frequency of checkpoints taken and the size of the operator state. For instance, a recovery of an operator with a state size of 4 GB from local disk would require at least 40 seconds under the assumption of a throughput of 100 MB/s for reading a checkpoint from a conventional magnetic disk. Hence, several hybrid approaches have evolved over time with the goal to the lower recovery time.

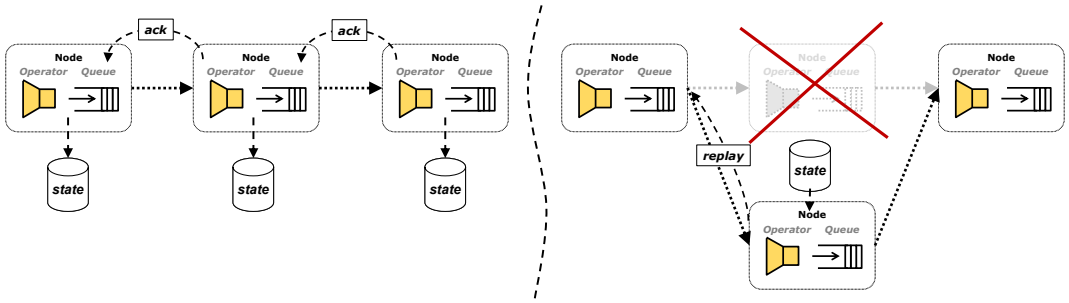

Figure 2.6: Passive replication - before (left) and after (right) a crash.

One of such hybrid approaches is ***passive standby*** where checkpoints are stored at a *suspended* operator replica instead of using an external stable storage. Since such a replica is already equipped with an up-to-date state, only events between the last taken checkpoint and the time the system crash occurred must be pulled and replayed as shown in Figure 2.7.

Contrary to passive standby, the secondary peer in ***active standby*** receives and processes events from upstream operators similar as in active replication, however, output produced by the secondary peer is not being sent downstream saving bandwidth costs as depicted in Figure 2.8. Active standby comes with the advantage of providing a quick fail over as only the connection to the downstream operator must be established in order to resume event

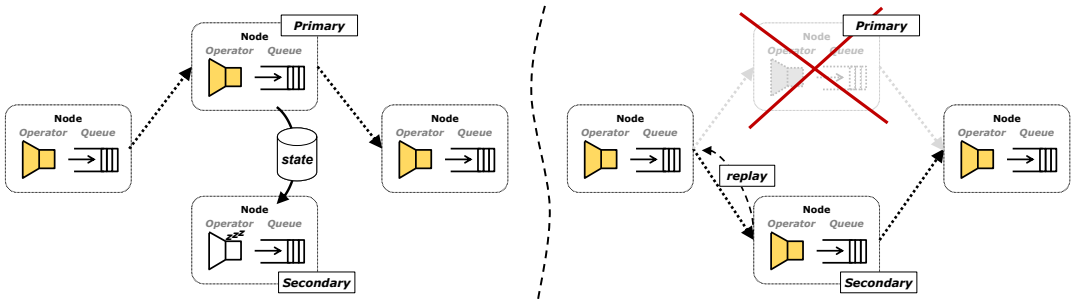

Figure 2.7: Passive standby - before (left) and after (right) a crash.

processing. However, the approach has almost the same costs with regards to processing and network resources as active replication. Hence, the advantages and the associated costs must be carefully considered when choosing an appropriate fault tolerance approach for the ESP application at hand. In Chapter 7, we will provide the reader with a more condensed overview about resource requirements and recovery times for each of these different approaches, and propose a fault tolerance controller that takes the burden from the system user to choose the most appropriate scheme considering resource requirements and recovery times.

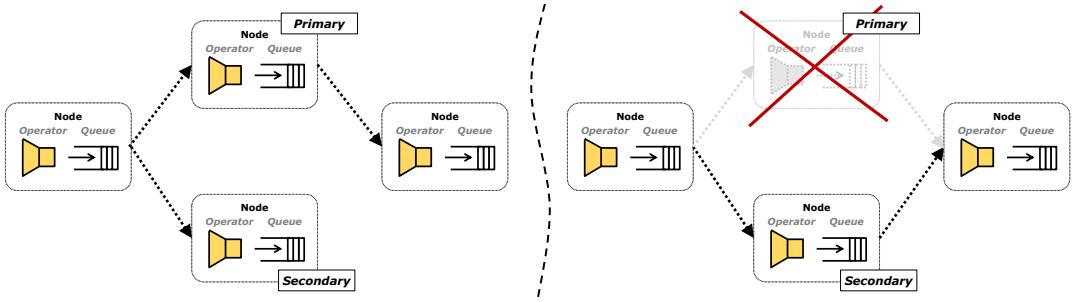

Figure 2.8: Active standby - before (left) and after (right) a crash.

2.3 Summary

In this chapter, we introduced the reader to the basic concepts of ESP and fault tolerance. We first presented our data, operator and query model we will use in the remainder of this work, described how queries are executed and mapped onto ESP systems and defined input and output adapters. We then defined stateful and stateless operators, the different types of windows and concluded with an overview about various standard operators commonly used in ESP systems.

In the second part of the chapter, we first introduced the failure model, failure detection mechanisms and recovery guarantees. We then provided an overview of the different failure tolerance mechanisms and schemes commonly used in ESP systems and discussed the

advantages and disadvantages for each of the presented approaches.

3 StreamMine3G Approach

In this chapter, we will present StreamMine3G, our approach for fault tolerant and elastic event stream processing. We will first present the general architecture of StreamMine3G, introduce the programming model, techniques for parallelization and scalability, fault tolerance and elasticity. We furthermore provide for each subsection an overview of related work comparing the proposed architecture, algorithms and protocols with the state-of-the-art. Finally, we conclude the chapter with an overview about existing open and closed source systems originating from industry and academia and discuss their architecture and shortcomings.

3.1 Overview and Architecture

StreamMine3G is a distributed ESP system designed to run on thousands of machines for carrying out large scale data processing at low latency. The system is composed of a set of nodes forming a StreamMine3G cluster where each node hosts an arbitrary number of operator partitions. In order to cope with varying load, the system allows the addition and removal of nodes at runtime. Furthermore, operator partitions can be *migrated*[1] between nodes in order to balance load and to prevent situations of overload or under-utilization.

Two distinct programming interfaces are offered to StreamMine3G users: An ***operator interface*** that targets data analysts interested in extracting useful information from continuous streams of data, and a ***management interface*** targeting operations of the system such as deploying queries on a StreamMine3G cluster, monitoring the system's health status, acquiring new resources and shifting load within a cluster in situations of over- or underload.

[1] Migration describes the process of moving an operator from one node to another node which includes moving the state for stateful operators as well as redirecting input streams to the new location of the operator.

3.2 Programming Model

In order to analyze streaming data using STREAMMINE3G, data analysts can either choose from a set of provided standard operators or implement their own special crafted operators using user-defined-functions (UDF) by simply sub-classing a well defined operator interface.

3.2.1 Event Format

Events in STREAMMINE3G are provided as simple Byte arrays giving its users a maximum of flexibility in data representation. However, when using custom operators in combination with standard operators, events must follow and use the same predefined data format and serialization mechanisms as used by the standard operators in order to ensure compatibility and a correct processing of events.

Users not tied to specific standard operators or serialization frameworks can choose from a wide variety of techniques and frameworks publicly available for their internal data representation. One of the most popular frameworks in this area is Google Protocol Buffers [Pro15] which not only provides convenient mechanisms for serializing complex data structures but also allows a seamless data exchange across platforms due to its generic binary format. However, the use of such frameworks imposes an additional overhead for event processing which can lower the overall system's performance. In case the data can be represented in a non-nested way using solely flat data structures, and the user opted to implement the business logic using STREAMMINE3G's native C++ interface, simple techniques such as static or dynamic casts as provided in C/C++ can be used instead.

Event Timestamp

In addition to the event itself, users can provide a ***timestamp*** with each event if an ordering of events is desired. Event ordering is crucial for order sensitive applications or applications which require precise recovery to fully mask system crashes. Order sensitivity describes the fact that the output of an operator strongly depends on the order events are being received and processed which is the case when using, e.g., non-commutative operations within an operator.

Timestamps in STREAMMINE3G are represented as 64bit integers which can be either interpreted as physical or logical timestamp by the application developer. However, in order to ensure a correct ordering of events, operators must emit events with strictly monotonically increasing timestamps.

3.2.2 Event Loop/Process Method

When implementing custom operators, user have to subclass from the provided operator interface and overwrite the ***process method***. The process method is the central point of operation when performing data processing in StreamMine3G, as it takes as input an *event* (provided as a Byte array), a *state object* (for implementing stateful operators), and a *collector object* needed for emitting new events to downstream operators.

Listing 3.1 shows an implementation of the classical word count application using StreamMine3G's Java interface. The implementation consists of two operators, a *map* and a *reduce* operator. In the given example, the incoming event (provided as Byte array) is first interpreted and converted into a string object (Line 7). Since the string is representing a sentence, it is split up into chunks using Java's StringTokenizer (Lines 8-13) where each individual word is then emitted as new event using the collector's `emit()` method (Line 12). Note that the word is converted back into a Byte array prior passing it to the `emit()` method using the `getBytes()` method provided by Java's String class.

Program Listing 3.1 Word count map and reduce operator.

```
1  public class WordCountMapper implements Operator
2  {
3      ...
4      /* process method of the map operator*/
5      public void process(..., byte[] event, Object state, Collector collector)
6      {
7          String sentence = new String(event);
8          StringTokenizer st = new StringTokenizer(sentence);
9          while (st.hasMoreElements())
10         {
11             String word = st.nextToken();
12             collector.emitEvent(..., word.getBytes());
13         }
14     }
15     ...
16 }
17
18 public class WordCountReducer implements Operator
19 {
20     ...
21     /* process method of the reduce operator */
22     public void process(..., byte[] event, Object state, Collector collector)
23     {
24         String word = new String(event);
25         Integer count = new Integer(1);
26         Map<String, Integer> map = (Map<String, Integer>)state;
27         if (map.containsKey(word))
28         count = new Integer(map.get(word).intValue()+1);
29         map.put(word, count);
30     }
31     ...
32 }
```

As with the map operator, the reduce operator converts the input event (i.e., the word) in a string first (Line 24). However, contrary to the map operator, the reducer is *stateful* where the state is composed of a hash-map keeping track of the word frequencies. Hence, the generic

state object which was provided as an additional parameter in the `process()` method is first casted to a Java HashMap object (Line 26) in order to update its counters accordingly (Lines 27-29).

3.2.3 Topology

Once the set of operators has been defined, i.e., in our example the map and reduce operator, the flow of events must be specified for carrying out a complete word count application. The flow of events is defined by an *operator topology* which is also often referred as a *query*. An operator topology in STREAMMINE3G is a directed acyclic graph[2] (DAG) specifying the set of upstream and downstream neighbors for each operator. Note that an operator in our system can have multiple upstream as well as multiple downstream neighbors. However, in our given example, the topology for the word count application is quite simple as the map operator has only a single downstream operator, i.e., the reduce operator as shown in Figure 3.1. Using this topology, events produced by the `process()` method of the map operator (using the collector object) will be routed through the network and provided as input events to the `process()` method of the reduce operator.

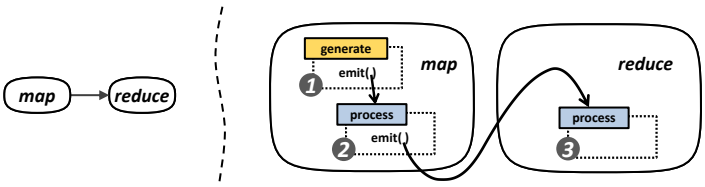

Figure 3.1: Word count topology (left) and event flow (right).

3.2.4 Input/Output Adapter and Event Generator

In order to feed an ESP application with events, some form of a *source operator* is needed for converting data coming from external systems into the application specific event format or, for generating events based on some predefined patterns. Instead of defining an explicit source operator for data conversion and event production, the operator interface offers additional methods and callbacks for reading and writing raw data from TCP and file streams through the callbacks of the `onReadAdapter()`, `generate()` method, respectively. Similar as in the `process()` method, those callbacks take a collector object as input for emitting events. However, those events are provided locally to the first operator defined in the topology rather than routed across the network as shown in Figure 3.1.

Listing 3.2 shows an implementation of the `generate()` method for our word count example application. First, a static string array has been defined with an assignment of random

[2]The architecture of STREAMMINE3G supports also cycles, however, cycles in ESP impose certain challenges with regards to flow control which will not be discussed here.

sentences in the constructor of the map operator (Lines 6-10). The periodically called `generate()` method selects randomly one sentence out of previously defined pool of phrases, i.e., the static array (Line 17) and emits it for consumption at the `process()` method of the map operator using the collector object. Hence, the `generate()` method can be envisioned as the `process()` method of a *virtual source operator*.

Program Listing 3.2 Word count application.

```
1  public class WordCountMapper implements Operator
2  {
3      ...
4      public WordCountMapper()
5      {
6          randomSentences[0] = "the cow jumped over the moon";
7          randomSentences[1] = "an apple a day keeps the doctor away";
8          randomSentences[2] = "four score and seven years ago";
9          randomSentences[3] = "snow white and the seven dwarfs";
10         randomSentences[4] = "i am at two with nature";
11         rand = new Random(System.currentTimeMillis());
12     }
13
14     /* generate method of the map operator of the word count applicatin*/
15     public boolean generate(Collector collector)
16     {
17         String sentence = randomSentences[rand.nextInt(5)];
18         collector.emitEvent(..., sentence.getBytes());
19         return true;
20     }
21     ...
22 }
```

Note that from a software engineering point of view, it is also possible to make the source operator explicit through simple wrapping techniques, i.e., users would create a source operator class and override the `process()` method as in regular operators. The newly defined source operator is then instantiated as an object (class member) inside it's successor operator, i.e., the map operator in our word count example where the `process()` method of the source operator is called with each `generate()` call of the mapper instance by passing a null pointer as input event and state as arguments.

3.3 Parallelization

ESP systems are built to process millions of events per second. However, processing such an amount of data will quickly overload an ESP system if neither partitioning nor parallelization techniques are used. Fortunately, as in ancient Rome, *divide and conquer* is the key to success to cope with this sheer amount of data.

Scalability in StreamMine3G is achieved through (i) the partitioning of input streams and (ii) the parallel execution of operator partitions consuming subsets of the previously partitioned input stream.

3.3.1 Parallelization Degree

Prior to the partitioning and parallelization of operators, it is important to decide for an appropriate parallelization degree. While a high parallelization degree, i.e., a high number of partitions, allows to scale out the system to a large number of nodes (i.e., acquiring more nodes and shifting partitions to those newly acquired resources), it also introduces a non-negligible overhead. Ideally the parallelization degree matches the maximum number of available nodes in a cluster or cloud the ESP system is running on.

The parallelization degree in STREAMMINE3G is specified on a per operator basis rather than on query basis. This comes with the advantage of taking into account operator specific characteristics when deciding for an appropriate degree. For example, a sink operator publishing accumulated results to a single instance of a UI dashboard should not be parallelized at all while a CPU intensive pattern matching operator can greatly benefit from parallelization and partitioned input streams.

3.3.2 Semantic Transparency

A major challenge when partitioning data is to ensure *semantic transparency*, i.e., the results and conclusions drawn from processing events sequentially using a single operator instance must be equivalent to the partitioned execution. For stateless operators, semantic transparency is always guaranteed as the output of an event solely depends on its input. However, for stateful operators, the subsets of events must be carefully chosen in order to achieve semantic transparency. In case the operator is also order sensitive, the input stream from a partitioned upstream operator must be furthermore *merged deterministically* so that the input sequence is equivalent to the production of events of a non-partitioned version of the upstream operator.

Consider the previously presented word count example: In order to be scalable, the map and reduce operator must be partitioned. The partitioning of the map operator is trivial as it is a stateless component with the sole purpose of splitting phrases into individual words. Hence, each instance of the map operator can work on any subset of the provided input stream. In contrast to the map operator, the reducer has to keep track of word frequencies, hence, events with the same routing key (in this case the word itself) must be always routed to the same partition in order to update the frequency counters correctly. Since the addition operation used in the reduce operator is a *commutative* operation, the operator is not order sensitive. Hence, the input streams coming from the partitioned instances of the map operator must not be merged deterministically in order to produce correct results.

3.3.3 Partitioner Interface

The decision to which operator partition an event should be routed to for processing is taken by a *partitioner*. The partitioner takes an input event, parses it and extracts relevant information

for an assignment of the event to an appropriate partition of the downstream operator.

Similar as in Hadoop [DG08, Had15], users can either opt for writing their own custom partitioner in STREAMMINE3G using the provided interface by sub-classing and overriding the appropriate methods, or use the provided default key-range partitioner. When using the key-range partitioner, users have to specify a key range (a 32 bit integer) which is then broken up into n-equally sized chunks based on the previously defined parallelization degree. A routing key (32 bit integer) provided in addition to the timestamp is then used for the evaluation and partition assignment. Since the assignment is based on an extra field (i.e., routing key) rather than the content of the event itself, the approach does not impose any de-serialization and parsing overhead while still being applicable for the majority of applications.

3.4 State Management

STREAMMINE3G was designed for stateful event processing, hence, an operator can access, read and modify state. In order to maximize the developer's flexibility, the state of an operator can comprise any kind of data structure, hence, users are provided with a pointer where any kind of data structure can be bound to, such as linked lists, hash-maps or custom data structures.

In order to free the user from the burden of having to implement their own error prone mechanisms for consistent state modification and persistence, STREAMMINE3G provides its users with an explicit state management interface: The interface consists of callbacks for the initialization of the data structure (`stateInit()`, deallocation of state (`freeState()`) and callbacks for serialization and de-serialization of state (`serializeState()` and `deserializeState()`) needed for state persistence (i.e., checkpointing) and operator migration.

For consistent state modification, STREAMMINE3G uses mutual exclusion mechanisms to ensure that at no point of time two events access and modify concurrently the same portion of state. However, to harness nowadays multi-core systems, events that work on disjoint portions of state are still being processed in parallel.

Similar as with input streams, state is partitioned where one state partition is mapped to exactly one operator partition. Hence, an operator partition can only access and modify its own state residing in local memory.

In addition to the user-defined state, each state partition is equipped with a *timestamp vector* to keep track of the last event that performed a state modification. The timestamp vector is used to filter out duplicates as we will describe more in detail in Section 3.7.

3.5 Management Layer

Contrary to the operator interface which is most predominantly used by data analysts, the management interface provides various mechanisms to fully manage a StreamMine3G cluster which includes the registration of new operators in the system, deploying and removing operator partitions, wiring operators to form topologies (i.e., queries) or moving operator partitions within a cluster in case of over- or underload situations.

In addition to the various actions that can be performed from the management layer, the interface comes also with various callbacks to notify the manager instance about events such as when a new node joins or leaves the system (for example due to a crash of the virtual or physical node a StreamMine3G instance was running on) and various performance related metrics such as CPU, memory and network utilization of all participating nodes in a StreamMine3G cluster including operator partition level metrics such event throughput and state size. The management interface allows therefore the implementation of a manager component that provides a resource and fault tolerant aware operator placement as well as elasticity.

In order to prevent the manager component being the single point of failure in the system, the interface is also equipped with two additional callbacks (`serializeState()` and `deserializeState()`) in a similar fashion as operators for persisting state. However, the manager's state will be persistently stored in Zookeeper [HKJR10] along with StreamMine3G's internal data rather than in a filesystem as done for operators. Although Zookeeper does not scale well with frequent writes, it is sufficient to use for persisting the manager's state as (*i*) it is modified relatively seldom only during system reconfiguration such as when operators must be moved around or new ones are deployed, and (*ii*) the state is relatively small as it comprises only the deployment state of the cluster.

The manager component can be instantiated on any node within a StreamMine3G cluster. If a cluster does not run a manager instance yet, the first active node will be chosen as a host automatically.

In case the node hosting the manager becomes unavailable due to a system or hardware crash, a new manager instance will be automatically deployed in the cluster resuming the operation based on the latest checkpoint stored in Zookeeper. The selection of the node running the manager instance is performed via Zookeeper's lock service which prevents that two nodes acquire manager status at the same time.

3.6 System Architecture

In the following section, we will present several entities and components of the StreamMine3G architecture which are essential for understanding algorithms and protocols employed in the system for carrying out fault tolerance and elasticity.

STREAMMINE3G comprises a set of nodes forming a ***cluster***. Each node in this cluster is a C++ process running on either a virtual or physical machine in user space. A node is uniquely identifiable using a ***nodeName*** which can be any kind of string, however, the hostname of the machine the process is running on is usually used in order to easily locate the host machine within a data-center or cloud infrastructure. In addition to the nodeName, each node receives a unique ***nodeId*** (integer) upon its launch and registration with Zookeeper. NodeIds allow the system to identify a restart/relaunch of a process using the same nodeName.

In order to carry out data processing in the system, users provide the functionality in form of ***operators*** by implementing the `process()` method of the provided operator interface. Operators are compiled as dynamic shared libraries which are only loaded upon instantiation of an operator. Similar as with nodes, an operator is uniquely identifiable using an ***operatorName*** where the choice for the name is entirely left to the user. In addition to the operatorName, a unique ***operatorId*** is assigned to an operator during registration with Zookeeper.

Prior registering an operator at the system, users have to specify the parallelization degree for the operator at hand which will determine the number of partitions used for processing data with this operator later on. ***Operator partitions*** are identified using ***sliceIds*** where 0 represents the first partition and $\pi - 1$ the last one based on the parallelization degree of π.

Operator partitions can be instantiated (i.e., deployed) on any node and at any point of time in the system. An instance of an operator partition is called a ***slice*** and receives a ***sliceUId*** for unique identification within the system. A slice holds a pointer to the concrete instance of the user's operator code for calling the `process()` method upon reception of a new event.

In order to efficiently manage slices within a STREAMMINE3G node, the following hierarchy of entities exists: A node comprises a set of *operator containers* where each container hosts a set of slices.

3.6.1 Cluster Configuration and Management

STREAMMINE3G uses Zookeeper to store its cluster configuration in a reliable way. In order to run separate STREAMMINE3G clusters using the same Zookeeper instance, different namespaces (i.e., different root nodes in Zookeeper) can be specified.

For a cluster configuration, four separate trees are maintained in Zookeeper: 1) nodes, 2) operators, 3) wiring and 4) slice deployment.

Node records contain information such as the host and port a STREAMMINE3G process is listening on in order to establish TCP connections for data exchange while operator records provide essential information about the registered operators such as the parallelization degree and the location of the shared library file. The wiring tree specifies how operators are wired to form topologies. For this, each record contains the name of the specific operator and two sets with names for their upstream and downstream neighbors. The last record type maintains

the information on what node a slice (i.e., an instance of an operator partition) is currently running on using a quadruple consisting of *nodeName*, *operatorName*, *sliceId* and *sliceUId*.

3.6.2 Network Communication and Event Dissemination

In order to ship data between slices based on the configured operator wiring, i.e., topology, each slice maintains a ***routing table*** with a list of its downstream slices. In addition to the routing table for disseminating events to downstream slices, a slice is equipped with an additional table listing all its upstream slices. The ***upstream table*** is needed for mechanisms such as event ordering and duplicate detection as we will explain in Section 3.6.4.

The two tables are updated by the manager instance whenever a new slice is being deployed or removed from the cluster. Or, in situations where a *topology is being altered* through the removal or addition of operators serving either as new downstream or upstream neighbors. Both tables maintain their records as quadruples consisting of the nodeName, operatorName, sliceId and sliceUId.

With each slice deployment or removal, the manager instance has to update (i) the routing table of all upstream slices, (ii) the upstream and routing table of the (new) slice being deployed (or removed), and (iii) the upstream table of all downstream slices. In order to do so, the manager instance computes the set of all communication partners based on the operator wiring and slice deployment to correctly update the tables for all affected slices as shown in Listing 3.3.

Program Listing 3.3 Routing and upstream table update algorithm for new slices.

```
1: function WIREOPERATOR(newSlice)
2:     for each operator in upstreamOperatorsOf(newSlice.opName) do
3:         for each sliceUp in deploymentOfOperator(operator) do
4:             updateRoutingTable(sliceUp.node, sliceUp, newSlice)
5:             updateUpstreamTable(newSlice.node, sliceUp, newSlice)
6:     for each operator in downstreamOperatorsOf(newSlice.opName) do
7:         for each sliceDn in deploymentOfOperator(operator) do
8:             updateRoutingTable(newSlice.node, newSlice, sliceDn)
9:             updateUpstreamTable(sliceDn.node, newSlice, sliceDn)
```

In a first step, the set of upstream operators for the new slice is retrieved (Line 2). Next, we iterate through the deployed slices for each of those upstream operators (Line 3) and update the routing table of the upstream slices (Line 4) and the upstream table of the new slice (Line 5). In an analogue way, the set of downstream operators is retrieved (Line 6) in order to update the routing table of the new slices (Line 8) and upstream table of the downstream slice (Line 9) for every downstream operator and slice.

For slice removal, the same algorithm is applied, however, the update will now perform a removal of the entries from the routing and upstream tables rather than the creation of them.

The use of such routing tables does not only allow us to deploy and remove slices at runtime but also to alter topologies while the system is running. Hence, new links between operators can be established or removed at any point of time.

Listing 3.4 depicts the steps taken for updating tables for two newly linked operators by computing the cross product of all affected slices through the iteration over all slices deployed on upstream as well downstream side.

Program Listing 3.4 Routing and upstream table update algorithm for topology changes.

```
1: function WIREOPERATOR(operatorUpstream, operatorDownstream)
2:     for each sliceUp in deploymentOfOperator(operatorUpstream) do
3:         for each sliceDn in deploymentOfOperator(operatorDownstream) do
4:             updateRoutingTable(sliceUp.node, sliceUp, sliceDn)
5:             updateUpstreamTable(sliceDn.node, sliceUp, sliceDn)
```

In addition to upstream and routing tables, ***peer slice tables*** are maintained for each slice. Peer slice tables are updated along with routing and upstream tables and contain only records for partitions of the same operator including replicas that presently exist in the cluster. The table is used to establish communication between replicas in order to perform actions such as checkpointing and storing operator's state in memory of replica slices rather than in files, and for synchronization protocols among operator partitions and replicas as we will present in Chapter 5. Checkpointing state to replicas is used for slice migration and fault tolerance as we will describe more in detail in Sections 3.7 and 3.8.

STREAMMINE3G uses *lazy connections*, i.e., connections to peer nodes are only established if nodes must exchange data based on the information stored in routing, upstream and peer slices tables. In case previously established connections are not in use any more since slices have been removed, they are torn down along with tables updates.

Although the routing table allows in principle to send events to any of the downstream slices based on the topology's definition, it may not be confused with the actual routing of events which is controlled by the *partitioner*. The partitioner is the sole instance to decide to which downstream partition (using the sliceId as identifier) an event should go to whereas the routing table is just used for locating recipient slices within the network in order to ensure a correct delivery of events.

3.6.3 Network Batching

ESP systems are built to process high velocity data at low latency. However, achieving high throughput and low latency at the same time is a challenging task: Low latency data processing requires the system to send data items immediately which introduces a considerable overhead as events are relatively small comprising often only as little as a KB of data. This results in a high amount of system calls consuming a considerable amount of CPU time which lowers the computing power for CPU intensive operators such as joins.

An alternative to the instant transmission of events is ***event batching*** where several events are consolidated into a packet prior to the transmission to its destination. Although event batching reduces the amount of system calls when sending and receiving data packets, it also introduces latency if (i) the batch size was chosen too big, or if (ii) events are produced at low frequency due to the nature of the application or workload.

When implementing event batching, two different approaches can be considered to consolidate events: Events can be either consolidated based on their common sliceUId, i.e., ***slice-level-batching*** or common destination node, i.e., ***node-level-batching***. The first approach comes with the advantage that only a single packet header with the sliceUId of the destination slice is needed while the second approach requires individual headers for every event. However, node-level-batching allows the implementation of publish-subscribe systems [EFGK03] on top of STREAMMINE3G using an efficient broadcast mechanism where no redundant information is transmitted as we have shown in [BHM+14].

Since STREAMMINE3G allows the adjustment of a batch size at runtime, latency can be kept at considerable low levels also when experiencing workloads with fluctuating throughputs using a controller-based approach.

Event batching is executed at the operator container level where each container maintains the set of slices for the specific operator with access to the routing tables to determine the downstream slices in order to insert events into the correct batch for transmission. Hence, events originating from different slices but the same operator container may be merged into the same batch in FIFO order.

Micro Benchmarks

In order to evaluate the benefit of event batching, we performed micro benchmarks where we varied the batch size while measuring the latency and throughput. For the evaluation, we used a simple application consisting of three operators: A *source* operator generating events, a *filter* operator forwarding events that match a certain condition and a *sink*.

The latency we measured is the *end-to-end* latency, i.e., when the event was generated at the source operator until it arrives at the sink. The batch size is specified in Bytes and defines the threshold for the amount of data to be accumulated in a batch prior to its transmission. The size of an event used in the benchmark comprises approximately 150 Bytes including 100 Bytes used as payload by the application and the remaining Bytes for header information. Hence, a batch size of 300 Bytes accommodates approximately two events.

The outcome of the benchmark is depicted in Figure 3.2. As shown in the plot, the latency decreases first until the batch size reaches a size of approximately 5 KB, stays then almost constant at around $180ms$ and increases then quickly for sizes larger than 1 MB. In contrast to the latency that follows the shape of the bathtub, throughput increases constantly until it stagnates at batch sizes of 24 KB or more.

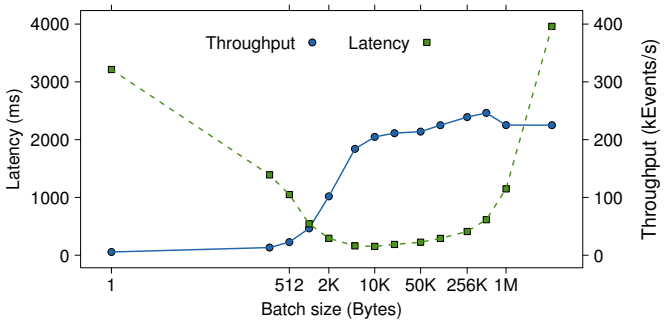

Figure 3.2: Throughput and latency with varying batch sizes.

As expected, throughput increases with larger batches as the amount of system calls is reduced. On the other hand, latency also increases for batch sizes greater than 256 KB as events spend more time in batches until they are full and being sent off to downstream operators. Interestingly, latency is still very high for small batch sizes such as 1 KB and less. We account this bathtub effect to lock contention on the receiver side as a significantly higher amount of system calls and small packets must be handled and processed appropriately.

As the results show, a batch size of approximately 10 KB provides already a good trade off between low latency and high throughput while smaller or larger batches have a negative impact on either throughput and/or latency, respectively.

3.6.4 Event Ordering

In order to achieve scalable data processing, ESP systems such as StreamMine3G are carried out as distributed systems where several nodes cooperatively perform data processing at large scale. Since data in those systems is exchanged via TCP/IP where packets may be delayed for various reasons, the order of arrival of such packets originating from different nodes is neither predictable nor deterministic. Furthermore, in order to fully harness the available processing capacity of nowadays multi-core systems, data processing must be performed using multiple threads. However, thread scheduling performed at operating system level is also non-deterministic which introduces another level of unpredictability when processing continuous streams of data.

Although some applications can tolerate disorder in event streams such as a simple stateless filter, others which may employ order sensitive operators require an ordered event stream in order to produce correct results. Moreover, when it comes to fault tolerance, an ordered event stream allows to perform a *precise recovery* without duplicates as event production is repeatable under the assumption of a deterministic behavior of the user-provided operator code. Hence, event ordering is required (i) to support sensitive operators and (ii) to provide fault tolerance with strong guarantees such as precise recovery.

STREAMMINE3G follows the *partition-merge* model as presented in Section 2.1.4 where the output stream of an operator is first partitioned (using the default or the user-provided partitioner) and then merged downstream with peer-streams prior to the consumption by an operator. Hence, event ordering is achieved by applying the *merge function* f_m which takes events originating from a set of streams as input and produces a single event stream with a well defined order, i.e., a *total order*.

The input of the *merge function* f_m for an operator o_d is a set of channels C which comprises the output streams of the set of partitioned upstream operators. Hence, the set C can be specified as follows: $C = \{s^1_{o_1}, \ldots, s^{\pi^1}_{o_1}, s^1_{o_2}, \ldots, s^{\pi^2}_{o_2}, \ldots, s^1_{o_k}, \ldots, s^{\pi^k}_{o_k}\}$ where $s^j_{o_i}$ is an output stream, i.e., a channel delivering events produced by the j^{th} partition of operator o_i with partition degree π_i. The set of operators contributing to this set of channels comprise all upstream operators of operator o_d as specified by the user's operator wiring, i.e., the topology. Furthermore, the set of channels C is ordered using the pair (*operatorId*, *sliceId*) associated with each stream coming from the upstream operator partitions.

Events in a channel c are ordered using *strictly monotonically increasing* timestamps, hence, $\tau_1 < \tau_2$ where $\tau_1 \epsilon e_1 \wedge \tau_2 \epsilon e_2$ and event e_1 arriving prior to e_2.

The output of the merge function is a single stream s_i of events following a total order as defined by the function.

In case the output of the merge function is always repeatable, i.e., deterministic, we call the merge a *deterministic merge* following the definition provided by [KR05]:

Definition 1. ***Deterministic Merge*** *For events e_1 and e_2, and operators j and k that satisfy: Operators j and k are downstream operators of operator u that produces e_1 and e_2, hence, both e_1 and e_2 are received at j and k the following condition is satisfied: The order in which e_1 and e_2 are delivered at j and k is the same, and e_1 and e_2 are eventually delivered.*

Since the input sequence for both operators j and k is identical and can be reproduced through the deterministic merge, the input is *deterministic*. We therefore define a *deterministic input* as follows:

Definition 2. ***Deterministic/Repeatable Input*** *For events e_1 and e_2, and operators j and k, the order in which e_1 and e_2 are provided as input sequence to j and k is always the same, regardless if the input has been replayed or produced the first time.*

We furthermore define an operator as deterministic if it always produces the same output sequence for the same (deterministic) input sequence. The output sequence of such an operator is then deterministic. We therefore define a *deterministic output* as follows:

Definition 3. ***Deterministic Output*** *An operator j produces a deterministic output if it satisfies the following conditions: (i) the operator is deterministic and (ii) the provided input is deterministic.*

Round-Robin Deterministic Merge

An ordering of events using the *merge function* f_m can be enforced in several ways: One possibility is to disregard the timestamps associated with each event and taking one event from each channel at a time in a *round-robin* fashion as depicted in Figure 3.3. Since the channels are ordered, the traversal of the round robin scheme is deterministic producing a reproducible output stream.

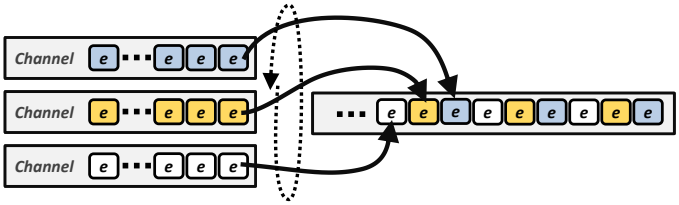

Figure 3.3: Round-robin merge.

Although this scheme is simple to implement, it requires a balanced workload where all channels deliver events at roughly the same data rate. For unbalanced workloads, an enqueueing of events at channels with higher data rates may introduce an undesirable latency.

A *weighted round-robin* scheme where more than a single event is consumed from high-data-rate-channels at each turn can mitigate this problem, however, it requires that the workload, i.e., the data rates for each of the channels do not change over time. For changing data rates, the weights must be adjusted at runtime which requires a *deterministic reconfiguration*, i.e., the point in time when the reconfiguration happened must be recorded including the full set of parameters (in this case the weights for each channel) to ensure a deterministic replay which is not trivial to implement.

Listing 3.5 depicts a simplified version for the weighted round-robin merge algorithm in pseudo code. The round-robin scheme is expressed through the usage of a *foreach*-loop as shown in Line 2 where we iterate over the *sorted set* of upstream channels. For every channel, we process as many events as specified by the channel's weight through the use of another nested for-loop as shown at Line 3. An additional check at Line 4 ensures that the channel we are currently consuming events from has still sufficient events in its buffer available. If not, we simply wait for more events to arrive. Note that the wait in StreamMine3G is implemented by saving the current state of the iteration and returning the currently active thread for the processing of other tasks. Once new data is available, the operation resumes the processing of events at the exact same position it was paused previously.

Timestamp Based Deterministic Merge

An alternative to the previously described round-robin scheme is the use of timestamps associated with each event. The key idea here is to always proceed with the event carrying

Program Listing 3.5 Round-robin deterministic merge.

```
1: while true do
2:     for each channel in upstreamChannels do
3:         for i=0; i<channel.weight; i++ do
4:             if channel.hasMore() ≠ true then channel.waitForMore()
5:             event ←channel.poll()
6:             PROCESS(event,state)
```

the *lowest timestamp* provided by any of the channels as shown in Figure 3.4. Contrary to the round-robin scheme where the order of merging events is predefined through the weights associated with each channel, the order in this scheme is predefined by the event's timestamp.

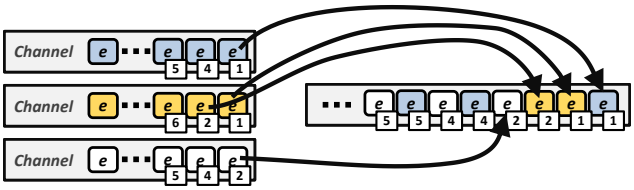

Figure 3.4: Timestamp-based merge.

The advantage of considering timestamps rather than channels and associated weights for event ordering is the implicit and dynamic adjustment to unbalanced and changing workloads. However, this requires the use of *synchronized clocks* such as wall-clock time for timestamps assigned to events at upstream operators where channels that produce temporarily less frequent data items will produce a sequence of events with larger gaps in timestamps than channels with high data rates.

The disadvantage of the timestamp-based approach is that at any point of time each channel must be filled with at least one event in order to make progress. Considering a channel which does not produce an event for some time t due to the nature of application, this can considerable increase processing latency as no progress can be made during this time until the slow channel provides an event. A solution to keep latency at a predefined low level is the use of ***silence propagation*** (also commonly referred to ***punctuations*** [DMRH04, LMT+05] in literature) where slow channels emit a NOOP event after a previously defined timeout in order to let the *merge function* f_m at downstream operators make progress. Note that NOOP events are discarded at downstream operators and serve solely for the purpose of ensuring progress.

Silence propagation in STREAMMINE3G can be easily implemented at operator level by utilizing the `generate()` method which is triggered periodically based on a user-specified time interval. The interval determines in this case an upper bound for the processing latency introduced by slow channels.

Implementation Details

Since ESP systems are naturally exposed to fluctuating workloads as they often consume data from real-time data sources such as Twitter fire hose, the timestamp-based approach has been proven successful for the majority of applications running on top of StreamMine3G.

In StreamMine3G, the *merge function* f_m is implemented as a separate component named *sequencer* which is part of the slice abstraction holding a reference to the user-provided operator code. The sequencer consists of a *priority queue* with upstream channels as its entries. The channels are carried out as simple FIFO queues where the timestamp of the tail element is used to discard duplicated events due to the existence of replica slices of upstream operators. The timestamp of each channel's head element is used to determine the order of channels within the priority queue. As previously mentioned, the event with the lowest timestamp is processed first, hence, the next event to process is always the head element of the channel which is the first element in the priority queue as channels are sorted by their head element's timestamps. Since a poll of the head element changes the channel's timestamp, as now the event which was previously second moved to the head, a reordering of the priority queue is needed.

Priority queues can be implemented in several ways, either by using heaps or self-balancing binary trees as underlying data structure. Since updating the head element is the most frequent operation performed on the queue rather than insertions and removal of elements, a heap-based implementation serves for the priority queue in StreamMine3G which exposes the worst case complexity of $\log(n)$ for updates to the head element. Note that an insertion or removal of an element translates to a change in topology which occurs relatively seldom.

Listing 3.6 depicts the pseudo code for the timestamp-based deterministic merge algorithm. Since the ordering of channels is based on the timestamps of each channel's head element, the algorithm checks first if each channel holds at least one event in its buffer to perform the sort. If the pre-condition is not met, we simply wait for more data (Lines 1-2) prior adding the channels to the priority queue for sorting (Line 3).

Program Listing 3.6 Timestamp-based deterministic merge.

```
1:  for each channel in upstreamChannels do
2:      if channel.hasMore() ≠ true then channel.waitForMore()
3:  priorityQueue.addAll(upstreamChannels)
4:  while true do
5:      channel ←priorityQueue.top()
6:      event ←channel.poll()
7:      PROCESS(event,state)
8:      for each channel in upstreamChannels do
9:          if channel.hasMore()≠ true then channel.waitForMore()
10:     priorityQueue.adjustTop()
```

The timestamp-based merge is reflected in the algorithm by always selecting the head element

of the priority queue, i.e, the channel with the lowest timestamp (Line 5) and adjusting the head element afterwards (Line 10) which triggers a reordering of the priority queue based on the new timestamp of the former head element of the priority queue. The head element, i.e, the event of the previously selected channel is then passed to the user-provided `process()` method as shown in Lines 6-7. Prior to the adjustment of the priority queue, an additional check is performed ensuring that at least one event resides in each channel's buffer again (Lines 8-9).

3.6.5 Memory Management

Although nowadays ESP systems are classified as big data systems as they process high volumes of data, the actual size of such data items is rather small, comprising often only as little as a few hundreds of Bytes in average. Moreover, since ESP systems operate on moving data, the lifetime of such data items is also relatively short compared to other distributed systems such as key-value stores. Hence, ESP systems need to manage efficiently huge amounts of small as well as short-lived data items.

Allocating memory in an operating system is generally a costly operation, especially if the allocated chunk is only used for a short amount of time as it is the case in ESP systems. In a naïve ESP system implementation, memory is requested for every incoming event and free-ed once the event has been processed and becomes obsolete. While this approach is the natural way of building systems, it imposes huge and unnecessary overheads in ESP systems as small memory chunks are frequently allocated and immediately free-ed again.

Two different mechanisms are used in STREAMMINE3G in order to reduce the memory footprint: (*i*) The utilization of network batching and (*ii*) the reuse of previously allocated memory chunks.

While network batching not only improves performance due to the lower amount of system calls required to send or receive data as presented in Section 3.6.3, they also reduce the number of system calls needed for memory allocations when receiving data packets. Instead of having to allocate small chunks of buffers for each arriving event, a single larger memory chunk comprising a few KBs will now serve multiple events at once.

Network Batching

In order to avoid costly calls to `memcpy()`, simple pointer arithmetic is used in STREAMMINE3G in order to pass the individual events sharing the same network-batch-buffer to the `process()` call of the user-provided code. In a similar fashion, multiple events are consolidated into a network batch using a `writev()` call rather than using a sequence of costly `write()` or `memcpy()` calls.

Memory Reuse

Since events in ESP systems have a relatively short lifetime while not varying much in size, the overhead of memory allocation can be greatly reduced by reusing previously allocated memory chunks. Instead of implementing our own algorithms for reusing memory chunks, we use the mature implementation of `tcmalloc` [TcM15] which intercepts systems calls such as `malloc()` and `free()` in order to reduce the number of memory allocations executed by the operating system's kernel. In addition to memory reuse, `tcmalloc` provides also thread-level caching of memory chunks reducing contention in multi-threaded systems such as StreamMine3G. In our experiments, we saw a 19% increase in throughput in average when linking StreamMine3G with `tcmalloc`.

Zero-Copy Message Passing

StreamMine3G allows the co-location of several slices in the same node in order to save resources in case of low system utilization. This can lead to situations where two consecutive operators of a topology share the same host. In case the first operator simply forwards events such as when using filters, StreamMine3G performs a zero-copy message passing instead of creating a new copy of the forwarded event using a `writev()` call. Zero-copy message passing is implemented in StreamMine3G using a similar technique as provided with smart pointers of the `c++11` standard where a reference counter is maintained tracking the bindings of the memory region.

Zero-Copy State Management

In a similar fashion as when forwarding events, StreamMine3G passes pointers of the user-managed operator state directly to a `writev()` call when performing checkpointing of state for persistence. In case the data structure used for the operator's state is already flat in memory as authors propose in [KBG08], checkpointing can be performed without any extra costs of `memcpy()`.

3.6.6 Non-blocking I/O and Flow Control

Despite the fact that ESP systems perform most of the operations in memory, they are also in need of exchanging data with peer nodes which involves I/O operations on the network stack. Since I/O operations are generally long running tasks, we opted to use asynchronous network operations as it allows us to overlap these long running I/O operations with processing tasks resulting in an overall better performance and system utilization.

StreamMine3G employs the asynchronous model by utilizing the `boost.io` [Boo15b] library which provides a convenient abstraction of the operating system's low level primitives for asynchronous operations on network streams and files. Concept wise, the framework provides

an io-service as central entity which is essentially a thread pool. Processing tasks can be explicitly scheduled, i.e., submitted to the io-service by passing a function pointer for future execution. In case an asynchronous operation completes such as reading a data packet from the network stack, the completion handler (i.e., a function pointer) of this operation is automatically scheduled for execution by the io-service.

Flow Control

Employing an asynchronous communication model in ESP systems imposes several challenges: First, ESP systems operate on constantly moving data where flow control is essential in order to prevent an overflowing of event queues at overloaded operators. Second, asynchronous communication using multiple threads imposes the risk of processing data packets in a different order rather than their original arrival order at the node.

Flow control in STREAMMINE3G is carried out by using a combination of asynchronous and synchronous communication patterns in order to establish *back-pressure* in overload situations: While data packets in STREAMMINE3G are read in an asynchronous fashion, sending of data packets is performed in a synchronous way. This pattern creates implicit back-pressure as blocking write calls occupy threads in the io-service which would be otherwise used to accept new incoming data originating from upstream operators. Hence, back-pressure is implicitly propagated up to the source operator in a topology. Alternatively, back-pressure can also be implemented explicitly by monitoring the number of outstanding write requests and adjusting the read request rate accordingly.

In order to provide a tune-able back-pressure mechanism, event reception and processing are handled as separate tasks: The first task takes data packets from the wire and inserts the transmitted events into the processing queues of slices while the processing task consumes events from those queues and applies the user-provided `proces()` method as depicted in Figure 3.5. Since both tasks are scheduled by the same io-service instance, back-pressure can be adjusted by changing the amount of enqueued processing tasks in the io-service. A higher number of processing tasks translates to a higher weight for processing rather than event reception which results in mostly empty event queues while a low number of processing tasks will cause the opposite situation in which more data is being accepted from upstream nodes rather than processed which causes an increase of items waiting in event queues. Hence, the ratio of event processing tasks to event reception tasks determines the degree of back-pressure enforced by an operator.

Reading data in an asynchronous fashion can introduce disorder depending on the order read requests are issued and data is being processed. For example, issuing multiple read requests prior to the return of the first completion handler can lead to a situation where the first packet is erroneously handled by the completion handler of the second request. Hence, to preserve the FIFO order provided by the TCP channel, read requests may only be issued after the return of the previous one. However, this can still introduce disorder if no explicit

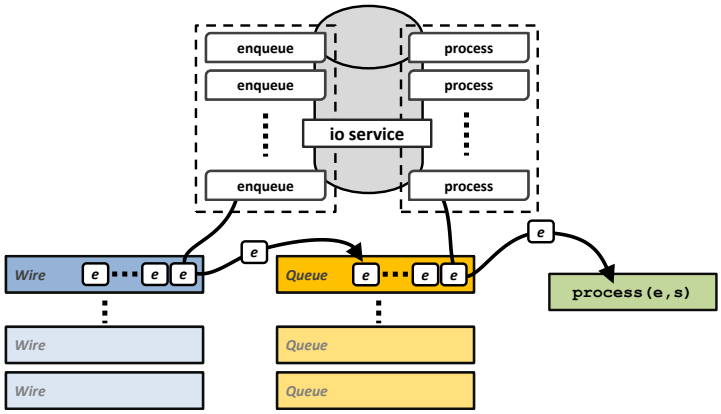

Figure 3.5: IO service and separation of tasks for event reception (enqueue) and processing.

locking scheme is used as a second packet may be processed (i.e., inserting events in queues) concurrently with the first one. STREAMMINE3G supports both schemes: *read-queue-read* and *read-read-enqueue,* boths with explicit locking for preserving TCP's FIFO order.

3.6.7 Techniques for Improving Scalability

Although applications with partition-able state can conceptually fully scale within the two dimensions (horizontal and vertical), real world applications often exhibit a non-optimal behavior that limits scalability. For example, a poorly chosen partition key may overload certain operator partitions while leaving others underutilized. Similarly, if long sequences of events need to access the same state partition, concurrent access protection will limit the processing close to a single threaded execution.

In the following, we describe mechanisms employed in STREAMMINE3G that improve scalability and performance for real world applications that may exhibit such a counter productive behavior such as when performing pattern detection in a click stream analysis application, where certain patterns are likely to be much more frequent than others.

Network Batch Delegation

In order to improve throughput, events in STREAMMINE3G are sent in batches to downstream operator slices (as described in Section 3.6.3). However, some operators may naturally emit longer sequences of events to a single downstream slice. This effect results in a higher contention on that specific network batch and limits parallelism as threads are blocked until the network batch is fully sent. To avoid such a blocking, we introduce *delegation* for network batches: if a thread is currently busy with sending a complete batch downstream other threads can, in a non-blocking fashion, append ready-to-send events to a list which will then later be passed on to the thread currently in charge of sending events on that channel. Appending ready-to-send events to such lists (note that a separate list exists for each downstream

channel), frees the previously blocked thread for the processing of events for idle downstream channels. The delegation algorithm is depicted in Listing 3.7.

Program Listing 3.7 Network batch delegation.

```
 1: function EMITEVENT(event)
 2:     mutexBatch.lock()
 3:     batch.add(event)
 4:     if batch.size() > threshold then
 5:         batchesToSend.add(batch)
 6:         batch← new batch
 7:         if mutexSendChannel.trylock() then
 8:             cntr ← 0
 9:             while batchesToSend ≠ empty do
10:                 if cntr < procSize then break
11:                 mutexBatch.unlock()
12:                 SENDBATCH(batchesToSend.pop())
13:                 mutexBatch.lock()
14:                 cntr ← cntr + 1
15:             mutexSendChannel.unlock()
16:         else if batchesToSend.size() > maxBPend then
17:             mutexBatch.unlock()
18:             mutexSendChannel.lock()
19:             mutexBatch.lock()
20:             for each batch in batchesToSend do
21:                 SENDBATCH(batch)
22:             mutexSendChannel.unlock()
23:             batchesToSend.clear()
24:     mutexBatch.unlock()
```

Every time an event is emitted by an operator, the event is first passed to the partitioner in order to determine the target partition of the downstream operator. STREAMMINE3G maintains a so called downstream channel object for each target partition. A downstream channel object consists of a queue where events are appended to during emission, and a list where those batches of events are appended once they are full and ready for transmission.

In a first step, the event is appended to the batch for the previously selected downstream channel (Line 3). Since STREAMMINE3G allows the hosting of several slices in a single node/process, appends to the same batch are protected through a mutex (Line 2). With every append, the size of the batch is checked (Line 4). If the size exceeds a dynamically adjustable *threshold*, the batch is marked as full and appended to the list of full batches in order to be shipped over the wire later on (Line 5). Note that the adjustable *threshold* parameter allows us to flush events immediately in order to reduce latency if desired.

Once a batch is full and has been appended to the list of full batches, a new empty batch is being created (Line 6), and the currently active thread attempts to acquire the sender mutex

lock (Line 7) for the channel in order to send the batch downstream. If the acquisition was successful, i.e., no other thread is currently busy with putting complete batches for the selected channel onto the wire, the thread iterates over the list of full batches and puts the full batches one-by-one onto the wire (Lines 9-15). Since a batch comprises multiple events, we use a single `writev()` call to reduce the number of system calls. Prior to this process, the mutex for appending new events to the previously created batch has been released (Line 11) in order to allow previously competing threads to append new events to the batch in a non-blocking fashion which increases substantially throughput as we will show in our micro benchmarks at the end of this section.

In case the system experiences an imbalance in the workload for a longer period of time, the above algorithm would exhibit the following behavior: (i) the list of full batches would grow with time, hence, increase latency and memory consumption, and (ii) the thread that entered the loop for sending the full batches first will not leave the processing loop until the imbalance dissolved.

In order to address the first issue, the algorithm in Listing 3.7 contains an implicit back pressure mechanisms which prevents an infinite growth of the full batches list. The growth of the list is bounded by the dynamically adjustable parameter *maxBPend* (Line 16) which specifies the maximum amount of pending batches to be sent over the wire. The parameter can be adjusted during runtime in order to postpone back pressure on upstream operators if desired. In case the limit has been exceeded, back pressure is enforced by keeping the batch mutex (Line 19) locked while sending the accumulated full batches (Lines 20 and 21). Note that batches are sent over the wire in a synchronous fashion, hence, the send call is blocking.

In order to allow a thread to leave the processing loop while sending batches, we limited the amount of batches a thread is allowed to send with each turn by using a simple counter mechanisms and a *procSize* limit variable (Lines 8, 10 and 14).

State Access Delegation

In situations where the state for an operator partition is further partition-able such as when using hash-maps for storing some kind of information, *state accessing delegation* can be used in order to improve the scalability of the system. State access delegation prevents the queuing of threads if multiple consecutive events require access to the same portion of state, hence, the processing of such events is serialized in order to ensure a consistent state modification. However, this leads to a sequential execution of events although recently enqueued events that require access to disjoint parts of the state could be processed in parallel. Inheriting the idea for network batch delegation, we also implemented and evaluated the benefit of state access delegation by extending the operator interface with a `stateAccess()` method that shares the semantics of a partitioner.

Micro Benchmarks

In the following, we present the results of micro benchmarks we performed in order to evaluate the benefit of network and state access delegation in the context of ESP systems. We consider both horizontal and vertical scalability.

For the evaluation, we implemented a click stream analysis application that subscribes to a stream of Apache HTTPd log entries from a cluster of web servers using piped logging [Apa15]. The application consists of the three pipelined operators: The first operator groups http requests by user-ids (through session-id cookies). The following operator keeps a sliding window of the requests per user-id. With each update to the sliding window, an output is produced representing the identified pattern. The last operator is a *top-k* which continuously tracks the frequency of the identified patterns. The frequency can then be used to trigger actions that consider the most popular access patterns in a site (or set of sites, if technologies such as third-party cookies are used for tracking).

Our experiments were executed on a 40-node cluster where each node is equipped with 2 Intel Xeon E5405 (quad core) CPUs and 8 GB of RAM running a Debian Linux 7.4 operating system with kernel 3.2.0. All nodes are connected via Gigabit Ethernet (1000BaseT full duplex). The click stream analysis application has been implemented in C++ using STREAMMINE3G's native operator interface.

Horizontal Scalability For the first experiment, we continuously added nodes for the pattern detection and *top-k* operator and increased the workload imposed on the application. The number of nodes where source operator partitions are deployed for accepting log entries from web servers is kept constant. The aggregated throughput was measured at the pattern detection operator.

As depicted in Figure 3.6, STREAMMINE3G scales almost linearly with the increasing workload and added computing resources. Note that the delegation mechanism for network batches considerably improves throughput. Each STREAMMINE3G node is processing around 160k events per second. Through the back-pressure mechanism in our delegation mechanism, we keep the amount of pending network batches constant, hence, imposing an upper bound on end-to-end latency.

Vertical Scalability In the next experiment, we control the size of the thread pool to evaluate the throughput as the number of used cores increases. Figure 3.7 depicts the result of this experiment. Again, the system performance increases with the amount of available threads, i.e., the number of used cores in a node. However, if delegated state access is disabled, the throughput will decrease when more than six threads are being used. This decrease is due to the high contention on the state which gets worse with an increasing number of threads. Although delegated state access improves throughput, the increase is not linearly as with

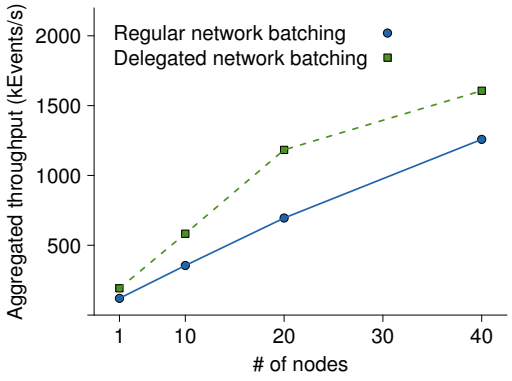

Figure 3.6: Horizontal scaling - aggregated throughput with increasing number of nodes.

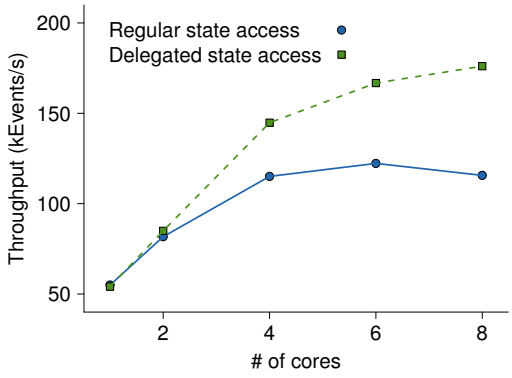

Figure 3.7: Vertical scaling - per node throughput with increasing number of threads.

horizontal scaling. This is due to the usage of a single event queue that exists for each operator slice. The queue is needed for ordering events if the user opted for deterministic execution. However, throughput can still be improved through a more fine grained partitioning, i.e., the usage of more slices for an operator by raising the partitioning degree.

3.7 Fault Tolerance

Achieving fault tolerance in ESP systems is a challenging task as there is constantly moving data that must be continuously filtered, aggregated and transformed in order to extract useful information in near real-time. Replication is the key mechanism used in ESP systems to mask system failures where either a combination of checkpointing and logging can be used in order to provide fault tolerance based on *passive replication*, or simply the replication of operators which is commonly known as *active replication*.

While passive replication is primarily based on the replication of data such as the events (though event logging) and accumulated operator state (though checkpointing), active replication utilizes the replication of the actual activity, i.e., *redundant processing of events*. Hence, ESP systems must provide mechanisms for replicating data such as event logging and checkpointing as well as replicating operators to tolerate faults in one or another way.

In the following, we will outline how the different mechanisms for fault tolerance are employed in the StreamMine3G system.

3.7.1 Event Logging

Event logging describes the mechanism of keeping a copy of events produced and sent downstream in an additional buffer. The objective of this buffer (i.e., the event log) is to allow a

retransmission of events (without the need of regenerating/reprocessing them) whenever a downstream operator is in need of those events due to a recovery from a recently experienced crash of a node. In order to achieve a reasonable performance, the buffer is solely kept in main memory, however, periodically persisted and written to disk with every checkpoint taken in order to tolerate multiple failures at once.

In theory, it is possible to provide fault tolerance solely based on event logging which is also known as *upstream backup* in literature [HBR+05]. However, since ESP systems process high velocity data, such event logs would grow quickly and exhaust the node's memory. Hence, regular log pruning is needed in order to keep the logs at reasonable sizes. Furthermore, recovering solely from logged events can take a considerable amount of time especially for long running applications.

In order to keep event logs small, *acknowledgment* messages are used where downstream operators inform their upstream peers about the set of events that can be considered as obsolete and pruned from logs. During recovery, downstream operators send *replay* messages upstream to request a retransmission of previously logged events.

Events can be considered as obsolete if either their information has been made externally visible, or incorporated into the downstream's operator state that is already on stable storage through checkpointing. Hence, acknowledgment messages are either sent after a successful completion of a checkpoint or if events became externally visible through some sink operator.

Since operators are required to produce and emit events with strictly monotonically increasing timestamps, a simple *less than relation* can be used in order to define the set of events that should be pruned or replayed at upstream event logs. Hence, acknowledgment and replay messages solely carry a timestamp and the sender's identification information in form of a tuple such as (ts, operatorId, sliceId).

Implementation wise, an event log can be carried out as a simple FIFO queue which can be installed at the level of slice abstraction. However, this has the disadvantage that events must go through the partitioner during event replay again which introduces additional overhead and slows down the overall recovery process. Therefore, event logs in STREAMMINE3G are installed at the operator container level where they are maintained in a partitioned way using a two-dimensional hash-map. The two-dimensional map uses the operators as a first and the partitions of those (using the sliceIds as key) as a second dimension. The event log itself is split up again using a one-dimensional map where the sliceUId is used as key and the FIFO queue as value for maintaining the association of events and their slices they are originating from as depicted in Figure 3.8 which is needed for persisting event logs as described in Section 3.7.2. The head of the queues contain the event with the smallest timestamp which is the oldest one while at the tail, new events are continuously appended.

Although this design seems to introduce redundancy as the same event is maintained multiple times if more than a single downstream operator exist, the approach requires only negligible

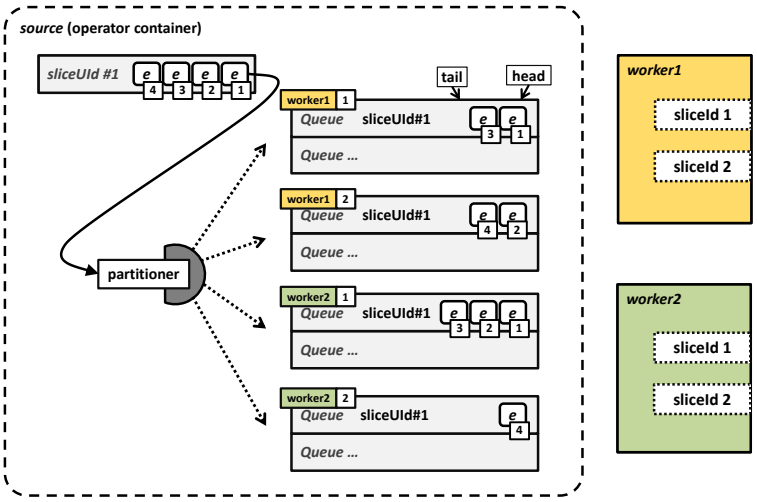

Figure 3.8: Event log.

more space as only pointers to the payload are maintained and not a full copy of the event. Hence, this approach trades space for processing cycles of the partitioner.

Event Pruning

The objective of log pruning is to free memory by purging outdated information. However, determining what piece of information can be considered as outdated or not, is not a trivial task as an operator in an ESP system can have multiple downstream operators which externalize events and perform checkpointing in an uncoordinated fashion at different points in time. Hence, an event *e* might be obsolete for one operator while for another one the event is still needed in order to survive future crashes.

In order to determine correctly the set of obsolete events, we need to track the acknowledgments and their timestamps received from downstream slices. For tracking, we do not distinguish between individual slices but consider only the *maximum timestamp* if multiple replicas for one operator partition exist. The reason for this simplification is that events become automatically obsolete if one of the replicas externalized information or successfully completed a checkpoint. While this not only reduces space in event logs, it comes also with the advantage that replicas can recover with a much more up-to-date state although the slice that had crashed might have been slightly behind originally.

Using the acknowledgments, we can determine (theoretically) the set of events to purge from the queues of the event log by computing the minimum timestamp delivered through the acknowledgment messages as depicted in Figure 3.9. However, in the actual implementation, events in the individual queues are purged with every arrival of an acknowledgment. Since events are maintained only as pointers referencing the payload as described in Section 3.6.5, the actual event is transparently removed from memory if no references exist anymore which

is equivalent to the minimum computation of the timestamps.

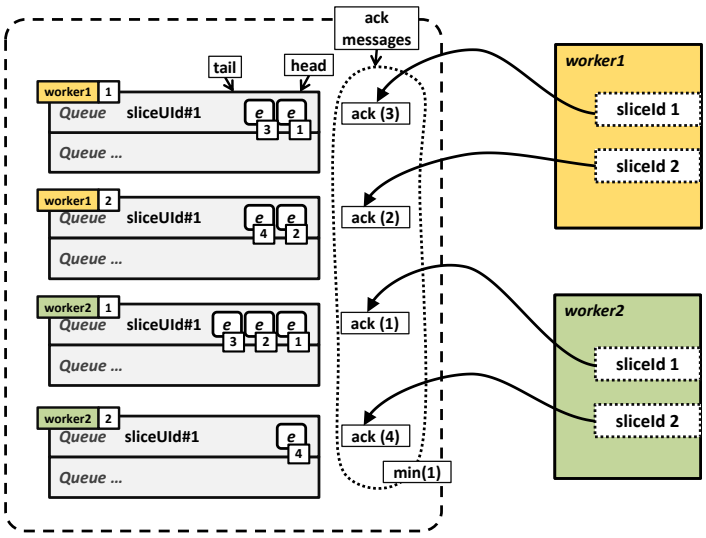

Figure 3.9: Event log (pruning).

Event Replay

While event pruning only serves for the purpose of preventing the system from collapsing due to memory exhaustion, replaying events is one of the key mechanisms in ESP systems to recover from crashes without loosing vital information. The relatively complex data structure with hierarchically managed queues as presented in the previous section greatly simplifies the retransmission of events during a recovery process as we will outline in the following.

As mentioned in Section 3.1, a node in STREAMMINE3G can host an arbitrary number of slices. Therefore, a crash due to a hardware or software failure will immediately lead to the unavailability of several slices at once. Since the manager instance is transparently informed about lost nodes and slices, new backups can be quickly deployed on spare nodes within a STREAMMINE3G cluster to initiate a recovery procedure. The recovery procedure comprises essentially two steps: (*i*) loading the most recent checkpoint available for the operator partition in recovery, and (*ii*) replaying events from upstream partitions prior resuming normal operations.

Since slices are aware of their upstream neighbors due to the existence of the upstream table as described in Section 3.6.2, slices can request the retransmission of events based on the state's timestamp vector. In situations where two or more replica slices for an upstream operator partition exist, the replay request is sent only to one of those replicas. Due to the hierarchically managed queues, the replay request can be immediately served by simply constructing a packet using the pre-partitioned events from the queue and a `writev()` call for delivering the events directly to the slice that requested the replay.

ESP systems process naturally an unbounded and continuous flow of events. Unfortunately, the flow of events does not stop during a crash or a recovery. Therefore, a slice in recovery mode may receive two different types of events at the same time: new events as well as replayed ones due to the retransmission. In order to distinguish the two types of events, events are equipped with an additional flag in their header/metadata to indicate their type.

The type information in the event can now be used to ensure a correct order of processing events regardless of their simultaneous arrival. In order to achieve this, the system enqueues events originating from upstream logs immediately into the ***merge queue*** (i.e., the sequencer) for ordering and further processing whereas new events are put into a separate ***on-hold queue*** for later enqueuing in the merge queue as depicted in Figure 3.10. However, such a design could lead to a situation where the event log and the on-hold queue buffer the same set of events in the end. In order to prevent such a situation, we extend the replay protocol with an additional *replay-stop* message which is sent upstream with the timestamp of the on-hold queues's head element as a marker to indicate where to stop the replay.

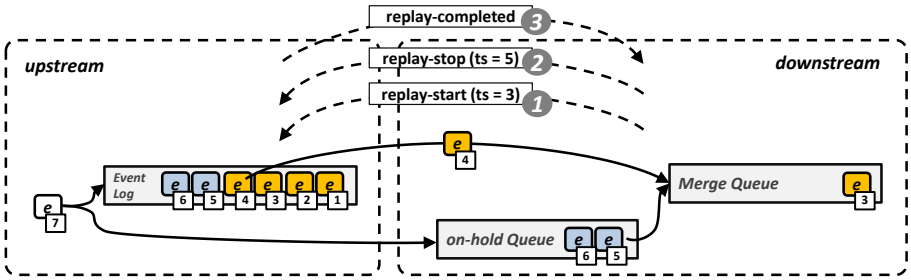

Figure 3.10: Event log (order preserving replay).

An example execution of the protocol is shown in Figure 3.10: Let's assume a slice has been deployed for recovery. After loading the most recent available checkpoint, the timestamp vector associated with the state partition indicates that the last seen event was $ts = 2$, therefore, a *replay-start* ① message with $ts = 3$ is sent upstream to initiate a replay of events with timestamps $ts \geq 3$. Since the slice is set in recovery mode, events produced by upstream slices in the meantime are enqueued into the on-hold queue rather than directly into the merge queue while at the same time, replayed events go straight into the merge queue. Once the first new event arrived downstream which is the event with $ts = 5$ in the example, the downstream slice sends a *replay-stop* ② message upstream to indicate that only events with timestamps up to $ts < 5$ are needed for a replay. After sending the last replayed/logged event, the upstream slice notifies the downstream slice that no more events are coming (*replay-completed* ③) which marks the end of the recovery process where in a final step all events from the on-hold queue are appended to the merge queue at once in an atomic fashion.

Listing 3.8 shows the implementation in pseudo code of the previously described protocol. The listing contains two methods, ONMESSAGEFROMUPSTREAM() and ONMESSAGEFROMDOWNSTREAM(). While the first one (Lines 1-15) is called whenever a message arrives from an

upstream node, the second one (Lines 16-27) handles messages received from downstream peers.

Program Listing 3.8 Event replay protocol.

```
1: function onMessageFromUpstream(data)
2:     if data.type =EVENT then
3:         if slice.state =RECOVERY then
4:             if data.event.header =REPLAY then
5:                 mergeQueue.append(data.event)
6:             else
7:                 onHoldQueue.append(data.event)
8:                 if onHoldQueue.size = 1 then
9:                     sendUpstream(REPLAYSTOP, onHoldQueue.head.ts)
10:        else
11:            mergeQueue.append(data.event)
12:    else
13:        if data.controlMessage =REPLAYCOMPLETED then
14:            mergeQueue.appendAll(onHoldQueue.getAll())
15:            slice.state ←NORMAL
16: function onMessageFromDownstream(data)
17:    if data.type =CONTROLMESSAGE then
18:        if data.controlMessage =REPLAYSTART then
19:            stopper ← ∞
20:            for each event in eventLog do
21:                if event.ts<stopper then
22:                    sendDownstream(event)
23:                else
24:                    sendDownstream(REPLAYCOMPLETED)
25:                    break;
26:        if data.controlMessage =REPLAYSTOP then
27:            stopper ← data.controlMessage.ts
```

In a first step when executing the onMessageFromUpstream() method, the type of the message is evaluated in order to determine if it is either an event or a control message that has been received from upstream. In case it is an event, and the slice is currently in recovery mode (Lines 3-9), events are either appended to the mergeQueue (Line 5) or the onHoldQueue (Line 7) depending on their type, i.e., a replayed event or not. If the slice is operating normally, events always go straight to the mergeQueue (Line 11). During the first append of an event to the onHoldQueue (Line 8), a *REPLAYSTOP* message containing the timestamp of the head element is sent upstream notifying the sender when to stop.

In case the message received is a *REPLAYCOMPLETED* notification (Line 13), all events of the onHoldQueue are appended atomically to the mergeQueue (Line 14) and the slice is set to normal operation mode (Line 15).

On upstream side, only control messages are handled (Line 17) as events in ESP systems flow only in one direction, i.e., from upstream to downstream. In case the message requests a replay, i.e., a *REPLAYSTART* message, first a stopper variable is set to infinity (Line 19) as the timestamp for stopping the replay has not yet been reported from downstream side. In the following, events are replayed and sent downstream (Line 22), as long as their timestamps are still less than the one defined by the stopper (Line 21).

As soon as a *REPLAYSTOP* notification arrives (Line 26), the stopper variable is updated (Line 27) appropriately which will break the for-loop in Lines 20-25 once the timestamps of the logged events are larger than the stopper one. As a final step, a *REPLAYCOMPLETED* message is sent downstream.

3.7.2 Checkpointing

The majority of operators in ESP systems accumulate some form of state over time. Examples for those stateful operators are joins, aggregations and pattern matching. Recreating the state of those operators by simply replaying events from event logs is often difficult or even not possible: While the state of operators working on short windows can be easily recreated, a recreation of state for longer windows comprising several hours or even days is simply impossible. Hence, mechanisms are required to persist operator state for such types of applications.

One way of achieving state persistence is the use of checkpointing where a snapshot of the state is taken periodically and persistently stored on some stable storage. During recovery, the most recent checkpoint is loaded to resume operation at the point of time the checkpoint was taken before.

Overhead of Checkpointing

Checkpointing is often used in combination with event logging as it allows a gap-less recovery without any information loss. Therefore, the recovery process naturally consists of two phases, loading the checkpoint and replaying events that have been produced since the last checkpoint. The replay time during a recovery depends on the checkpointing interval. The more frequently checkpoints are taken, fewer events must be replayed reducing the overall recovery time. However, taking checkpoints introduces a non-negligible overhead. First, the state must be locked in order to prevent concurrent modifications during serialization to ensure consistency. Second, serialization can take a considerable amount of time depending on the complexity of the data structure and the serialization framework used. After serialization, the state snapshot must be either written to some stable storage which can be a local disk or a (fault tolerant) distributed filesystem, or sent to a peer node to keep a copy of the state in memory. Sending the checkpoint to peer nodes or using a distributed filesystem can temporarily saturate the network especially if the serialized state comprises hundreds of MBs. Hence, there is a trade-off

in overhead introduced by checkpointing and recovery time.

Checkpointing Scheduling

In STREAMMINE3G, checkpointing can be either performed automatically by the system or manually upon user interaction through the manager interface. However, in either case, checkpoints are only taken at ***epoch boundaries***. An epoch boundary is a marker in an event stream which specifies a point in time where a certain action such as taking a checkpoint can be performed. Epoch boundaries are automatically created by the system based on the user-defined ***epoch length*** which is essentially a tumbling window. The epoch length also specifies the minimum checkpointing interval as no checkpoints are taken outside of epoch boundaries. However, it is also possible for the system not to perform a checkpoint when passing an epoch boundary, hence, the checkpointing is prolongated by a multiple of the epoch length then. Note that the epoch length can also be set to one which basically allows the system to perform checkpointing at any point of time and not at fixed intervals, however, as mentioned previously, too frequent checkpointing can substantially degrade system performance.

Checkpoint Components

The objective of a checkpoint is to allow a recovery of accumulated operator state. However, persisting solely the operator state is not sufficient to mask a failure completely, i.e., allow a gapless recovery since information such as what was the last event that modified the checkpointed state or even the event log itself, needed for replaying in-flight events, is lost which would either imply a partial information loss or duplicated processing. Hence, it is important to decide carefully what information is essential and must be included in a checkpoint and what can be safely omitted as it can be easily recreated.

Whenever an operator performs a checkpoint, an acknowledgment message is sent to upstream peers in order to notify those that the previously sent information is included in the state and will never be requested again for a replay in future. However, this means in turn that the operator sending such an acknowledgment message may not be able to reproduce previously sent events anymore. Hence, the checkpoint must not only include the operator state but also the event log in order to serve downstream peers with such information from the past. Note that the checkpoint of a slice only preserves events the slice produced by itself. We therefore introduced an additional dimension in the hierarchical managed event log where the sliceUId is used as identifier to maintain the association of events to its origins, i.e., the producer slices.

In addition to the event log, the checkpoint must also include the timestamp vector associated with the state in order to preserve the information about the set of events which modified the state last. This information is essential to correctly discard past and obsolete events that may occur during a replay from the event log. In summary, a checkpoint in STREAMMINE3G

consists essentially of the following three components: The operator state, the timestamp vector and the event log.

Since checkpoints comprise important meta information such as the timestamp vector and the event log, checkpointing cannot be simply omitted for stateless operators although they do not hold any application specific state that must be preserved. However, checkpointing also triggers the generation of acknowledgment messages to notify upstream operators about when to prune events from the logs which prevents the system from memory exhaustion.

Checkpoint Recycling

Since there are several checkpoints taken during the lifetime of an ESP application, it is also important to recycle and purge obsolete snapshots in order to free resources for other tasks. StreamMine3G performs checkpoint recycling transparently whenever a new checkpoint has been successfully created. The recycling procedure starts by scanning the folder of the local or distributed filesystem for previous checkpoints. Previous checkpoints can be easily identified through their filenames as they carry the maximum timestamp of the state's timestamp vector in addition to the regular identifiers such as operatorName and sliceId. After identifying obsolete checkpoints, they are transparently removed from the filesystem.

3.7.3 Replicated Processing

In order to replicate the processing of an event stream, at least two slices for an operator partition must be deployed on a StreamMine3G cluster. In order to tolerate a system crash, those replicas must be placed on different (virtual or physical) nodes within a cluster using appropriate operator placement strategies.

Using replicas imposes several challenges: First, events must be duplicated and routed to both replicas. Second, the replicas must receive events in identical order in order to produce identical results (as with state machine replication), so that, third, the downstream operators can reliably filter out duplicates produced by the upstream replicas in order to ensure exactly once processing semantics.

The replication of events for consumption at replicated downstream slices is done transparently through the use of the routing table as described in Section 3.6.2. In fact, not the event itself is replicated but a pointer to the event is appended to the appropriate network batches for the various recipients which causes an implicit replication of the events when writing event batches on the network through a `writev()` call.

The routing table can be envisioned as a simple topic-based publish-subscribe mechanism where the slices subscribe to events published by their upstream peers based on the current operator wiring in the system, i.e., user's specified topology. This mechanism ensures that all replicas receive the same set of events regardless of their current location within a clus-

ter. Note that contrary to a real publish-subscribe system, direct routes are established in STREAMMINE3G between event producers and consumers without the use of any brokers in between.

In order to produce identical results which is essential for filtering out duplicated events downstream, replicas must receive events in identical order. Although STREAMMINE3G uses TCP for data exchange between nodes, the FIFO ordering guarantee offered by TCP is not sufficient as a slice may receive events from several upstream slices running on different nodes. Hence, events originating from different nodes must be *merged deterministically* so that the resulting input sequence is identical for all replicas. For replicated processing, a deterministic merge such as described in Section 3.6.4 is required in order to ensure correctness regardless of order-sensitivity of the downstream operators.

Since replicated operators produce duplicated results, a mechanism is needed to filter out those redundant information. Duplicate detection and filtering is performed through the timestamp vector associated with the state of each operator partition. Since events are emitted with strictly monotonically increasing timestamps, the timestamp vector can be used to reliably decide which events must be discarded and which ones must be still considered for processing.

3.7.4 Related Work

In this section, we will briefly present related work to the algorithms and protocols presented previously.

Event logging has been first proposed by Hwang et al. [HBR+05] as a method for providing fault tolerance in ESP systems. In their work, the authors propose to log events at upstream nodes (*upstream backup*) and a two level acknowledgement protocol in order to prune event logs. However, the approach lacks of a support for stateful operators. Hence, operator state can only be recreated by replaying a potentially infinite number of events depending on the window size of the operator.

An approach that combines event logging and state persistence is the *sweeping checkpointing* approach presented by authors in [GZY+09]. Similar as in our approach, precise recovery is provided in case the operator exposes a deterministic behavior. The objective of their work is to decrease the checkpoint size by keeping the output queues included in the checkpoint short by checkpointing immediately after log trimming. In our approach, however, we allow uncoordinated checkpointing due to the use of deterministic execution which ensures replay-ability of events. In uncoordinated checkpointing, operators may take state snapshots independently of others at any point of time as we assume operator topologies where multiple different upstream operators with diverse characteristics may contribute to the input stream of a downstream operator. We furthermore provide appropriate data structures used to efficiently handle event logging, pruning and replay for those complex topologies.

Several approaches has been proposed in order to reduce the overhead of state checkpointing: Authors in [KMR+13] present an approach for rollback recovery without using checkpoints. In order to recreate state, events that contributed to a state modification are logged using safe points which essentially form a dependency graphs. State recovery is then performed by replaying those logged events. Although this approach reduces the overhead of checkpointing and the amount of events required to be logged for state recreation, the approach requires a more verbose operator API where the underlying ESP system is informed about events that performed a state modification.

Another approach that relies on context information of the operator such as the window type and size is the work presented by [SM11]. The objective of the approach is to lower the overhead of state persistence by checkpointing individual keys in a rolling fashion upon window closures rather than taking a complete state snapshot for an operator at once. However, such an approach is not applicable to our programming model as no such context information is provided in StreamMine3G.

Since arbitrarily designed data structures can be used for keeping state in operators in StreamMine3G, users are required to provide appropriate code for serializing and de-serializing those data structures for state persistence. The overhead imposed through serialization can be reduced by utilizing data structures which do not require any serialization as they are already flat in memory as proposed by authors in [KBG08]. By simply returning the pointer of the starting address of the flat data structure, the state can be checkpointed using the previously described `writev()` mechanism without having to perform costly `memcpy()` operations as presented in Section 3.6.5 of this chapter.

An alternative to rollback recovery using checkpoints and in-memory logging is active replication. Authors in [SHB04] introduce two new operators (*F-Prod* and *F-Cons*) to replicate the processing of data. While the *F-Prod* operator replicates events and ensures that all replicas receive the same set of events as input, the *F-Cons* operator removes duplicates and enforces event ordering. The FLUX approach is very similar to our approach, however, in our system, the functionality of those two operators is hidden from the system user while in the FLUX system, those operators are explicit.

The active replication approach has been also explored in the context of data processing in wide area networks as authors have shown in [HCZ07]. However, the approach does not provide strong consistency as we do we with our StreamMine3G approach. Moreover, window punctuation/epochs are used which is similar to the approach we are presenting in Chapter 4 and 5.

3.8 Elasticity

Since ESP systems provide low latency data processing enabling its users to trigger actions in near real-time, such systems are often used to process data originating from live data sources

such as Twitter fire hose, log data, or streams such as Google AdWords for real-time bidding. However, the data rate of such sources often fluctuates over the course of a day where at peak times the throughput can increase by several orders of magnitude. In order to cope with those situations, the system must be provisioned with sufficient capacity for those peaks in order to prevent an overload of the system resulting in service degradation. However, using a static provisioning as shown in Figure 3.11 leads to idle resources for most of the times imposing unnecessary costs to the system user, or even to overload situations in case the system was under-provisioned. Hence, a dynamic provisioning is desired where the amount of available resources follows the system's actual demand.

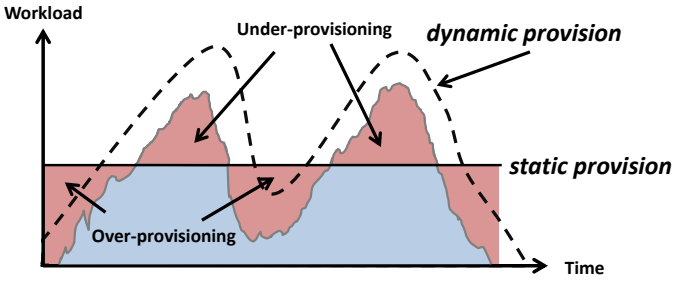

Figure 3.11: Static vs. dynamic resource provisioning.

Cloud computing is a new paradigm enabling companies to acquire resources on demand which is a cost efficient alternative to the hosting of dedicated data-centers. The new paradigm allows its users to acquire resources such as virtual machines almost at an instant and releasing them if they are not in use anymore. Cloud users are billed on a pay-as-you-go basis without high up-front investments such as needed when operating an own data-center.

Although cloud computing allows the quick addition of new resources, it is equally important that the system running on top of such an infrastructure is also capable of using such newly acquired resources as well as releasing resources if not needed anymore.

While most of the big data systems are designed to be scalable, where scalability translates to the property of being capable of handling more data if more resources are available, only few systems are also cloud enabled, i.e., elastic where resources can be added or removed during runtime of the system without the need of a reconfiguration and restart. STREAMMINE3G was designed from ground up to be a fully elastic system to operate in cloud environments such as Amazon EC2. Hence, it allows the addition and removal of resources during runtime without manual user interaction or disruption of service.

Elasticity in big data systems is carried out through one key mechanism: migration. Migration describes the process of moving data or an ongoing operation from one host to another in order to balance load in situations of overload, or to free resources by co-locating several less utilized operations on a single node.

Implementing elasticity in ESP systems is not trivial as ESP systems operate on constantly

moving data. Moreover, ESP systems often use stateful operators which cannot be simply restarted on the new host. Hence, the migration mechanism in an ESP system consists of several actions such as (i) redirecting the stream of events and (ii) moving the state of an operator while guaranteeing that during this process neither events are lost nor processed twice.

Elasticity in StreamMine3G is carried out by utilizing several mechanisms which are also used for fault tolerance such as active replication and checkpointing. We will now present in detail how operator migration is carried in our system.

In StreamMine3G two different kinds of migration mechanisms exists: An interruption free variant where at no point in time the stream of events must be stopped, however, which requires deterministic execution, i.e., event ordering, and an alternative variant which does not require strict event ordering, however, involves a short pausing of the system in order to prevent duplicates to occur in the system.

3.8.1 Interruption Free Migration

Figure 3.12 depicts the steps involved in an interruption free migration: Lets consider the situation where a node becomes over- or underloaded and the system decides to migrate a slice to a new host ①. In order to start such a migration, a replica is deployed onto the system first ②. In case the migration was triggered due to a detected overload, the new replica would be deployed on a spare node to evenly balance the load across multiple nodes, or in the opposite case, co-located with other slices in order to free a node for releasing it to the resource pool. The newly instantiated slice receives already events but does not perform any processing yet as its state is still in virgin state. Prior to a state synchronization where the primary slice takes a snapshot of the operator's state and sends it to its new replica (i.e, the secondary), the state of both queues must be examined first in order to ensure a migration without any information loss.

A slice migration is usually triggered in situations of over- or underload. In case of an overload, a node often does not provide sufficient resources to process events as quickly as they arrive. This leads to the situation where the merge queue at the primary will quickly grow. However, a newly deployed replica will only receive recently produced events from upstream slices unless the upstream peers are instructed to replay those events from the log the primary is holding in its merge queue. Therefore, the state synchronization must be *postponed* until the moment where the timestamp of the head of the primary's merge queue (ts_{prim}) is larger than the head of the secondary one (ts_{sec}). In other words, the primary must be ahead of time to the secondary. Only if this condition is met, the secondary can resume event processing with (i) the state copy received from the primary and (ii) the events enqueued in the merge queue since the birth of the slice in the system.

The timestamps of the primary and secondary's merge queues are exchanged periodically until

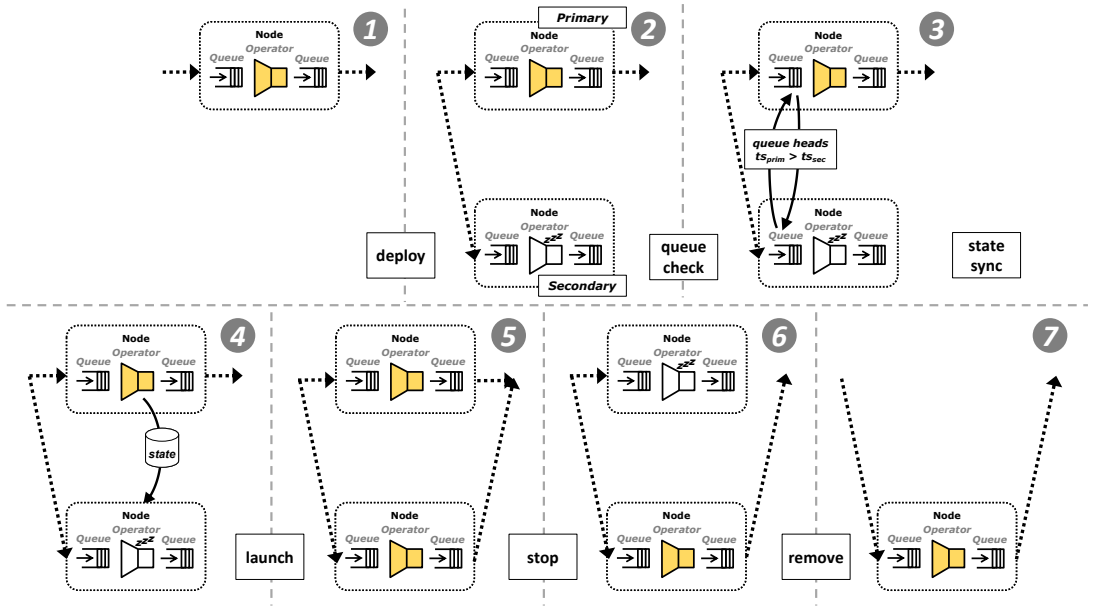

Figure 3.12: Interruption free migration (with postponed state synchronization).

the condition to proceed is met ($ts_{prim} > ts_{sec}$) ③. If so, the slices are synchronized where the operator state of the primary is checkpointed and sent to the secondary in order to bring its state up-to-date ④. The updated secondary slice may now start processing events ⑤ which will cause that two slices process concurrently the same set of events. Since events are now processed redundantly, a deterministic merge is required to reliably filter out duplicates at downstream nodes. After the launch of the secondary which will in fact turn it into a primary, the old primary is stopped first ⑥ and finally removed ⑦, leaving only a single slice running on the new host in the system now.

As mentioned previously, an alternative implementation of the migration protocol is possible which uses event replay instead of a postponed state synchronization to ensure a gap-less migration. The protocol is shown in Figure 3.13 where instead of the queue timestamp exchange phase the state synchronization is performed immediately ③. Since the state is equipped with a timestamp vector, an event replay ④ as described in Section 3.7.1 can be initiated in order to receive all events from upstream since the state snapshot/checkpoint was taken at the primary.

The advantage of the alternative implementation is that an already overloaded primary does not have to continue processing events until the event in the queue has been reached which is also available at the secondary. Instead, the state synchronization can be executed immediately followed by an event replay at the secondary which allows an early shutdown of the overloaded primary. Although this approach allows a much quicker transition, it requires event logging to be enabled in the system which might not be always the case especially for users not interested in fault tolerance.

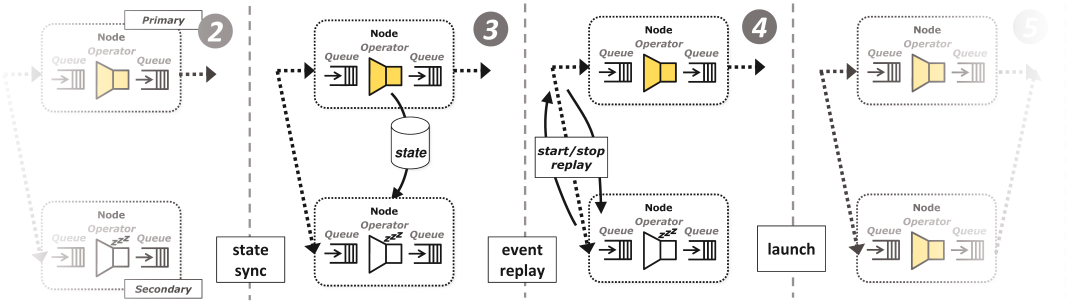

Figure 3.13: Interruption free migration (with event replay).

3.8.2 Legacy Migration

Although the previously introduced migration protocol allows a seamless and pause-free migration of operator slices, it requires a deterministic merge of events which introduces a non-negligible overhead as we will show in Chapter 4. However, there are classes of applications that use solely non-order-sensitive operators such as filters and do no require fault tolerance mechanisms. Those applications do not necessarily require an ordered stream of events, however, would still benefit from load balancing techniques such as operator migration. In order to provide also operator migration for those applications, STREAMMINE3G employs a legacy migration protocol in addition to the pause-free variant which requires a short stopping of the event stream in order to ensure a duplicate and gap-less free migration.

The protocol is depicted in Figure 3.14 where the first three steps are equivalent to the protocol of the interruption free migration with postponed state synchronization as depicted in Figure 3.12. However, since event streams are not deterministically merged which means that two replica will not produce the same order of events, duplicate detection downstream cannot be established in a reliable way, hence, we cannot allow to have redundant processing at any point of time which includes slice migration. In order to prevent such a situation, the primary is stopped ④ before the state synchronization ⑤ while the secondary is only started after the synchronization completed ⑥ which introduces a short stop in the event flow and implies a latency increase.

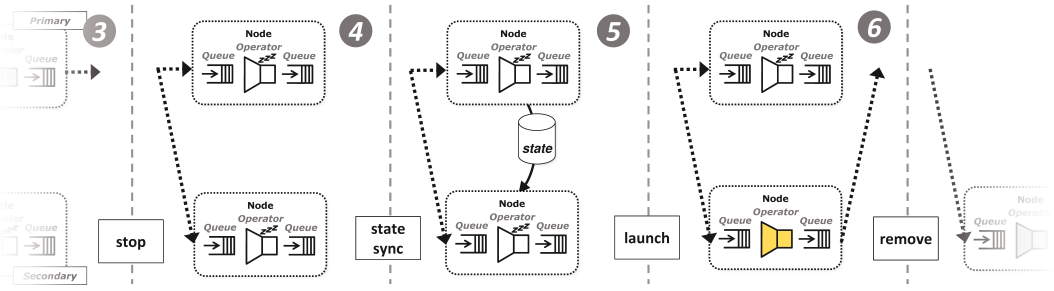

Figure 3.14: Legacy migration (with postponed state synchronization).

Similar as in the interruption free migration protocol, an alternative implementation using

event replay from upstream peers event logs is possible, however, the primary is still paused prior to the execution of the state synchronization.

The introduced processing latency greatly depends on the size of the state, the serialization overhead for the data structure as well as available network bandwidth. Hence, in situations where the state comprises hundreds of MBs, latency can increase up to tens of seconds which might be not tolerable for the application user.

3.8.3 Related Work

In the following, we will present related work with regards to the migrations mechanisms and protocols presented in this chapter:

Migration mechanisms can be broadly distinguished into two classes: (*i*) *pause-drain-resume* and (*ii*) *parallel track* [CSA05]. While in the first approach, the processing is paused for the duration of migration such as presented as *legacy migration* in this chapter, the second approach is characterized through redundant and concurrent processing, i.e., both operators process simultaneously the same sets of events in order to produce partial or fully identical results during the migration phase such as in the *interruption free* approach.

Authors in [SSC+02] propose a migration protocol (*state movement protocol*) which falls into the first category and is very similar to our *legacy migration* approach, however, in their approach, upstream operator partitions are instructed to pause the transfer of events for the downstream partition being moved while in our approach, we only stop the processing of events directly at the moving partition. In order to prevent an increase in processing latency for operators which are not moved during the *quiescence phase* but still consume and rely on events originating from the same upstream operator, a *transient skew buffer* is used at upstream partitions. In our approach, the incoming queues installed at downstream operators serve the same purpose, i.e., prevent a latency increase during the quiescence phase due to paused processing.

In case the user provides context information with the operator such as if an operator is a windowed operator including its window size, more advanced and optimized state movement protocols can be applied. Authors in [GJPPM+12] propose two protocols (window and state recreation) where the state is being recreated based on such information. Although the proposed protocols aim at reducing the imposed communication overhead and state recreation time, the approach still requires context information while we target the black-box operator approach for STREAMMINE3G where no such information is provided at all by the application user.

An alternative to the pause-drain-resume approach is *parallel track* as proposed by Zhu et al. [ZRH04] which does not require the pausing of an operator. Although the approach is very similar to our first presented migration protocol, there are still major differences: For example, in our approach, we rely on deterministic execution using a deterministic merge to ensure

identical output for both concurrently running replicas while Zhu et al. does not require any ordering. Using the timestamp vector of the state allows us to reliably filter out duplicates at downstream operators. Moreover, in our approach, we do not rely on any context information of the operator as the work presented in [ZRH04].

3.9 Related Event Stream Processing Systems

In the following section, we will present several open-source implementations that have emerged during the past five years from industry as well as academia. We will discuss briefly their programming models, system architecture as well as the guarantees provided with regards to event ordering and fault tolerance and compare them to our StreamMine3G approach.

3.9.1 Apache S4

Apache S4 [S4215, NRNK10], originally developed and pushed by Yahoo! Inc. was a first attempt towards real-time computation at large scale. The ESP system is entirely written in Java which simplifies debugging and bug tracking while still being portable for various platforms. S4's programming model was inspired by the simplicity of MapReduce [DG08] where users are enabled to implement operators through user-defined-functions which are instantiated as processing elements (PEs) during runtime. Similar as with StreamMine3G, users are provided with an event as input and a stream object (which mimics a collector object) to emit events to downstream peers. However, contrary to our system, PEs are instantiated based on the emitted keys, hence, a PE operates solely on a single key rather than on a partitioned stream of events comprising a set of keys. In order to mimic event partitioning, the provided `KeyFinder` interface can be used in order to assign events to predefined keys based on the attributes provided.

The flow of events in S4 is specified through streams that are attached to PEs. PEs consume events from those attached streams where the streams themselves can be made available to any of the peer PEs in order to allow them to emit and route events transparently to interested parties forming a topology of operators. In order to feed the system with events, a special adapter operator exists which has access to the input stream attached to the first operator in the chain of operators.

For message transport, `netty.io` [Net15b] is used, a Java-based library providing a convenient abstraction to the non-blocking native I/O interface in a similar way as `boost.asio` for C++. However, contrary to StreamMine3G which sends events in batches when facing high volumes of data, events in S4 are always disseminated without any batching capabilities. Moreover, S4 reads messages asynchronously without throttling read requests (`AUTO_READ` mode) which can quickly saturate input buffers where events will be dropped transparently in case of an overload of the operator which is also known as *load shedding* [TcZ07].

Similar as with STREAMMINE3G, S4 uses Zookeeper in order to store its cluster configuration in a reliable manner. However, the cluster configuration is static and cannot be changed during runtime. Hence, nodes cannot be added or removed during the course of processing in order to accommodate new load situations through elasticity mechanisms. Moreover, a S4 process can only host a single PE where co-location of PEs for better resource utilization can only be achieved through launching of several S4 processes on the same node.

While the initial releases of S4 were only equipped with very basic fault tolerance by simply restarting crashed PEs, the latest version 0.6.0 offers state persistence through uncoordinated checkpointing where the fields of a PE are transparently serialized and written to stable storage. Although checkpointing preserves the state of long running operations, S4 does not provide strong consistency since events are neither merged, processed deterministically nor logged for a replay of in-flight events during an initiated recovery. Hence, S4 provides only gap recovery without strong consistency for state as well as events.

Although S4 was open-sourced by Yahoo Inc. in early 2011 and became an Apache incubator project just two years after, the project itself is can be considered as dead since its last release (0.6.0) in summer 2013.

3.9.2 Apache Storm

A project that evolved roughly at the same time as S4 is Apache Storm [Sto15] which was originally pushed by Twitter Inc. and is written in Clojure [Hal09], a Java-based dialect. Operators in Storm are implemented using UDFs in a similar way as in S4 and STREAMMINE3G. However, contrary to S4 where operators (processing elements) solely operate on a single key, Storm operates on a partitioned stream of events comprising several keys specified through *stream-groupings* which is essentially a partitioner. Instead of providing several streams to an operator in order to have events delivered downstream as done in S4, Storm provides a single collector object to emit events and handles transparently the event dissemination to downstream peers based on the user's specified topology.

Topologies in Storm are established by specifying the set of upstream operators a downstream operator is required to consume events from. The topology definition also contains the partitioning scheme (i.e., the stream grouping) as well as the parallelization degree in a similar fashion as in STREAMMINE3G. Storm distinguishes between two types of operators: bolts and spouts. While spouts are solely used as input adapters to ingest events, bolts serve as regular operators within a topology that receive, process and emit events to downstream bolts.

Earlier versions of Storm were based on `zeroMQ` [Zer15] for message dissemination across the network as it provides a high-level abstraction for networking similar to `boost.asio`. However, since `zeroMQ` is written in C++, the use of a JNI wrapper [Lia99] library was needed which introduced additional overhead through (un-)marshaling of data and complexity when deploying the system on diverse platforms. Therefore, authors recently switched to the `netty.io` frame-

work which not only simplifies the deployment of the system but also improves performance as micro benchmarks have shown [Net15a].

Storm clusters are configured through Zookeeper, however, contrary to S4, new nodes can be added at runtime. In order to mimic elasticity, Storm provides a *re-balance* feature where operators (i.e., bolts) will be evenly redistributed across the cluster taking into account the newly added resources. Although re-balancing allows a scale out, a contraction of the system can only be achieved by actively crashing running nodes.

In order to cope with system failures, Storm provides the concept of *transactional topologies* where unacknowledged events will be automatically replayed. Hence, Storm employs mechanisms to prevent event loss, however, accumulated operator state will be lost. Authors of Storm propose to use RDBMS systems instead for storing the state of long running applications.

In 2013, Storm became an Apache Incubator project and is currently maintained and supported by a steadily growing community.

3.9.3 Apache Samza

A quite recent development in the area of ESP systems is Apache Samza [Sam15] which evolved from LinkedIn Inc. The system is written in Scala [OSV08] and uses Apache Kafka [Kaf15], a publish-subscribe system to exchange messages. Operators in Samza are called `StreamTasks` that operate on a partitioned stream similar as in Storm and StreamMine3G.

Instead of defining an explicit topology which cannot be modified during runtime as in S4 and Storm, Samza follows a more flexible approach with a loose communication model where `StreamTasks` communicate solely through named streams. This model allows a deployment and removal of `StreamTasks` without any interference. In order to feed Samza with new events, `StreamConsumers` are used which serve as input adapters for the system.

Operators, i.e., `StreamTasks` in Samza are deployed using Apache YARN [Yar15], a resource aware scheduler which evolved from the Hadoop eco-system. However, contrary to Storm and StreamMine3G which use multi-threading to process events, a set of single threaded processes is used in Samza to execute `StreamTasks` in a sequential way. While Storm supports re-balancing where threads and nodes can be added during runtime, Samza follows a more static approach using a fixed resource pool.

For message dissemination, Samza uses Kafka which employs its own low level implementation to exchange messages between peer nodes based on bare-metal Java TCP sockets. However, to store the configuration of the system, Zookeeper is used.

Samza provides fault tolerance using two mechanisms: Periodic checkpointing of state and event logging. Event logging is realized through persistently writing events to stable storage which is executed at the transportation layer, i.e., Kafka while state persistence is provided by

Samza itself. In order to offer exactly once processing semantics, the state is equipped with a timestamp vector to track the progress of each upstream partition. However, this requires that every partition emits events with strictly monotonically increasing timestamps as in STREAMMINE3G.

Since `StreamTasks` can explicitly define the order of upstream partitions to consume events from at runtime, a deterministic merge can be easily implemented at operator level giving the user the possibility to recover from crashes in a precise manner as in STREAMMINE3G.

Similar as with Storm, the project benefits from a large community and has been recently promoted to a top level Apache project.

3.9.4 SEEP

In contrast to the previous projects which evolved mainly from industry, we will now briefly present SEEP [See15, CFMKP13], an ESP system which originates from academia and is publicly available as open-source. SEEP is completely written in Java and shares many commonalities with STREAMMINE3G as the architecture of the system was designed with the focus to provide fault tolerance and elasticity for stateful operators at the same time.

As for the programming model, users are provided with a simple MapReduce-like interface which provides an event as input and a collector object (as field) to emit events downstream. Topologies in SEEP are defined using the provided Java API where first the operators are defined and then explicitly wired by defining to what downstream operators events should be sent to.

The partitioning of data is done explicitly through the `send()` method which takes a second parameter *streamId* to specify to which downstream operator and partition an event should be routed to rather than encapsulating the data partitioning in a separate interface. In order to harness the power of nowadays multi-core architectures, SEEP provides multi-threading support for stateless operators.

Messages in SEEP are exchanged through the use of bare-metal Java socket connections. Similar as in STREAMMINE3G, event batching is used to reduce serialization costs and improve network efficiency.

Contrary to the previous presented systems, SEEP does not rely on Zookeeper for storing its configuration. Furthermore, nodes can be added at runtime which allows a scale-out of the system. Since SEEP comes with an explicit state management interface, stateful operators can be easily migrated to spare nodes during a scale-out without information loss.

The provided state management interface does not only serve for load balancing when scaling-out of the system but also for fault tolerance through periodic checkpointing. Since events are also logged prior leaving the node for dissemination to downstream operators, a gap-less

recovery as in StreamMine3G can be provided. However, providing precise recovery is only possible through the application of a deterministic merge which is currently only supported through the use of an explicit ordering operator that must be plugged in into the operator pipeline right before the operator requiring the ordered input.

3.9.5 IBM System S

A system that originates from industry, however, is not publicity available as open-source software is IBM's System S [AAB+06, GAW+08]. The system comprises of the StreamProcessingCore (SPC) [AAB+06] which can be considered as the actual streaming engine similar to StreamMine3G with a query language layer on top called SPADE [GAW+08]. The language allows the definition of queries including the information about its distribution and parallelization. In addition to a high-level definition of queries using SPADE, users can also implement their custom operators using UDFs, maximizing the user's flexibility.

SPC shares many similarities with StreamMine3G's architecture and design: Firstly, operators are executed as processing elements (PEs) where they form a topology in order to carry out a specific query. Furthermore, topologies can be modified at runtime through stream composition. In order to increase the efficiency of the system, applications can consume public streams avoiding unnecessary re-computation of previous results. In situations where several succeeding operators are deployed on the same local node, data is passed via simple pointers as in StreamMine3G.

In order to handle system failures, System S is also equipped with a set of fault tolerance mechanisms such as active replication and rollback recovery preventing event and state loss.

3.10 Summary

In this chapter, we presented StreamMine3G, our approach for fault tolerant and elastic event stream processing. We first introduced the programming model and the general architecture of the system. We then provided details on how we support stateful event processing and achieve scalability through our explicit state management interface and data partitioning, respectively. We furthermore presented several techniques to improve performance and scalability such as event batching, efficient memory management and delegation mechanisms tailored to ESP systems.

With regards to fault tolerance, we presented several protocols and their underlying implementations to efficiently perform checkpointing, event logging and redundant processing in StreamMine3G. In order to operate StreamMine3G in the cloud, we furthermore described two protocols for stateful operator migration. The chapter concludes with an overview and comparison about recently evolved ESP systems originating from industry and academia.

4 Lowering Runtime Overhead for Passive Replication

In this chapter, we will present an approach for lowering the runtime overhead of event ordering in the context of roll-back recovery. Event ordering ensures that events are processed deterministically producing repeatable output. Deterministic execution allows applications to recovery precisely without information loss nor duplicated processing.

Our approach is based on the observation that many windowed operators share the property of *commutativity*. We therefore introduce the notion of an *epoch*, matching the window of an operator. Instead of enforcing a strict ordering of events, we *assign events to epochs and process events within such epochs (i.e., windows) out-of-order*. Checkpointing and recovery is then solely performed at epoch boundaries providing a deterministic snapshot and allowing the application to recover in a precise manner.

4.1 Motivation

Since ESP applications are long running operations, one requirement developers of ESP applications often face is fault tolerance. Fault tolerance is not only important to mitigate crash failures of the system but also to allow a migration of the system to a new machine. Since the majority of ESP applications accumulate some form of state over time which must not be lost when facing a system failure or when migrating the application, appropriate mechanisms are required to persist state as well as events. Depending on the underlying ESP system, application developers are offered with a set of fault tolerance mechanisms and replication schemes they can choose from such as roll back recovery which uses a combination of checkpointing and logging or active replication.

Although active replication offers a quick recovery, it is often not the preferred solution as it consumes almost twice the resources through its redundant processing. Moreover, active replication requires deterministic execution in order to reliably filter out duplicates at downstream operators which adds another layer of complexity.

However, many applications are less critical and can easily tolerate recovery times of a few

tens of seconds where simpler mechanisms for fault tolerance such as rollback recovery can be applied. Moreover, rollback recovery uses a combination of checkpointing and event logging which consumes far less resources than the use of active replication.

Despite the fact that system crashes occur relatively seldom, application developers are often in favor of a precise recovery for their application where neither events are lost nor will be processed twice. Precise recovery simplifies the implementation of an application, and reduces the developer's effort for guaranteeing and reasoning the correctness of the application at hand.

In order to let an application recover precisely, deterministic execution (as in active replication) is required where events originating from different upstream operators are merged and processed, i.e., passed to the user's operator code in a predefined order. The strict order allows operators[1] to produce the exact same sequence of events at any point of time, i.e., even during a replay. The replay-ability/reproducibility of events allows a reliable filtering of duplicates using a timestamp vector associated with the operator state and a gap-less recovery if event logging is used. Moreover, checkpoints can be taken independently without coordination which lowers the overhead imposed onto the system through checkpointing.

Although the use of rollback recovery saves computational resources, users are still required to run their application using deterministic execution in order to benefit from the properties provided by a precise recovery. However, deterministic execution with a strict ordering of events imposes a non-negligible overhead onto the system as we will show in the evaluation Section 4.3 of this chapter.

The majority of ESP applications perform some form of aggregation where either several events are summarized over some window or several streams joined in order to identify correlations between events within a fixed window. Many of those operators such as aggregation and joins share the property of ***commutativity*** where the order of arrival within a window does not distort the result. Commutativity also applies to non-order-sensitive custom-written operators (using user-defined-functions) as well as stateless ones such as simple filters. Since applications that solely employ commutative operators do not require a strictly ordered event stream in order to produce correct results, deterministic execution would solely serve here for the purpose of achieving a precise recovery.

In the following, we will present an alternative event ordering scheme which consumes far less resources than a strict ordering while still allowing applications to recover in a precise manner.

4.2 Epoch-Based Processing

Our approach is based on the observation that many applications use commutative operators and operate on jumping windows. Commutative operators share the property of non-order-

[1] Under the assumption the operator itself behaves deterministically.

sensitivity where the result does not depend on the order of arrival of events within a window. Hence, the key idea of our approach is to process events within a window in arbitrary order but guaranteeing that windows themselves are processed in order.

In order to ensure the correctness of our weak ordering scheme, the set of events for a window must be always well defined. Events are assigned to such sets, which we call ***epochs***, using their timestamps. An epoch ω is essentially a time based tumbling window with two boundaries: A lower bound b_{lower} and an upper bound b_{upper} where both bounds are defined using timestamps with $ts_{b_{lower}} < ts_{b_{upper}}$. The assignment of events to epochs fulfills the following relation: $\omega = \{e | ts_{b_{lower}} < ts_{event} \wedge ts_{event} \leq ts_{b_{upper}}\}$

In order to ensure a deterministic assignment of events to epochs, the incoming event stream must provide events in a monotonically increasing order. Note that no *strict* monotonicity is required as it is the case when applying strict event-ordering (complete determinism) since multiple consecutive events can now share the same timestamp as they will be assigned to the same epoch anyhow.

To illustrate our approach, we will first briefly recall how strict event ordering is carried out in StreamMine3G (as described in Section 3.6.4), then explain the properties of no-ordered event processing followed by a description of our epoch-based approach.

4.2.1 Complete Determinism

Strict event ordering provides *complete determinism* as the sequence of events to process is always well defined. Therefore, the state of an operator can be checkpointed and recovered at any point of time. During a recovery, the checkpoint is loaded first which is equipped with a timestamp vector indicating the last recorded set of events that modified the state. Since events are processed in the exact same order as prior to the system crash, the timestamp vector can now be used to discard outdated events originating from the replay of event logs and resume the processing of events precisely at the point of time the checkpoint was taken before masking the failure completely.

Although complete determinism allows to checkpoint and recover the system at any point of time, it limits the system's performance as events must be first put in the correct order (using a deterministic merge) prior to processing them. Moreover, the processing can be stalled until the next in-order event for an upstream channel is available in order to proceed with the ordering. This does not only introduce latency but also synchronization overhead as the state of the upstream channels must be continuously checked prior to the merge so that at no point in time an event e_2 is going to be processed prior to an event e_1.

4.2.2 No-Order

In contrast to complete determinism, *no-ordered* processing does neither introduce latency nor synchronization overhead as events are immediately processed in the order as they arrive from upstream channels. However, the absence of ordering imposes other challenges: In order to recover an application precisely, the order of events must be recorded which can be achieved through logging of the input sequence on stable storage. Although this approach does not introduce latency, logging events to stable storage incurs a high runtime overhead which lowers the overall throughput of the system. Note that the no-order approach can only be applied to commutative operators such as joins or aggregations which are not order-sensitive.

4.2.3 Epoch-based Determinism

Our epoch-based approach is inspired by virtual synchrony [BJ87] which allows an out-of-order execution, yet it restricts the execution to predefined intervals (i.e., epochs) limiting the overhead of synchronization. Within an epoch (originally called views), events are processed in arbitrary order which decreases the time and processing cycles spent for synchronization. Only at the end of each epoch, synchronization is required in order to ensure that all events falling into the current epoch are being processed prior to the start of the following epoch.

Table 4.1 illustrates the differences in time spent on waiting for the different ordering schemes, i.e., deterministic, no-order and the virtual-synchrony inspired epoch-based ordering. In the provided example, we assume two upstream channels which represent either two different operators or two partitions of the same operator named as source #1 and #2. The sources send events downstream to some stateful operator. The first column in the table depicts the arrival time of these events while the second column indicates from which of the two sources the events originate. The subscript in the events is the timestamp assigned by the sources. The latencies imposed by each of the approaches, i.e., for the epoch-based, complete deterministic and no-order execution, are depicted in the three rightmost columns of the table.

Arrival	**Source**		**Waiting time**		
time	1	2	det	no-order	epoch
4	e_1		1	0	0
5		e_3	4	0	0
6		e_4	3	0	3
7	e_2		0	0	0
8		e_6	?	0	1
9	e_5		0	0	0
			8	0	4

Table 4.1: Example execution and waiting time for deterministic, no-order and epoch-based ordering scheme.

As mentioned above, complete determinism limits performance as processing must be paused

until the next in-order event is available. In our example, e_3 arrived from source #2 but cannot be processed until it is certain that no event with a timestamp of less than three will arrive from source #1. However, this information becomes only available with the arrival of e_5 at time $t = 9$. By then, the processing of e_3 and e_4 has been already delayed for four and three turns, respectively. In our example, those delays lead to an accumulated waiting time of eight turns when using strict ordering as indicated in the last row of the table. In case where the system experiences a slight overload, this delay is usually much lower as an overload causes an enqueuing of events, hence, the next in-order event is already available. Although an overload decreases the average delay for the availability of the next in-order events, the overall processing latency will increase as events will spend more time in processing queues due to the overload.

In the virtual synchrony case, synchronization is only executed at epoch boundaries which is represented as a horizontal line between row #3 and #4. At this point, the system must ensure that all events e_n with $n \leq 3$ must have been processed prior entering a new epoch. For this reason, e_4 is assigned to the second epoch and is delayed for three turns until the arrival of e_5. Only with the arrival of e_5, sufficient information exits in order to close the first epoch comprising events e_1, e_2, and e_3. The exact order how these events are processed within the first epoch is irrelevant as it does not distort the result. This allows an immediate processing of event e_3 without having to wait for the arrival of e_5 as it is the case in complete determinism.

In contrast to complete determinism which allows to checkpoint the state at arbitrary points in time, virtual synchrony limits checkpointing to epoch changes, i.e., the point in time when one epoch has been closed and prior to the start of the following epoch. At those boundaries, i.e., the epoch changes, the system provides a ***deterministic snapshot*** whose state can always be reproduced since the set of events included in the state is fully known. Hence, the epoch-based approach permits us to recovery the state of an operator to known points in time while keeping the overhead for synchronization low. Note that the lengths of an epoch can be either defined using counts or timestamps, hence, the approach is applicable to operators utilizing count- or time-based windows.

In order to fully benefit from the epoch-based ordering approach, users are required to configure the length of an epoch to match the window size used in the application. Furthermore, epochs should be aligned to the window boundaries without an offset so that the closure of an epoch is in sync with the closure of the supported window.

4.2.4 Implementation

After giving an intuition about the epoch-based ordering, we will now provide more details on how the approach is implemented in StreamMine3G.

The epoch-based ordering is performed by the *merge function* $f_m$which is encapsulated in the sequencer component of an operator. The function takes a set of upstream channels C as

input, however, contrary to definition of an upstream channel as in Section 3.6.4, a channel in our epoch-based approach does only accept events from a single upstream slice rather than a set of slices in case they are replicated. This restriction is necessary since events within an epoch are processed in arbitrary order, hence, the outputs of the replicated upstream slices may differ unless a single event is solely emitted at the end of each epoch which is the ideal case. In this chapter, we will only consider the ideal case while in Chapter 5, we will allow intermediate output of an epoch and show how determinism can still be enforced.

In addition to the set of upstream channels, the size of an epoch must be provided as input to the *merge function* f_m. If no size has been provided by the developer, the size is set to one by default which is equivalent to an execution with complete determinism as synchronization happens after every single event. The provided epoch size ($epoch_{size}$) parameter is used to compute the end of the current epoch, i.e., $ts_{b_{upper}} = ts_{b_{lower}} + epoch_{size}$. Upon initialization of an operator, the start of the first epoch, i.e., b_{lower} is set to 0.

The *merge function* f_m consumes events from the set of upstream channels in arbitrary order rather than in a round robin or strictly ordered fashion via a priority queue as in complete determinism. Hence, the order for consuming events is solely defined by the arrival of packets coming from the underlying network stack. Prior to the processing of a recently consumed event, the function determines if the event belongs to the current epoch, i.e., the timestamp is in range of the boundaries of the current epoch or if it is an event for any future epoch. In case the event cannot be processed immediately, the event is appended to a FIFO queue associated with each upstream channel.

In order to determine an epoch change, the merge function tracks the timestamps for each upstream channel using a timestamp vector. An epoch change can be initiated as soon as every channel received an event with a timestamp greater than the upper bound of the current epoch. After closing an epoch and prior to the start of a new one, checkpointing can be performed in order to allow a recovery of the operator at a later point of time.

Listing 4.1 depicts a simplified version for the epoch-based event ordering algorithm in pseudo code. As mentioned above, events are consumed in arbitrarily order from any upstream channel (Line 2). In order to determine if an epoch can be closed or not, the timestamp vector is updated with the timestamp of the consumed event (Line 3). After updating the timestamp vector, we determine if the event at hand belongs to the current epoch or to a future one (Line 4). In case it falls into the current epoch, the event is processed immediately by passing it directly to the `process()` method of the user-provided operator code (Line 5), otherwise it is appended to the backup queue (Line 7).

After processing an event, a check is performed whether the current epoch can be closed or not through an iteration over the timestamp vector in order to determine if all upstream channels have passed the current epoch's upper bound or not (Lines 9-12). In case the epoch can be closed (Line 13), first a snapshot of the state is taken and written to stable storage (Line 14), the epoch's upper bound is updated (Line 15), and the backup'ed events of each upstream

Program Listing 4.1 Epoch-based event processing.

```
1:  while true do
2:      event ←anyReadyUpstreamChannel().poll()
3:      upstreamChannels[event.channel].ts ← event.ts
4:      if event.ts ≤ epoch.upperBound then
5:          PROCESS(event, state)
6:      else
7:          upstreamChannels[event.channel].backupQueue.append(event)
8:      epochChange ←True
9:      for each channel in upstreamChannels do
10:         if channel.ts ≤ epoch.upperBound then
11:             epochChange ←False
12:             break
13:     if epochChange = True then
14:         CHECKPOINT(state)
15:         epoch.upperBound ← epoch.upperBound + epoch.size
16:         for each channel in upstreamChannels do
17:             channel.queue.prepend(channel.backupQueue)
```

channel are pre-pended to the individual channels for a future consumption (Lines 16-17).

In order to further reduce processing latency, it is also possible to process events of future epochs immediately rather than enqueuing them by providing each epoch with its own state instance which is then merged with the state instance of the previous epoch upon epoch closure using a *combiner* as in MapReduce. However, this requires an extension of the programming model.

Note that an alternative implementation using a listener pattern is used in StreamMine3G in order to determine if all upstream channels have passed the boundary of the current epoch. The listener-based implementation reduces the number of costly iterations depicted in Lines 9-12 of Listing 4.1.

4.3 Evaluation

In the following section, we will present the results from various experiments we performed in order to asses the performance of our proposed solution. All experiments were executed using a homogeneous 50 nodes cluster running a Debian Linux 5.0 with kernel 2.6.26. Each machine in the cluster is equipped with 2 Intel XEON E5405 CPUs (quad core) and 8 GB of RAM. The nodes communicate via a Gigabit Ethernet (1000BaseT full duplex) network.

In order to evaluate our approach, we implemented a canonical word count application for continuous streams of data. Hence, the application comprises two operators, a stateless mapper which parses lines from a text stream which are then broken up into individual words in oder to be consumed by a stateful reducer operator for summarization. However, contrary

to the original word count application as implemented in MapReduce, word frequencies comprise only the corpus received within a tumbling window rather than the entire data set.

For the experiments, we co-located mapper and reducer operators on the same nodes. The co-location follows the original MapReduce approach where map and reduce tasks are running on the same physical host to benefit from a reduced network communication as well as an efficient use of computational resources.

As input for the application, we use a 25 GB large Wikipedia dump which is consumed through a file stream rather than using a live data source. In order to mimic a partitioned data source with multiple input streams, the dump is stored in a pre-partitioned way on the local disk of each node.

4.3.1 Failure-free Experiments

In the first set of experiments, we evaluate the performance of our proposed solution in a failure-free execution, hence, fault tolerance mechanisms are enabled, however, we do not trigger a recovery by actively crashing components of the system.

We start our performance evaluation with a comparison of our epoch-based approach with similar systems in the area of MapReduce-like systems, followed by several experiments to asses the scalability and the runtime overhead of our proposed approach.

System Comparison Experiments

In our first experiment, we compare the performance of our epoch-based approach implemented in StreamMine3G with Hadoop, a MapReduce-based and state of the art big data system.

Despite the fact that MapReduce systems are tailored for batch processing whereas our StreamMine3G approach targets continuous data processing, both systems shares many commonalities: First of all, the simple and intuitive programming interface allows developers to process unstructured data in a convenient way using user-defined-functions. Second, both systems are highly scalable through the partitioning of data which allows users to apply the systems to arbitrary large data sets. Third, MapReduce as well as our epoch-based approach employed in StreamMine3G require operators to share the property of commutativity.

Although Hadoop has been designed as a batch processing system, several extensions have been proposed such as the HadoopOnline-prototype [CCA+10], which allows to break the strict phasing of the traditional MapReduce paradigm enabling the system to operate on continuous stream of data as in ESP systems. We therefore used the following three systems for our first experiment: An unmodified version of Hadoop, the streaming enabled HadoopOnline-prototype and our StreamMine3G system. All three variants consume their input via file

streams even though ESP systems would technically consume data from live data sources via network streams. However, in order to keep the comparison across those different systems fair, the same data sets as well as methods for ingesting the data into the system were used.

For the experiment, we used fixed sized data sets ranging from 100 MB up to 8 GB and measured the execution time for each of the systems to complete the processing of the prepared data sets. Figure 4.1 shows the result of our measurements.

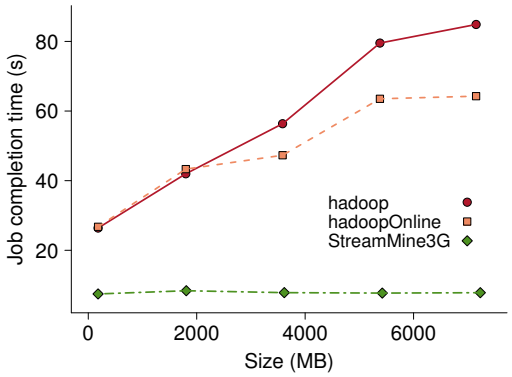

Figure 4.1: Job completion time of `hadoop`, `hadoopOnline` and StreamMine3G for increasing problem sizes.

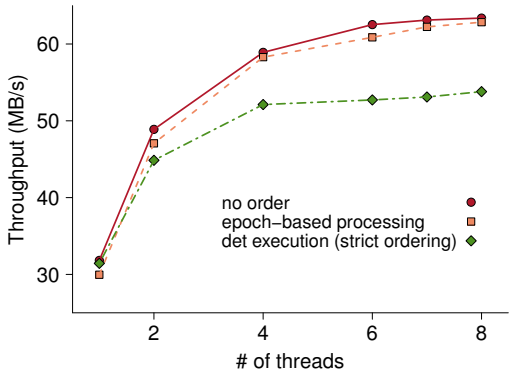

Figure 4.2: Vertical scalability in a single multi-core processing node.

As shown in the Figure, the open-source Hadoop implementation exposes the longest completion time even for small problem sizes comprising only as little as 100 MB. The reason for this long execution time is the job setup time in Hadoop which is typically in the range of 30 seconds.

With increasing job size, the HadoopOnline-prototype outperforms the unmodified Hadoop implementation. The improvement is due to the fact that *data pipelining* is used in the HadoopOnline-prototype where intermediate results are immediately pushed to the next stage, i.e., next operator, as in ESP systems rather than stored on local disks. Furthermore, the HadoopOnline-prototype supports incremental processing where smaller portions of data are incrementally passed to the reducer rather than performing a multi-pass merge sort as in Hadoop's legacy implementation.

Although the job completion time significantly improved in the HadoopOnline-prototype compared to the legacy MapReduce implementation, it is still dominated by the set up time especially when processing small data sets comprising a few hundred MBs.

Since the operator deployment in StreamMine3G does not involve complex and time costly scheduling actions as in Hadoop, applications in StreamMine3G can be deployed much faster reducing the overall job completion time as shown in Figure 4.1. This leads to an almost constant completion time in StreamMine3G even for large problem sizes of 8 GB of input

data. Note that there is an increase in completion time beyond that point which is not shown in the figure. Moreover, Hadoop-based systems provide fault tolerance based on a disk-based logging of intermediate results which introduces additional overhead and reduces the overall event throughput at the same time. Using our epoch-based approach, accumulated state is only checkpointed at predefined intervals (i.e., epoch intervals) reducing the overhead for event logging (to stable storage) as well as checkpointing.

Scalability Experiments

After the performance comparison, we will now asses the performance and the imposed overhead for the different ordering schemes, i.e., no-order, deterministic and the epoch-based approach. In a first experiment, we evaluate the vertical scalability of the system in order to verify if our approach can fully harness nowadays many core systems. Therefore, we varied the number of available threads in the thread pool of a StreamMine3G node. Ideally, the size of a thread pool matches the number of available cores of the underlying virtual or physical machine a StreamMine3G node is running on in order to fully utilize the available processing capacity.

The outcome of the experiment is depicted in Figure 4.2. As expected, the no-ordered execution performs best achieving the highest throughput. Note that input logging for the no-order execution was disabled, hence, the depicted throughput represents the best case, i.e., an upper bound of the achievable throughput. Ideally, logging will not have any effect on throughput but only add to latency and stable storage demand.

The execution using the strict deterministic ordering can be considered as the lower bound as it shows the lowest overall throughput regardless of the number of cores used. Moreover, deterministic execution exposes only limited scalability when more than four threads are used. This limitation is due to thread contention on the priority queue which is used for synchronization. The effect of contention becomes even more pronounced with an increasing number of threads.

Our epoch-based approach shows almost an identical behavior with regards to scalability as the no-order approach, yet at a slightly slower throughput. Thus, the throughput is close (within 2, or 1 MB/s difference) to the no-ordering and up to 20% (4 to 5 MB/s) better than deterministic execution.

In the next experiment, we investigate the horizontal scalability of our approach, hence, we varied the number of nodes used for processing from a single node up to 50 nodes. The results of the experiment are depicted in Figure 4.3 and 4.4. While Figure 4.3 depicts the aggregated throughput, Figure 4.4 shows the per node throughput.

As shown in Figure 4.4., the per node throughput decreases with an increasing number of nodes as more contention on network connections as well as synchronization is required. However, the figures also indicate that the benefit of our epoch-based approach over deterministic

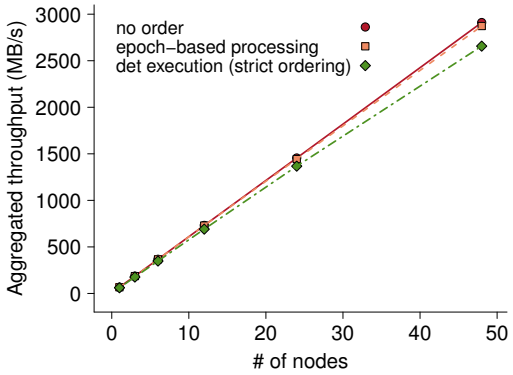

Figure 4.3: Aggregated throughput with increasing number of nodes.

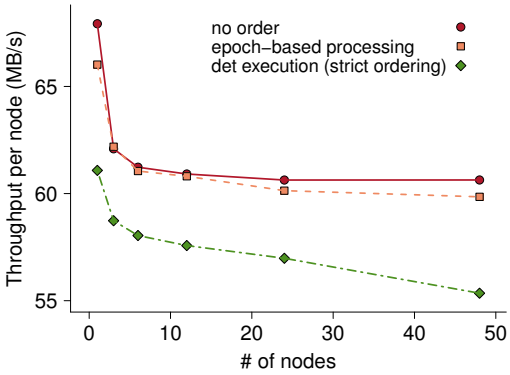

Figure 4.4: Per node throughput with varying number of compute nodes. The epoch-based approach significantly outperforms deterministic execution.

execution becomes even more pronounced with an increasing number of nodes. This is expected as the number of out-of-order arrivals for events increases with the number of nodes involved, hence, more synchronization is required to ensure a strict ordering of every single event while in the epoch-based approach, the synchronization is reduced to epoch boundaries leading to an overall higher throughput.

In order to provide a complete deterministic behavior using our epoch-based approach, the output of an epoch must be deferred until the end of each epoch. However, this increases latency if the user opted for longer epoch intervals as output is less frequently emitted then. The advantage of longer epoch intervals is an overall lower overhead for fault tolerance since synchronization and checkpointing is only performed during epoch changes which happens less frequently. Hence, there is a trade-off between latency and synchronization overhead in the epoch-based approach.

Figure 4.5 shows the impact on latency and throughput with varying epoch intervals. For the experiment, we used the file position in the Wikipedia file as timestamps for the events and varied the epoch interval between 25 MB and 1 GB. As expected, the latency increases with longer intervals as updates from the mapper operator are sent less frequently to the reducer. The measured throughput however approaches the maximum of 62 MB/s rather quickly. At a checkpoint interval of 100 MB, the throughput is already very close to its maximum. Considering throughput and latency in unison, epoch intervals should be around 100 MB which translates to a window of 1.6 seconds considering a throughput of roughly 62 MB/s in average. Larger intervals result only in a marginal increase in throughput.

The latency measured and computed (half the epoch interval size) in Figure 4.5 is the mean value for an entire epoch. For example, if the user opted for an epoch/checkpoint interval of 500 MB and the event throughput reaches around 60 MB, the maximum latency should be

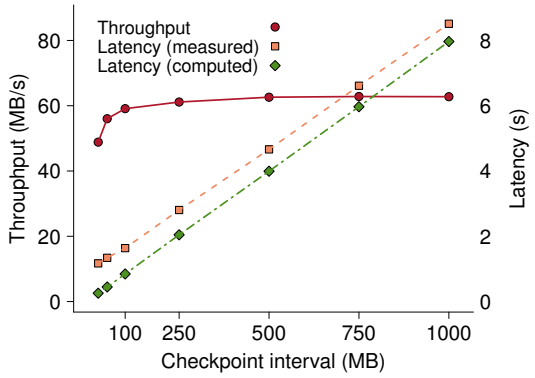

Figure 4.5: Per node throughput and latency with varying epoch interval sizes. nodes. For 100 MB epoch size, StreamMine3G already approaches the maximum throughput.

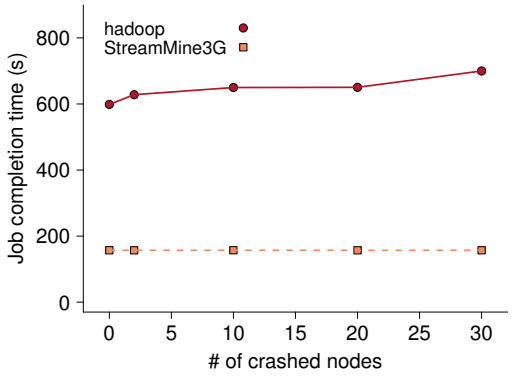

Figure 4.6: Completion time as a function of number of transient crash failures during execution.

around 8 seconds. This can be confirmed visually by looking at Figure 4.7 where the latency is plotted over time for the same epoch interval size. Note that around 15 seconds into the experiment, a failure was inserted which we will discuss in the following section.

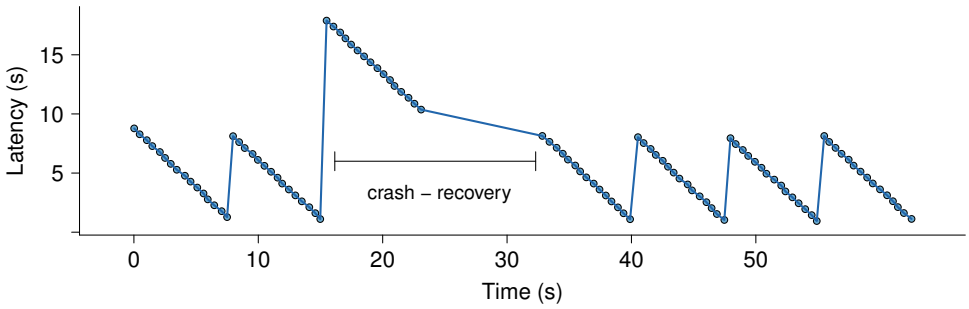

Figure 4.7: Impact of mapper crash on latency for an epoch interval of about 8 seconds.

4.3.2 Failure Experiments

We will now evaluate the behavior of our system when failures occur. As mentioned above, Figure 4.7 depicts also the impact on latency during a crash of a mapper. At around 15 seconds into the experiment, we actively crashed a node hosting a mapper operator. Subsequently, the latency increases to around 18 seconds, 10 more than the previously seen maximum. This increase is due to the time it takes to restart and redeploy the operator on a new node, including loading the most recent checkpoint from disk as well as replaying events for the current interval. Also, note that no updates arrived between seconds 23 and 33. This silence period is an effect of the stopped processing during the recovery after the failure occurred.

In the next experiment, we investigate the impact of crash failures on the throughput of the whole system. In this setup, we consider transient failures where the processes and operators are restarted in the same machine they were prior to the failures. This is equivalent to consider software crashes or software (including infrastructure software) updates, and it is also a common scenario for situations where two nodes, a primary and passive backup, share a disk via eSATA and the passive assumes when the primary fails. In this case, no state needs to be transferred or read from remote nodes. However, later we will consider failures where a recovery is executed on remote nodes which do not own a copy of the state of the failed node locally.

Figure 4.8 depicts the evolution of throughput over time when the system experiences a node crash. We successively crashed more nodes in each subplot starting from 10 up to 40 nodes. The crash was initiated after 18 seconds in the experiments where we actively killed StreamMine3G processes. With failures in 10 machines (second plot from left), there is a drop to 2,400 MB/s from the usual 2,750 MB/s aggregated throughput. As more machines are crashed, the drop in throughput becomes even more pronounced and it takes longer to fully recover.

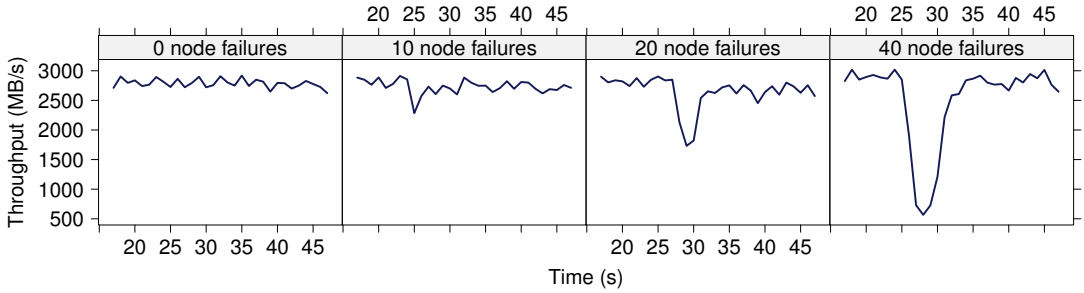

Figure 4.8: Impact of transient crash failures on throughput. Successively more components were crashed for each experiment.

In case the checkpoint cannot be loaded from a local disk, the checkpoints must be transfered from remote nodes (via a distributed filesystem). This not only has an impact on throughput but also on the overall recovery time for such permanent failures as shown in Figure 4.9. Hence, the increase in recovery time can be seen when comparing the graphs of Figure 4.8 and 4.9.

In the last experiment, we show the impact of transient failures in job completion time. For this experiment, we compare the recovery time from Hadoop and StreamMine3G using the word count application. We varied the number of failures and measured the impact on total job completion time. The results are shown in Figure 4.6. The experiment reveals that job completion time for StreamMine3G is almost unaffected regardless of the number of crashed nodes while the completion time visibly increased. The reason for the increased job completion time in the Hadoop case is due to the failure detection and recovery mechanisms used in Hadoop. After the detection of a lost task tracker, first the task is rescheduled to spare nodes where intermediate results are pulled from remote disks which takes a considerable amount of

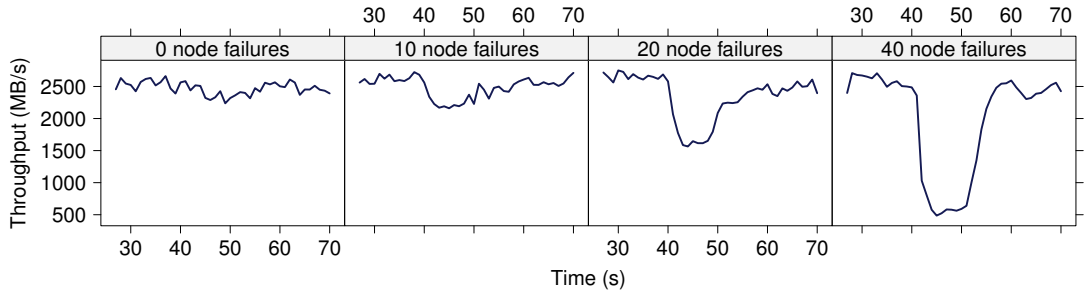

Figure 4.9: Impact of permanent crash failures on throughput. Successively more components were crashed for each experiment.

time while StreamMine3G performs a replay directly from the memory of upstream nodes.

4.4 Related Work

Many distributed systems achieve fault tolerance by using logs and checkpoints which is also known as rollback-recovery [EAWJ02]. In this case, nodes of the system rely on 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 such as StreamMine3G this is especially important as past inputs cannot be kept in memory for long periods of time due to their huge volume.

Checkpoint-based protocols perform state snapshots periodically where the state of the system is rolled back to some globally consistent checkpoint upon a failure. Checkpointing can be performed either in an uncoordinated or coordinated fashion. Uncoordinated checkpointing is easy to implement as nodes can checkpoint their states at any point in time. Once a rollback is initiated, however, 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* [EAWJ02], which can force the system to roll back to the initial state. To avoid the domino effect, event logging can be used in order to ensure the replay-ability of past events. However, if no deterministic execution is used which allows the recreation of past events, event logging must be performed using a fault tolerant stable storage rather than an in-memory log which introduces latency and lowers throughput.

Coordinated checkpointing on the other hand 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 orchestration overhead and complexity. In our case, determinism by ordering events provides such implicit coordination. Therefore, we need neither to log messages to disk in order to gain repeatability of events nor to explicitly force nodes to checkpoint at certain points of time.

Finally, if multiple failures occur 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 as well. Replicating the log imposes continuous burden to the system as all messages received by a node need to be forwarded to other nodes.

Authors in [BFF09a] propose a different approach for stream processing systems based on the speculative execution of events to overlap and minimize logging and checkpointing overhead. However, speculative execution requires the system to rollback the state using software transactional memory (STM [ST95]) in case of crash failures which is currently not supported by our state management model.

4.4.1 Determinism and Virtual Synchrony

By assuming completely deterministic execution as proposed by authors in [AS00, SDFZ09], the coordination problem can be avoided as mentioned above. Deterministic execution requires coordinating when each message is processed such as performing a deterministic merge. However, complete determinism is not always necessary and, as we have shown, it imposes an additional overhead to the system. We explore the fact that many operators are normally commutative and therefore force only the minimum determinism needed to guarantee periodically consistent checkpoints.

Virtual synchrony [BJ87] was originally used to provide consistency in active replication. Here we extend this idea to rollback recovery by limiting event ordering to moments where it is necessary to provide determinism.

Regarding determinism in multi-threaded applications, several approaches have been proposed: Jiménez-Peris et al. [JPPnMA00] proposed a deterministic scheduler that guarantees that multi-threaded 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., [BKI06]) or by forcing a total order in code sections processed by a transactional memory (e.g., [BFF09b]). In our case, multi-threading is not an issue as our approach to parallelism guarantees that independent data is mapped to independent operator partitions and, that related data is serialized and processed in the same partition.

StreamCloud [GJPPMV10] 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 using a full deterministic merge induces more overhead than our epoch-based 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 [BHCG10, AWHF10, CWTY10]. We focus on determinism at the application level. Whether or not OS-level mechanisms for deterministic execution could

be of help in our case, it paves the way for interesting future work.

4.5 Conclusion

In this chapter, we presented an approach for reducing the overhead imposed by ordering events required for deterministic execution. Our approach is based on the observation that many operators used in ESP applications are commutative and operate on tumbling windows. We therefore define epochs which ideally match the windows defined by the application and assign events to those epochs. Events within such epochs are processed out-of-order and only the order of epochs is preserved. At the change of an epoch, i.e., an epoch boundary, we perform checkpointing or initiate a recovery after a crash which enables us to recover the system in a precise manner.

Our approach is inspired by virtual synchrony [BJ87] that has been proposed and intensively investigated in the context of state machine replication. Adapting the ideas from virtual synchrony, we can almost achieve the same throughput as processing events without event ordering as shown in the evaluation previously. This is remarkable insofar as our no-order implementation can be considered as a base-line implementation that does not support fault tolerance due to the absence of replay-ability of events.

5 Lowering Runtime Overhead for Active Replication

In the previous chapter, we presented an approach to lower the overhead of event ordering by introducing the notion of an *epoch*. Events within such an epoch have been processed out-of-order increasing throughput and lowering latency in comparison to a strict ordering while still offering a complete deterministic snapshot at epoch boundaries.

In this chapter, we extend the previous approach for active replication: Instead of processing events out-of-order within epochs, we delay the processing until the start of the following epoch and perform a *deterministic merge on epoch rather than event level* so that all replicas are provided with identical sequences of input. We furthermore propose a *light-weight consensus protocol* that allows us to *stop the propagation of non-determinism originating from unreliable sources* and to overcome stragglers reducing overall processing latency.

5.1 Motivation

With the recent trend of fairly-priced on demand computational resources as enabled by cloud computing, a constantly growing number of businesses is moving from traditional batch processing to online processing. Online processing of data streams offers new business opportunities through a quick reaction to relevant situations (e.g., fraud detection or automated trading). However, such applications do have tight QoS constraints with respect to response time during a recovery after node failures.

One way of providing fault tolerance for ESP applications is passive replication which uses a combination of in-memory logging and checkpointing [GZY+09]. Passive replication comes with the advantage of a low resource footprint since only a limited set of events is kept in memory and checkpoints are stored on stable storage. During a recovery, the most recent checkpoint is loaded and events from in-memory logs are replayed in order to mask the failure completely. Despite the fact that passive replication imposes only a small resource overhead onto the system during normal operation, it suffers from long recovery times which is often unacceptable for critical applications such as fraud detection or high frequency trading.

The recovery time in passive replication is primarily dominated by (*i*) the size of the state that must be recovered and (*ii*) the amount of events needed to be replayed since the last taken checkpoint. Although it is possible to reduce the recovery time by taking checkpoints more frequently as it lowers the amount of events that must be replayed, the size of the state cannot be reduced that easily. However, reducing the checkpointing interval will also decrease the overall throughput due to lock contention where checkpointing and event processing compete for the acquisition of state locks needed to ensure consistency when modifying the state.

For applications that require a quick fail over, active replication can be used as an alternative to passive replication. In active replication [Sch90, MFB11], fault tolerance is achieved by using two or more replicas to process events where a crash of one of those replicas does not affect the processing of events for other operators. Unfortunately, active replication has a high resource footprint as it consumes almost twice of the resources compared to passive replication. In addition to the high resource footprint, active replication also requires deterministic processing of events in order ensure that replicas produce identical results for a reliable filtering of duplicates at downstream operators.

Since ESP applications are targeted to operate on unbounded streams of data, the majority of such applications compute over event windows where several events originating from different streams are either correlated or summarized using operators such as joins or aggregates. Many of those windowed operators share the property of commutativity where the input order for a window is irrelevant for the computation of the correct result. Hence, the underlying ESP system must solely ensure that events are assigned to the correct window rather than enforcing a strict order for those classes of operators.

In the previous Chapter 4, we introduced the notion of an *epoch*. An epoch is essentially a window, however, only known and used by the underlying ESP system. The size of an epoch must match the size of the operator window so that an ESP system can correctly assign events to the appropriate windows. The advantage of such an assignment is that no strict ordering must be enforced for events processed within such epochs/windows.

Although the epoch-based ordering approach reduces the overhead of deterministic execution while still offering a system to recover precisely, the approach is not always applicable if replication is used. Consider for example a replicated join operator that outputs multiple join partners based on a user-specified key for some window. Since the input sequence for both replicas can differ due to the absence of a strict ordering, join partners may be identified at different stages within the join algorithm. Although the set of identified join partners would match for both replicas satisfying the property of commutativity, the output order may still diverge.

An example for such a situation is shown in Figure 5.1. In the provided example, we assume a join operator which consumes events from two different sources and tries to find join partners for events among a time-based tumbling window comprising three time units. The operator is a custom written join implementation which allows to join several streams based on a set of

attributes. In the given example, events with the same key a or b are joined.

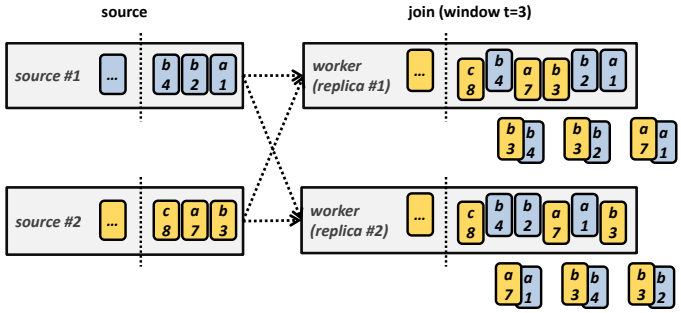

Figure 5.1: Event ordering join example.

The join is implemented using a simple nested-loop join where events from both sources are buffered in separate linked lists first, in order to find join partners once the window has closed. Since replica #1 received $(a,1)$ as a first event while replica #2 received $(b,3)$, both algorithms encounter different configurations for the traversal of lists to identify join partners. This may lead to the emission of the following sequence of pairs for replica #1 $\{(a,1,7),(b,2,3),(b,4,3)\}$ while replica #2 may emit $\{(b,2,3),(b,4,3),(a,1,7)\}$. Although both sets are identical with regards to their contents, events are still output in different orders.

One way to filter out duplicates for replicated operators is the use of a timestamp vector where timestamps serve as identifiers to detect and discard duplicates. However, this mechanism cannot be applied if the epoch-based ordering approach is used in conjunction with replication since two replicas may emit different events sharing the same timestamps as identifiers.

An alternative to the timestamp vector-based filtering approach is to disregard events based on channels they originate from rather than timestamps, i.e., the output of the second replica is completely ignored. However, this approach restricts migration mechanisms and recovery to window/epoch boundaries. Moreover, a precise recovery cannot be guaranteed anymore since partial output of one replica may have already propagated through the system which cannot be reverted anymore.

In the following, we will present an epoch-based deterministic merge approach which does not require a strict ordering of events, however, provides complete deterministic input for actively replicated operators.

5.2 Epoch-based Deterministic Merge

Our approach is driven by the observation that commutative operators do not require a strict ordering of events in order to produce correct results, however, if replication is used, the input sequence for all replicas must be identical in order to produce identical results in *identical*

order.

The key idea of our approach is to order the sets of events received from each upstream channel prior passing them to the operator code instead of ordering every single event. However, this requires to postpone the processing of events until an epoch has been closed. Although this introduces end-to-end latency based on the chosen epoch size, it does not negatively impact throughput as events comprising future epochs can be collected in parallel in a pipelining fashion.

Figure 5.2 depicts the effect of our epoch-based merge approach with regards to the input sequence for the replicated join operator taken from the previous example. As shown in the Figure, the input sequence for both replicas is identical now providing a complete deterministic input to the user-provided operator code. As a consequence, both replicas produce identical output streams which allows a filtering of duplicates using the timestamp vector based approach.

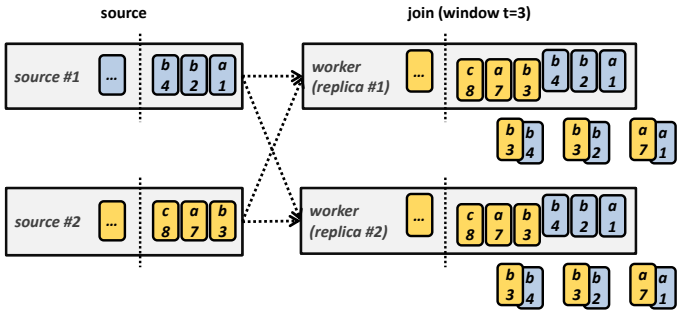

Figure 5.2: Epoch-based deterministic merge example.

In order to provide such a deterministic input, we will buffer events received from each of the channels (source #1 and 2) in separate queues for each epoch. In the following, we will call those queues *epoch-bags*, as they comprise events that belong to a specific epoch for some channel. In our example, we have the following two epoch-bags: $\{(a,1),(b,2),(b,4)\}$ for channel source #1 and $\{(b,3),(a,7),(c,8)\}$ for source #2. The two epoch-bags comprise only the events that belong to the first epoch, i.e., $ts \leq 3$. Moreover, each epoch-bag inherits the identification properties from a channel it backs up such as the operator name and its sliceId.

Once the epoch-bags for an epoch can be closed, i.e., events delivered by all upstream operators have a timestamp $ts > 3$, the epoch-bags are sorted according to their identification properties (operator name and sliceId). Since the identification properties are globally defined, the ordering of those bags is consistent across all replicas. In the provided example, the epoch-bag from source #1 will be the first one while source #2 the second one. After the ordering of the epoch-bags, the events from those bags are then passed one-by-one to the `process()` method of the user-provided operator code resulting in the following final input sequence: $\{(a,1),(b,2),(b,4),(b,3),(a,7),(c,8)\}$.

In order to provide such a deterministic input, the processing of events must be postponed until the end of an epoch in contrast to the approach presented in the previous chapter where events are processed immediately without any delays. However, the concept of epoch-bags allows us to achieve a certain degree in parallelism where the waiting time for the completion of an epoch can be overlapped with the collection of events for future epochs originating from faster channels.

5.2.1 Implementation

In a similar fashion as for the epoch-based processing approach presented in the previous Chapter 4, the epoch-based deterministic merge is executed by the *merge function* f_m and part of the sequencer component of an operator. The input for the *merge function* f_m is a set of upstream channels C. We assume first that replicated upstream operators emit the same sequences of events so that duplicates can be simply filtered using their timestamps, hence, only a single channel exists per operator partition. Note that in Section 5.3 of this chapter, we will present an extension to handle also situations where replicated upstream operator partitions may not emit identical sequences of events.

For each of the upstream channels, an epoch-bag is transparently created upon the arrival of an event. An epoch-bag comprises only the events that belong to that specific epoch. Once the events for an epoch-bag have been processed, an epoch-bag is transparently purged to free memory. Note that multiple epoch-bags that belong to the same channel can co-exist in case other channels are delivering events more slowly delaying the closure of the current epoch.

Listing 5.1 depicts a simplified version for our epoch-based merge algorithm: As mentioned previously, we maintain a set of epoch-bags for each channel. The set is implemented as linked list to preserve the order for epochs. Each list holding epoch-bags is added to an ordered hash-map called *epochBagsChannelsMap*. We use an ordered map in order to iterate in a fixed order over all channels to ensure a deterministic input for all replicas. For the keys of the hash-map, we use a pair comprising the operator name and sliceId of the respective channel.

The algorithm is implemented using a simple while loop as depicted in Listing 5.1 where an event is taken from any upstream channel as soon as new data is available (Line 2). In a first step, we retrieve the linked list of epoch-bags for the channel the event was consumed from (Line 3). Next, we use a simple boolean flag to track if we were able to append the event to the last epoch-bag or if it is too far from the future which requires the creation of a new epoch-bag. The boolean flag is set to false first indicating that no epoch-bag has been found for insertion yet (Line 4).

In a next step, we check first if there is at least one epoch-bag (Lines 5-9) and insert the event in the last epoch-bag but only if the event's timestamp is less than or equal to the timestamp of current epoch boundary (Line 7). In case the event was inserted (Line 8), the boolean

Program Listing 5.1 Epoch-based deterministic merge.

```
1:  while true do
2:      event ←anyReadyUpstreamChannel().poll()
3:      epochBags ← epochBagsChannelsMap[event.channel]
4:      inserted ←False
5:      if epochBags.size > 0 then
6:          epochBag ← epochBags.last()
7:          if event.ts ≤ epochBag.boundary then
8:              epochBag.append(event)
9:              inserted ←True
10:     if inserted ≠ True then
11:         newEpochBag.append(event)
12:         epochBags.append(newEpochBag)
13:     epochIsClosed ←True
14:     for each epochBags in epochBagsChannelsMap do
15:         if epochBags.size()< 2 then
16:             epochIsClosed ←False
17:             break
18:     if epochIsClosed = True then
19:         for each epochBags in epochBagsChannelsMap do
20:             for each event in epochBags.front() do
21:                 PROCESS(event,state)
22:             epochBags.pop()
```

variable (*inserted*) is set to true (Line 9). In situations where no matching epoch-bag has been found (Line 10), a new epoch-bag is created, the event inserted (Line 11) and the bag finally appended to the end of the linked list for the channel the event originated from (Line 12).

While the first part of the algorithm handled the insertion of events into the appropriate epoch-bags, the second part will evaluate if an epoch can be closed, and if so, it will pass the events from the bags to the `process()` method of the user-provided operator code. In order to check if an epoch can be closed, we first set a boolean variable optimistically to true (Line 13). Next, we iterate over the epochBagsChannelsMap and check if for any of the channels, there exist less than two epoch-bags (Line 15). If that is the case, it indicates that we are still awaiting events for those channels, hence, cannot close the current epoch and immediately terminate the iteration (Line 17).

In case the current epoch can be closed since every channel contains already more than one epoch-bag, we iterate over the channels (Lines 19-22), take the first epoch-bag (Line 21) and pass its events to the `process()` method of the user-provided operator code.

If we compare the two pseudo-code listings (Listing 4.1 and 5.1) from the approach present in the previous Chapter 4 and the epoch-based deterministic merge of this chapter, we can quickly identify the fundamental differences: In the previous chapter, we checked if an event

belongs to the current epoch, and if so, we passed it immediately on for processing (Lines 4-5 in Listing 4.1). Only events which are ahead of time are buffered for future consumption (Line 7 in Listing 4.1), whereas in this chapter, all events are added to buffers first (Lines 3-12 in Listing 5.1) and only passed on for processing if the current epoch has been closed successfully (Lines 14-22 in Listing 5.1) .

5.3 Light-weight Consensus-Based Deterministic Merge

In the previous section, we have seen an approach to achieve determinism for commutative operators without enforcing a strict event order. Determinism in this approach is achieved by applying an **epoch-based deterministic merge** algorithm which ensures that replicated operators receive identical sequences of events as input. In case the user-provided operator code behaves deterministic, operators produce deterministic output which comes with the benefit of replay-ability required to recover from a system failure in a precise manner. In addition to replay-ability, a deterministic output also allows a reliable filtering of events if replication is used at downstream operators.

Although the approach provides complete determinism, it has a few limitations: First, since the deterministic merge relies on the output of a *fixed set of replicas* (i.e., those with the lowest sliceUId per partition which is used to order entries in the epochBagsChannelsMap), the approach can suffer from *stragglers* which in turn can negatively impact throughput and latency. Second, there might be situations where an upstream operator may not be able to produce a deterministic output at all such as when the upstream operator is an external source.

In the following, we will present an extension to the epoch-based deterministic merge algorithm which overcomes those limitations, i.e., does not rely on a specific set of epoch-bags and stops the propagation of non-determinism in the system by utilizing a light-weight consensus protocol which ensures consistency across operators partitions.

Figure 5.3 depicts an example for the propagation of non-determinism if a topology employs replicated operators producing non-deterministic output. In the given example, we assume a simple topology consisting of three operators: Two upstream operators where one of them is replicated (source) providing the input for a join operator. Although both source replicas produce identical sets of events, the output still differs with respect to its order: While replica #1 emits $\{(a,1),(b,2),(b,4)\}$ as output, replica #2 produces a slightly different sequence of events consisting of $\{(b,2),(a,1),(b,4)\}$.

The inconsistency in output across replicas of the source operator can have several reasons: One reason may be if the user opted to use the epoch-based processing approach as presented in Chapter 4 where events are immediately processed and emitted, lowering the overhead and providing low latency. However, the operator may have produced more than a single event as output per epoch leading to such divergence. Or, if the topology consumes events from

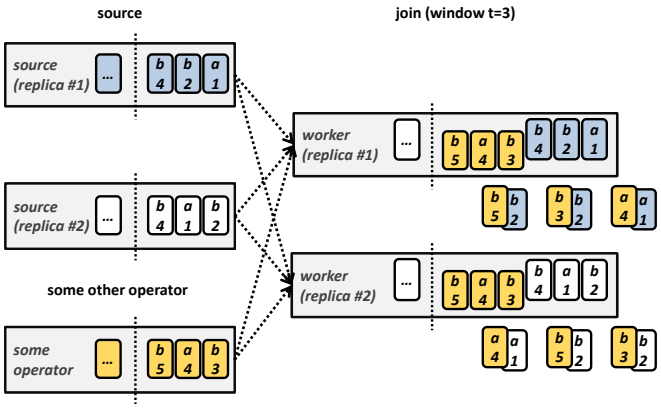

Figure 5.3: Replicated source with non-deterministic output example.

some sensor network where sensor readings can be consumed from two different network endpoints, providing a set of identical measurements, however, in different orders.

Since both source replicas emit different sequences of events, downstream operators can only avoid processing duplicates by solely accepting the output of one of the replicas and ignoring completely the output of all other replicas. However, if the downstream operator is replicated, the replicas of the downstream operator need to agree on the set of events to ignore and accept from the replicated upstream source. If no such agreement is implemented, replicas may end up receiving different sequences of input events as depicted in Figure 5.3: In our example, no agreement has been established between the two replicas of the join operator: While worker replica #1 decided upon the acceptance of output coming from source replica #1, worker replica #2 consumes the output originating from source replica #2 in addition to the events coming from the other upstream operator. Hence, the two worker replicas encounter different sets of input sequences such as $\{(a,1),(b,2),(b,4),(b,3),(a,4),(b,5)\}$ for worker replica #1 and $\{(b,2),(a,1),(b,4),(b,3),(a,4),(b,5)\}$ for worker replica #2, respectively.

As for the implementation of the join operator, we assume a custom written implementation again based on a nested-loop join as in our previous example. Since both replicas encounter different configurations when applying the join algorithm, the output stream of join partners differs with respect to the order although the sets of identified join partners are identical. Therefore, worker replica #1 emits $\{(a,1,4),(b,2,3),(b,2,5),...\}$ while worker replica #2 $\{(b,2,3),(b,2,5),(a,1,4),...\}$, respectively.

Since the replicated join operator received a non-deterministic input, the output is consequently non-deterministic which propagates down to the next operator stage until the sink operator. This propagation can lead to wrong results if one of the following operator stages employs order sensitive operators where the correctness of the result strongly depends on the input order of events. In order to avoid such propagation of non-determinism in the system, and to decrease latency introduced by stragglers, a simple agreement can be used where all replicas agree on what output to accept prior to processing. Note that such an agreement is not

only required to achieve consistency across replicas but also when operators are partitioned.

5.3.1 Consensus Protocol

In the following, we will describe a consensus protocol that achieves agreement across replicas (and partitioned operators) if upstream operators provide non-deterministic output. The key idea of our approach is to have a leader within a group of participants who decides what output the group members may accept and what to discard. The decision is taken once all participants have completely consumed an epoch from one of the upstream replicas which translates to the closure of an epoch. As a next step, the epoch-bags with the collected events are merged using the epoch-based deterministic merged algorithm presented in Section 5.2 in order to ensure a consistent input order across all group participants. The final sequence of events is then passed to the `process()` method of the user-provided operator code for processing.

We will now provide the details for the protocol. The protocol consists of two phases:

- Phase 1: **Reporting**
- Phase 2: **Decision**

During the first phase, the reporting phase, the leader receives from all group participants *epoch-completion notifications* messages. An epoch-completion notification is sent to the leader once an epoch for a channel is complete. With every reception of such an epoch-completion notification, the leader evaluates if all group participants have received an epoch for (*i*) *any of the available replicas* and (*ii*) *for all partitions*. If that is the case, the leader notifies all group members about its selection using an *epoch-commit* message. The epoch-commit message triggers an epoch-based deterministic merge on the set of epoch-bags and the processing of events for the specific epoch by passing the final sequence of events to the `process()` method of the user-provided operator code.

An example for the execution of the protocol is depicted in Figure 5.4. In this simplified example, we assume two partitioned operators, one upstream and one downstream operator. Both operators are partitioned using a partitioning degree of three, hence, the upstream operator is split into partitions which are named *a*, *b* and *c* while the downstream one *x*, y and *z*, respectively. Since both operators are replicated, every partition has a replica-pendant named $a\prime$, $b\prime$ and $c\prime$ while the downstream one $x\prime$, $y\prime$ and $z\prime$, respectively. The operator partitions are spread across six nodes as shown in the Figure. Furthermore, the operator partitions, i.e., slices are uniquely numbered using the sliceUIds ranging from 0 up to 11.

One slice of the group is elected as the leader taking the decisions later on. For the leader election, we use the mechanism provided by Zookeeper. A leader election is initiated whenever the group membership changes due to the deployment or removal of replicas or when slices become unavailable due to system failures.

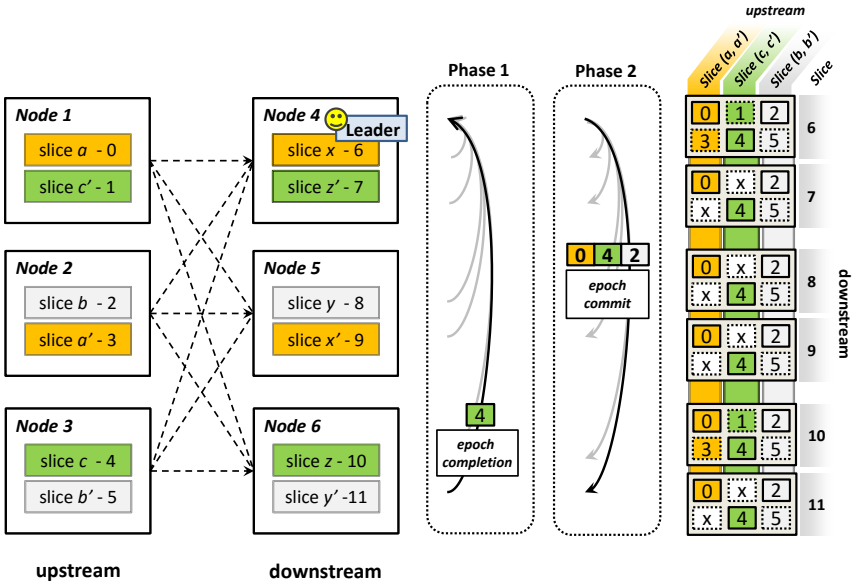

Figure 5.4: Consensus protocol example.

In order to take decisions, the leader maintains a two-dimensional matrix as depicted to the left of Figure 5.4. The x-dimension marks the different partitions (a, b and c) we received events from for an epoch while the y-dimension represents the recipient slices. In the y-dimension, we also distinguish from what specific replica a complete epoch has been received from, e.g, from a or $a\prime$ etc. An entry in the matrix can have two states, either **x** which represents that not sufficient events to complete the epoch have been received so far for the specific slice, or a concrete number, i.e., the sliceUId of the slice that delivered sufficient events to complete the epoch successfully.

As mentioned previously, the protocol is a two staged process where first all peers notify the leader about the epochs they have completed. In the shown example, slice $y\prime$ (with sliceUId #11) received sufficient messages from slice c (sliceUId #4) to complete an epoch. Hence, a message carrying sliceUId #4 is sent to the leader who then makes an appropriate entry in the matrix, i.e., in the box for the slice with sliceUId #11, for slice c. Note that the leader maintains such a matrix for each epoch, hence, matrices are filled simultaneously with epoch-completion messages in case some slices are ahead of others.

The arrival of an epoch-completion message initiates the evaluation if the current epoch can be processed in a consistent manner by all participating group members or not. The evaluation is performed on a per partition basis by computing the intersection of epochs received from upstream replicas. Consider for example the state of the matrix as shown in Figure 5.4: For partition a, which is served by two replicas a and $a\prime$, only slice 6, and 10 received sufficient events from upstream slice 0 and 3 to complete an epoch, while slice 7, 8, 9 and 11 have only received complete data from slice 0 so far. The intersection of those sets results in the acceptance for the data received from slice a (i.e, slice with sliceUId #0) rather than slice $a\prime$

(slice with sliceUId #1). In an analogous way, slice 4 and slice 2 have been selected to serve for partition *c* and *b*, respectively. In case none of the result sets is empty, i.e., for every partition, all group members have received events from at least one and the same upstream replica, group members are notified about the taken decision using the epoch-commit message finally.

5.3.2 Implementation

A simplified version for the implementation of the light-weight epoch-based consensus protocol is provided in Listing 5.2. The pseudo code shows the actions taken by the leader slice upon reception of an epoch-completion message from one of its group member operator slices identified by the **sliceUIdMbr** variable. Furthermore, the message contains the information from which of the upstream operator partitions the data has been received from (**sliceIdUpstr**), and the unique identifier of the operator slice to distinguish replicas (**sliceUIdUpstr**).

Program Listing 5.2 Light-weight epoch-based consensus protocol.

```
1:  function onEpochCompletionMessage(epoch, sliceUIdMbr, sliceIdUpstr, sliceUIdUpstr)
2:      matrix[sliceIdUpstr][sliceUIdMbr]∪sliceUIdUpstr
3:      for each sliceIdUpstr in sliceIdUpstrSet do
4:          intersection[sliceIdUpstr]←matrix[sliceIdUpstr].first()
5:          for each sliceUIdMbr in sliceUIdMbrSet do
6:              intersection[sliceIdUpstr]←computeIntersection(intersection[sliceIdUpstr],
                 matrix[sliceIdUpstr][sliceUIdMbr])
7:      finalSet← ∅
8:      for each sliceIdUpstr in sliceIdUpstrSet do
9:          if intersection[sliceIdUpstr]=∅ then exit
10:         finalSet←finalSet∪intersection[sliceIdUpstr]
11:     broadcastEpochCommitMessageToAllMembers(finalSet)
```

As mentioned previously, the leader maintains a matrix to track acknowledgments (i.e., the epoch-completion messages) received from its group members. The matrix is carried out as a two-dimensional hash-map with **sliceIdUpstr** as first dimension, **sliceUIdMbr** as second, and a set of values holding the **sliceUIdUpstr**. The **sliceIdUpstr** represents the columns in the matrix shown in Figure 5.4 while the rows are represented by the **sliceUIdMbr**. In order to distinguish the upstream replicas, a set is used representing the boxes holding the sliceUIds within each row and column of the depicted matrix in Figure 5.4.

The *sliceIdUpstrSet* set contains all sliceIds of the upstream operators while the *sliceUIdMbrSet* set holds sliceUIds of all participating group members.

In a first step, the **sliceUIdUpstr** is added to the appropriate position and sliceUId-set in the matrix indicating that a slice with identifier **sliceUIdMbr** has sent an epoch-completion-message for data it received from an upstream partition identified via **sliceIdUpstr** and replica **sliceUIdUpstr** (Line 2).

As mentioned previously, with every reception of such a message, the matrix is being evaluated if progress can be made, i.e., if all member received for each upstream partition (identified by sliceId) sufficient identical information. In order to do so, we compute the intersection for all those sets in the two dimensional matrix on a per partition basis using a one dimensional hash-map (Lines 3-6). This step is identical to looking at the matrix in Figure 5.4 and determining the intersection of numbers for each column.

As a final step, we create a final set by first checking if each of those result sets, i.e., the intersections, are not empty (Line 9), and if so, we add the selection to the final set (Line 10) which is then sent to all members to initiate the processing of the epoch (Line 11). Note that if any of the partitions contains an empty (intersection) set, the algorithm is aborted with an exit (Line 9).

5.4 Evaluation

To evaluate our approach, we implemented a canonical application operating on jumping windows. The application comprises three operators: a partitioned source operator, an equi-join that combines the output from the source operator and a sink. The experiments were executed 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 two Gigabit Ethernet (1000BaseT full-duplex) network adapters. The compute nodes for the experiments run a Debian Linux 5.0 operating system with kernel 2.6.32.

5.4.1 Scalability and Overhead

In the first set of experiments, we evaluated the performance of our approach with regards to scalability and induced overhead. The operator topology used for the measurements consists of three operators as shown in Figure 5.12: A partitioned *source* operator generating events and hence simulating several incoming event streams, a *worker* operator computing the equi-join on a jumping window, and a *sink* operator receiving the results. Note that in this experiment the sources are also replicated in order to evaluate the overhead of the consensus protocol at the worker operator stage. As discussed in Section 5.3, consensus may be needed in order to handle stragglers and to stop the propagation of non-determinism to consecutive stages. We therefore use up to 48 nodes in our experiments (24 sources, 12 workers, and 12 sinks), and the throughput values depicted in the graphs are measured at the worker nodes.

To evaluate the induced overhead for our two proposed approaches, i.e., (*i*) the **epoch-based deterministic merge** and (*ii*) the **consensus-based deterministic merge**, the experiments were executed using a set of six different implementations: In the *no-order* variant, no event order is enforced at all and hence can be considered as an upper bound for the best achievable throughput. As an extension to *no-order*, duplicates are discarded in the *no-order no-duplicates* variant by only accepting the output of one of the replicas and completely ignoring

others. Although this avoids duplicates, events originating from different upstream operators are not merged in a deterministic fashion leading to possibly incorrect results. The throughput labeled with *epoch-based processing* represents the approach and implementation described in Chapter 4 where events are assigned to epochs, however, processed immediately leading to wrong results when replication is used. The approach described in this chapter (in Section 5.2) is marked as *epoch-based det merge* in the graphs while the consensus-based variant as described in Section 5.3 as *epoch-based det merge+con* . The *deterministic execution* finally represents the strict ordering approach (as described in Section 3.6.4) which therefore represents the lower bound.

In the first experiment, we assess the horizontal scalability (scaling with an increasing number of nodes) of our approach. For this experiment, we successively increased the number of processing nodes as well as the incoming data rate to fully utilize the processing nodes. As shown in Figure 5.5, the system scales almost linearly with an increasing number of nodes. As expected, the *no-order* variant exhibits the best performance whereas the *deterministic execution*, confirming our initial beliefs, produces the most modest throughput.

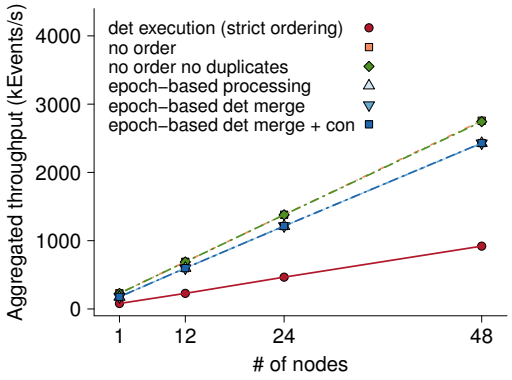

Figure 5.5: Horizontal scaling – aggregated throughput with increasing number of nodes.

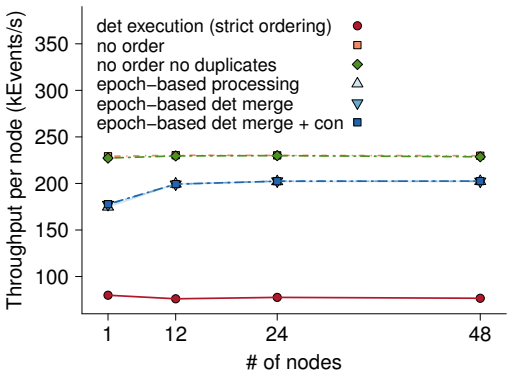

Figure 5.6: Horizontal scaling – per node throughput with increasing number of nodes.

Discarding duplicates using the *no-order no-duplicates* variant does not introduce any overhead as events are simply ignored and garbage collected. Therefore, the throughput of the *no-order no-duplicates* variant follows closely the *no-order* variant.

Although the throughput for the *epoch-based* variants is noticeable reduced in comparison to the *no-order* variants, all three variants exhibit a quite similar behavior with regards to throughput. Hence, neither the delaying of the processing of events in order to perform an epoch-based deterministic merge, nor the consensus protocol have a noticeable impact on throughput in comparison to the epoch-based processing approach as presented in the previous chapter.

Figure 5.6 depicts the per-node throughput for the same experiment. Naturally, throughput decreases slightly with an increasing number of nodes due to additional overhead as well

as more frequent contention on incoming and outgoing communication channels. In our experiment, throughput stays almost constant across the variation nodes used. However, throughput increases slightly consistently for our epoch-based variants when going from one to three nodes. This increase is due to the nature of the system where a single-node setup cannot fully utilize the system due to contention in the communication channels when sending events downstream.

To fully utilize nowadays many-core systems, we have also evaluated the vertical scalability of our approach: For the evaluation, we varied the number of processing threads available in the thread pool of StreamMine3G to match the number of available processing cores of the underlying multi-core system. In Figure 5.7, the impact on throughput with increasing number of available processing threads/cores in a single node is shown. Throughput increases almost linearly with an increasing number of available processing threads/cores for all epoch-based and no-order variants. Nevertheless, the deterministic execution exhibits a different behavior as the throughput increases only up to four processing threads/cores. This behavior is a consequence of the strict ordering where threads content on the locks of the priority queue used for ordering limiting the overall throughput.

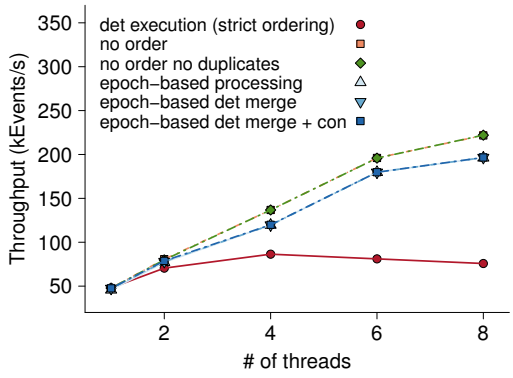

Figure 5.7: Vertical scaling – throughput with increasing number of threads.

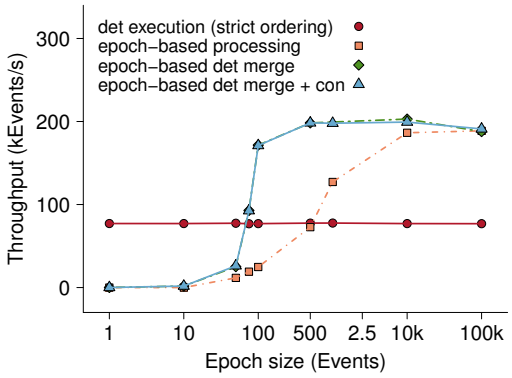

Figure 5.8: Event throughput with increasing epoch size.

In general, the epoch-based deterministic merge exhibits a better performance compared to a strict ordering of events. This gain is mainly due to the saved computational resources required to order events as well as lower contention on shared data structures used for synchronization. Hence, if we decrease the sizes of the epochs, synchronization will happen more frequently which will result in a decrease in throughput. Eventually, the performance will even drop below the performance of strict ordering (deterministic execution).

To evaluate this effect, we also investigated the throughput with varying epoch sizes. The results are depicted in Figure 5.8. Regardless of the epoch size, the throughput for deterministic execution stays constant. This is due to the ordering of every single event prior processing it while in the epoch-based variant, overhead is mainly dominated by (i) determining epochs boundaries, (ii) inserting events into epochs and agreeing on a final set for the epoch-bags.

As expected, the *epoch-based deterministic merge* variants exhibit almost the same performance with regards to the chosen epoch sizes, i.e., they reach the maximum throughput if a window/epoch consists of at least 500 events. However, *epoch-based processing* only performs best with significant larger epoch sizes. This slight drift is due to the immediate processing of events which increases contention on the data structures to check if epochs can be closed or not, which therefore has a negative impact on throughput as shown in Figure 5.8.

Note that although the horizontal axis in Figure 5.8 is logarithmic, the vertical is still linear. As a consequence, choosing an adequate epoch size is not hard. For very small epochs, the system can default to the strict ordering approach. For the second half of the graph, which represents the largest portion of the domain, the results are better than the ones obtained with the strict ordering approach. For the mid ground, we believe that simple adaptive approaches can tune the system dynamically (e.g., the system could periodically alternate between full determinism and the epoch-based approach).

The epoch-based deterministic merge approach requires delaying the processing of an epoch until it is complete/closed. This naturally introduces latency in comparison to an instantaneous processing of events as in no-order. We therefore measured also the introduced latency for the different implementations. The outcome of the measurements is shown in Figure 5.9.

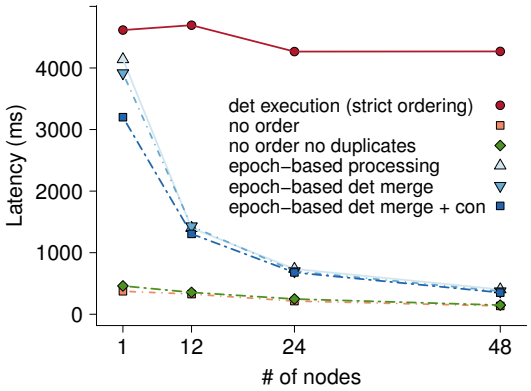

Figure 5.9: Latency with increasing number of nodes.

As expected and depicted in the graph, the latency for the no-order variants represent a lower bound. Interestingly, latency for the deterministic execution, i.e., strict ordering stays constantly high regardless of the number of nodes used, while the epoch-based variants exhibit a lower latency when more nodes are used. We account this effect to (i) the simultaneous collection of events for future epochs and to (ii) an overload of the system in the strict ordering case. Hence, imbalances in event production at upstream nodes do not necessarily increase latency for the epoch-based approach, i.e., once the last events arrives to close an epoch, the processing can be immediately started while in strict ordering, future events are only appended at the end of queues where the actual ordering only happens once the events moved up to the head, hence increases latency.

Since windowed operators expose a different behavior with regards to the number of events emitted per window, we executed two additional experiments investigating the impact on throughput when (i) varying the number of events emitted in a window, and (ii) the relative throughput gain for those operators in comparison with strict event ordering. For the first experiment, we define the output ratio as follows: A ratio less than one translates to some windows not emitting any event. For example, for an output ration of 0.5, a single event will be emitted only with every second window.

Figure 5.10 depicts the outcome of the first experiment described above. With strict event ordering, throughput constantly decreases with increasing event emission per window. This decrease is mainly due to the overhead of the *sequencer* component at the next stage which is responsible for ordering events prior passing them to the operator code of the sink operator. For the remaining variants, event throughput stays almost constant during the experiment.

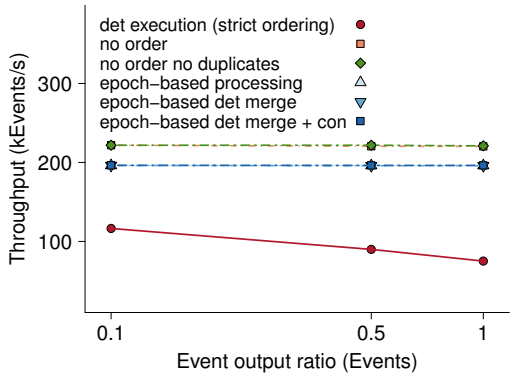

Figure 5.10: Event throughput with varying output ratio. A ration of 0.5 means only a single event is emitted for every two input events.

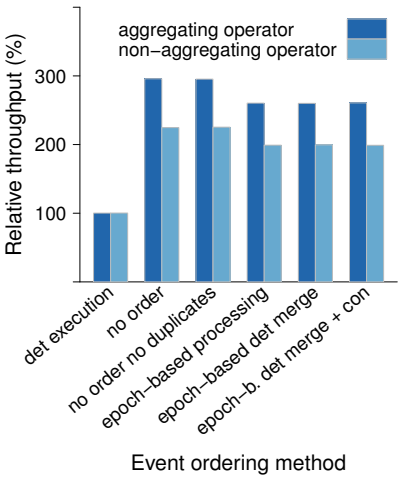

Figure 5.11: Relative event throughput gain with the epoch based deterministic merge approach for different operator types.

Finally, Figure 5.11 summarizes the previous performance experiments by depicting the overall performance gain when using the epoch-based variants for aggregating operators, i.e., with an output rate lower than the input rate as it is the case for e.g. a moving average operator as well as for non-aggregating operators, i.e., input and output rates are the same, such as in operator for classification or enrichment of events. Our epoch-based deterministic merge and consensus approach targets aggregation operators. Nevertheless, as it can be seen in the graph of Figure 5.11, it can also be applied to non-aggregating operators.

5.4.2 Failure Experiments

The last experiment evaluates our approach in the case of a node failure. Figure 5.12 depicts the experimental setup and an overview of the throughput in some nodes. The setup consists of nine physical machines and three operators, a source, worker and sink operator. Each operator is partitioned into three partitions (*a*, *b* and *c*) where each partition is also replicated with a replication factor of two. As before, we associate a unique id with each slice (i.e., sliceUId). The worker slices receive events from all source slices and produce events which are then sent to all sink slices in a broadcast manner. The replicas are spread across the physical machines in an interleaving manner. Therefore, a crash of one physical node can be tolerated without service disruption. In this experiment, *Node 4* has been crashed after around 30 seconds in the execution and restarted by redeploying a slice on a spare node shortly after.

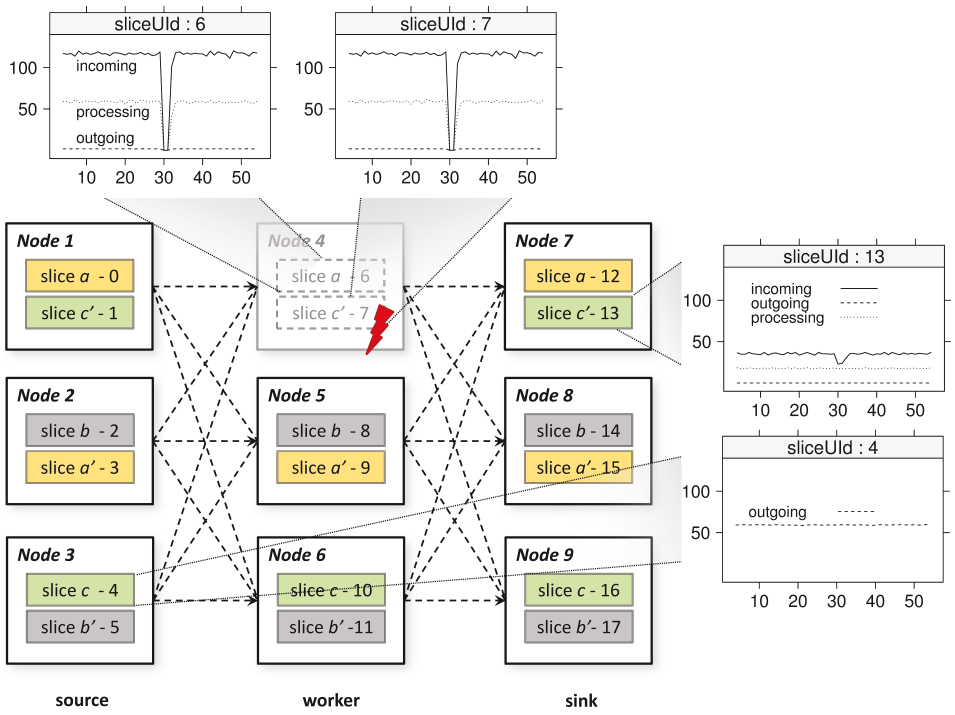

Figure 5.12: Crash and restart of node hosting leader after 30 seconds.

Note that the incoming rate of slices 6, 7, and 13 is twice as high as the processing rate (i.e., the throughput) for those slices due to replication. However, even though *Node 4* has been unavailable for some time, no throughput decrease in event processing is noticeable at the sink slice 13: Only the incoming event rate decreases during the time when *Node 4* and, hence, slices 6, 7 were unavailable.

Figure 5.13 depicts the *worst-case* latency. In this experiment, the node hosting the leader crashed after around 40 seconds. This failure included an epoch boundary. A crash of the

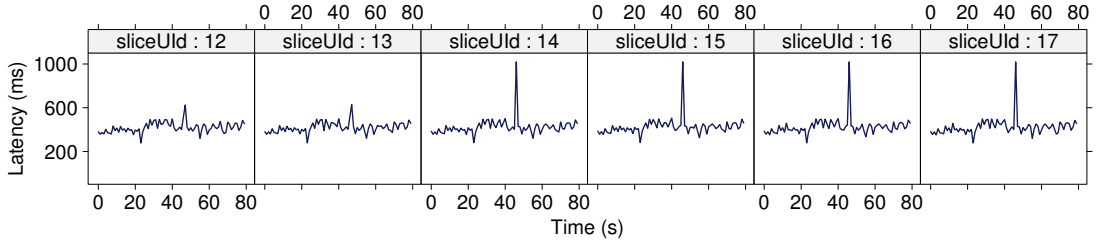

Figure 5.13: Latency over time and crash after 30 seconds.

leader requires a leader election and the retransmission of epoch-completion notification messages (as described in Section 5.3). As a consequence, latency increases as the completion and processing of an epoch is delayed. For this setup we used a custom failure detector for dedicated machines in a local area network. This failure detector has an average detection time of 14 ms. In scenarios that consider other system architectures and, especially, in distributed scenarios where perfect failure detection can be much more expensive, the detection time should be added to these values in order to derive the expected worst case scenario.

5.5 Related Work

In this section, we provide an overview about related work that is based on active replication:

Authors in [BBMS05] propose an approach that uses active replication, however, with no strong consistency guarantees. In the work, users can trade availability with consistency by having operators emit *tentative* results (and hence non-correct results) if an upstream replica is temporarily not available which would otherwise prevent making progress when applying the deterministic merge. In contrast to this work, our solution provides strong consistency as only final events are emitted and a deterministic merge is executed at all times.

There is several work such as [BBMS05, AS00, SHB04] which applies a strict ordering of events in order to avoid a costly atomic broadcast to ensure consistency for actively replicated operators: In [SHB04], authors implemented a deterministic merge component through an explicit operator (Ex-Cons) which performs a strict deterministic merge whereas our approach follows the pattern as we presented in the previous Chapter 4 where the merge is performed by a merge function hidden from the user.

An approach that offers fault tolerance using active replication, however, deviates from a strict ordering is the work presented in [HCZ07]. In this work, authors use a *punctuation*-based approach similar as the notion of an epoch as we propose here. However, in [HCZ07], replicas may emit events in different orders which imposes a non-negligible overhead to reliably filter out duplicates. Our approach, however, ensures that replicas emit identical sequences of output so that downstream operators can simply filter redundant content using the proposed timestamp vector associated with state.

An alternative to the processing of individual events is the work presented by [ZDL+12] which introduces the notion of discretized streams where small batches of events are used for processing. The approach is similar to our epoch-based deterministic merge approach as it also delays the processing of event batches but simplifies fault tolerance on the other hand. However, the approach in [ZDL+12] is strictly tied to the Spark/MapReduce programming model [ZCF+10] with no stateful operator support whereas our approach targets stateful operators and allows users to form arbitrary operator topologies.

An approach similar to our light-weight consensus-based deterministic merge is the work presented in [Pow91, DTT99] and commonly referred as *semi-active replication* in literature. In semi-active replication, a leader entity proposes the order events/messages to be processed by its followers. However, in our approach we use the leader-follower approach to decide on a *common set of messages* to be processed *by data partitioned and replicated* operators rather than proposing an actual event order for replicas. In our case, the order of messages is already defined through the application of the epoch-based deterministic merge where the globally defined channel identifiers serve as ordering criterion. Moreover, StreamMine3G uses TCP as transport mechanism that guarantees FIFO order when delivering messages, hence, no ordering of messages as in semi-active replication must be performed. We furthermore allow followers to send their processing results downstream which is not the case in semi-active replication.

5.6 Conclusion

In this chapter, we presented an epoch-based deterministic merge algorithm including a light-weight consensus-based deterministic merge for actively replicated operators. The merge algorithm as well as the consensus protocol provide deterministic input for operators and therefore replace an atomic broadcast at a much lower cost. Our approach is based on the observation that many operators in ESP applications are commutative. This is also the case for many applications originally written for MapReduce-based systems such as Hadoop [Had15] which are often migrated for online processing nowadays. Hence, our approach provides the same guarantees with regards to event ordering as in MapReduce during the execution of the reduce phase.

The key idea of our approach is to allow non-determinism within certain time periods, which are named epochs but ensure consistency across replicas. For non-deterministic sources and to handle stragglers, we enforce an agreement using our light-weight consensus protocol at downstream operators in order to avoid the propagation of non-determinism and to reduce latency. This agreement ensures that downstream replicas process the same sets of previously unordered events. Our evaluation results show that throughput can be more than doubled when comparing our approach using a reasonably epoch size and full deterministic.

6 Improving Resource Utilization and Availability through Active Replication

In the previous two chapters, we presented mechanisms for reducing the overhead of event ordering in ESP systems using an *epoch-based processing* and *epoch-based deterministic merge*. Event ordering serves two purposes with regards to fault tolerance: First, it allows to recover an application in a precise manner when using passive replication, and second, it is a low cost alternative for an atomic broadcast needed when replicating operators using the state machine approach [Sch90].

In this chapter, we present an approach for reducing the overhead for fault tolerance by utilizing paid but unused cloud resources for fault tolerance. Our approach is driven by the observation that ESP applications in cloud environments typically run at conservative load levels as low as 50% in order to accommodate sudden load changes such as spikes. We therefore propose a hybrid approach that combines the two fault tolerance schemes *active replication* and *passive standby* where the system runs in active replication mode as long as sufficient resources are available, i.e., under moderate load, however, switches instantly to passive standby once a peak occurs. Load spikes are transparently detected using a *priority scheduler* which performs the transitions between the two fault tolerance schemes. Furthermore, we propose an *interleaved partitioning* placement for primary and secondary operator partitions in order to evenly balance load imposed on operators during a recovery of crashed nodes.

6.1 Motivation

With the recent advent of high-end mobile devices such as blackberries, smart phones and tablets, we have witnessed a massive growth of applications, offering various services to their users. Examples for such applications range from social networking apps such as Facebook and Whatsapp to GPS tracking services such as Runtastic targeting athletics and runners. Although all of those applications run on end-user's devices, the majority of such services require communication with some back-end system in order to carry out their services. For example, the taxi service provider Uber constantly communicates the user's geo-location to its

back-end system in order to determine if an empty taxi or a potential new customer is in close proximity.

In order to run the back-end system, the service provider can either opt to use a dedicated data-center with the disadvantage of a high upfront investment and the risk of idle-ling resources in times of low system utilization, or settle on pay-as-you-go solutions such as Amazon EC2 instead. In fact, cloud computing is often the preferred choice for many of those companies as it allows an adjustment of the resource pool to the actual demand in a fine grained manner.

Although the cloud computing model allows to reduce costs in general as applications can scale up and down at almost any point in time, the provided mechanisms are often not suitable for applications that require quick expansions or contractions such as when performing data analytics in near real-time where the load of the system may suddenly change from one minute to another. For example, an application that processes data originating from a real-time data source such as Twitter fire hose, may experience a sudden load increase by several orders of magnitude whereas the acquisition of a new virtual machine in Amazon EC2 may take several minutes until fully available. In order to prevent unresponsiveness of an application for those situations, application developers have therefore to consider conservative load levels in the nodes in order to be able to accommodate such eventual short term load changes (i.e., spikes). Although suitable load levels can be determined for each application individually, rules-of-thumb are normally the guidance for application developers [Mil10] where load levels as low as 50% are often chosen.

The majority of cloud providers charge their customers based on the number of VM hours used rather than the actual consumption of CPU cycles. Hence, data processing using ESP systems in cloud environments such as Amazon EC2 is generally more expensive than processing an equal amount of data in a batching manner using the MapReduce [DG08] approach. The reason for the difference in cost is the higher amount of resources needed in order to cope with unpredictability such as sudden load changes and spikes. However, those paid resources are mostly idle-ling in times of moderate system load which is the case for the majority of time.

Another important aspect for ESP applications to consider is fault tolerance: Since unresponsiveness is generally not acceptable, the system may not only be capable of handling load spikes that may occur during the course of processing but also handle process and node crashes appropriately in order to prevent unresponsiveness. Although various fault tolerance schemes can be applied in order to provide fault tolerance in ESP systems, the only approach offering recovery at almost at an instant is *active replication*. In active replication, operators are replicated in order to mask a fault. However, active replication comes with a high price as it requires almost twice the resources, in CPU and network.

In this chapter, we propose a novel approach to improve the resource utilization in cloud environments. Our approach is driven by the observation that ESP applications in cloud environments such as Amazon EC2 typically run at conservative load levels as low as 50% in

order to cope with sudden load changes such as spikes. The key idea of our approach is to use those spare but paid resources to run replicas for each operator which provides a quick fail over in case of node or process crashes through *active replication.* However, in order to ensure responsiveness of the application in times with increased load or sudden load spikes, replicas serving solely as backup are transparently paused once the amount of resources becomes scarce. In order to keep those previously paused replicas up-to-date, the state of the primary is periodically checkpointed which therefore offers fault tolerance through *passive standby* where a recovery can take place by simply processing the events accumulated in the incoming queue since the last received state update.

6.2 Approach

In this section, we will provide implementation details of our approach that can be envisioned as a hybrid approach for fault tolerance as it combines the two schemes *active replication* and *passive standby* with the goal of increasing the availability of ESP applications, however, without incurring additional costs when running in cloud environments such as Amazon EC2.

Our approach is based on several components and concepts StreamMine3G employs and have been introduced and presented in great detail throughout Chapter 3 in this work. However, in order to better understand the interaction between those components, we will briefly summarize the functionality of each component in this chapter prior describing the interactions between them. The components and concepts used to achieve the hybrid approach are as follows:

1. Event ordering, i.e., deterministic merge
2. State management & synchronization
3. Priority scheduler
4. Operator placement using interleaved partitioning

Furthermore, we define the term *high availability unit* (HA-Unit) which consists of two replicas for an operator partition which we will call *primary* and *secondary* in the following.

6.2.1 Event Ordering & Deterministic Merge

In order to use active replication, events must be processed deterministically across all operator replicas what can only be achieved if such replicas receive identical sequences of input events. However, since an operator partition may receive events originating from several upstream partitions, and since network packets may be delayed, identical input can only be provided by performing a *deterministic merge* using a predefined scheme such as presented in Section 3.6.4 in Chapter 3.

The implementation for the deterministic merge algorithm uses a heap-based priority queue

which serves two purposes: First, it preserves an order across incoming events, and second, the queue can be used to buffer events, i.e., when setting an operator partition explicitly in *passive standby* mode where the operator may still receive and buffer events but does neither consume them nor perform any processing.

6.2.2 State Management & Synchronization

In our approach, we allow operator instances to be temporarily disabled, i.e., put in passive mode, in situations of overload where no events are processed. However, since such passive instances are still accepting events which are enqueued in the previously described priority queue, those buffered events can be quickly processed once enough resources become available again. While this mechanism works well if a load spike lasts only a few seconds which is the case in the majority of situations, there may be times where the system experiences an overload for a longer period of time. In those situations, the disabled operator instances would quickly accumulate thousands of events and eventually run out of memory. In order to prevent such situations, the state of an equivalent and active operator instance, i.e., the primary is periodically checkpointed to the passive instance, i.e., the secondary (as shown in Figure 6.1, right ②) in order to update the operator state and to allow the instance to purge accumulated events using the timestamp vector associated with the previously received state update.

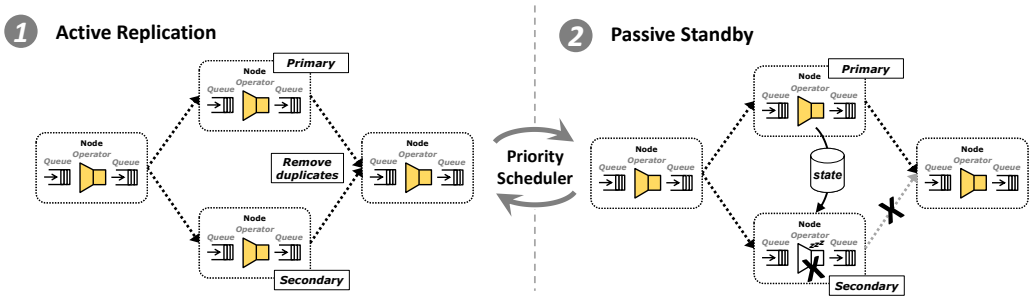

Figure 6.1: Active replication and passive standby.

6.2.3 Priority Scheduler

In order to temporarily disable operator instances in case of an overload of the system, we use a *priority scheduler* where tasks with higher priorities are always executed prior to tasks with lower ones. The priority scheduler is carried out using multiple queues where each of those task queues represents one priority. StreamMine3G allows to define an arbitrary number of priorities, hence, scheduling decisions can be executed in a very fine grained manner, however, for the approach presented in this chapter, only two priorities are required.

The priority scheduler is a component that can be considered as a proxy between the io-

service presented in Section 3.6.6 and the operator container. Hence, processing tasks issued by an operator container are first put into the queue of the priority scheduler according the operator instance's priority prior to the actual submission and execution through the io-service component.

Priorities can be assigned on either operator or operator instance level. Furthermore, priorities can be changed during runtime as we will describe later when we change the roles of a primary and secondary after a crash.

Program Listing 6.1 Priority Scheduler

```
1:  function onProcess
2:      processingCycles ← 0
3:      priority ← 0
4:      while processingCycles < cyclesMax do
5:          task ←processingTaskQueue[priority].pop()
6:          if task ≠NULL then
7:              ioService.Process(task)
8:              processingCycles ← processingCycles + 1
9:          else
10:             priority ← priority + 1
11:             if priority > priorityMax then return false
12:         if processingCycles = cyclesMax then return true
13:     return false
```

The algorithm for the priority scheduler is depicted in Listing 6.1. For the scheduler, we first define the number of processing cycles (*cyclesMax*) to execute, i.e., the number of events that should be processed at most with every time slice. In a first step, we set a cycle counter to zero (Line 2) and start processing tasks with the highest priority, i.e., zero (Line 3). To process a task, we first pop the head element from the processing task queue for the currently active priority (Line 5). In case the element reveals to be not NULL indicating that the queue is not empty (Line 6), the task is then passed to the `process()` method of the io-service (Line 7). In addition to that, the cycle counter is incremented (Line 8). In case the queue was empty, i.e., there are no more tasks scheduled for the current priority, the priority counter is incremented (Line 10). With every increase of the priority counter, we check if there are more priorities defined (Line 11) and leave the loop if this is not the case.

After processing a task, we furthermore check if still sufficient cycles are available in the current time slice. If not, we exit the loop with true (Line 12) indicating that there is more work left to be done. Otherwise, the `onProcess()` method of the scheduler is exited with false indicating that in the current time slice all events from all slices with different priorities have been served successfully.

6.2.4 Operator Placement using Interleaved Partitioning

In order to increase the availability of the system and reducing the additional load imposed on a node due to the unavailability of a recently failed node, we use a combination of *interleaved* and *chained partitioning* as proposed by authors in [HD93] to place primaries and secondaries across a set of nodes. While in chained partitioning, a primary and a secondary copy are placed on two adjacent nodes since the probability of a failure in two adjacent nodes is much lower than a failure of any two nodes, the objective of interleaved partitioning is to spread the additional load imposed due to an unavailability of a failed node across a set of nodes. An example for an operator placement using interleaved partitioning across four nodes is depicted in Figure 6.2.

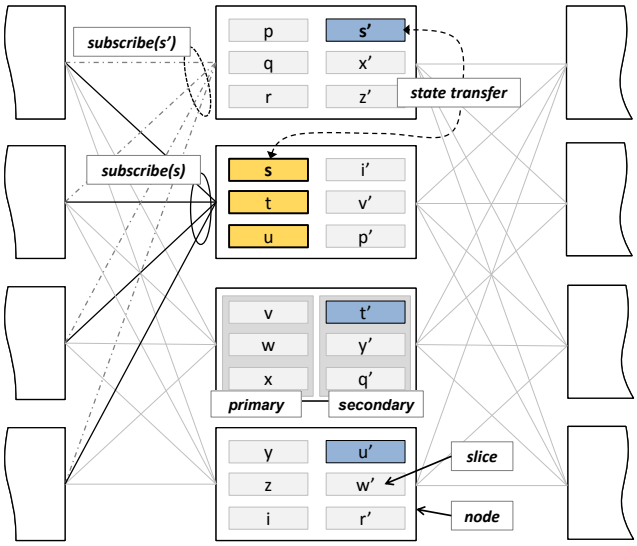

Figure 6.2: Primary and secondary slice placement.

The figure depicts four nodes and four operators which are partitioned into three partitions each: $\{p,q,r\},\{s,t,u\},\{v,w,x\}$ and $\{y,z,i\}$. Each of those operator partitions is backed up with secondaries such as $\{p\prime,q\prime,r\prime\}$ etc. As mentioned previously, interleaved partitioning reduces the load imposed on each node serving as a backup. Let's assume that in our example the second top node fails which hosted the primary slices $\{s,t,u\}$. However, since the corresponding secondary slices $s\prime$, $t\prime$ and $u\prime$ are deployed on the first, third and fourth node from top, the work performed originally by the second top node is now taken over by three nodes, i.e., increasing the load for each node only by 1/3. Using a more fine grained partitioning, i.e., a higher number of partitions, the load for each node can even be reduced further. As a rule-of-thumb, the number of partitions for an operator should match the maximal number of available physical or virtual nodes. This not only allows to scale out the system to the maximum number of nodes available in a data-center (as mentioned in Section 3.8 using StreamMine3G's migration capabilities) but also spreads the load evenly across available nodes.

6.2.5 Interaction Between the Components

In the following, we will describe the interaction of the mechanisms such as the event ordering, priority scheduler, state synchronization and the previously proposed operator placement in order to carry out the low cost fault tolerance based on active replication and passive standby.

As shown in Figure 6.2, each node hosts a set of primary and secondary slices. We assign the primary slices a priority of 0 while secondaries are assigned a priority of 1. Using the priority scheduler, primary slices are prioritized during processing, i.e., events for secondaries are only processed if sufficient processing cycles are available. Under moderate load when sufficient processing capacity is available, both primary and secondary slices are processing events. However, in case of sudden load spikes or longer periods of increased work, the priority scheduler will only serve primaries while events arriving at the secondary are only enqueued in the incoming queue of the slice used for event ordering.

The queue status such as the number of elements in a queue for a slice is continuously reported to the manager component using heart beat messages as described in Section 3.5. Using fixed or dynamic thresholds for the number of items enqueued, and the amount of remaining main memory, the manager component can trigger a slice synchronization where the state of the primary slice is copied to the secondary. This situation is also depicted in Figure 6.2: Let's assume that after some time the first top node becomes overloaded, hence, events arriving for the secondary slices $s\prime$, $x\prime$ and $z\prime$ are not being processed. The manager component is notified about the overload situation at this node through a constantly increasing number of events being enqueued at the secondary slices in addition to the performance metrics of the node that typically indicates a 100% CPU utilization. This information can be used in order to trigger a *state synchronization* where the state of the primary slice s is copied to $s\prime$ which enables $s\prime$ to purge outdated events from the incoming queue freeing the node's main memory.

We will now consider the situation of a node crash. For this scenario, we assume that the second top node crashes which was hosting the primary slices s, t and u, and the secondary slices $i\prime$, $v\prime$, and $p\prime$ previously. The crash of the node is detected through the failure detection mechanism employed in Zookeeper which then notifies the manager component about the lost node. The manager component performs hereafter the following actions: First, the secondary slices $s\prime$, $t\prime$ and $u\prime$ of the corresponding lost primaries s, t and u are *upgraded to primaries* by changing the assigned priority levels, and second, new secondaries/backups are deployed for those lost slices in order to tolerate subsequent failures.

6.3 Evaluation

We will now present the results of various experiments we performed in order to evaluate the scalability and effectiveness of our proposed solution and its implementation.

6.3.1 Experimental Setup

In total, we implemented four applications on top of StreamMine3G for our evaluation: (1) SLA (service level agreement) conformance monitoring, where the application monitors if a pattern of events that indicates the completion of a service 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 commonly used for an evaluation of MapReduce-like systems.

In a first step, we want to evaluate how the system behaves with sudden load variations that are typically found in the forthcoming applications as they monitor highly dynamic data originating from systems such as social networks or disaster detection systems. 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, state size and event distribution. The usage of the load generator allows us to study and visualize interesting cases, such as the behavior 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 utilize 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 a Gigabit Ethernet (1000BaseT full duplex). The nodes run a Debian Linux 5.0 operating system with kernel 2.6.32 and the applications have been implemented using StreamMine3G's native C++ interface.

6.3.2 Power Consumption

In the first experiment, we measure the power consumption against CPU utilization as depicted in Figure 6.3. 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 6.3, 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. However, nowadays cloud providers such as Amazon AWS also increase energy efficiency through massively over provisioning of hosts machines.

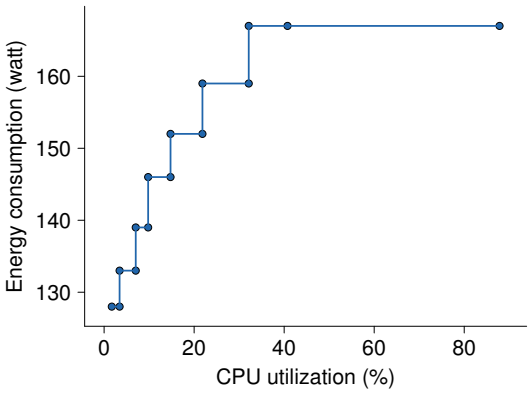

Figure 6.3: Energy consumption with increasing CPU load.

6.3.3 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 operators forming a query: a *source*, a *worker*, and a *sink*. Each operator uses 16 nodes. The source operator constantly generates events and sends those downstream to the worker operator where some computation is performed to simulate real work. The sink receives the results from the previous worker operator 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 operator.

To compare the overhead introduced by (i) the deterministic merge and (ii) the routing mechanism as described in Section 3.6.2 needed to run our approach, 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 while in *deterministic execution,* events are first enqueued and ordered using the priority queue as described in Section 3.6.4. This ordering introduces noticeable overhead as events from one node may have to wait for events from another node before they can be processed. Finally, in *active replication*, events are merged deterministically and routed using the mechanism described in Section 3.6.2. The overhead introduced by the deterministic merge and the routing mechanism in comparison to the no-oder version can be seen in Figures 6.4, 6.5 and 6.6. Note that the overhead of active replication (deterministic merge and the routing mechanisms) is negligible in comparison to the sole deterministic merge.

In addition to the introduced overhead, Figure 6.4 depicts the *horizontal* scalability of the system: with increasing number of nodes, the aggregated throughput increases. Figure 6.5 shows the per node throughput which only slightly decreases with the addition of new nodes. This

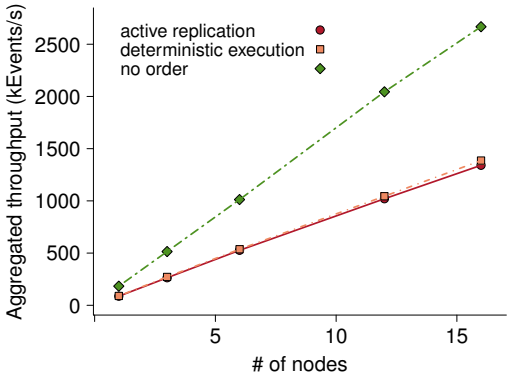

Figure 6.4: Horizontal scaling – aggregated throughput with increasing number of nodes.

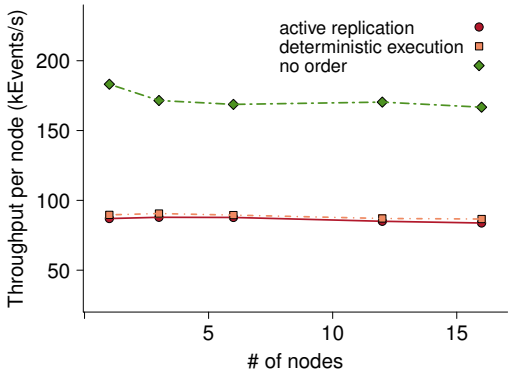

Figure 6.5: Horizontal scaling – per node throughput with increasing number of nodes.

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 routing tables that need to be inspected for each event.

Vertical scalability, i.e., scaling with the number of threads, is shown in Figure 6.6. As expected, this experiment shows that StreamMine3G can fully utilize modern multi-core processors.

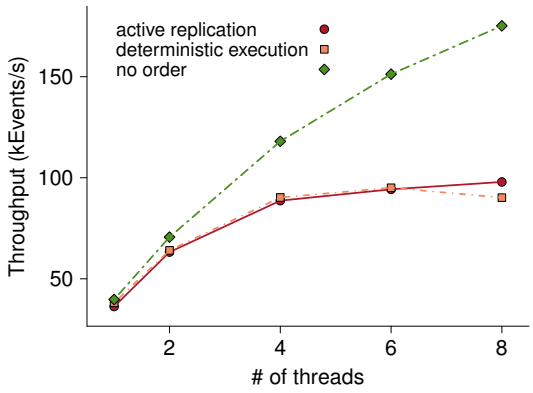

Figure 6.6: Vertical scaling – throughput with increasing number of threads.

6.3.4 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 6.7 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 two seconds.

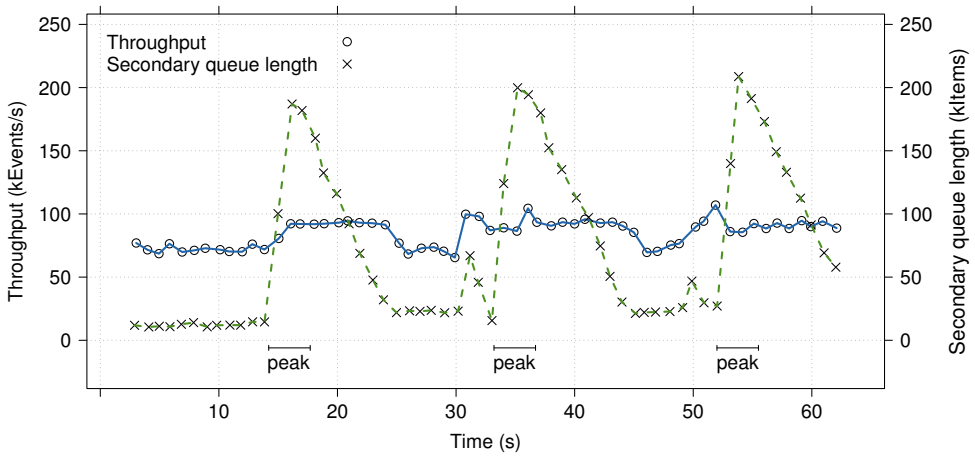

Figure 6.7: Event throughput and queue length behavior of secondary slice queues with induced load spikes.

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 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.

Next, we introduced load spikes more frequently, every 15 seconds and for four seconds. The behavior of the system is shown in Figure 6.8. More frequent and longer spikes result in shorter recovery periods. Consequently, queues of secondary slices grow with each new load peak. Once the amount of enqueued events of a primary and its corresponding secondary slice exceeds a certain threshold, a *state synchronization* action is triggered such as shown after around 43 seconds in Figure 6.8. State synchronization allows the secondary slice to prune previously enqueued events which became obsolete due to the state update. Note that 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 6.9 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, StreamMine3G will operate in *passive standby* mode until load balancing techniques such as migration re-stabilizes the system (e.g., by contracting more nodes from the cloud) and reduces the average loads to the original, safe levels.

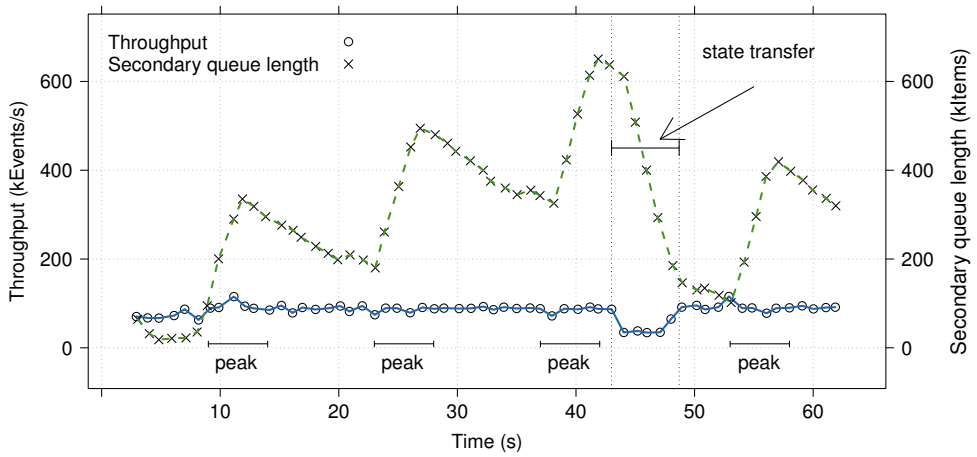

Figure 6.8: Event throughput and queue length behavior of secondary replica queues with induced load spikes and state transfer/synchronization.

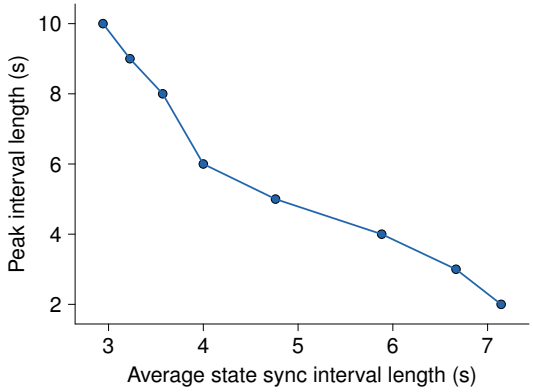

Figure 6.9: Spike lengths and state synchronization update interval length.

6.3.5 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 6.2. However, we used three nodes instead of four and only two primary and two secondary slices that run on each individual node. Each group with four graphs in Figure 6.10 and 6.11 represents one physical node. Inside a group, the two graphs on the left column depict the throughput over time of the *primary* slices, while the two on the right depict the *secondary* slices.

Figure 6.10 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, $b\prime$ and $e\prime$ are not functional anymore. The unavailability of the crashed node is detected by the manager component, hence, the previous backup slices $c\prime$ and $d\prime$ (running on

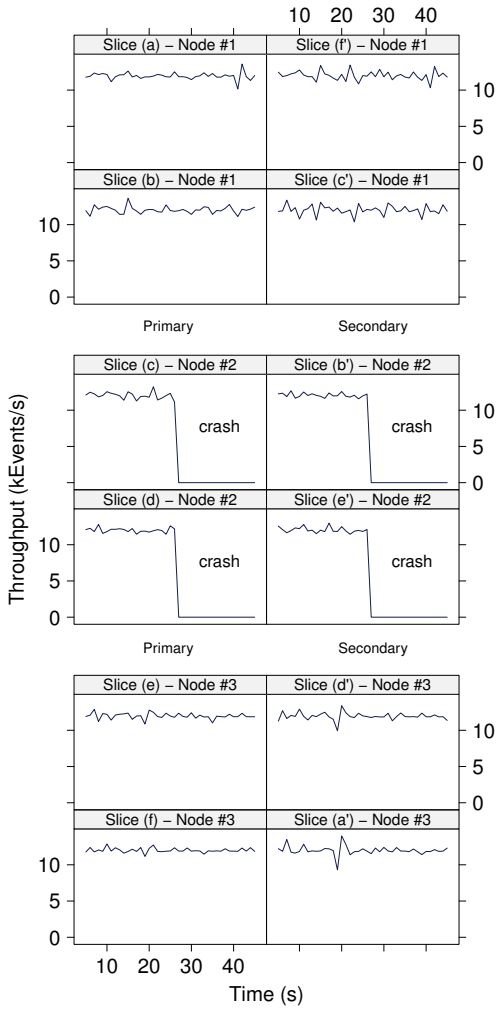

Figure 6.10: Node failure under moderate load.

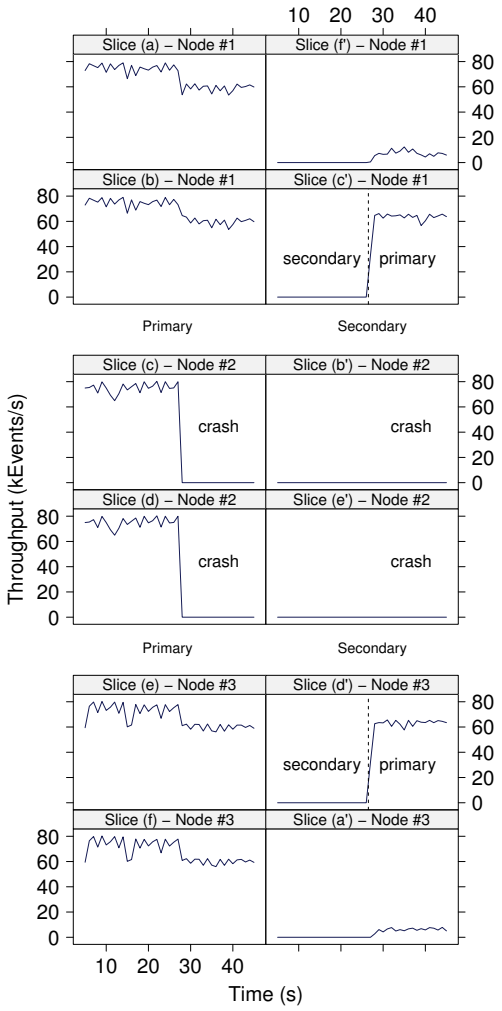

Figure 6.11: Node failure under high load.

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 6.11, the system is running constantly under high load. Therefore, secondary slices $a\prime$, $b\prime$, $c\prime$, $d\prime$, $e\prime$ and $f\prime$ have not processed events until the failure of the second node after 26 seconds. At this point, as secondary slices $c\prime$ and $d\prime$ 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.

6.4 Related Work

The idea of interleaved partitioning of operator state in online processing systems was first used in Flux [SSC+02]. 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 focuses 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.

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 [SHB04] and Borealis [HBR+05]). 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. [BFF09b] for ESP systems and by Jiménez-Peris et al. [JPPnMA00] for general distributed systems, address a scenario where multi-threading 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. [ZGY+10]. 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.

6.5 Conclusion

In this chapter, we presented an approach that transparently switches between the two fault tolerances schemes *active replication* and *passive standby* based on the availability of resources. The objective of the approach is to provide high availability without incurring additional costs by utilizing temporarily resources which have been originally reserved to accommodate sudden load spike where other load balancing techniques such as slice migration are not applicable due to large latencies. The transition between the two schemes is transparently performed using a priority scheduler which ensures that the previously reserved resources are only used for fault tolerance, i.e., replicated processing, if sufficient CPU cycles are avail-

able. In situations of an overload for longer periods of times, passive standby ensures that secondary operator slices are periodically refreshed through the state synchronization mechanism employed in STREAMMINE3G which furthermore prevents memory exhaustion as well as decreases the overall recovery time.

Our evaluation shows that a full utilization of the system through the use of redundant processing does only marginally increase energy consumption and that our system scales well with an increasing number of nodes and number of cores. Moreover, the figures show that our system transitions between to the two fault tolerance schemes ensuring that load spikes are accommodated by pausing immediately the processing of events at secondary slices. In case load spikes occur too frequently or the system is overloaded for a longer period of time, the mechanisms provided by passive standby enable the system to recover in the event of a crash.

7 Adaptive and Low Cost Fault Tolerance for Cloud Environments

While the objective of the approach presented in the previous chapter was to improve system availability by utilizing spare but already paid cloud resources through a transition between *active replication* and *passive standby* at runtime, the approach presented in this chapter will take a slightly different direction with the goal to *reduce of the overall resource consumption for fault tolerance:* Instead of switching between available fault tolerance schemes based on the system's resource availability and utilization, the system will now select a fault tolerance scheme that consumes the least resources while still ensuring a recovery of the system within the user-provided *recovery time threshold* and *recovery semantics*.

7.1 Motivation

Inspired by the simplicity of MapReduce, a number of new and open-source ESP systems have emerged and gained traction over the past three years such as Apache Samza [Sam15] (LinkedIn), Storm [Sto15] (Twitter) and S4 [NRNK10] (Yahoo!). Although all of those systems have a simple MapReduce-like interface in common, they have different guarantees when it comes to fault tolerance. For example, Apache S4 can recover from faults by restarting an operator on a new node and loading a previous checkpoint of the operator's state. However, in-flight events, which were not included in the checkpoint are simply lost, hence, only gap recovery is provided. On the contrary, Apache Storm guarantees no event loss through its transactional topologies, however, lacks appropriate mechanisms for state persistence.

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

Since the majority of ESP systems often employ only a single fault tolerance scheme, users have to choose from a pool of ESP systems rather than a pool of schemes that best matches the

requirements of the application at hand. In fact, a wide variety of fault tolerance schemes for ESP systems is known in literature ranging from active replication [SHB04, HCZ07], active or passive standby [MFB11, HBR+05], to passive replication where a combination of checkpoint and logging (i.e., upstream backup [HBR+05]) is used.

Choosing the right fault tolerance scheme is often not a trivial task as there is a trade-off between recovery time and resource overhead imposed by each scheme. For example, using active replication, an operator can recover almost instantaneously, however, at the cost of consuming twice of the resources (CPU, memory and network). On the contrary, passive replication consumes only little additional resources for state persistence (disk) and the in-memory log upstream. However, it comes with the price of a long recovery time comprising the time it takes to load the most recent checkpoint from disk and replaying events from the upstream node's in-memory log.

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

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

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

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

replication may be sufficient to satisfy the user's specified recovery time threshold, schemes providing faster recovery such as active standby must be used in times of high system load.

In this chapter, we present six different fault tolerance schemes we employed in STREAMMINE3G a user can choose from such as passive and active replication as well as intermediate alternatives such as active and passive standby. In order to free the user from the burden of choosing the right scheme for the application at hand, we propose a self-adaptive fault tolerance controller that transitions between the employed schemes during runtime based on evolution of the given workload and the user's provided constraints (acceptable recovery time and recovery semantics, i.e., gap or precise recovery). Our evaluation shows that the overall resource footprint for fault tolerance can already be reduced by 50% with a recovery time threshold of 3 $seconds$ using our adaptive scheme compared to a conservative use of active replication.

7.2 Background

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

For example, *active replication* provides the quickest possible recovery, however, at the cost of consuming twice the resources: two identical copies of the same operator are deployed and run on two different nodes, hence, redundant processing and communication is the key mechanism in order to mask a fault. Not only this scheme requires twice the processing nodes and the duplication of the events coming from upstream operators, but also, in order to produce identical results, atomic broadcast [DSU04] and deterministic execution [AS00] is required so duplicates can be reliably filtered at downstream operators imposing additional overhead onto the ESP system. In summary, active replication requires twice the CPU, network and memory resources, however, provides a nearly instantaneous recovery.

An approach that consumes the least resources, at the cost of a long recovery time, is *passive replication*. In passive replication, only a single instance of an operator runs on the ESP system. Application robustness is provided through periodic checkpoints to save the state of (stateful) operators either to a local disk or a distributed, fault-tolerant filesystem. Since ESP systems work on continuous streams of events, an in-memory log at upstream operators is used to buffer events which have been produced since the last taken checkpoint. The events in the buffer can be replayed and hence reprocessed in case a failure occurs, ensuring a gap-less recovery. Although the approach consumes only little additional resources for providing fault tolerance, such as disk space for storing checkpoints and memory for the buffered events, its recovery time comprises the loading of the most recent checkpoint, de-serializing the stored state, and the reprocessing of the events from the upstream in-memory log. Depending on the size of the checkpoint interval, replaying events can take a considerable amount of time.

Also, if state is big (e.g., in an application that holds a 24-hour data window), loading and de-serializing the state will further delay the recovery. In summary, passive replication can be considered as the opposite of active replication as it consumes considerably less resources, avoiding replicated communication and processing, at the cost of recovery time.

Besides active and passive replication, there exist several approaches which can be considered as a composition of active and passive replication. For example, in passive standby, an identical copy of the operator is deployed on the system, however, it does not perform any event processing (i.e., resides in standby mode). Instead, the replica in standby mode is used to hold a copy of the state rather than having it stored on a filesystem. For an overview of major approaches and their properties we refer the reader to Section 7.3.1, where we describe how StreamMine3G manages the available fault tolerance schemes.

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

7.3 Approach

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

7.3.1 Fault Tolerance Mechanisms

The downtime of an ESP systems comprises two components: (*i*) The time it takes to detect the failure, and *(ii)* the time it takes to execute compensation actions such as state recovery and event replay until normal operation resumes. In StreamMine3G, we rely on Zookeeper's failure detection mechanism where the detection time can be bounded through the configuration of the session timeouts and the tick time (i.e., heartbeat interval). In the remainder of this chapter we use the term recovery time only for the second component, i.e., the time it takes to execute the recovery steps excluding the detection time.

An operator in StreamMine3G is equipped with several components that contribute to fault tolerance as shown in Figure 7.1.

First, an outgoing event queue ① (i.e., upstream buffer/in-memory log) is used to log events

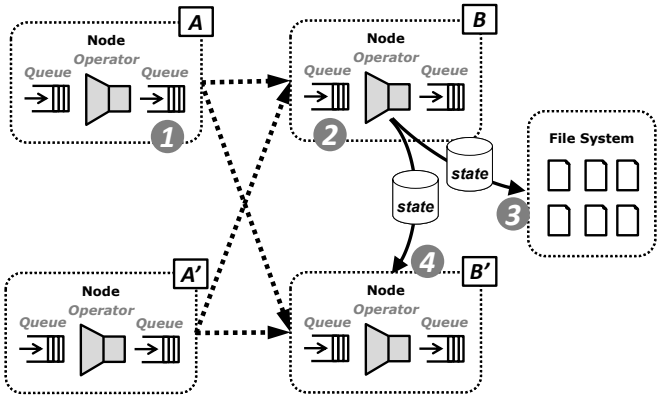

Figure 7.1: Operator components: ① Upstream/output queue for replaying events in flight (upstream backup), ② processing queue for ordering events to detect duplicates, ③ checkpoint to stable storage (filesystem), ④ state synchronization (i.e., checkpoint to peer node).

for a replay in case of a crash of a downstream operator. Of course, event replay comes only into play if the user opted for a precise recovery where event loss is unacceptable. In order to prevent memory exhaustion, the queue is purged whenever the state of the downstream operator has been included in a checkpoint successfully. STREAMMINE3G uses an acknowledgment protocol in combination with the sweeping checkpoint algorithm as described in [GZY+09].

Second, an incoming event queue ② installed at each operator instance is used for merging and ordering events coming from different upstream operators to ensure a consistent processing across replicas. Events are merged and ordered using application timestamps and a variant of the Bias algorithm as presented in [AS00].

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

While the incoming and outgoing event queues ensure that events are neither lost nor processed twice, operators often accumulate state which must be protected though appropriate mechanisms as well. A well established approach is to make checkpoints, where the state that can comprise potentially any kind of data structure is first serialized in binary form and then either written to some stable storage ③ or sent to a peer node ④ for a take over in case of a system failure.

STREAMMINE3G employs in total six different fault tolerance approaches the controller can choose from as shown in Figure 7.2. In each of the subfigures (①-⑥) in Figure 7.2, three

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

We will first start with the fault tolerance approach which guarantees the quickest recovery time, however, at the cost of consuming twice of the resources. As depicted in subfigure ① of Figure 7.2, in *active replication* replication, an operator partition is replicated where both replicas receive, process and send out events to downstream operators. In order to prevent processing events twice originating from the two replicas, the downstream operator transparently filters duplicates using the previously mentioned incoming event queue with its associated state timestamp vector. The advantage of active replication is that it can practically recover within zero seconds as a crash of either of the two replicas will not affect the downstream operator in any way.

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

Figure 7.2: Fault tolerance schemes and their impact with regards to resource consumption and recovery times.

Passive standby hot describes the state where the secondary still receives events, however, does not perform any processing saving computational resources in addition to network bandwidth. Since the secondary still receives events which are enqueued in the operator's incoming event queue, state can be safely recreated by simply reprocessing enqueued events. However, this approach would lead soon to memory exhaustion and an increased recovery time. Hence, the state from the primary is periodically included in checkpoints and these are stored at the operator's replica. We call this process *state synchronization* as it updates the secondary's state as depicted in subfigure ③. During state synchronization, the state timestamp vector is updated so outdated events can be pruned from the incoming event queue. In case the primary crashes, the following steps must be executed before resuming to normal operation:

First, the links between the secondary and its downstream operators must be established so that the results the secondary produces will arrive at its interested parties. Second, events enqueued in the incoming queue of the secondary must be processed before accepting any new events coming from upstream operators. Note that since a priority queue is used here, events are always accepted, however, newer events are transparently shifted to the end of the queue until it is their turn (as described in Section 3.7.1 of Chapter 3).

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

Keeping a copy of the operator's state in a peer node (i.e., secondary) rather than on a distributed filesystem comes with the advantage of a fast recovery as it saves the time (i) to load the checkpoint from disk and (ii) to de-serialize its data structures. However, it comes with the price of memory consumption which can be a problem if memory resources are scarce and if applications accumulate a considerable large amount of state over time. An approach to cope with this problem is to store the state on disk rather than in memory as depicted for the *deployed* and *passive replication* approach in subfigure ⑤ and ⑥. The two approaches differ in the way that in the deployed case, the internal data structures and binary code to execute the operator are already loaded and present in the system (but the state is uninitialized/virgin) while for passive replication the secondary is simply absent. Hence, the deployed state can be considered as a way of operator preloading that is beneficial for operators that rely on static data for example a lookup table which can take a considerable amount of time to be constructed during operator initialization.

Table 7.1 summarizes the different approaches with regards to actions required to be executed during a recovery.

7.3.2 Fault Tolerance Components

In order to provide adaptable fault tolerance where the system can transition between the fault tolerance approaches as presented in Figure 7.2, specific components of an operator must be enabled and disabled during runtime. In the following, we will provide an overview about the controllable components of an operator:

An operator can be set either in *sleep* or *processing mode*. In case the operator is in processing mode, events are taken out of the incoming event queue and passed to the `process()` method of the user-provided operator for processing whereas in sleep mode, the queue is not being touched and will grow with every new event arriving at the operator. Putting an operator in sleep mode, will save computational resources, i.e., CPU time which can be used for other

Fault Tolerance Schema	deploy operator	read checkpoint	wire to receive events	wire to send events	replay events*
Active Replication					
Active Standby					(✓)
Passive Standby Hot				✓	(✓)
Passive Standby Cold			✓	✓	(✓)
Deployed		✓	✓	✓	(✓)
Passive Replication	✓	✓	✓	✓	(✓)

Table 7.1: Recovery steps required to perform for each fault tolerance schema.
Note: Replay events is only needed for precise recovery.

tasks or operators running on the same node.

In order to manage network bandwidth, an operator can be enabled or disabled for *receiving* and *sending* out events. Event reception and dissemination of an operator can be controlled independently as they have semantically different outcomes: If an operator instance is not enabled to receive events, none of the upstream operator partitions will route any produced event to that specific operator instance whereas if an upstream operator is not enabled to send-out any events, none of the downstream operator instances will receive any events from that specific operator instance. Consider as an example the two operators instances A and B and their replicas $A\prime$ and $B\prime$ as depicted in Figure 7.1: Let's assume that operator instance $B\prime$ is not enabled to receive events: In this case, neither A nor $A\prime$ will route any events to $B\prime$ while B is still receiving the complete output coming from upstream. On the contrary, if operator instance $A\prime$ is not enabled to send-out any event, neither B nor $B\prime$ will receive results produced by $A\prime$, however, they will still receive results produced by operator instance A.

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

In the following, we will describe the interplay of the previously described components in order to transition between the different fault tolerance states as depicted in Figure 7.3:

First, we define the term of a *high availability unit (HA unit)*. A HA unit is an operator partition that consists of two replicas which we denote as *primary* and *secondary*. We use checkpoints[1] rather than more than two replicas as it allows us to tolerate an arbitrary number of concurrent node failures while keeping our model simple.

[1] A checkpoint comprises the operator state as well as its outgoing queue.

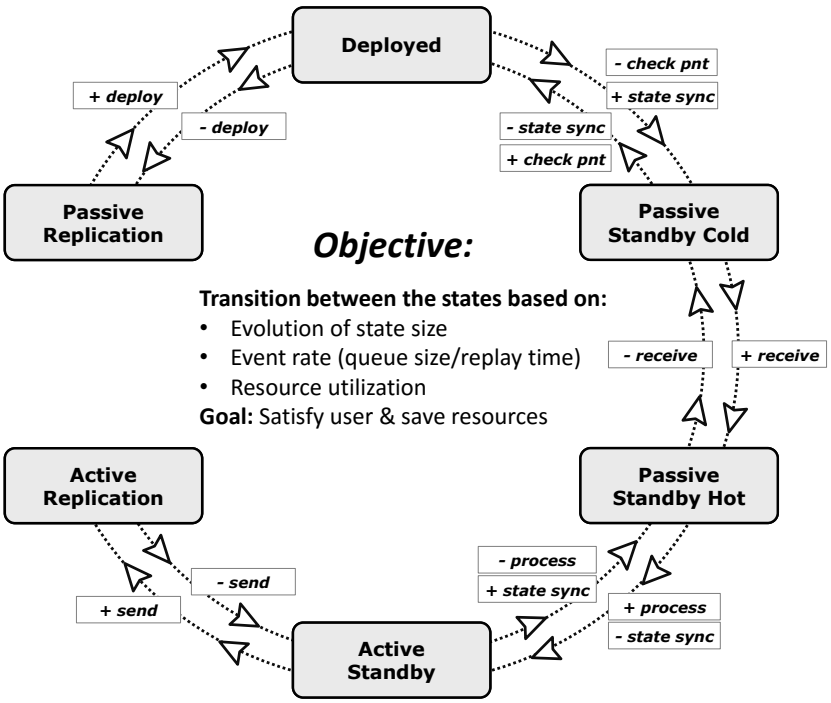

Figure 7.3: Fault tolerance schemes state transition wheel.

HA units can reside in any fault tolerant state (e.g., active replication, active standby etc.) as depicted in Figure 7.3. A transition between the states requires several actions to be taken where certain components of the secondary operator instance are enabled or disabled. For example, in order to switch from active replication to active standby, the secondary must be instructed to stop emitting events denoted as $[-send]$ in Figure 7.3, while going back to active replication requires re-enabling the secondary to emit events $[+send]$. Although Figure 7.3 shows all possible states that can be reached, we reduced the number of transitions to a minimum for better readability. Hence, in our system it is also possible to directly switch from active replication to passive standby. In fact, our fault tolerance controller employs a complete graph with all possible transitions and their required actions which allows a quick transition from one state to another.

7.3.3 Adaptive Fault Tolerance Controller

The objective of our fault tolerance controller is to choose the best fault tolerance state for each HA unit ensuring that the user-defined constraints are met at any point of time. Hence, the controller evaluates constantly the recovery time for each scheme and transitions to another state if needed. In order to prevent the system to oscillate, the controller has a cool down period where no transition to a new state is performed. Choosing the best scheme involves the following four steps the controller has to perform:

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

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

7.3.4 Recovery Time Computation

We will now describe, how we compute the recovery time for each of the schemes. Since each fault tolerance scheme requires several steps to be taken before the system can resume normal operation after a crash, the overall recovery time comprises several components as depicted in Table 7.2. For example, if a HA unit has to recover from a failure using passive replication, first a backup (secondary) operator instance must be deployed in the system which takes time t_{dep}. In addition to the deployment of the operator, the most recent checkpoint (in case the operator is stateful) must be loaded and de-serialized which is reflected by $t_{readChkpt}$. Establishing connections to upstream and downstream operator partitions as defined through the given topology can be executed in parallel. Hence, the maximum of t_{wrec} (time it takes to wire upstream operators) and t_{wsend} (time it takes to wire downstream operators) is used. In case the user opted for a precise recovery, the replay time t_{replay} is added to the overall recovery time as the last step contributing to a complete recovery.

Fault Tolerance Schema	Recovery Time
Active Replication	$t_{recover} = 0$
Active Standby	$t_{recover} = 0 + (t_{replay})$
Passive Standby Hot	$t_{recover} = t_{wsend} + (t_{replay})$
Passive Standby Cold	$t_{recover} = \max(t_{wsend}, t_{wrec}) + (t_{replay})$
Deployed	$t_{recover} = t_{readChkpt} + \max(t_{wsend}, t_{wrec}) + (t_{replay})$
Passive Replication	$t_{recover} = t_{dep} + t_{readChkpt} + \max(t_{wsend}, t_{wrec}) + (t_{replay})$

Table 7.2: Overall recovery time for each fault tolerance schema.
Note: Replay events is only needed for precise recovery.

For the time it takes to execute a certain recovery step, we use a mixture of historical collected values and an estimation-based approach as depicted in Table 7.3.

We first define a function $h(ts)$ to retrieve a historical measurement for some point in time specified by a timestamp ts. For example, the function $h_{chkptSize}(ts)$ returns the size of a checkpoint (i.e., the serialized form of an operator's state) reported at time ts. Hence, we

Recovery Step	Recovery Time
read checkpoint	$t_{readChkpt} = h_{chkptSize}(ts_{lastCkpt}) / \min(\{h_{writeChkptTP}(ts_1), \ldots, h_{wChkTP}(ts_n)\})$
replay events	$t_{replay} = \sum h_{rec}(ts_{now}), h_{rec}(ts_{now-1}), \ldots, h_{rec}(ts_{now-lastCkpt}) / h_{procTP}(ts_{now})$
wire to rec events	$t_{wrec} = \max(\{h_{wrec}(ts_1), \ldots, h_{wrec}(ts_n)\})$
wire to send ev.	$t_{wsend} = \max(\{h_{wsend}(ts_1), \ldots, h_{wsend}(ts_n)\})$
deploy operator	$t_{dep} = \max(\{h_{dep}(ts_1), \ldots, h_{dep}(ts_n)\})$

Table 7.3: Recovery time for each recovery step.

define the following functions which are instantiated per operator partition:

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

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

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

number of entries, we favor a simple estimation-based approach as described previously.

In addition to the recovery of state, events must be replayed in case the user opted for a precise rather than a gap recovery. Since the amount of events that must be replayed is strongly influenced by the checkpoint interval, event replay can take a considerable amount of time. Hence, we use the number of events received by the primary since the last checkpoint and divide it by the current processing throughput in order to retrieve an estimate for the replay step. Note that it is also possible to adjust the checkpointing interval in order to reduce the replay and recovery time. However, there is a trade-off in overhead imposed through a more frequent checkpointing and the gain in a decrease of the recovery time as we will show in the evaluation in Section 7.4.

For a refinement of the parameter estimation, advanced techniques such as Kalman filters [Kal60] or machine learning-based approaches can be used as they may result in a more positive estimation of the recovery time compared to our approach. However, in order to keep the system model simple, we left the exploration of such techniques for future work.

7.3.5 Cost Savings Adaption

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

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

7.4 Evaluation

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

7.4.1 Experimental setup

For our evaluation, we used two different applications and workloads. The first application performs a sentiment analysis using Twitter streams we collected over a period of a month. The application comprises two operators where the first one performs a simple filtering based on certain hash-tags or keywords while the second one performs a sentiment analysis and an aggregation using a sliding window of ten seconds length. The workload is depicted in Figure 7.4.

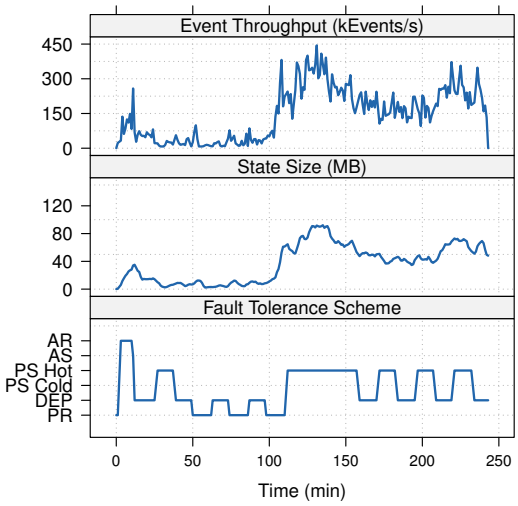

Figure 7.4: Throughput, state size and fault tolerance scheme evolution over time using Twitter workload with a recovery threshold set to 5.5 seconds.

The second application performs a short term energy consumption prediction for Smart Grids [MMBF14]. As with the first application, two operators are used where the first one performs a data conversion of the data tuples coming from smart plugs while the second one performs the short term load prediction using several sliding windows.

Both applications have in common that the query/topology consists of stateless and stateful operators, and the stateful operators use a time-based sliding window. Time-based sliding windows have the property of quickly accumulating state once the throughput rises, hence, the evolution of the state follows the pattern of the throughput as shown in Figure 7.4.

We implemented the applications in C++ to run on top of StreamMine3G's native interface. However, application developers can also use Java as their language of choice by using the

supplied Java interface wrapper. As for the environment, we performed our experiments on a 50-node cluster where each node is equipped with 2 Intel Xeon E5405 (quad core) CPUs and 8 GB of RAM. The nodes are inter-connected via Gigabit Ethernet (1000BaseT full duplex) and run an Ubuntu Linux 14.04.1 LTS operating system with kernel version 3.13.0.

7.4.2 Validation

In our first experiment, we performed a sanity check to validate our approach. We first analyzed the given Twitter workload with regards to the evolution of event throughput and state size. As mentioned previously, there is a correlation between throughput and state size for time based sliding windows where the state size follows the pattern of the throughput, as shown in Figure 7.4. Since the throughput is quite low during the first 100 seconds (around 100 kEvents/s), the amount of events being kept in the sliding window accumulates to roughly 20 MB, while with the sudden increase in throughput the state size quickly rises to 90 MB. In our experiment, we set the recovery time to 5.5 seconds since application criticality regards only user experience. However, we chose precise recovery rather than gap recovery for two reasons: first, precise recovery provides repeatability, a very useful feature for debugging distributed applications; second, because it is the safest approach it is typically the one selected. Since the state is quite small, we set the checkpoint and state synchronization interval to a rather small value of 3.5 seconds. We did not specify a cost weight vector, hence the default one is used where the approach that consumes the least CPU, network, memory and virtual machine resources is selected.

As shown in the beginning of the bottom plot in Figure 7.4, the system starts with active replication, as it is the safe choice. Once enough measurements have been collected, the controller quickly switches to the deployed scheme as the state and the throughput are quite low and, 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 schemes. The cool down time of five seconds prevents the system from oscillating due to sudden load spikes which are common in workloads originating from live data sources such as Twitter streams. In summary, the controller chose a combination of passive replication and deployed during the first half of the experiment, whereas the second half was dominated by passive hot standby.

7.4.3 Resource Overhead and Savings

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

are depicted in Figure 7.5.

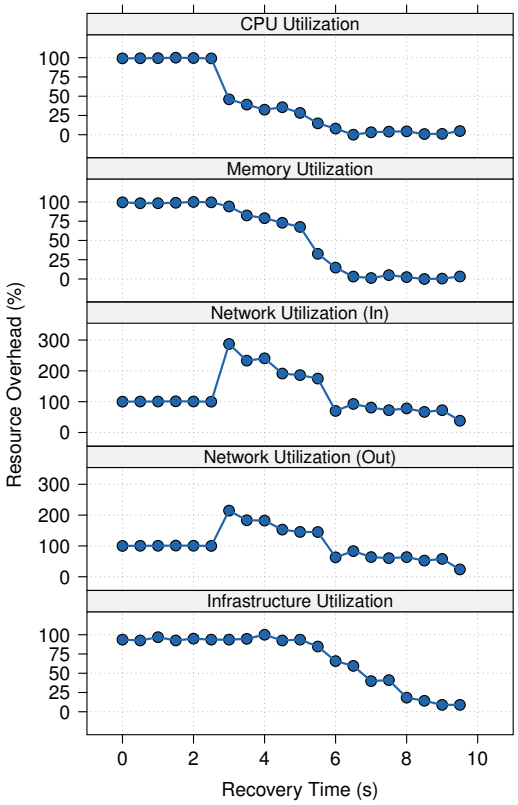

Figure 7.5: Resource overhead with varying recovery time thresholds using the All Costs cost model.

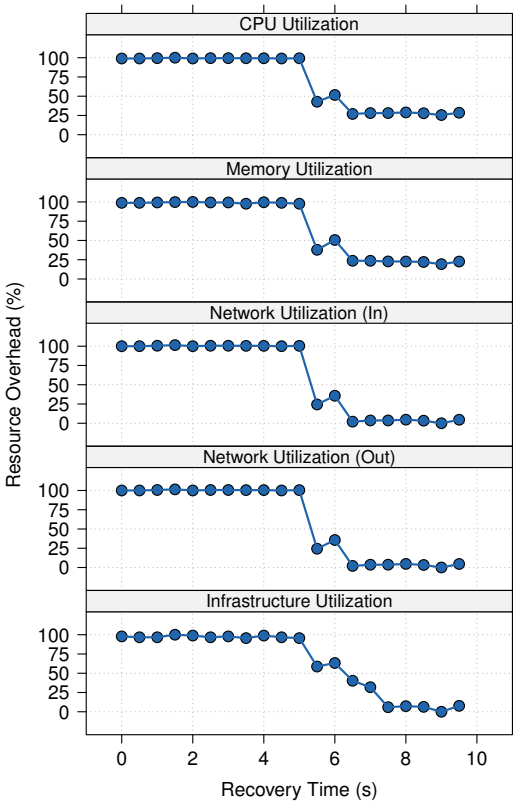

Figure 7.6: Resource overhead with varying recovery time thresholds using the Amazon EC2 cost model.

The plot in Figure 7.5 shows the overhead for CPU, memory, network (incoming and outgoing) and infrastructure utilization, where the infrastructure utilization reflects the number of virtual machines used. The resource utilization has been normalized to the execution of active replication. Hence, when the user chooses zero seconds recovery time, we can witness an 100% overhead for CPU, memory, network and infrastructure utilization as the system runs solely in active replication mode. However, with an increasing recovery threshold, resources can be saved. For example, using a recovery threshold of four seconds or more, the CPU overhead already drops to 50%, whereas the network utilization rises up to 300%. This is due to the fact that the system predominately uses hot passive standby where CPU cycles are saved due to the suspended secondary, however, at the cost of network bandwidth due to the periodic state synchronization mechanism. In fact, an overhead of 300% also indicates that the state synchronization mechanism consumes more network bandwidth than event dissemination alone. The overhead can be reduced by lowering the state synchronization frequency as we will show later.

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

7.4.4 Cost Model

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

The outcome of this modification is shown in Figure 7.6. If we compare the infrastructure utilization depicted in Figures 7.5 and 7.6, we can see that with a recovery time threshold of 5.5 seconds, the overhead (i.e., the number of required nodes) decreases faster than with the default cost weight vector, confirming the emphasis on infrastructure costs rather than on individual resources. On the other hand, the overhead for CPU, memory and network resources stays constant for recovery time thresholds of less than 5.5 seconds. This is due to the fact that the controller favors active replication as it provides a faster recovery compared to approaches that incur similar costs.

7.4.5 Relation between State Size and Resource Savings

As mentioned previously, the size of the state has a strong impact on the recovery time. A larger state requires significantly more amount of time to be loaded from disk and reconstructed in memory. Hence, in the following experiment, we extended our sentiment analysis application with an extra data field attached to each event. We then varied the event size to investigate its impact to resource savings. It is expected that applications with a relatively small state allow more potential resource savings when considering fault tolerance than applications with larger state. Figure 7.7 depicts the lower bound, i.e., the lower limit for the recovery threshold a user must choose in order to achieve resource savings.

The results reveal that regardless of the state size a recovery threshold of more than four seconds allows already saving resources since the system can then transparently switch to passive standby mode, reducing the CPU overhead. However, one has to keep in mind that with increasing state size, state synchronization can be performed less frequently which increases the number of events in upstream logs.

In order to save memory resources, users have to provide a large recovery time threshold prior saving resources. This is due to the fact that for the default cost weight vector, CPU

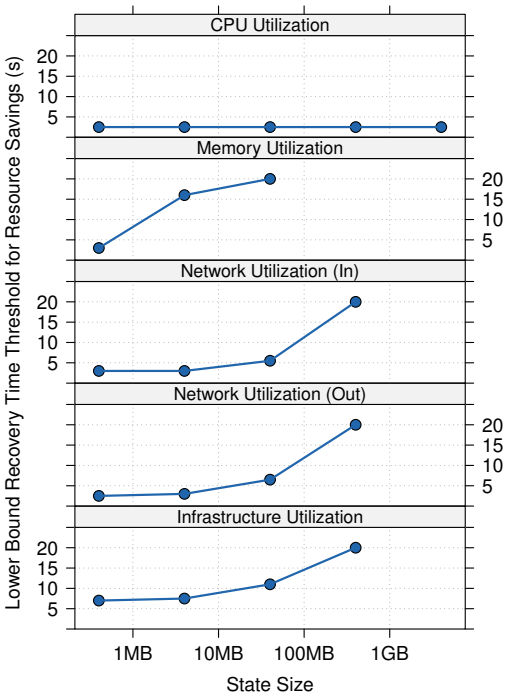

Figure 7.7: Lower bound recovery time threshold with different state sizes.

hungry schemes such as active replication and active standby are replaced with memory consuming states such as passive standby. For network and infrastructure utilization we can observe similar trends where with increasing state size, higher recovery time thresholds must be provided prior to benefit from resource savings.

Note that for state sizes of 500 MB and more, a recovery time of more than 20 seconds is needed. This case is not directly shown in the graphs, but indicated through the omitted data points for larger state sizes.

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

7.4.6 Relation between the Cost Models and the Use of Fault Tolerance Schemes

In the next experiment, we varied the recovery time threshold for different cost models to get an insight about the time the system spends in each of the schemes. As with the previous experiment, we can identify a clear correlation between the chosen cost model and the used fault tolerance scheme as shown in Figure 7.8.

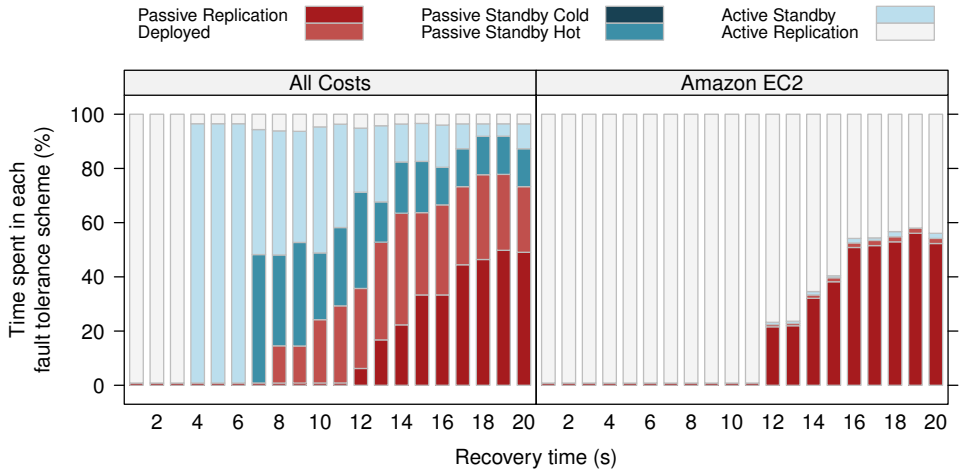

Figure 7.8: Fraction of time spent in each fault tolerance scheme with varying recovery time thresholds for different cost models.

Using the default cost weight vector, we can see that the system stays in active replication for recovery time thresholds lower than three seconds. With increased thresholds, active standby is used, and, then, schemes such as passive standby. Since the system starts always with active replication, a fraction of the time is always associated to active replication regardless of the specified recovery time threshold.

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

In addition to the analysis in what state the systems stay predominantly for each of the different recovery time thresholds, we also compared the resource consumption for the two cost models. The break down is shown in Figure 7.9 where the difference in demand for each resource is shown.

As depicted in the graph, the Amazon EC2 cost model consumes considerably more resources if the user specified more than four seconds as a recovery threshold. On the other hand, fewer nodes are acquired which is indicated by the negative bars for the infrastructure if a recovery time threshold of six or more seconds has been specified.

A summary of the break down is depicted in Figure 7.10. As expected, the Amazon EC2 cost model consumes more CPU (roughly 30%), memory and network resources while reducing the number of required nodes in comparison to the AllCost cost model.

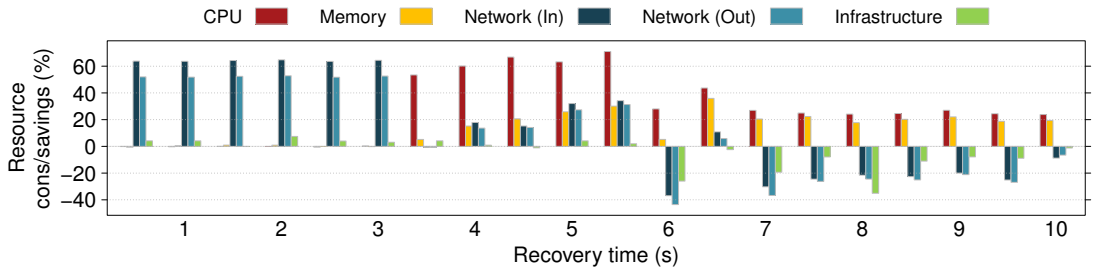

Figure 7.9: Resource consumption difference between the two cost modes. A positive value indicates that the AllCost model consumes more of the specific resource in comparison to Amazon EC2.

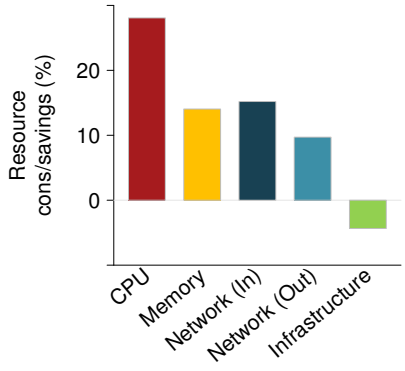

Figure 7.10: Resource consumption difference between the two cost models (mean).

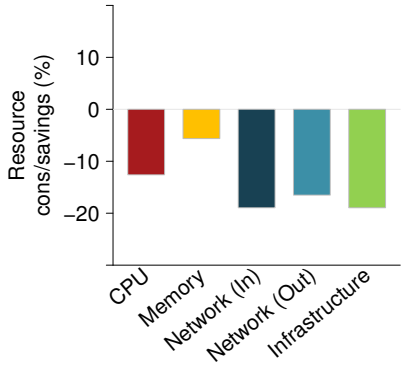

Figure 7.11: Resource consumption difference between the recovery guarantees precise and gap recovery (mean).

7.4.7 Relation between the Recovery Guarantees and the Use of Fault Tolerance Schemes

Similar to the previous experiment, we varied the recovery time threshold, but now with different recovery guarantees. Requiring only gap recovery leads to lower recovery times in comparison to precise recovery as the event replay can be omitted. As shown in Figure 7.12, requiring precise recovery keeps the system more time in active standby compared to cases in which the user opted for gap recovery. Moreover, using gap recovery, the system can already remain in passive replication when a recovery threshold of 18 or more seconds was specified.

Figure 7.11 shows the comparison of the resource consumption for the two different recovery guarantees. As expected, precise recovery consumes considerable more resources as the recovery takes per se longer due to event replay which forces the system to stay more time in more resource hungry states such as active replication consuming more resources CPU, memory, network and infrastructure wise.

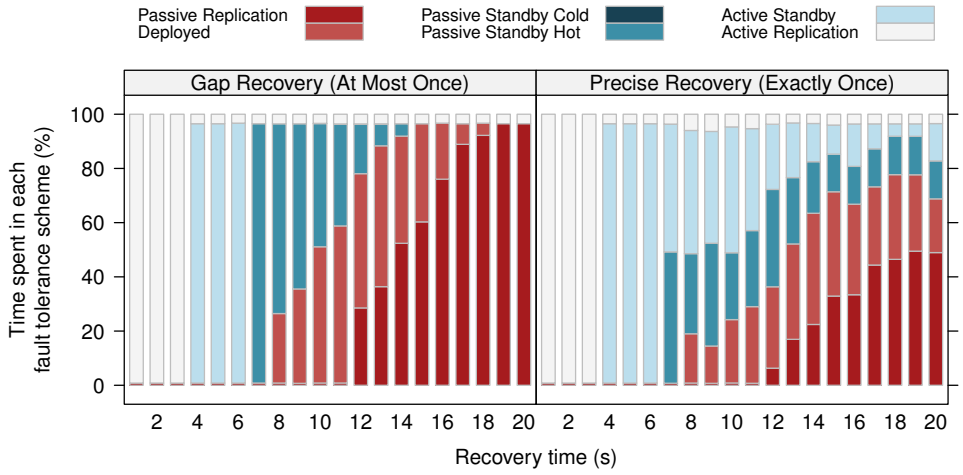

Figure 7.12: Fraction of time spent in each fault tolerance scheme with varying recovery time thresholds for different recovery semantics (gap and precise recovery).

7.4.8 CPU and Network Consumption Trade-off

In the last of our experiments, we investigated the trade-off between saved CPU resources and network resources when varying the interval for state synchronization and checkpoints. A smaller interval leads to a more up-to-date state and shorter outgoing and incoming queues, which improves recovery time, however, at the cost of network bandwidth as every state synchronization imposes additional overhead on the network.

Figure 7.13 depicts the overhead for CPU and network utilization with varying state synchronization interval. If state synchronization is performed continuously, without pauses, we can experience up to 600% peak overhead depending on the nature of the application and the state. For example, an application with average state size of 10 MB would exhibit an overhead of up to 200%. However, the savings regarding CPU are only marginal, hence, state synchronization should not occur more often than every two seconds.

7.5 Discussion

In the previous evaluation, we have seen that considerable cost savings can be achieved when adapting replication schemes at runtime. However, such an adaption-based approach comes with a cost if the experienced workload is *hostile* and *erratic*. For example, an operator state that suddenly increases from a few KBs to GBs within a few seconds will most-likely result in longer recovery times than estimated by the controller. One way to deal with such hostile workloads is the accumulation of QoS violations and a transition to less costly states only when there is still sufficient safety margin. Alternatively, traces can be used to drive a prediction-based approach in case the load fluctuations occur in fixed intervals. However, in this work, we assume workloads that change only moderately as we have seen when analyzing

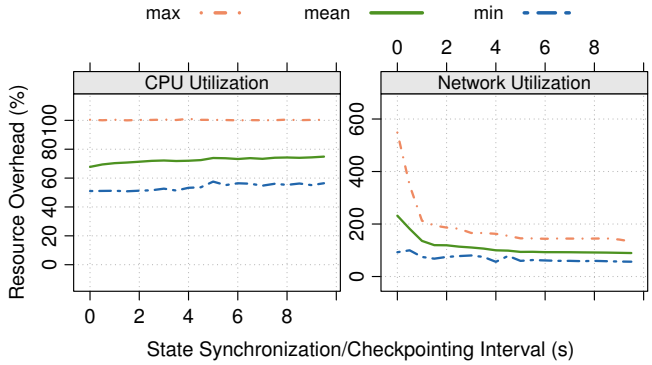

Figure 7.13: Trade-off between saved CPU and additional network traffic for different state synchronization and checkpoint intervals.

real-time Twitter streams.

The current approach uses several metrics such as disk throughput in order to estimate the time it takes to perform checkpointing and recovery. However, with new technologies such as RDMA and SSDs, access times and throughput are only a fraction and several orders of magnitude higher than traditional storage media such as HDDs, respectively. Although those new technologies have different properties, it does not affect our adaption-based approach as access times and throughput measurements are taken into account for estimating the recovery times for each of the individual replication schemes. Only the transition paths may change as now a different fault tolerance scheme may be chosen in favor of another.

In the previous chapter, we have seen an approach that switches between *active replication* and *passive standby* at runtime in order to use spare resources and to increase availability while the approach presented in this chapter switches between a larger set of fault tolerance schemes, however, with the objective of reducing the overall resource usage. Both approaches have clearly contradicting goals, however, a combination of both is still possible. Let's consider cloud providers such as Amazon that charge users on a *full hourly rate* rather than on a per minute basis. Although the controller may decide to deallocate a virtual machine at some point of to save resources, keeping the machine running until the end of the paid period is beneficial as now the approach presented in the previous chapter can be applied increasing the overall availability of the system.

Although the resource savings presented in this approach are quite remarkable, the work does not consider resource aware and fault tolerant operator placement strategies. Resource aware placement strategies such as the application of bin-packing etc. will reduce the overall resource usage while fault tolerant placement strategies may require more resources again. We leave the exploration to combine those different strategies for future work.

7.6 Related Work

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

Inspired by the schemes used in database systems, several approaches based on checkpoints and logging have been proposed, such as [HBR+05] and [GZY+09]. While Hwang et al. [HBR+05] proposes upstream backup where events are logged at upstream nodes for recovery, Gu et al. [GZY+09] combines logging with checkpoints by introducing the sweeping checkpoint algorithm. We used an adaption of the second approach, however, improved it so that the checkpoint intervals are adjustable to allow us to directly control the recovery time.

A more recent approach which does not require checkpoints is the work of Koldehofe et al. [KMR+13] where safe-points are used to track state modifications through a dependency graph. While this approach can save the overhead of doing and keeping checkpoints, which are not negligible, it is not suitable to our current operator model and API as it requires a notification mechanism to track state modification in relation to the incoming events.

A system similar to StreamMine3G is the one presented by Castro Fernandez et al. [CFMKP13] where the ESP system comes with explicit state management support, which serves for elasticity and rollback recovery at the same time. Although both systems (SEEP and StreamMine3G) share many similarities, SEEP does only support a single fault tolerance scheme while our system covers a whole range of well established schemes.

An approach which guarantees a user-specified recovery time similar to ours is presented by Balazinska et al. [BBMS05]. However, instead of switching between appropriate schemes and keeping consistency guarantees, temporarily inconsistency is introduced by forwarding only partial results due to the unavailability of upstream operator partitions.

Finally, several approaches have been proposed to combine more than a single fault tolerance scheme. However, they do so with different objectives: Authors in [ZGY+10] and the approach we presented in Chapter 6 use a combination of active replication and passive standby. However, while in the approach from the previous chapter the system runs in active replication during normal operation using spare and already paid cloud resources, Zhang et al. [ZGY+10] use passive standby and switch only in failure cases to active replication. In the approach presented in this chapter, we now combine several approaches in a single system and switch between the schemes based on the user-specified recovery time. Using the cost weight vector, our approach can also be used to harness spare resources at no additional cost for fault tolerance similar as presented in the previous chapter.

An approach which is closest to our adaptation mechanism was presented by Upadhyaya et al. [UKB11]. The authors propose an optimization algorithm which is tailored to specific operators rather than a whole query with the goal of guaranteeing a user-specified recovery time. However, in contrast to our work, the approach is not adaptive and does not consider

resource overheads.

7.7 Conclusion

In this chapter, we presented a set of six different fault tolerance schemes that is supported by our STREAMMINE3G approach. To our best knowledge, STREAMMINE3G is therefore the first ESP system that combines several fault tolerance schemes in a single system. In order to free the user from the burden of choosing the most appropriate scheme, we propose a fault-tolerance controller where users are only required to specify a recovery time threshold, a recovery guarantee (precise or gap recovery), and, optionally, a cost weight vector used for resource and cost optimization. Using the provided input, the controller will adapt and select the fault-tolerance scheme at runtime that ensures the user-specified recovery time threshold while keeping the costs for resources low. For adaption, the controller uses an estimation-based approach that utilizes performance metrics collected during the execution of the application.

8 Application Examples

In the previous chapters, we presented several approaches on how to lower the overhead of fault tolerance by either introducing a weak ordering scheme using epochs, utilizing spare resources for fault tolerance or transitioning between several fault tolerance approaches at runtime. We furthermore presented the programming model and system architecture of StreamMine3G, the ESP system we used to study, implement and evaluate the proposed solutions.

In this chapter, we provide implementation details and performance evaluations of several real word applications we implemented on top of StreamMine3G in order to assess the applicability of the proposed programming model and system architecture for a wide variety of potential applications. We first showcase how StreamMine3G can be used to detect outliers and to provide a short term load and energy consumption prediction in the context of smart grids, while in the second part of the chapter, we present an approach on how to perform a continuous analysis of taxi rides using StreamMine3G in order to identify frequently driven routes and profitable areas. Both applications have been carried out as part of the annual DEBS Grand Challenge [JZ14, JZ15].

8.1 Energy Consumption Prediction

In the following section, we will present an approach for analyzing energy consumption data in near real-time using StreamMine3G. The use case for the application is taken from the annual DEBS grand challenge [JZ14] where the application is required to provide a short term energy consumption prediction as well as to detect outliers.

8.1.1 Introduction

Smart grids is a general term to describe the usage of information technologies to increase efficiency and robustness of the power grid. The fourth edition of the DEBS Grand Challenge calls for applications centered around sensor data recordings originating from so-called smart

plugs in private households. A smart plug continuously monitors power consumption of a power outlet. Applications are provided with a continuous stream of smart plug recordings with the objective of deriving new streams that provide (i) a short term prediction of the energy consumption and (ii) detect outliers [JZ14]. A major obstacle in the course of solving this challenge is the incompleteness of the provided data stream as well as the amount of data provided, which exceeds the processing capacity of any single-node solution, requiring a distributed data processing system.

As the application is provided with a data set in the form of a continuous stream of events, and results must be produced as quickly as possible, well established approaches such as MapReduce [DG08] cannot be applied. Instead, the challenge calls for ESP applications implemented on top of ESP systems such as Storm [Sto15] and Apache S4 [NRNK10] originating from the open source domain, or commercial ones such as Esper [Esp15].

Since the challenge requires estimating future energy consumption, which can only be accomplished using historical data as reference, the ESP system to power the solution must support stateful operators. However, open source systems such as Storm [Sto15] and Apache S4 [NRNK10] provide no explicit state management, hence, users are forced to use other forms of storage, such as databases or key-value stores, for managing application state. Nevertheless, using external state management tools limits throughput, scalability and elasticity.

Commercial CEP systems such as Esper [Esp15] and SAP ESP [SAP15] provide implicit state management (at the cost of throughput) and a convenient abstraction for performing queries on a continuous stream of data using a high level query language (CQL - Continuous Query Language). However, those solutions are not well suited for solving the challenge as lookups and updates of historical data for deriving a future energy consumption are brittle to express through those CQL-based languages. Instead, the challenge calls for an ESP solution supporting explicit state management, for efficiency, as well as a MapReduce-like interface, for scalability.

8.1.2 Approach

In the following, we will describe the implementation of the challenge on top of StreamMine3G: In a nutshell, the challenge defines two queries to be addressed by the system:

1. A **load prediction query** that provides a load forecast based on current and historical load measurements according to the given prediction model, and
2. An **outlier detection query** to retrieve the ratio of plugs exceeding a certain energy consumption level.

Both queries have in common that they use the same input data set, hence, the resulting operator graph consists of three operators: **Source**, **Prediction** and **Outlier detection** where the latter ones consume the stream produced by the source operator. We use operator par-

titioning as provided by STREAMMINE3G and, as partitioning key, we use the houseId (hId) provided in the measurement records since the grouping of data within the queries is done at the level of houses. As the given data set provides only measurements for 40 different houses, the maximum scale out factor for our evaluation is limited to 40 machines running a single operator (or 120 machines in total). Note that with a custom partitioner and an additional aggregation stage, much higher scalability limits could be achieved. Current scalability levels are, however, more than enough for the job at hand.

Source Operator

The source operator acts as data converter as it consumes events through a file or network stream (binary or text) and performs a conversion to a STREAMMINE3G compatible message format. As the given input data set is provided as a single 135 GB large file, we partitioned the file to spread it across multiple cluster machines. The sources read the data through memory mapping and parse the data to generate the events. Once a record has been successfully parsed, the corresponding event will be sent downstream to the prediction and outlier detection operators for further processing.

Prediction Operator

The prediction operator is responsible for providing a load forecast based on current and historical measurements. Contrary to the source operator, the prediction operator is a stateful component keeping history in order to compute such forecasts.

Implementation of the Prediction Algorithm

As the prediction must be provided at two different levels, houses and plugs, we consider a data structure for keeping the history such that we can support both levels at no extra memory cost. We achieve this by using the plugs as our most fine granular level in a multidimensional hash-map (5-dimensional) as depicted in Figure 8.1. The hash-map is used to keep the load averages for every house, household, plug and time window. As requested by the challenge, a prediction must be provided for houses and plugs for different window sizes ranging from one to 120 $mins$. In order to speed up accesses as well as insertions into the hash-map, implementations such as `boost unordered map` [Boo15a] can be used as an alternative to the standard STL maps in C++ resulting in constant complexity rather than logarithmic.

In an overview, the prediction operator needs to provide a load prediction for a time window which is two time steps ahead in the future. A load prediction is only generated on the completion of a window and based on the load average of the current window and a median of previous window averages. Therefore, the operator performs two steps as depicted in Listing 8.1: In a first step, the history is updated using the new load measurement received (Lines 2-7), while in a second step, a load prediction is provided (Lines 9-29) if and only if a

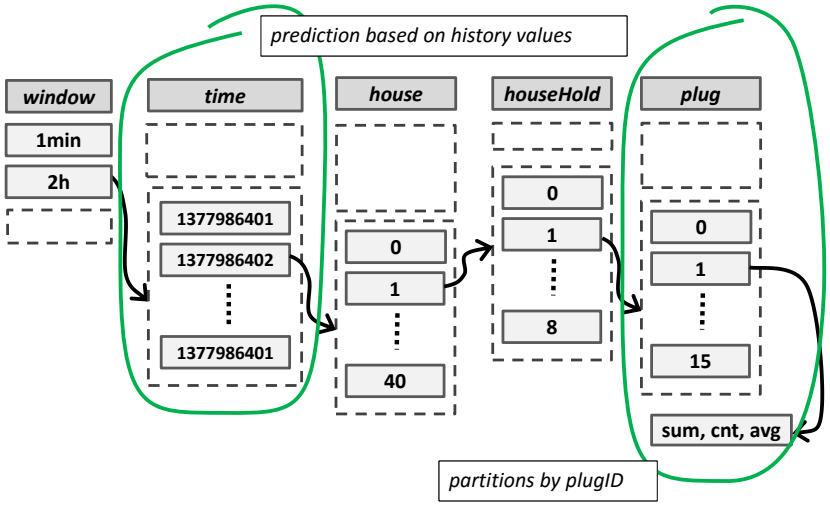

Figure 8.1: Data structure prediction operator.

window boundary has been crossed (Line 8).

In order to update the history, we first determine the time sliceId (slcId) of the current measurement (Line 2). With each arriving measurements, we update the number of measurements received so far for the current time window (Line 6), the accumulated load (Line 5) and the computed average (Line 7) for each individual window size and plug. The first two fields (cnt and val) allow us to continuously update the average upon arrival of a new measurement while the last field saves computational overhead as the precomputed average can be reused multiple times in the second step of the operator. Updating the history is performed through accessing the multidimensional map (Line 4), retrieving a plug data record and updating its values as described previously. Those updates are performed for every window size using a surrounding *foreach*-loop (Line 3).

If the boundary for a time window has been crossed (Line 8), the second part of the operator will be executed: In order to provide a prediction for every window size, the computation is performed multiple times using a surrounding *foreach*-loop (Lines 9-28). With every iteration, first the sliceId of the current time window as well as the future one used for prediction is retrieved (Lines 10 and 11). Furthermore, the number of time windows (n) within a day based on the window size (Line 12) and the number of days (k) since the beginning of the history is determined (Line 13).

The prediction is then computed for every plug (Line 17) within a household (Line 15) and house (Line 14) by first creating a set containing all load averages measured at the same time of day in the past (Lines 20-22), and computing the average between the current average and the median of this set (Line 23). If no history is available (Line 19), the prediction will solely be based on the current average (Line 25). The resulting prediction tuples are then emitted (Line 27) to some downstream operator to trigger further actions or simply for visualization

Program Listing 8.1 Prediction Operator

```
1:  function PROCESS(ms, state)
2:      slcId ← (ms.ts - initialTime) / sliceLen
3:      for each wndSize in state do
4:          plug ← timeslice[ms.hId][ms.hhId][ms.plgId][slcId]
5:          plug.val ← plug.val + ms.val
6:          plug.cnt ← plug.cnt+1
7:          plug.avg ← plug.val / plug.cnt
8:      if ms.ts < nextLdPred then return
9:      for each wndSize in state do
10:         slcId ← (ms.ts - initialTime) / wndSize.len
11:         futureSlcId ← initalTime + (slcId + 2) * sliceLen
12:         k ← dayLength / sliceLen
13:         n ←(slcId + 2) / k;
14:         for each house in timeslice do
15:             for each houshold in house do
16:                 preddHLd ← 0
17:                 for each plug in household do
18:                     cur ← plug[slcId]
19:                     if n > 0 then
20:                         prevAvgs ← new List
21:                         for i ← 1..n do
22:                             prevAvgs.add(plug[(slcId + 2) - i * k])
23:                         predPlgLd ← (cur.avg+med(prevAvgs))/2
24:                     else
25:                         predPlgLd ← cur.avg
26:                     predHLd ← predHLd + predPlgLd
27:                     EMIT({wndSize.len, futureSlcId, hId, hhId, plgId, predPlgLd})
28:             EMIT({wndSize.len, hId, predHLd})
29:     nextLdPred ← nextLdPred + 30;
```

purposes (e.g., in a dashboard).

In order to provide a prediction of the load for a house, we summarize individual plug loads (Lines 16 and 26) and emit the sum as a prediction tuple (Line 27) resulting in another output stream.

As the outgoing tuples contain the window size and an identifier (e.g., hId), they can be handled either as a single stream or as separate streams depending on the partitioner or message delivery service used in the operator or application consuming that stream.

Discussion

In order to verify the accuracy of the prediction model provided by the DEBS challenge committee [JZ14], we extended the algorithm given in Listing 8.1 by adding another field in the multidimensional hash-map to keep both, the predictions and the actual consumptions. By keeping this information, we can compare the actual consumption with the previously predicted one. This extension enables us to modify the prediction model, and to compare its accuracy with the original one provided by the challenge.

As the provided model improves its prediction accuracy by taking into account a constantly growing history, it does not consider weekly or monthly patterns. However, in western countries, work and private life are dominated by a weekly pattern that is also reflected in energy consumption: The energy consumption for an office or factories is likely to be considerably lower during weekends, if compared to weekdays. This applies also to large commercial consumers, for example, bakeries, which usually close on Sundays. We therefore extended the prediction operator to also consider those weekly and monthly patterns and adjusted the original prediction using a weighting factor.

Outlier Detection Operator

The objective of the outlier detection operator is to provide a stream of events for each house with the ratio of plugs which have a median load higher than the median of all plugs of *all* houses. Similarly to the prediction query, different window sizes must be considered: a one and a 24 hours long sliding window.

In order to keep the history needed for the medians of the two different window sizes, we use a multi-dimensional map again for maintaining the measurements at plug level, and a second map for maintaining the global measurements. The two maps are instantiated for the two windows for which an output stream must be produced. We are using two maps as we keep the measurements in an ordered sets for a fast retrieval of the median value. While this approach consumes more memory as we keep measurements at two levels (plug and global), we save computational resources needed for the sorting of the time measurements needed to determine the medians. To reduce the additional memory usage, we summarize the messages by preserving only the load values and timestamps. This approach considerably reduces memory requirements.

Contrary to the prediction operator where an infinite history must be maintained, measurements can be discarded after expiration (i.e., do not belong to the current sliding window anymore). The pseudo-code for the implementation of the outlier detection operator is depicted in Listing 8.2.

First, the measurement is included in the current time window and outdated measurements are discarded (Lines 3-11). In order to include the measurement in the time window, we perform a lookup by *plugId* (Line 3) to retrieve an ordered set of measurements and add the

Program Listing 8.2 Outlier Detection Operator

```
1:  function PROCESS(ms, state)
2:      for each wndSize in state do
3:          msSet ← wndSize.mDimMap[ms.hId][ms.hhId][ms.plgId]
4:          msSet.add(ms)
5:          wndSize.wndSizeMap.add(ms)
6:          for each ms in wndSize.glblMss do
7:              if ms has expired then delete ms
8:          for each house in wndSize.multiDimMap do
9:              for each household in house do
10:                 for each plug in household do
11:                     if ms has expired then delete ms
12:         for each house in wndSize.multiDimMap do
13:             plugs ←0;
14:             plugsGlblM ←0;
15:             for each household in house do
16:                 for each plug in household do
17:                     plugs ← plugs + 1;
18:                     if median(plug) > wndSize.glblM then
19:                         plugsGlblM ← plugsGlblM + 1
20:             ratio ← plugsGlblM / plugs
21:             if wndSize.ratio[hId] ≠ ratio then
22:                 EMIT({timeWndwEnd, hId, ratio})
23:                 wndSize.ratio[hId]← ratio
```

measurement to it (Line 4). In addition to the measurement sets on *plugId*-level, we add the measurement to the global set (Line 5) needed to determine the ratio of outliers later on.

Prior to computing the ratio of outliers, outdated measurements are removed from the global set (Lines 6-7) and from *plugId*-level (Lines 8-11) by iterating over the ordered sets and evaluating the timestamps of each measurement. In order to compute the ratio of outliers, we iterate over the individual plugs existing in each household and house (Lines 12, 15 and 16), and count the number of total plugs existing in each house (Lines 13 and 17), and the number of plugs that are above the global median (Lines 14, 18 and 19). In a last step, the ratio is computed for a house (Line 20) and emitted to downstream operators, if it changed from the previous computation (Lines 21 and 22).

The median computation itself can be performed in two different ways: Through a sorted set which exposes little computational costs, but comes with a high price for the removal of expired entries as an iteration over all entries in the window is required, or, alternatively, through a linked list, which allows an inexpensive removal of expired entries but comes with the cost of sorting all the items each time the median is needed. As the computational cost for sorting is higher than iterating over the sorted set for removing expired entries, the algorithm presented in Listing 8.2 already depicts the best performing solution.

Dealing with Incomplete Data

In this section, we describe our approach for dealing with incomplete data. As outlined by the challenge description [JZ14], load and work measurements might be missing, which is a common case in large scale data analytics. However, since work measurements are provided in addition to the load measurements where the former one accumulates over time, it is possible to derive missing load values from the accumulated work stream by computing the difference of the current work value and the last work value, when we have also seen a load measurements. In this way, we can complete the load measurement stream and improve data quality. Note that as we use only load measurements for both, the prediction and the outlier detection, we only improve the data quality of load measurements (i.e., completing the load measurement stream). Listing 8.3 shows the implementation of our data completion operator.

Program Listing 8.3 Data Completion Operator

```
1:  function PROCESS(ms, state)
2:      trace ← state[ms.hId][ms.hhId][ms.plgId]
3:      if ms.type = WORK then
4:          if ms.val > trace.lastVal and trace.lastTs != -1 then
5:              timeDiff ← ms.ts - trace.lastVal
6:              expectedLd ← workDiff * 1000 * timeDiff / 3600
7:              loadDiff ← expectedLd - trace.totalLoad
8:              if loadDiff > 0 then
9:                  EMIT({id, ts, loadDiff, LOAD, plgId, hhId, hId})
10:         trace.lastTs ← ms.ts
11:         trace.lastVal ← ms.val
12:         trace.totalLoad ← 0
13:     else
14:         trace.totalLoad ← trace.totalLoad + ms.val
15:         if ms.type = LOAD then
16:             EMIT(ms)
```

The intuition of the algorithm is as follows: We accumulate the load values with every load measurement that arrives (Line 14) since the last time we received a work measurement. Every time we receive a work measurement, we reset the counter (Line 12). Once the next work measurement arrives, we check if the amount of work changed until the last measurement in order to derive missing load measurements if needed (Line 4). If the accumulated work has changed since the last measurement, we can verify if a load measurement is missing and improve data quality the following way: We derive the load measurement by splitting the accumulated work in equal sized load consumption units (Line 6) which is then subtracted by the accumulated load from the load measurements received (Line 7), if any. If the difference is greater than zero (Line 8), then the smart meter has obviously counted more energy consumption than it was reported through the received load measurements. The difference will then be used in order to generate a synthetic measurement that will then be taken into the consideration for both the prediction and outlier detection. Note that the completion of the data stream is also

performed at the level of plugs, hence, the above algorithm will be applied to every registered plug by accessing a multi-dimensional map as in the prediction operator (Line 2).

The data completion operator can be installed at two different locations: It can be installed as a successor operator of the source operator for providing a complete data stream to the downstream operators executing the prediction and the outlier detection, or it can be tightly integrated into the prediction and outlier detection component. The former approach comes with the advantage of the reduction of data transfered to the downstream prediction and outlier detection operator while the latter one saves the extra lookup costs in the multi-dimensional hash-map. We decided for the first design as it results in a cleaner, more modular and scalable design. Through a co-location of the source operator and the data completion operator on the same physical host, we can furthermore reduce the extra cost in network traffic imposed by this modular design of utilizing a dedicated operator.

8.1.3 Evaluation

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

Experimental Setup

The experiments were performed on a 40-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) and run a Debian Linux 7.4 operating system with kernel 3.2.0. StreamMine3G and the operators needed for the queries are written in C++.

In order to evaluate the scalability regarding the number of nodes used in the system as well as for the given workloads (10, 20 and 40 houses), we partitioned the provided 135 GB large data set into 40 partitions and spread them across our 40 nodes cluster. Each partition contains the whole trace for one house. We chose this partitioning scheme as it prevents the source operator from being a potential bottleneck in the system as data will be read and parsed in parallel. In a real, live deployment the data would be coming directly from the network and, hence, could be routed appropriately, not suffering from this disk bottleneck.

In our experiments, we measured the event throughput at the worker operators (prediction or outlier detection operator) while the latency represents an end-to-end from the source operator where the event has been created after parsing until it is emitted at the worker operator. The latency also includes the overhead of deriving new measurements from incomplete data as described in Section 8.1.2.

Workload Experiments

In our first experiment, we measured the per-query throughput and latency as a function of the workload for 10, 20 and 40 houses as depicted in Figure 8.2 and 8.3.

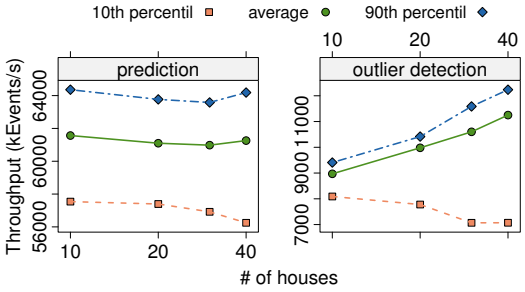

Figure 8.2: Per-query throughput as a function of the workload for 10, 20, and 40 houses (average, 10th and 90th percentile).

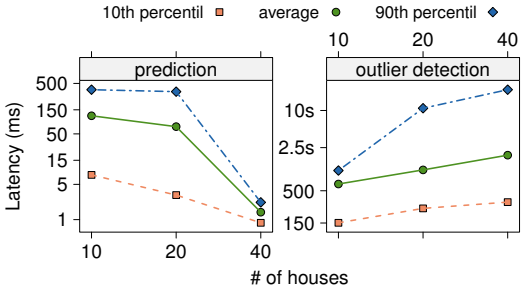

Figure 8.3: Per-query latency as a function of the workload for 10, 20, and 40 houses (average, 10th and 90th percentile).

In order to generate the required workload, we varied the number of source operators in a way that only the data partitions containing the data for the first 10, 20 or 40 houses are being taken into account while keeping the number of worker operators (i.e., the prediction or outlier detection) constant. Our experiment revealed that for the prediction query, the source operator is the bottleneck, saturating the CPU of the nodes. For the outlier-detection query, the worker operator (the outlier algorithm itself) is the bottleneck.

Because the source operator is the bottleneck in the prediction query, we see no throughput increase when increasing the number of houses in Figure 8.2 (left). At the same time, we can see a decrease in latency (left on Figure 8.3), which is due to the batching behavior in StreamMine3G, where higher volumes of events (in this case more diverse data is being consumed, i.e., the different houses) result in lower latency as events stay shorter periods of time in buffers prior to being sent over the network.

As noted above, in the outlier detection query, the worker operator (outlier detector) achieves 100% CPU utilization. Due to the extreme overload of this operator, we also experienced a high queuing behavior in StreamMine3G which accounts to a constant increase in latency as events remain longer in the incoming event queue that is constantly growing. As shown in Figure 8.3 (right), latency can go beyond seconds range and reach up to 23 seconds (peak) once a node is fully saturated.

Since events are being enqueued in the incoming event queue, we can see an increase of throughput when more houses are added to the workload as depicted in Figure 8.2 (right). Nevertheless, the net throughput for the outlier detection operator stays almost constant at around 10 kEvents/s while the remaining events end up in queues resulting in such high latencies and the slight increase in throughput with an increasing number of houses as shown in Figure 8.2 (left).

In order to prevent memory exhaustion in STREAMMINE3G when enqueuing events during a temporary or permanent overload, a built-in back-pressure mechanism kicks in once a predefined threshold has been exceeded slowing down upstream operators. The back-pressure mechanism can operate either on a fixed threshold value or dynamically based on metrics such as current memory consumption of the physical node.

Scalability Experiments

In the last set of experiments, we assessed the scalability of the system by varying the number of worker operators while keeping the number of source operators constant at 40, hence providing the whole data set to the queries. The results of the experiment are shown in Figure 8.4 and 8.5.

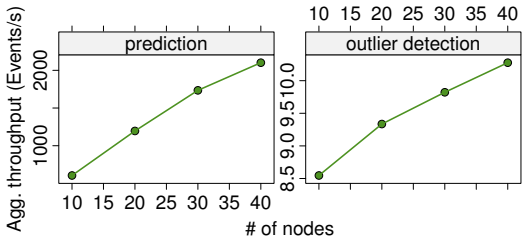

Figure 8.4: Per query throughput as a function of the number of processing nodes.

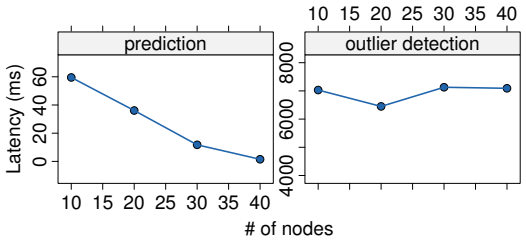

Figure 8.5: Per query latency as a function of the number of processing nodes.

In our experiment, we saw a throughput increase of roughly 42% when doubling the number of worker nodes for the prediction query while we got only a 2% increase for the outlier detection query which confirms our initial assumption about the limited scalability of the outlier detection query.

Moreover, an increase in the number of worker operators decreases latency as each partition of the worker operator (prediction) maintains only a subset of historic energy consumption profiles resulting in faster processing and lower overall latency. In contrast to the prediction operator, the outlier detection operators needs to maintain *a global view of all plugs of all houses*, hence, with the current, naïve implementation, it cannot be sub-partitioned in a similar fashion as the prediction query.

Discussion

Both query implementations have different characteristics with regards to scalability:

The first query (forecast) can be trivially partitioned though the plugId identifier as each partition is operating completely independently. By joining and aggregating the individual plug load prediction output streams, a new stream can be derived comprising the load prediction for each house.

Although the proposed partitioning scheme makes the query highly scalable with regards to additional power plugs, the current implementation will eventually suffer from memory exhaustion due to the constantly growing history. However, since the forecast operator keeps only a load average for each time slice in memory, it is possible to postpone memory exhaustion to years if host machines with a memory capacity of 8 GB for each operator partition are used (i.e., the load balancing unit). Hence, we do not consider alternative partitioning schemes for this particular query. However, other applications may accumulate larger portions of state over time where a different partitioning scheme is required in order to maintain scalability.

Contrary to the forecast query, which can scale even though it may require considerable amounts of memory due to its accumulated history, the outlier detection query implementation as presented previously cannot handle large amounts of plugs. This limitation is due to the fact that a *global median* has to be computed spanning the whole set of power plugs connected to the grid. However, since the power grid is constantly growing with each smart meter installation, it is important to design the application in a way so it can accommodate such growth.

Alternative Implementations of Outlier Detection Query

In the following, we present two alternative implementations for the outlier detection query. We first present a version that uses a distributed median and a feedback loop to determine the median value while the second alternative utilizes a simple static array to determine the median, however, requires a discretization of the values.

The first alternative achieve scalability through a *distributed median* where the load values instead of the plugIds are used as partitioning keys for the input stream. However, in order to determine correctly the median value from this distributed setup, a more complex protocol is needed that comprises several operators as depicted in Figure 8.6.

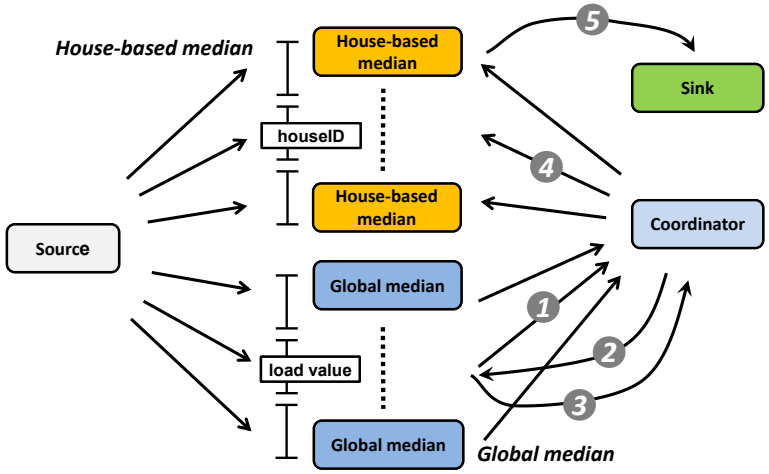

Figure 8.6: Revised outlier query operator topology with distributed median computation.

In a first step, the source stream (smart meter data) is duplicated and emitted to two different operators: A *global median* as well as the *house-based median* operator. While the input stream for the global median operator is partitioned by load values, the houseId is used as partitioning key for the house-based median operator. In addition to the load values, a multicast event is periodically sent to all operator partitions of the global median in order to trigger the computation of the global median. With the arrival of the multicast event, each global median operator partition will send the counts of load values it has received so far downstream to a coordinator operator ①. The objective of the coordinator is to determine the global median by requesting from the appropriate upstream partition the exact global median value ②. After the coordinator receives the reply, ③, it multicasts ④ the median to the house median operator triggering the evaluation of the percentage of plugs in a house exceeding the previously determined global median value. The percentage values are hereafter sent downstream to a sink operator ⑤. Note that this scheme involves a feedback loop where the global median operator and the coordinator are constantly exchanging messages.

Instead of using a distributed setup as described previously, a simple static array can be used to determine the median in an efficient way as authors have shown in [MLW+14]. In their approach, the measurements are discretized by only allowing three digits as fractions and defining an upper bound for load values. Using those constraints, a median can be efficiently maintained by counting only the occurrences of the discretized values using a static array.

In order to reduce furthermore the amount of load values to be considered, authors in [FWPG14] propose a filtering of load values if only marginal changes occur. Although such an approach reduces the amount of data needed to be processed for the median computation, it will lead incorrect results which is not permitted according to the regulations of the DEBS challenge.

8.2 Taxi Rides Analysis

In the following section, we will present an approach for analyzing geospatial data in near real-time using StreamMine3G. The use case for the application is taken from the annual DEBS grand challenge [JZ15] where the application is required to provide an always up-to-date view for the top-ten most frequently driven routes as well as most profitable areas.

8.2.1 Introduction

With the recent advent of fairly priced mobile devices and data plans, we witness an increasing amount of mobile applications that use geospatial data in order to carry out some form of service. Examples for such services range from simple dating apps where individuals can easily spot peers sharing common interests in close proximity, to services such as Uber, where customers can quickly find a nearby available taxi for a ride. Although all those applications follow a different purpose targeting a specific group of customers, they all have in common to process geospatial data in near real-time in order to carry out their tasks.

The processing of such data is often done by back-end systems on server-side rather than on the mobile device itself where the application solely acts as a client constantly sending its location updates and receiving notifications in exchange such as the availability of a taxi in close proximity if desired. Since the incoming data stream must be processed with low latency in order to provide up-to-date information to customers, such data processing tasks are often carried out using ESP systems rather than database or MapReduce [DG08] like systems.

The fifth edition of the DEBS Grand Challenge [JZ15] calls for applications centered around the processing of geospatial data originating from taxi rides. Applications in the challenge are provided with a continuous stream of records consisting of taxi rides with the objective to derive two new streams answering the following two queries: (*i*) what are the top-ten most frequently driven routes, and (*ii*) the top-ten most profitable areas?

8.2.2 Approach

In the following, we will provide implementation details of the two queries used to solve the challenge using StreamMine3G. The objective of the queries is to provide timely information for cab drivers about the **top-ten**:

1. most **frequently driven routes** within the past 30 *mins*, and the
2. most **profitable areas** within the past 15 *mins*,

where the profit for an area is defined by the median of all fares (i.e., fare + tip) for rides that started or ended in an area divided by the number of empty taxis in that area. A taxi is considered empty if it had a drop-off but no follow-up pickup within 30 minutes at its drop-off location.

Both queries have in common to provide a *top-k*, however, the *top-k* computation is only supposed to produce an output if a change occurred with regards to contents or order of elements in the *top-k* set. Hence, the operators have to track changes, either for **routes** that can be uniquely identified using the pair: $\{(x_1, y_1), (x_2, y_2)\}$ where (x_1, y_1) is the coordinate of the cell the ride departed and (x_2, y_2) the destination cell, or for **areas** which can be defined just using: $\{(x, y)\}$. Furthermore, routes and areas must be sorted according to the number of rides or profit in order to determine the desired *top-k*, respectively. Hence, the problem can be divided into two sub problems: **change tracking** and ***top-k* computation**.

In order to achieve scalability, the change tracking computation for both queries can be partitioned in the following way: For the first query, the pair: $\{(x_1, y_1), (x_2, y_2)\}$ can be used as partitioning key for the routes whereas $\{(x, y)\}$ will serve as key to partition the areas of the second query. Data partitioning serves the following two purposes: First, it allows to scale the system without bounds, i.e., an almost infinite number of routes and areas can be tracked using an arbitrary number of processing nodes (i.e., *horizontal scalability*), and second, data can be processed in parallel increasing the overall throughput of the system as well as harnessing the power of multi-core machines (i.e., *vertical scalability*).

Figure 8.7 depicts the resulting query graph for the application at hand. As mentioned previously, the change tracking and *top-k* computation can be considered as independent problems and have been therefore carried out as separate operators in each query. Furthermore, the tracking operators (routes and profitable areas) are partitioned in order to achieve horizontal and vertical scalability. However, since the application must produce a global *top-k*, only a single non-partitioned *top-k* operator instance is used in each query serving as sink as shown in Figure 8.7. In the following, we will describe the task of each operator used in the queries.

Source Operator

The purpose of the source operator is to perform a transformation of the incoming data originating from various sources such as sensors or mobile devices in a StreamMine3G compatible event format for processing. The input can be either provided via a simple network socket or a file stream.

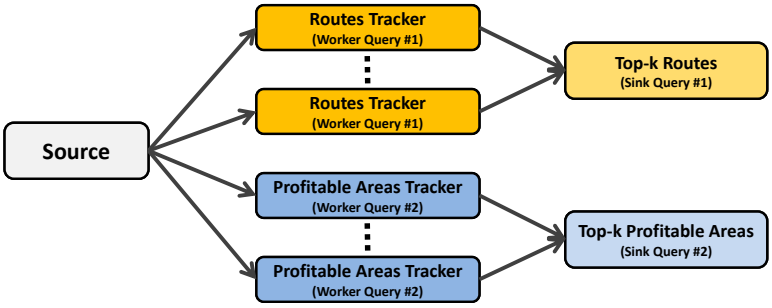

Figure 8.7: Query Graph

Since the test data set for the application is provided through a 32 GB large csv-file, the primary task of the source operator is to parse the records stored in the file and perform a data conversion to a format that can be consumed by both downstream operators, i.e., the routes and profitable areas tracker.

The conversion comprises a discretization of the pickup and drop-off location provided in latitude and longitude notation to coordinates of a predefined grid. Two grids are used, matrices sized 300×300 and 600×600 with cells measuring 500 or 250 meters in length and width for the first and second query, respectively. The grid covers an area of 22,500 square kilometers in total.

Since the two queries require different discretization for the given geographic coordinates, the source operator has to generate two output events per input event: One event with a transposition of the given coordinates to squares with 500 *meters* in length and width (query #1), and one using a more fine grained resolution with smaller squares measuring only 250 *meters* on each side (query #2). In order to avoid redundant and unnecessary network traffic, we use a custom partitioner that *selectively* routes events carrying the coarse grain discretized

coordinates only to the routes tracker operator of the first query while the remaining events are transparently routed to the profitable areas tracker operator of the second query instead.

Query #1 - Route Frequencies

In the following, we will provide details with regards to the algorithms we used in order to carry out the functionality of the tracker and *top-k* operator of the first query. Listing 8.4 depicts the pseudo-code for the routes tracker operator.

Tracker Operator

Contrary to the source operator which is implemented as a *stateless* operator as it solely performs a transformation of the input stream, a *stateful* implementation is required for the tracker operator as it operates on a 30 $mins$ sliding window. Therefore, the `process()` method is provided with a *state* object in addition to the incoming event as input as shown in Line 1 of Listing 8.4.

In a first step, the partitioning key is computed where we use a simple hash (Line 2) which comprises all four dimensions to uniquely identify a route in the given grid. Alternatively, a string can be used which however requires the usage of other hashing algorithms such as a Murmur Hash [Mum15] in order utilize data structures such as hash-maps later on.

Although data partitioning allows a system to scale, it requires synchronization mechanisms in order to ensure *semantic transparency*, i.e., the output of the partitioned execution must be equivalent to the non-partitioned one. In order to ensure semantic transparency in our application, we need to broadcast source events to all partitions of the tracker operator so that all tracker partitions are synchronously notified about a move forward in time in order to purge potentially outdated events from the predefined 30 $mins$ sliding window. However, in order to track a route only at a single partition, such notifications events must be filtered and may not be considered for tracking at that specific partition. The filtering of such notification events is performed at Line 4 in Listing 8.4 where only events are considered for tracking whose x-component matches the following equation: $x \mod slices = sliceId$ where $slices$ is the number of partitions used for the tracker operator and $sliceId$ the unique partition identifier. As an example consider the case where only two partitions are used, i.e., $slices = 2$. Using the previous equation, routes with an even x-component would be tracked by the partition with $sliceId = 0$ while odd ones by the partition with $sliceId = 1$.

In case the x-component of the event satisfies the equation in Line 4, a *Ride* object is created which is composed of the unique key to identify the route and the timestamps of the corresponding incoming event, i.e., pickup (*tsPp*) and drop-off time (*tsDf*) in Line 5. The Ride object is than appended to the tail of a linked list which is part of the state object (Line 6). The purpose of the linked list is to maintain all rides that occurred within the past 30 $mins$ in the order of occurrence where the head of the list represents the oldest ride while the tail contains

Program Listing 8.4 Query#1 Routes Tracker (Worker) Operator

```
1:  function PROCESS(rec, state)
2:      key ← rec.xPp * 300^3 + rec.yPp * 300^2 + rec.xDf * 300^1 + rec.yDf
3:      modifiedKeys ← ∅
4:      if rec.XPickup mod slices = sliceId then
5:          ride ← new Ride(rec.key,rec.tsPp,rec.tsDf)
6:          state.rides.append(ride)
7:          rRec ← state.map[key]
8:          rRec.count ← rRec.count + 1
9:          rRec.tsPp ← rec.tsPp
10:         rRec.tsDf ← rec.tsDf
11:         modifiedKeys.insert(key)
12:     if rec.tsDf ≠ state.oldTsDf then
13:         state.oldTsDf ← tsDf
14:         cutofftime ← rec.tsDf − 30 * 60
15:         while cutofftime > state.rides.front().tsDf do
16:             ride ← state.rides.front()
17:             rRec ← state.map[ride.key]
18:             rRec.count ← rRec.count − 1
19:             rRec.tsPp ← ride.tsPp
20:             rRec.tsDf ← ride.tsDf + 30 * 60
21:             state.rides.pop()
22:             modifiedKeys.insert(ride.key)
23:     for each key in modifiedKeys do
24:         rRec ← state.map[key]
25:         EMIT({rRec,sliceId}))
26:     EMIT({SILENCEPROPAGATION,rec.tsDf})
```

the most recent one. In addition to the linked list, the state object consists of a hash-map (*state.map*) which keeps track of the number of rides for each route in the grid, and what incoming event (using the timestamp pair *tsPf* and *tsDf*) caused an update to the counter for the route using the route record (*rRec*) object.

The hash-map entry is updated right after adding a new ride to the linked list. In order to do so, the corresponding route record object is retrieved from the hash-map first using the route's unique key, or it is transparently created if it did not exist before (Line 7). In a next step, the counter is incremented (Line 8) and the timestamps from the corresponding event that modified the counter are copied (Line 9-10). Finally, the key of the route that has been updated is added to a *modifiedKeys* set (Line 11) in order to notify the downstream *top-k* operator about this update later.

While the first part of the tracker operator (Line 2-11) adds information based on the incoming event, the second part of the operator (Lines 12-21) is solely responsible for updating information based on events leaving the window due to a time shift triggered by a new incoming

event.

In order to determine if an incoming event triggered a move of the sliding window, the drop-off time of the previous and the current event are compared first (Line 12). In case they differ, the new drop-off time is stored in the state object to indicate the move forward in time (Line 13), and the new *cutoff-time* is computed by subtracting 30 $mins$ from the drop-off timestamp of the incoming event (Line 14). Using the *cutoff-time*, the linked list can now be traversed starting from the head in order to purge outdated rides (Lines 15-21). In case the traversal encounters an expired element, the counter as well as the timestamps for corresponding route in the hash-map are updated (Lines 16-20). In a similar fashion as in the first part of the operator, the key for the route that has been updated is added to the set of modified keys (Line 22).

In a final step, the route records carrying the frequencies which have been recently updated are sent downstream to the *top-k* operator (Lines 23-25). In case the event triggered no updates at all, a *silence propagation* event is sent (Line 26) in order to notify the downstream operator that the current event has been fully processed but did not generate an update.

Top-k Operator

The previously generated frequency updates are then processed using the *top-k* operator as depicted in Listing 8.5. In a first step, the type of the event is examined. In case the event is a silence propagation event and originated from the *last* partition of the upstream operator (i.e., holding the largest *sliceId*) (Line 2), the `generateOutput()` method is called in order to determine if an output can be generated or not (Line 3). This constraint is an optimization to reduce the number of modification checks of the *top-k* set due to the partitioning of the upstream tracker operator. Note that the events arriving at the *top-k* operator are processed in a strictly *deterministic* fashion in STREAMMINE3G using a *round-robin* merge scheme so that the event with the largest *sliceId* represents the last event from the same *epoch* of the previously broadcast source event.

In order to determine if the update (*routeUpd*) received from the tracker operator belongs to a route that has not yet been registered at the *top-k* operator, a boolean flag (*newRoute*) is set to false first (Line 4).

The state for the *top-k* operator comprises three different data structures: (i) a hash-map (*state.map*) which contains a mapping of keys to route records in order to keep track of the frequencies, (ii) a *custom linked list implementation* (**allRoutes**) which preserves the order of elements and lets elements transparently move up or down in case they are updated, and (iii) a standard linked list (**oldTopK**) to store the previously computed *top-k*.

The first linked list is used to maintain an order across all elements required to determine the *top-k*. The advantage of our custom implementation is that insertions are of cost $O(n)$ in worst case while removal of elements $O(1)$. Moreover, since updates to list elements cause

Program Listing 8.5 Query#1 Top-k (Sink) Operator

```
1:  function PROCESS(routeUpd, state)
2:      if routeUpd=SILENCEPROPAGATION and routeUpd.sliceId=slices−1 then
3:          GENERATEOUTPUT(routeUpd,state)
4:      newRoute ← False
5:      if state.map[routeUpd.key]=NULL then
6:          state.map[rRec.key]← new RouteRec()
7:          newRoute ← True
8:      rRec ← state.map[routeUpd.key]
9:      rRec.count ← routeUpd.count
10:     rRec.tsPp ← routeUpd.tsPp
11:     rRec.tsDf ← routeUpd.tsDf
12:     if rRec.count = 0 then
13:         rRec.remove()
14:         state.map.erase(rRec.key)
15:     else
16:         if newRoute =True then
17:             state.allRoutes.prependToHead(rRec)
18:         rRec.update()
```

only marginal changes, i.e., counts are only incremented or decremented by one, an update of an element requires only a single swap with its neighboring elements in order to maintain order which is far less costly than performing a complete sort on each update.

Once the *top-k* operator receives an update, a lookup in order to update the route frequency based on the incoming event is performed first (Line 5). In case the route has not yet been registered, a new *RouteRec* object is being created and added to the hash-map (Line 6). In addition to the creation, the *newRoute* flag is set to true indicating that the update corresponds to a newly tracked route. In a second step, the new information carried by the incoming event is stored by updating the fields of the *RouteRec* object appropriately (Lines 9-11). This implies also copying the timestamps from the event causing the update.

In case the incoming event reveals that no rides occurred on that specific route within the past 30 $mins$ which is indicated by a count of 0 (Line 12), the route record (*rRec*) is purged in order to free memory where the rRec removes itself from the custom linked list **allRoutes** (Line 13), and from the hash-map (Line 14).

For all other cases, the update method of the route record (rRec) object is called (Line 18) which causes a swap down or up in the custom linked list in order to re-establish order. In case the route did not exist before, the route is also pre-pended to the head of the custom linked list (Line 17) where it transparently moves the element up to its correct position.

Listing 8.6 shows the pseudo-code of the `generateOutput` method. In order to track if there were modifications to the *top-k* that output previously or not, a boolean flag is set to false first

Program Listing 8.6 Query#1 Top-k (Sink) - Generate Output

```
1: function GENERATEOUTPUT(routeUpd, state)
2:     modified ← false
3:     state.oldTopK.resetIterator()
4:     rRec ← state.allRoutes.getTail()
5:     for i ← 0, i < k, i ← i + 1 do
6:         if state.oldTopK.next() ≠ rRec then modified ← true
7:         rRec ← rRec.prev()
8:     if modified = true then
9:         state.oldTopK.clear()
10:        rRec ← state.allRoutes.getTail()
11:        for i ← 0, i < 10, i ← i + 1 do
12:            state.oldTopK.append(rRec)
13:            rRec ← rRec.prev()
14:        PRINT(rRec.tsPp,rRec.tsDf)
15:        state.oldTopK.resetIterator()
16:        for i ← 0, i < k, i ← i + 1 do
17:            item ← state.oldTopK.next()
18:            if item ≠ NULL then
19:                PRINT(item)
20:            else
21:                PRINT("NULL")
22:            PRINT(rRec.latency)
```

(Line 2). In a next step, an iteration over both data structures is performed, i.e., the **oldTopK** linked list and the **allRoutes** custom linked list (Lines 3-7). If the traversal detects a mismatch for at least one of the first *k*-elements (Lines 5-6), the modification flag is set to true (Line 6).

In case the modification flag was set to true, the **oldTopK** list is cleared (Line 9) and filled with the current *top-k* from the up-to-date **allRoutes** custom linked list (Lines 10-13).

Finally, the current *top-k* is output to screen. If less than *k* elements are available (Lines 18-19), `NULL` is printed to screen for the missing values (Line 21). In addition to the *top-k*, the latency (Line 22) is output.

Query #2 - Area Profit

In the following, we will describe the algorithm used in order to track the profit for an area. Since the computation of an area profit is way more complex than the tracking of route frequencies as presented in the previous query, additional data structures are required as the area profit is composed of (i) the amount of empty taxis for an area and (ii) the median of all fares for drop-offs and pickups that occurred in an area.

We therefore introduce the following entities: area, taxi and fare record:

1. An ***area record*** (*aRec*) contains (*i*) a profit field, holding the value of the most recently computed median value of recently collected fares, the number of empty taxis, and the timestamp (*tsPp* and *tsDf*) of the relevant event that caused a modification to one of the two fields, i.e., the profit or the number of empty taxis. In addition to the median value, the record holds also a *self-balancing double linked* with fare records as entries used to efficiently compute and lookup the median.

2. The ***taxi record*** contains a pointer to the *area record* where the customer was dropped-off. In addition to the area record, it holds the timestamp (*tsPp* and *tsDf*) of the relevant event for the trip.

3. The ***fare record*** holds similar as the taxi record a pointer to the *area record* the fare was collected at. Again, it also holds the timestamp (*tsPp* and *tsDf*) of the relevant event for the trip.

In order to efficiently compute the median of fares collected for an area, we use two different types of linked lists: (*i*) a *self-balancing double linked* list (named **faresVal** ③) which is associated with an area record, and (*ii*) a regular linked lists (named **faresTs** ⑤) which is associated with the state object as depicted in Figure 8.8. While the first list is used to efficiently perform updates and lookups for the median value in an area, the other list is solely used to efficiently purge elements from a time window due their expiry. Hence, the first one orders elements according to their fare amount (fare+tip) while the second one (**faresTs**) maintains its elements according to the drop-off time.

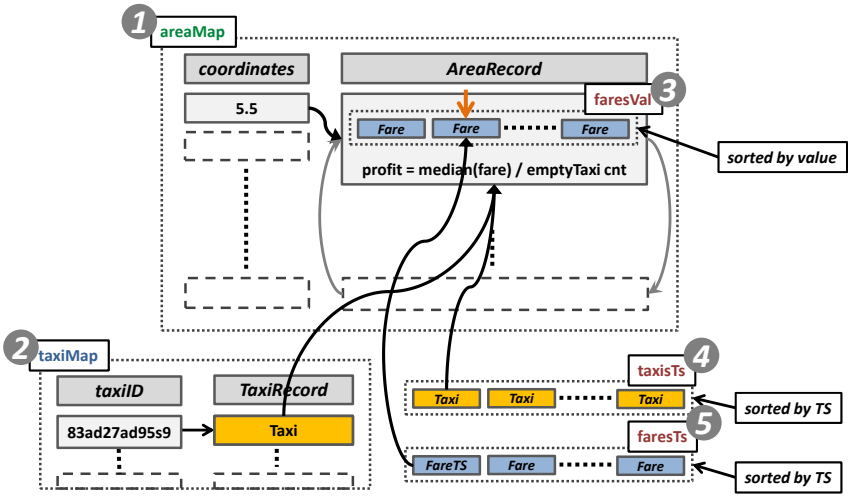

Figure 8.8: Data structure profitable areas tracker operator.

The *self-balancing double linked list* maintains a pointer to the median element in the following way: An insertion of an element with a lower value than the current median one causes a shift of the median pointer to its left neighbor while an insertion to the left causes a shift to the

right. Elements "bubble" up in the same ways as described for the **allRoutes** custom linked list where an element swaps the position with its neighbor to the right or left depending on the value used for comparison. Hence, the insertion of an element in the list is of cost $O(n)$ (worst case) while deletions and median lookups are of cost $O(1)$. Only if one of the median's neighbors is being removed that shared the same value as the current median, the new median and its position within the list must be determined again which is of cost $O(n/2)$.

The current location of a taxi is tracked using a one-dimensional hash-map (**taxiMap** ②) where the taxi's unique identifier is used as key and the *taxi record* object as value that holds a pointer to the drop-off location (area record) and the timestamps of the pickup and drop-off time.

We will now describe how those data structures are used in concert to provide the needed functionality. The pseudo-code for the operator is depicted in Listing 8.7 and can be broken up in two parts: In the first part (Lines 2-23), new information from the incoming event is added while the second part (Lines 24-34) handles the removal of outdated events due to a shift of the time window.

Program Listing 8.7 Query#2 Area Tracker (Worker) Operator

```
1:  function PROCESS(rec, state)
2:      modifiedAreas ← ∅
3:      aRecPickup ←NULL
4:      aRecDropoff ←NULL
5:      if rec.xPp mod slices = sliceId then
6:          key ← rec.xPp * 600 + rec.yPp
7:          aRecPickup ← state.areaMap[key]
8:      if rec.xDf mod slices = sliceId then
9:          key ← rec.xDf * 600 + rec.yDf
10:         aRecDropoffRec ← state.areaMap[key]
11:     taxi ←new Taxi(aRecPickup, aRecDropoff, rec.id, rec.tsPp, rec.tsDf)
12:     oldTaxi ← state.taxiMap[taxi.id]
13:     if oldTaxi ≠ NULL then
14:         oldTaxi.recordPickup(modifiedAreas)
15:         oldTaxi.remove()
16:     state.taxiMap[taxi.id]← taxi
17:     taxi.recordDropoff(modifiedAreas)
18:     farePickup ←new Fare(aRecPickup, rec.fare, rec.tip, rec.tsPp, rec.tsDf)
19:     farePickup.recordStart(modifiedAreas)
20:     fareDropoff ←new Fare(aRecDropoffRec, rec.fare, rec.tip, rec.tsPp, rec.tsDf)
21:     fareDropoff.recordStart(modifiedAreas)
22:     state.faresTs.append(farePickup)
23:     state.faresTs.append(fareDropoff)
```

Contrary to the first query where an incoming event causes an update only to a single route, an input event in the second query always causes updates to two areas at once, i.e., the area

where a customer was picked up and dropped-off. We handle those two cases independently through the use of separate variables associated with the pickup and drop-off location as depicted in the code at Lines 3 and 4: First the area record for the drop-off location is retrieved from the global **areaMap** ① hash-map (Lines 5-7) followed by the pickup location (Lines 8-10). Since we use data partitioning, only the drop-off and pickup locations for the areas that match the modulo computation (Lines 5 and 8) are retrieved from the hash-map. In other words, only the partition that is responsible for the specific area maintains and handles updates to the area at hand. As a reminder, events are still broadcast to all tracker partitions in order to remove expired records in a consistent fashion from all partitions otherwise semantic transparency cannot be ensured.

In a first step, a taxi record is being created which holds two pointers, to the areas it had a pickup and a drop-off, and the timestamps the pickup and drop-off occurred (Line 11). Next, we check if the same taxi had a drop-off previously by searching for a taxi record registered with the same identifier in the **taxiMap** ② hash-map. In case no taxi record was found, a NULL value will be assigned to the *oldTaxi* variable. In case we found a record matching the taxi's identifier, we call the `recordPickup()` method on the taxi object (Line 14) which decrements the counter for empty taxis in the area it had previously a drop-off, and removes itself from a the **taxisTs** ⑤ linked list.

Next, the new taxi record is added to the **taxiMap** hash-map (Line 16), and the `recordDropoff()` method on the taxi object called (Line 17) which increments the counter for empty taxis in the area it had a drop-off just now. Note that both methods (`recordPickup()` and `recordDropoff()`) copy the timestamps from the event causing the update to the area record object and inserts the area record object to the *modifiedAreas* set in order to trigger an update notification for the following *top-k* operator.

In order to compute the median fares for the pickup and drop-off location, two fare objects are created (Lines 18 and 20) and their `recordStart()` methods are called (Lines 19 and 21). The call to the method adds the fare object to the *self-balancing double linked list* (**faresVal** ③) associated with each area record. Note that the fare object holds a pointer to the area record it belongs to. Adding the new fare object to the **faresVal** list also causes an update of the median computation and the insertion of the area to the *modifiedAreas* set.

Next, the two fare objects are also appended to the timestamp ordered linked list **faresTs** (Lines 22 and 23).

As previously mentioned, the second part of the area tracker operator is responsible for updating the area profit based on expired events: First, we iterate through the timestamp ordered list of recorded taxi rides (**taxisTs** ⑤) in order to purge empty taxis that have expired of the 30 $mins$ sliding window (Lines 24-26). We call again the `recordPickup()` method in order to update the count of empty taxis for the associated area (Line 26).

In a similar fashion, fares for the areas pickups and drop-offs are purged from the 15 $mins$

```
24:  while state.taxisTs.front().tsDf + 30 * 60 < rec.tsDf do
25:      taxi ← state.taxisTs.popfront()
26:      taxi.recordPickup(modifiedAreas)
27:  while state.faresTs.front().tsDf + 15 * 60 < rec.tsDf do
28:      fare ← state.faresTs.popfront()
29:      fare.recordExpire(modifiedAreas)
30:  for each aRec in modifiedAreas do
31:      EMIT({aRec.profit,sliceId})
32:      if aRec.profit.medianFare = 0 and aRec.profit.emptyTaxies = 0 then
33:          state.areaMap.erase(aRec.key)
34:  EMIT({SILENCEPROPAGATION,rec.tsDf})
```

sliding windows (Lines 27-29) where the call to `recordExpire()` causes an update to the median computation to the respective area.

In case an area was updated, the corresponding object is added to the *modifiedAreas* set in order to send updates downstream to the *top-k* operator. After sending an update (Line 31), area records that contain an empty count of taxis and a median fare of zero are removed from the hash-map in order to free resources (Lines 32-33). For the sink, we use the same implementation as for the first query, hence, omit the pseudo-code here.

8.2.3 Evaluation

In this section, we present the results of various micro benchmarks we executed in order to assess the performance of our proposed solution. For the evaluation, we deployed the query using the *single-node* as well as the *distributed execution mode* option the StreamMine3G runtime offers:

In single-node setup, the query can be executed in a completely *lock free* manner as only a single instance per operator is used for carrying out the processing, hence, neither data partitioning nor a parallel execution of operators takes place. Furthermore, event dissemination across network as well as event ordering mechanisms, i.e., a deterministic merge, can be disabled which allows an easy assessment of the achievable baseline performance of the query at hand. The single-node execution mode can therefore be envisioned as a naïve implementation of the query which does not scale as no parallelization techniques are applied.

In contrast to the single-node setup, multiple operator partitions are deployed on a set of either physical or virtual machines in the distributed node with the benefit of a scalable data processing. However, the distributed setup comes with a price as data must be exchanged using TCP connections introducing additional latency. Moreover, events originating from multiple upstream partitions must be merged in a deterministic fashion prior passing them to the next operator in order to ensure semantic transparency which adds additional overhead.

Single Node Execution

In the first set of experiments, we assessed the performance of our implementation using the single-node setup. For the experiment, we used different configurations in order to determine the overhead of individual operators and sub-queries as shown in Figure 8.9. The configurations are as follows: *top-k freq + top-k area* represents the execution of both queries, i.e., all operators while *source only* represents the execution using only the source operator. *Top-k freq + top-k area w/o sinks* represents an execution of the full query except the sink operators that maintain the *top-k* sets while in *top-k freq only* and *top-k area only,* only one of the two queries is executed. For the last two configurations, we disabled the sink operator either of the first or second query. Furthermore, we executed each configuration in *single* and *multi-threaded* mode. In multi-threaded mode, operators are executed in a *pipelining* fashion using producer-consumer queues rather than in a chained fashion.

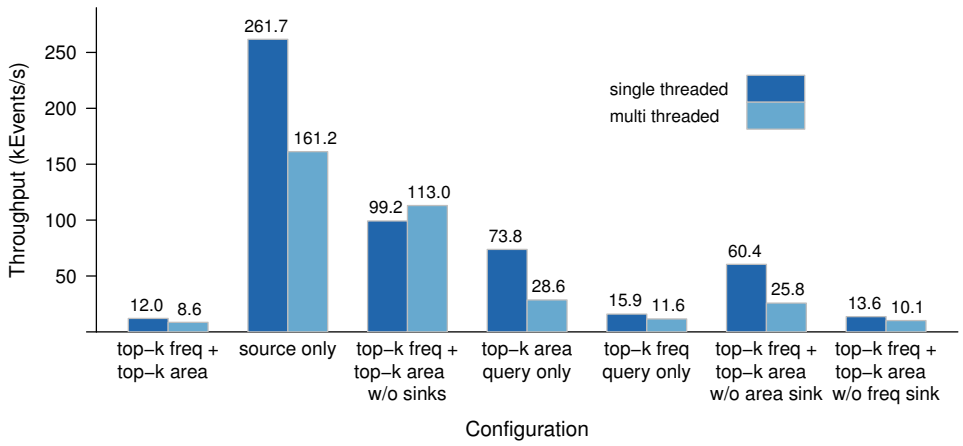

Figure 8.9: Throughput for different configurations, i.e., sub queries or individual operators.

The results of the experiments are depicted in Figure 8.9: When executing both queries, we can achieve a throughput of roughly 12.0 kEvents/s using a single-node and single-threaded execution whereas only 8.6 kEvents/s can be achieved using pipelining. In order to determine how quickly events can be read and parsed from the provided csv-file, we used solely the source operators in the second configuration. As shown in the Figure, roughly 261 kEvents/s can be processed which therefore represents the upper bound of the performance that can be achieved.

An interesting effect can be observed when disabling the sink operators for the two queries since we can achieve a noticeable higher throughput when using the multi-threaded version, hence, pipelining of the source and worker operators increases the throughput. In the following two configurations, we used only one of the two queries for execution. As shown in the graph, an order of magnitude higher throughput can be achieved when solely executing the *top-k* profitable areas query (73.8 kEvents/s) while only 15.9 kEvents/s are processed when

running the *top-k* frequent routes query. As shown in the last two configurations, the sink operator for both queries have also different runtime behaviors: Disabling the *top-k* profitable area sink operator increases the throughput by an order of magnitude compared to having both queries running as shown in the first configuration. This can be easily explained as with every event, two areas are updated while in the first query only one element in the *top-k* set is updated at a time.

Distributed Execution

In the second set of experiments, we used the distributed setup in order to asses the scalability of our system. For those experiments, we partitioned the source operator in order to prevent the source being the bottleneck in our system, i.e., eight source operator instances are emitting events to the worker operators of the two queries concurrently.

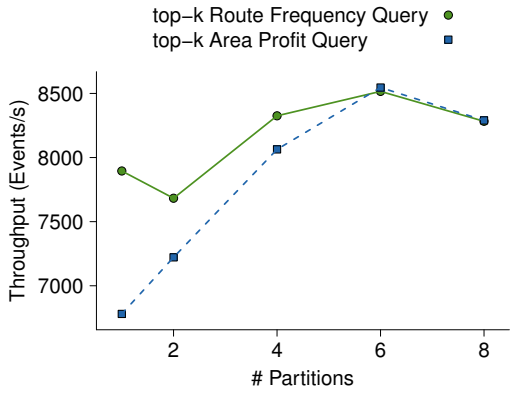

Figure 8.10: Aggregated throughput with increasing number of partitions.

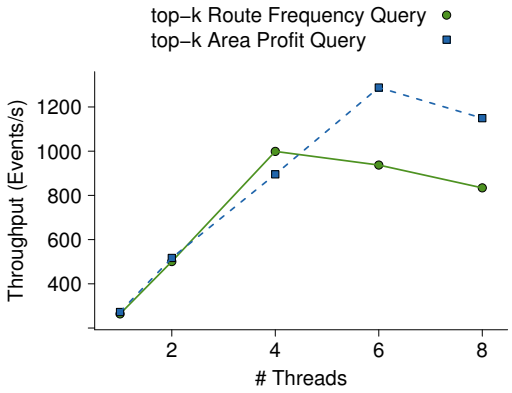

Figure 8.11: Per slice throughput with increasing number of threads.

In the first experiment, we varied the number of partitions used for each of the worker operators. The results are depicted in Figure 8.10. As shown in the graph, the throughput increases with the number of partitions. This is an effect of data partitioning since each partition is only responsible for a subset of routes/areas, i.e., hence with every event arrival, fewer routes/areas must be traversed in order to check for expired events due to a window move which therefore increases the throughput. Unfortunately, with an increase of partitions, also an increasing amount of events are sent to the sink operators which then becomes a bottleneck as one can see in the graph where the throughput saturates and finally decreases when using more than six partitions.

In the last experiment, we varied the size of StreamMine3G's thread pool, i.e., the number of threads used for parallel execution. As shown in Figure 8.11, the throughput increases with the number of threads. However, as in the previous experiment, the single instance of the sink prevents further scaling of the system which even lowers the throughput beyond six threads due to heavy contention. However, contrary to the multi-threaded single node

execution, multi-threading when applied on partitioned data can fully harness nowadays modern multi-core architectures.

8.3 Conclusion

In this chapter, we presented the implementation and performance evaluation of two real world applications that have been implemented on top of StreamMine3G. Both applications share the property of operating on moving time windows.

We showed how the applications can be intuitively implemented using StreamMine3G's provided programming model and how programmers can benefit from its explicit state management support. We furthermore demonstrated the scalability for both applications due to data partitioning while still providing correct results through the use of StreamMine3G's deterministic merge/execution.

9 Conclusions

In this chapter, we summarize the main contributions and achievements of this thesis, and provide an outlook for possible future research in the area of fault tolerance and ESP.

9.1 Summary of Contributions

Implementing and providing fault tolerance in ESP systems imposes several challenges. First, it requires a flexible architecture for the implementation of fault tolerance mechanisms that provide strong recovery semantics such as exactly once. Second, fault tolerance imposes per se a non-negligible overhead onto the system by consuming a considerable amount of resources and lowering the processing throughput at the same time. In this dissertation, we addressed several of those challenges as follows:

1. **StreamMine3G Approach:** In Chapter 3, we presented our StreamMine3G approach, a flexible architecture that combines scalable, elastic and fault tolerant event stream processing. The architecture of our system allowed us to implement and incorporate several fault tolerance mechanisms including complex recovery protocols that go beyond of state-of-the-art of publicly available ESP platforms. StreamMine3G served as a solid foundation for the fault tolerance and elasticity mechanisms we designed, implemented and evaluated throughout this work.

2. **Lower Overhead for Deterministic Execution:** Providing precise recovery and exactly once semantics in ESP systems requires the use of deterministic execution. In Chapter 4 and 5, we proposed a weak ordering scheme for a deterministic merge which provides precise recovery, however, at a much lower cost than with strict event ordering. The approach is based on the observation that most operators are commutative and operate on (jumping) windows, and can be used with active or passive replication.

3. **Improved Resource Utilization:** Since most applications run in cloud environments nowadays, we explored in Chapter 6 the possibility of improving the system availability by utilizing spare but paid cloud resources. The approach is driven by the fact that many

ESP applications run at conservative low utilization levels in order to accommodate sudden load spikes. In our approach, we use those idle resources by switching between active replication and passive standby at runtime, improving the system utilization and availability at the same time.

4. **Adaptive Fault Tolerance:** Choosing the right fault tolerance scheme is often cumbersome for application users as it requires a deep understanding of the implications with regards to recovery time and resource overhead imposed by the choice made. In Chapter 7, we presented a fault tolerance controller that frees the user from the burden to settle for a single fault tolerance scheme. Instead, our controller selects transparently the scheme that *(i)* satisfies the user-based constraints such as recovery time threshold and semantics and *(ii)* consumes the least amount of resources. In our evaluation, we showed that a considerable amount of resources can be saved in comparison to the conservative use of active replication.

5. **Real World Applications:** Finally, in Chapter 8 of this thesis, we showcased several real world applications where our StreamMine3G approach has been successfully applied to. The use cases and implemented applications prove that the proposed programming model and architecture is suitable for common problems to be solved using ESP system, and demonstrate that our system provides scalability and elasticity on top of fault tolerance.

9.2 Challenges and Future Work

During the execution of the research to lower the overhead for fault tolerance in ESP systems, we identified many potential paths to extend the approaches presented in this work.

First, we envision to combine our StreamMine3G approach with software encoded processing [WF07]. Since StreamMine3G is entirely written in C++, the code can be easily transformed into a secure version using the Silistra compiler [Sil15], hardening the system against control flow and value errors.

In Chapter 4, we have seen an approach that allows us to reduce the processing overhead and latency by applying an epoch-based event ordering and processing scheme. However, when applying this approach to active replication as presented in Chapter 5, we need to delay the processing of events until the end of an epoch in order to consistently agree on what set of epoch-bags to merge prior to processing. In order to reduce processing latency, a speculative approach can be applied where complete epoch-bags are merged and processed immediately, however, the output is not committed and released until the final decision has been made as presented in the approach of [BFSF08]. Alternatively, events/results can also be released immediately, however, marked as tentative results as authors proposed [BBMS05].

Since most ESP systems are running in cloud environments nowadays, trust becomes an equally important requirement besides fault tolerance. With the recent advent of Intel's new

SGX technology [SGX15], we envision to execute parts of our StreamMine3G system, such as the operator containers in *secure enclaves* in a similar fashion as authors have shown in [SCF+14] for MapReduce. This approach would enable cloud and PaaS providers to offer secure ESP in untrusted cloud environments opening an entirely new market.

One possibility to harden StreamMine3G against crash failures originating from software bugs in user code is the use of save execution environments where improper memory accesses etc. are caught prior to their propagation to the underlying ESP system. Since Java is one of the most popular languages used for data processing systems nowadays (Hadoop, Storm and S4 [Had15, Sto15, S4215, NRNK10]), a save execution environment can easily be offered by simply implementing a JNI-based [Lia99] operator wrapper. The advantage of such a JNI wrapper is that users familiar with Java can easily implement their operators and also reuse code fragments while not having to worry about garbage collection etc. However, the use of such a wrapper comes with a price as data passed to the UDFs must be marshaled and unmarshaled upon each call introducing a non-negligible performance overhead. Alternatively, it is also possible to provide such execution environments for other interpreter or byte-code-based languages such as Python, Javascript or Lua.

Although the majority of software bugs can be already captured and handled prior causing a severe system crash through various techniques such as static code analysis [ECH+01], the ever increasing complexity of software for distributed big data systems still requires mechanisms for handling system failures with a preferable low resource footprint such as presented in this dissertation.

Publications

- **[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 - **DEBS 2015 Challenge finalist paper**

- **[OFM+15]** Ana Cristina Oliveira, Christof Fetzer, André Martin, Do Le Quoc, and Marco Spohn. Optimizing query prices for data-as-a-service. In *2015 IEEE International Congress on Big Data*, pages 289–296, June 2015

- **[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 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 462–473, Washington, DC, USA, 2015. IEEE Computer Society

- **[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 - **UCC 2014 Cloud Challenge winner paper**

- **[BHM+14]** Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, pages 567–576, Washington, DC, USA, 2014. 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 - **DEBS 2014 Challenge finalist paper**

- **[MBF14]** André Martin, Andrey Brito, and Christof Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. In *Proceedings of the 8th ACM International*

Conference on Distributed Event-Based Systems, DEBS '14, pages 198–205, New York, NY, USA, 2014. ACM

- **[QMF13]** Do Le Quoc, André Martin, and Christof Fetzer. Scalable and real-time deep packet inspection. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, pages 446–451, Washington, DC, USA, 2013. IEEE Computer Society

- **[MQ13]** André Martin and Do Le Quoc. Tutorial: Elastic and fault tolerant event stream processing using streammine3g. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, pages xxxiii–xxxiv, Washington, DC, USA, 2013. IEEE Computer Society

- **[BMF+13]** Andrey Brito, André Martin, Christof Fetzer, Isabelly Rocha, and Telles Nobrega. Streammine3g oneclick – deploy and monitor esp applications with a single click. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 1014–1019, Washington, DC, USA, 2013. IEEE Computer Society

- **[HJM+12]** Thomas Heinze, Zbigniew Jerzak, André Martin, Lenar Yazdanov, and Christof Fetzer. Fault-tolerant complex event processing using customizable state machine-based operators. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 590–593, New York, NY, USA, 2012. ACM

- **[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

- **[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

- **[BMK+11]** Andrey Brito, André Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 48–58, Washington, DC, USA, 2011. IEEE Computer Society

Contributions By Topics

- **Fault Tolerance in ESP:**
  [MKC+11, MFB11, HJM+12, MSD+15]
  The content has been partially or fully incorporated in Chapters: 4, 5, 6 and 7 of this thesis.

- **StreamMine3G Approach:**
  [BMK+11, BMF+13, MQ13, QMF13, MBF14, MMBF14, MSBF14, MBF15]
  The content has been partially or fully incorporated in Chapters: 3 and 8 of this thesis.

- Other work executed within EU FP7 projects but not included in this thesis: [OFM+15]

Symbols

Term	Definition (Section)	Description
e	2.1.1	event, tuple
τ	2.1.1	timestamp
κ	2.1.1	key
ρ	2.1.1	payload
o	2.1.2	operator
$\mathcal{O}$	2.1.3	set of operators
s	2.1.3	stream
S	2.1.3	set of streams
q	2.1.3	query
f_o	2.1.2	operator function
f_p	2.1.4	partitioner function
f_m	2.1.3	merge function
π	2.1.4	parallelization degree/level
w	2.1.7	window

Table 9.1: Symbols

Bibliography

[AAB+06] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, DMSSP '06, pages 27–37, New York, NY, USA, 2006. ACM.

[Apa15] Apache httpd piped logs. https://httpd.apache.org/docs/2.2/logs.html#piped, July, 15th 2015.

[AS00] Marcos Kawazoe Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 209–218, New York, NY, USA, 2000. ACM.

[AWHF10] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[BBMS05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 13–24, New York, NY, USA, 2005. ACM.

[BFF09a] A. Brito, C. Fetzer, and P. Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 173–182, June 2009.

[BFF09b] Andrey Brito, Christof Fetzer, and Pascal Felber. Multithreading-enabled active replication for event stream processing operators. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, pages 22–31, Washington, DC, USA, 2009. IEEE Computer Society.

[BFSF08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings*

of the Second International Conference on Distributed Event-based Systems, DEBS '08, pages 265–275, New York, NY, USA, 2008. ACM.

[BHCG10] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[BHM+14] Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, pages 567–576, Washington, DC, USA, 2014. IEEE Computer Society.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 123–138, New York, NY, USA, 1987. ACM.

[BKI06] Claudio Basile, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Active replication of multithreaded applications. *IEEE Trans. Parallel Distrib. Syst.*, 17(5):448–465, May 2006.

[BMF+13] Andrey Brito, André Martin, Christof Fetzer, Isabelly Rocha, and Telles Nobrega. Streammine3g oneclick – deploy and monitor esp applications with a single click. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 1014–1019, Washington, DC, USA, 2013. IEEE Computer Society.

[BMK+11] Andrey Brito, André Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 48–58, Washington, DC, USA, 2011. IEEE Computer Society.

[Boo15a] Boost unordered map. http://www.boost.org/libs/unordered/, 2015.

[Boo15b] Boost.asio, a cross-platform c++ library for network and low-level i/o programming. http://www.boost.org/libs/asio/, 2015.

[CASD95] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Inf. Comput.*, 118(1):158–179, April 1995.

[CCA+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[CD88] George F. Coulouris and Jean Dollimore. *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[CFMKP13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[CSA05] Nauman A. Chaudhry, Kevin Shaw, and Mahdi Abdelguerfi, editors. *Stream Data Management*, volume 30 of *Advances in Database Systems*. Springer, 2005.

[CWTY10] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

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

[DMRH04] Luping Ding, Nishant Mehta, Elke A. Rundensteiner, and George T. Heineman. *Joining Punctuated Streams*, pages 587–604. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[DTT99] A. M. Déplanche, P. Y. Théaudière, and Y. Trinquet. Implementing a semi-active replication strategy in chorus/classix, a distributed real-time executive. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, SRDS '99, pages 90–, Washington, DC, USA, 1999. IEEE Computer Society.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[Esp15] Esper: Event processing for java. http://www.espertech.com/products/esper.php, 2015.

[Fet03] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, February 2003.

[FWPG14] Raul Castro Fernandez, Matthias Weidlich, Peter Pietzuch, and Avigdor Gal. Scalable stateful stream processing for smart grids. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 276–281, New York, NY, USA, 2014. ACM.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[GJPPM+12] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, December 2012.

[GJPPMV10] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, and Patrick Valduriez. Streamcloud: A large scale data streaming system. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 126–137, Washington, DC, USA, 2010. IEEE Computer Society.

[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.

[Hal09] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 1st edition, 2009.

[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.

[HCZ07] Jeong-Hyon Hwang, Ugur Cetintemel, and Stan Zdonik. Fast and reliable stream processing over wide area networks. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, ICDEW '07, pages 604–613, Washington, DC, USA, 2007. IEEE Computer Society.

[HD93] Hui-I Hsiao and David J. DeWitt. A performance study of three high availability data replication strategies. *Distrib. Parallel Databases*, 1(1):53–80, January 1993.

[HJM+12] Thomas Heinze, Zbigniew Jerzak, André Martin, Lenar Yazdanov, and Christof Fetzer. Fault-tolerant complex event processing using customizable state machine-based operators. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 590–593, New York, NY, USA, 2012. ACM.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University Ithaca, NY, USA, Ithaca, NY, USA, 1994.

[JPPnMA00] Ricardo Jiménez-Peris, Marta Patiño Martínez, and Sergio Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, SRDS '00, pages 164–, Washington, DC, USA, 2000. IEEE Computer Society.

[JZ14] Zbigniew Jerzak and Holger Ziekow. The debs 2014 grand challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 266–269, New York, NY, USA, 2014. ACM.

[JZ15] Zbigniew Jerzak and Holger Ziekow. The debs 2015 grand challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 266–268, New York, NY, USA, 2015. ACM.

[Kaf15] Apache kafka - a publish-subscribe distributed messaging system. http://kafka.apache.org/, 2015.

[Kal60] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[KBG08] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.*, 1(1):574–585, August 2008.

[KMR+13] Boris Koldehofe, Ruben Mayer, Umakishore Ramachandran, Kurt Rothermel, and Marco Völz. Rollback-recovery without checkpoints in distributed event processing systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 27–38, New York, NY, USA, 2013. ACM.

[KR05] Sandeep S. Kulkarni and Ravikant. Stabilizing causal deterministic merge. *J. High Speed Netw.*, 14(2):155–183, April 2005.

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 311–322, New York, NY, USA, 2005. ACM.

[MBF14] André Martin, Andrey Brito, and Christof Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 198–205, New York, NY, USA, 2014. ACM.

[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.

[Mil10] Cary Millsap. Thinking clearly about performance. *Queue*, 8(9):10:10–10:20, September 2010.

[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.

[MLW+14] Christopher Mutschler, Christoffer Löffler, Nicolas Witt, Thorsten Edelhäußer, and Michael Philippsen. Predictive load management in smart grid environments. In *Proceedings of the 8^{th} ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 282–287, New York, NY, USA, 2014. ACM.

[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.

[MQ13] André Martin and Do Le Quoc. Tutorial: Elastic and fault tolerant event stream processing using streammine3g. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, pages xxxiii–xxxiv, Washington, DC, USA, 2013. IEEE Computer Society.

[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 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 462–473, Washington, DC, USA, 2015. IEEE Computer Society.

[Mum15] Murmurhash. https://sites.google.com/site/murmurhash/, 2015.

[Net15a] Making storm fly with netty. http://yahooeng.tumblr.com/post/64758709722/making-storm-fly-with-netty, 2015.

[Net15b] Netty.io - a synchronous event-driven network application framework. http://netty.io/, 2015.

[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.

[OFM+15] Ana Cristina Oliveira, Christof Fetzer, André Martin, Do Le Quoc, and Marco Spohn. Optimizing query prices for data-as-a-service. In *2015 IEEE International Congress on Big Data*, pages 289–296, June 2015.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[Pow91] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Research Reports Esprit / Projects 818/2252 Delta-4. Springer Berlin Heidelberg, 1991.

[Pro15] Protocol buffers - google's data interchange format. https://github.com/google/protobuf, 2015.

[QMF13] Do Le Quoc, André Martin, and Christof Fetzer. Scalable and real-time deep packet inspection. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, pages 446–451, Washington, DC, USA, 2013. 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.

[SAP15] Sap event stream processor. http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html, 2015.

[SCF+14] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud. Technical Report MSR-TR-2014-39, February 2014.

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

[SDFZ09] R. Strom, C. Dorai, T.H. Feng, and Wei Zheng. Deterministic replay for transparent recovery in component-oriented middleware. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 615–622, June 2009.

[See15] Seep - a parallel data processing system. https://github.com/lsds/Seep/, 2015.

[SGX15] Intel sgx. https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx, 2015.

[SHB04] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 827–838, New York, NY, USA, 2004. ACM.

[Sil15] Silistra. http://www.silistra-systems.com/, 2015.

[SM11] Zoe Sebepou and Kostas Magoutis. Cec: Continuous eventual checkpointing for data stream processing operators. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 145–156, Washington, DC, USA, 2011. IEEE Computer Society.

[SSC+02] Mehul A. Shah, Mehul A. Shah, Sirish Chandrasekaran, Joseph M. Hellerstein, Joseph M. Hellerstein, Sirish Ch, Sirish Ch, Michael J. Franklin, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *In ICDE*, pages 25–36, 2002.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

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

[TcM15] Fast, multi-threaded malloc(). http://code.google.com/p/gperftools/, 2015.

[TcZ07] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 159–170. VLDB Endowment, 2007.

[UKB11] Prasang Upadhyaya, YongChul Kwon, and Magdalena Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 241–252, New York, NY, USA, 2011. ACM.

[WF07] Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'07, pages 356–369, Berlin, Heidelberg, 2007. Springer-Verlag.

[Yar15] Yarn - cluster resource manager. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2015.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[ZDL+12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[Zer15] Zeromq. http://zeromq.org/, 2015.

[ZGY+10] Zhe Zhang, Yu Gu, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. A hybrid approach to high availability in stream processing systems. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 138–148, Washington, DC, USA, 2010. IEEE Computer Society.

[ZRH04] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 431–442, New York, NY, USA, 2004. ACM.

Index

Declaration

Herewith I declare that this submission is my own work and that, to the best of my knowledge, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher education, except where due acknowledgment has been made in the text.

Dresden, July 30, 2015

André Martin