Pedro Miguel Fortunato Silvestre

Degree in Computer Science and Engineering

Clonos: Consistent High-Availability for Distributed Stream Processing through Causal Logging

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Asterios Katsifodimos, Assistant Professor,
Delft University of Technology

Co-adviser: João Carlos Antunes Leitão, Assistant Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2020

Clonos: Consistent High-Availability for Distributed Stream Processing through Causal Logging

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To friends and family.

Acknowledgements

My thanks go to the Department of Informatics of the NOVA University of Lisbon and the NOVA LINCS research center, for providing me with a great education, which aided me in achieving my goals. I must also express my gratitude to all the fantastic people who welcomed me at the TU Delft as one of their own, for showing me just how fun research can be. I am only sorry that due to the pandemic I could not spend more time with my colleagues from both universities. The work presented in this thesis is the result of many discussions, long nights and crowded white boards. I am thankful every person who ever participated in any one of the three.

I had the luck of having three people whom I could call adviser. I want to thank my official adviser João Leitão from our own NOVA University, for inspiring me to do research, supporting me through all the special arrangements I required and for his keen eye spotting possible problems that would emerge. Similarly, I would like to thank Asterios Katsifodimos from TU Delft, for always providing vision and encouragement, but most of all for believing in me. Finally, I would like to extend my gratitude to Marios Fragkoulis from TU Delft, for working closely with me, and being my main source of problem solving discussions. Without him, this work would surely not have been possible.

A special thank you is owed to Mariana Monteiro, who endured many months of distance, bad temper, and ramblings about causal logging, but who through it all helped me keep my spirits up. I would also like to thank my tireless mother, Ana Fortunato, who helped in any way she could possibly imagine through this process and for always ensuring I had access to the best education possible, even through the rougher times. Finally, I would like to extend my gratitude to every other member of my family and friend network. I realize that I have not been the most pleasant or present in the past few months, but I thank you for believing in me.

This work was partially supported by FCT/MCTES through the project NG-STORAGE (grantPTDC/CCI-INF/32038/2017) and NOVA LINCS (grant UID/CEC/04516/2013)

Abstract

Nowadays, distributed stream processing systems lie in the backbone of businesses, as a backend for critical event-driven applications such as real-time fraud detection or stock trading. Given their critical nature these systems should be expressive, performant, highly-available and maintain state consistency after failure. However, current fault-tolerance solutions forego one of these four requirements. Highly-available systems sacrifice either consistency or expressiveness and often performance, while more reliable systems have slow and blocking recovery.

In this thesis, we describe Clonos, a highly-available stream processing system that instantly switches the execution of failed operators to passive standbys. Our approach is non-blocking as it uses localized recovery, treating only the failed operators. By additionally forcing recovering operators to replay nondeterministic events Clonos achieves consistent recovery. To manage nondeterminism, we adapt causal logging, a rollback recovery method that logs such events in-memory and propagates them causally, to the stream processing paradigm. Clonos is configurable, allowing one to trade-off overhead for safety. To implement Clonos we re-engineered the distributed runtime of Apache Flink, a state-of-the-art stream processing system.

To evaluate the performance of Clonos in terms of throughput, latency, network bandwidth and recovery time we perform overhead and failure experiments using both realistic and synthetic workloads. Clonos delivers upwards of 10 times faster recovery times without blocking and with much lower latency, at the cost of 11% throughput overhead on realistic workloads, when compared to state-of-the-art reliable systems. Clonos is more expressive than past highly-available systems, supporting a much larger set of use-cases.

Clonos' use of causal logging also opens a plethora of new opportunities, such as transaction-less exactly-once delivery guarantees and consistent non-blocking reconfiguration.

Keywords: stream processing, dataflow, fault-tolerance, exactly-once, causal logging

Resumo

Hoje em dia, processadores de *streams* distribuídos são centrais em negócios, servindo como base para aplicações reativas de tempo real como deteção de fraúde e troca de ações. Dada a sua natureza crítica, estes sistemas devem ter flexibilidade, desempenho, alta-disponibilidade e consistência após falhas. Contudo, soluções para tolerância a falhas atuais sacrificam alguns destes requisitos. Sistemas altamente disponíveis sacrificam consistência ou flexibilidade e desempenho, enquanto sistemas confiáveis têm recuperação bloqueante e lenta.

Na presente tese, descrevemos Clonos, um sistema de processamento de *streams* altamente disponível que instantâneamente troca a execução de operadores falhados para outros em espera passiva. A nossa abordagem é não bloqueante dado que usa recuperação localizada, tratando apenas os operadores falhados. Ao adicionalmente forçar operadores em recuperação a reexecutar eventos não determinísticos, o Clonos atinge recuperação consistente. Para o fazer com performance, Clonos utiliza *causal logging*, um método de recuperação por retrocesso, que regista e propaga eventos não deterministicos em memória volátil. O Clonos é configurável, permitindo trocar sobrecarga por segurança. Para implementar o Clonos modificámos a camada de execução distribuída do Apache Flink, um sistema de *streaming* do estado-da-arte.

Para avaliar o desempenho do Clonos em termos de débito, latência, banda-larga e tempo de recuperação realizamos experiências de sobrecarga e de falha usando cargas de trabalho realistas e sintéticas. O Clonos permite recuperação até 10 vezes mais rápida, não-bloqueante e com latência muito mais baixa, em troca de 11% de sobrecarga de débito em cargas de trabalho realistas, quando comparado com sístemas confiáveis do estado-da-arte. O Clonos é muito mais expressivo que sistemas altamente disponíveis do passado, suportando um maior conjunto de casos de uso.

Causal logging abre um conjunto de novas oportunidades, como a entrega de resultados exatamente-uma-vez sem transações ou reconfiguração consistente não bloqueante.

Palavras-chave: processamento de streams, dataflow, tolerância a falhas, causal logging

Contents

List of Figures

List of Tables

Acronyms

ACID	Atomicity Consistency Isolation Durability
DAG	Directed Acyclic Graph
DBMS	Database Management System
DFS	Distributed File System
ETL	Extract Transform Load
FBL	Family-Based Logging
GFS	Google File System
HDFS	Hadoop Distributed File System
HTAP	Hybrid Transaction/Analytical Processing
IOP	In-Order Processing
IoT	Internet of Things
LSMT	Log-Structured Merge Tree
MPS	Message Passing System
MVCC	Multiversion concurrency control
ODD	Ordered Delivery Deterministic
OLAP	On-Line Analytical Processing
OLTP	On-Line Transaction Processing
OOP	Out-of-Order Processing
OWP	Outside World Process
PWD	PieceWise Deterministic

RDD	Resilient Distributed Dataset
RPC	Remote Procedure Call
SPS	Stream Processing System
SUT	System-Under-Test
WAL	Write-Ahead Log

Chapter 1

Introduction

For a long time, businesses have relied on data[31] to improve marketing efficacy, profits and customer satisfaction. Even when applications were designed as monoliths, as shown in Figure 1.1a, a Database Management System (DBMS)[87] was used to securely record customer and purchase information. This monolithic architecture was subsumed by the microservices architecture, shown in Figure 1.1b, which allowed for better separation of concerns, maintainability and even scaling. Though the application layer was scalable, there was still a reliance on single-node DBMSs. DBMSs are mostly built for On-Line Transaction Processing (OLTP) workloads and focus on guaranteeing ACID[87] transactions which are difficult to scale.

At the same time, as businesses began tracking minute details of their customer's interactions with their applications in hopes of gaining the upper hand on their competition, extremely large datasets began being generated. Vertically scaling single-node systems was infeasible and so Distributed File System (DFS)s[21, 44] were created. These systems forego some features of DBMSs such as ACID transactions in order to achieve better scalability and availability for file storage. The aim of collecting large datasets is to derive valuable insights from them, which can be used for anything from marketing and managerial decisions to consumption by other systems. To derive value from this raw data, it has to be processed which led to the implementation of the first batch processing systems, such as MapReduce [35], Dryad [55] and Spark [109]. These systems were specifically designed for On-Line Analytical Processing (OLAP) workloads and so are capable of performing analytics on terabytes of data in reasonably short time. To *ingest* data into the DFS, an Extract Transform Load (ETL) process is used to capture changes to the microservice DBMSs and load them into the DFS. This is architecture is shown in Figure 1.1d.

At this point, a transformation was occurring in data-driven application design. As

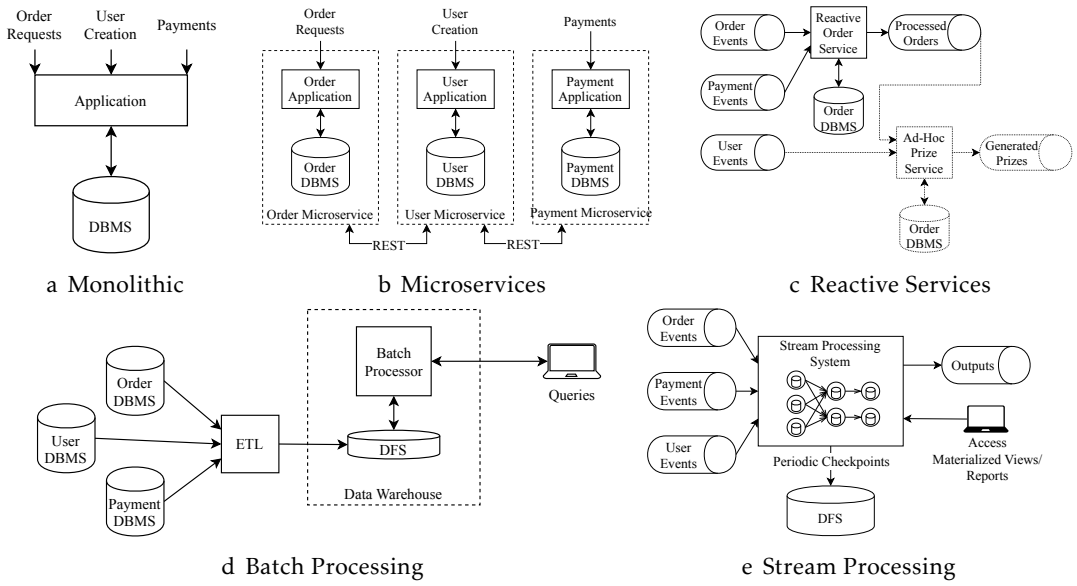


Figure 1.1: Data-driven architectures over time.

finer grained interactions with users were tracked, and the Internet of Things (IoT) [17] movement became widespread, it no longer made sense to model user interactions through single requests. Instead, user interactions appeared more as continuous and never ending *streams* of events. Applications thus began being designed as reactive services (shown in Figure 1.1c), services which react to events. These services are typically implemented in actor frameworks[1], such as Akka[47]. Several other reasons motivated this switch in thinking. First, given the aim of Big Data is to record as much data as possible, it makes sense to log all user events, instead of discarding them as in the microservices paradigm. Event logs such as Apache Kafka[62] are used to record these streams in a performant and durable way. Second, the microservices architecture uses blocking REST requests to communicate between microservices, which increases coupling and leads to cascading failures. Instead, in a reactive environment, the event logs are used as asynchronous communication channels. Furthermore, if event logs are kept, this allows a growing architecture to implement ad-hoc queries, even after requests have been made. This is possible because in a reactive architecture the ground-truth log of events is kept allowing it to be reprocessed, while in a microservices architecture only the post-processing application state is kept, and so further insights cannot be extracted.

Companies continue to attempt to gain an advantage over their competitors, and in recent years, the focus has switched to low-latency processing, a paradigm sometimes known as *Fast Data*. The aim is to process and react to a large volume of events with sub-second latency. Stream Processing System (SPS)s [32] were created in response to the high-throughput high-latency capabilities that batch data processing systems offered and specifically target real-time data processing by offering time-based operations such as windows and out-of-order processing capabilities. Use-cases were many, ranging from

analytics, to serving as ETL engines, to materialized view maintenance. Initially, stateless systems were developed [26, 99], but soon after the need for stateful stream processing became apparent. Stateful stream processing allows for more expressive computations with knowledge of past events. Modern SPSs offer fully managed state[22] reducing operational complexity further and handling fault tolerance often through periodic checkpointing of operator state to a DFS.

Typical data-driven companies nowadays mix and match these different components as necessary. During the initial stages of development of SPSs, both batch and stream processing were employed in a *Lambda architecture*[70], where batch jobs are computed daily to provide accurate results and a streaming path is used simultaneously for fast but often inaccurate results. However, as SPSs have matured, increasing their reliability[24], performance (comparable or higher than batch systems[59]) and expressiveness[4], many enterprises have adopted the *Kappa architecture*, where only reactive services and stream processing is used, reducing operational complexity. Furthermore, while initially designed for analytics processing, the success of SPSs has sparked interest in their use for building reactive services in general. In review, they offer nearly all the capabilities desirable[60] for building reactive services: failure detection, managed state, fault tolerance and consistency, scalability, performance and separation of concerns. Ongoing works aim to utilize SPSs as execution engines for reactive services frameworks[60, 92] and yet others target Functions-as-a-Service[2]. Nevertheless, even without these frameworks, SPSs are already utilized for critical real-time applications such as fraud detection[25], car-trip fare calculation[82], intrusion detection[83], stock trading[77] and even as a full social media application backend[38]. These use-cases inherently require both consistency and high-availability given that latency, and by extension downtime translates to a direct loss in revenue [48].

The volumes of data being continuously processed keep increasing and to keep up, stream processing deployments have scaled to thousands of nodes and terabytes of state [22]. However, as the number of participants in a computation increases, so does the probability that at any given moment, one of them may fail[54]. Alarmingly, annual failure rates in cloud environments seem to range from 4%[34] to 8%[103], with the likelihood of the failure of a machine increasing sharply with age. Fault tolerance in stateful stream processing is not as simple as in batch systems, due to the fact that all tasks are continuously operating, possess internal state and use nondeterministic operations. Both the state of each operator and every connection must be recovered in order to ensure that each input record affects the state of the computational Directed Acyclic Graph (DAG) *exactly-once*. Due to the difficulty of recovering consistent state in SPSs, the main fault tolerance mechanism used is converging towards periodic Chandy-Lamport-style [30] checkpoints of the system's global state [24, 29, 49, 56]. In this scheme, recovery requires rolling back the state of the computational DAG to a previous state, sometimes referred to as a *stop-the-world* approach [111], since all participating processes must pause for recovery. While a more localized treatment of the failure would be desirable, the entire

DAG must be rolled back to a previous state, because the nondeterminism present in the computations may cause participating processes' state to diverge.

While prior work has attempted to address high-availability in stream processing through the use of localized recovery and standby operators, they either support a limited set of pre-defined operators[50, 52, 73, 88] or they fail to address nondeterminism and state consistency[46, 65, 81, 95, 112]. Modern streaming users are not willing to make either concession, as they desire both the flexibility of user-defined functions and correct results for critical use-cases. Performance in these highly available systems is typically also limited due to the use of heavy synchronization protocols between replicas. Users are forced to choose between reliable production-grade systems or unreliable highly available research systems. Current stream processing systems are thus unable to provide both high-availability and consistency guarantees, while maintaining their original promise of low latency.

The evolution of SPSs can be split into four generations[16]. The first generation of systems offered no processing or consistency guarantees as they were merely adapted single-node DBMSs. The second generation focused on achieving distributed execution. The third generation improved on this by focusing on fault tolerance and managed state. We are currently undergoing a fourth generation, which focuses on IoT and edge processing. Parts of the processing (such as filters) are moved closer to the data sources, increasing performance by reducing the amount of data sent to the cloud deployment. Such scenarios increase the rate of perceived failures[17], as these are indistinguishable from disconnections, which are common at the edge. The current *stop-the-world* approach to fault tolerance is thus even more unfit for these systems, as it will lead to the inability to make progress under high churn. A more localized treatment of failures is thus imperative for these systems.

1.1 Motivation

The work presented in this thesis is motivated mainly by the following:

- Providing a thorough understanding of the stream processing landscape in aspects that relate to high-availability and consistency.
- Addressing the need for performant, consistent, and expressive localized recovery in stream processing, thus enabling high-availability in cloud settings and ability to make progress in edge settings.
- Exploring the design space for fault tolerance and adapting causal logging for use in distributed stream processing.
- Obtaining a comprehensive understanding of the efficacy and overhead of causal logging when applied to stream processing through practical evaluation, under synthetic and realistic workloads.

1.2 Problem Statement

The primary goal of this work is to explore causal logging as a fault tolerance solution for stream processing. With this, we aim to achieve a system capable of providing all the characteristics that users desire, namely high-availability and consistency under failure, without sacrificing expressiveness, and ensuring competitive performance both under failure and failure-free operation.

1.3 Contributions

The main contributions presented in this thesis are the following:

- A review of prior work on high-availability along with an analysis of rollback-recovery algorithms in the context of stream processing, providing a framework for understanding the challenges and pitfalls of different approaches.
- The design of Clonos, a highly configurable, scalable, performant and highly available SPS that offers faster non-blocking recovery with exactly-once processing guarantees without compromising expressiveness.
- A prototype of Clonos[1], implemented through the re-engineering of Apache Flink 1.7.2's distributed runtime.
- The evaluation of Clonos as a high-availability approach, in both synthetic and realistic workloads, for failure-free and recovery performance. We compare Clonos mainly to Apache Flink, as it is the state-of-the-art in regards to stream processing in both performance, reliability and expressiveness. We also configure Clonos in such a way that it emulates two classes of prior high-availability work and compare their performance to Clonos.

1.4 Articles

This thesis has generated a systems paper which we have submitted to SIGMOD 2021. No decision has been received however, as the process is currently delayed by the ongoing pandemic. We have received some positive feedback from reviewers, which we are currently in the process of integrating into the paper.

[1] github.com/delftdata/flink/tree/clonos1.7

1.5 Thesis Structure

The remainder of this thesis is organized thusly:

Chapter 2 studies the related work. In particular, this chapter begins by introducing some fundamentals on message passing systems, the system model used for the remainder of the thesis. Then, it introduces and analyses rollback recovery algorithms studied. Finally, we introduce dataflow systems in more depth and analyse the fault tolerance and high-availability mechanisms used in different systems. We also review previous work on high-availability in stream processing in detail, studying their capabilities and guarantees.

Chapter 3 discusses the implementation of Clonos in detail. To support such a discussion, it begins by justifying the choices we made in design and introduces the Apache Flink runtime in more depth.

Chapter 4 begins by explaining the experimental setting and methodology, and moves on to show the evaluation results, comparing Clonos with the base system and prior approaches.

Chapter 5 concludes the work with some closing remarks as well as an extensive list of future work directions.

Chapter 2

Related Work

In this chapter, we review past work related to this thesis. It begins by reviewing some preliminaries related to distributed and concurrent systems in Section 2.1. These preliminaries then allow for the introduction of the class of rollback recovery algorithms in Section 2.2, to which causal logging belongs. Finally, in Section 2.3 dataflow systems are presented. Before presenting streaming systems more formally, we first present batch dataflow systems, in order to highlight the differences between batch and streaming that make streaming fault-tolerance inherently difficult. We split our review of streaming systems into two classes. We present the design of production-grade Stream Processing System (SPS)s in Section 2.3.2.3 and argue that these target performance, expressiveness and consistency as they aim to support a variety of use-cases, but in doing so sacrifice high-availability. Highly-available systems are presented in Section 2.3.3, where we show that these systems have not left the research environment as solutions typically constrain expressiveness or sacrifice consistency. We conclude our review of the related work by examining the systems that most closely match the spirit of our work in Section 2.3.4.

2.1 Message Passing Systems

An asynchronous Message Passing System (MPS) is a common abstraction in distributed systems algorithms[1, 30, 41, 67]. In such a system, N processes communicate exclusively through message passing to achieve some computation. They communicate in an *asynchronous* manner, meaning no bounds can be placed on time taken to process or receive a message. Communication channels are assumed to be *reliable* and *FIFO*, as such we do not concern ourselves with the possibility of message loss or duplication, nor with the possibility of messages being delivered in the wrong order. Messages are first *received* by a process in the system, but are processed in the order they are *delivered* to the computation

running on the system, as shown in Figure 2.1. We assume a fail-recovery model, where processes fail independently, in a *crash-stop* fashion, meaning that they fail silently. They may later recover and rejoin the distributed computation. Similarly, network failures are modelled as crash-stop failures. Each process has access to a form of *stable storage*, whose contents survive failures. The form of stable storage need not be defined, but for our use-cases it is most commonly a Distributed File System (DFS). Of course, *volatile* memory is also present, and access to it is assumed to be faster than that of stable storage.

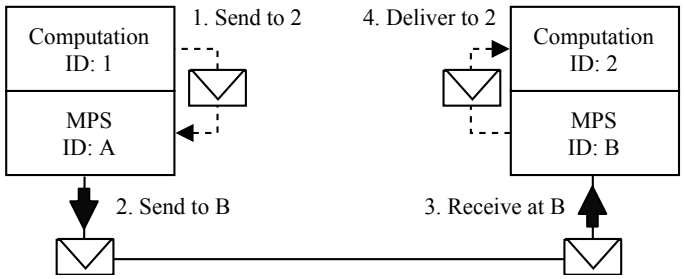


Figure 2.1: Difference between receiving and delivering a message.

Useful systems obligatorily communicate with the outside world, for example storing records in a database, sending a request to some external system, or writing to a terminal. We can model the entire outside world as a process, called the Outside World Process (OWP)[6], to which messages can also be sent.

2.1.1 Events

The transmission and reception of messages are events in an MPS. Sending a message is a deterministic event because a process always knows that it will send a message. However receiving a message is a nondeterministic event[6, 11], because a process does not know that it will happen. If two processes send messages to a third process, these may arrive in any order, which may affect the result of the computation. This is exemplified in Figure 2.2, where P_0 and P_2 both message P_1, however, these messages may take arbitrarily long to arrive, and thus may arrive in any order.

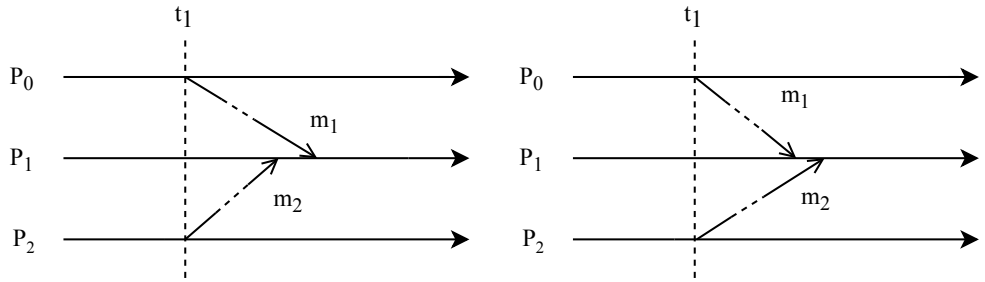


Figure 2.2: Nondeterministic order of arrival.

Similarly, many systems deliver messages in to the computation in a random order, by

for example delivering them in reception order, thus making order of delivery nondeterministic. Other kinds of nondeterministic events may happen in such a system. Accessing the current time is a nondeterministic event, because any given execution may return a different result. In much the same vein, so is generating a random number.

2.1.2 Virtual time and Logical Clocks

In order to build distributed protocols it is often very important to be able to order events. Synchronizing physical clocks in a distributed system is hard[67], so physical timestamps cannot always be used. Because of this, a notion of logical time needed to be developed. They were developed from the need to claim that an event preceded another.

Within a single sequential process, it is easy to see if one event happened before another. However, with a logical notion of time, it would be sometimes impossible to claim that one event on one process happened before another on another process, or the other way around. Because of this, the "happened-before" relationship is a partial ordering[67], denoted $\rightarrow$, defined as the smallest relation satisfying:

1. If a and b are two events within a single process, if a comes before b, then $a \rightarrow b$.
2. If a is the event representing sending a message, and b is the event representing receiving a message, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two events a and b are concurrent if $a \nrightarrow b$ and $b \nrightarrow a$.

Lamport clocks[67] are a method for ordering events in a distributed system, such that if an event a causally precedes an event b, then the timestamp $C(a)$ is smaller then the timestamp $C(b)$. Formally, $a \rightarrow b \Rightarrow C(a) < C(b)$. To implement them, processes follow the following protocol:

- Whenever a process i executes an internal event, its clock ticks: $C_i := C_i + 1$.
- Whenever a process i sends a message to another process, it piggybacks the timestamp of the send event, which is its current clock value.
- Whenever a process i receives a message from process j, it combines the timestamp received t_j with its own clock, by taking the maximum of the two: $C_i := max(C_i, C_j)$

However, Lamport clocks do not provide the reverse guarantee, $C(a) < C(b) \Rightarrow a \rightarrow b$, because even though one timestamp may be larger than another, the two events could be concurrent.

Vector clocks[74] are a generalization of Lamport clocks, which offers this guarantee. Each process i maintains an n-long vector clock C_i, such that $C_i[j]$ indicates the last event from j that causally affects i's current state. To achieve this, processes follow the following protocol:

- Whenever a process i executes an internal event, its clock ticks: $C_i[i] := C_i[i] + 1$
- Whenever process i sends a message to another process, it piggybacks its vector clock.
- Whenever process i receives a message from process j, it combines the vector received t_j with its own clock, by taking the component-wise maximum: $\forall k \in 1..n.C_i[k] := max(C_i[k], t_j[k])$

By having each process track the clocks of other processes in its own timestamps, we can now order events which with Lamport clocks would have been concurrent.

2.1.3 Consistency

A global state in a system such as described is the collection of the states of the individual processes and their communication channels. This global state is consistent if when the state of one process reflects the reception of a message, the state of the sender also reflect sending that message[30, 41]. Essentially, in the global state, a received message must have been sent, however a sent message may be *in-flight*, meaning it has been sent but not yet received or delivered.

A common way to visualize this is with cuts through system state as shown in Figure 2.3.

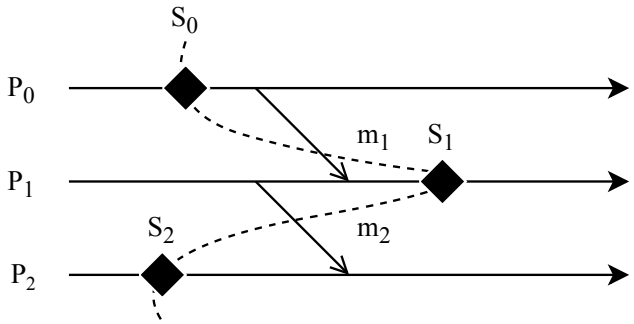


Figure 2.3: Inconsistent cut through a system

In this Figure, state S_1 is consistent with S_2 even though the sent message has not been received. However state S_0 is not consistent with state S_1 because a message which was received has not yet been sent. Thus as a whole, this cut is not consistent.

2.2 Rollback Recovery

Rollback recovery is a collection of methods to recover a consistent global state of an MPS after the failure of a set of processes. Recovery of consistency after a failure is achieved by executing a recovery algorithm, which will often roll back the state of some processes, such that all processes are consistent. However, we cannot roll back the state of the OWP,

as the outside world is a process that does not participate in recovery. As an example, imagine a fire detection system. Once the sprinklers have activated, you cannot roll back the fact that this happened. This is known as the *output commit problem*. Because of the output commit problem, when a process sends a message to the OWP, it must make sure that the state from which that is done is recoverable in the case of failures. Rollback recovery algorithms store information on stable storage, before output commit, so as to ensure that the state is recoverable.

Most of this section is heavily based on the excellent survey by Elnozahy et al.[41]. This section begins by introducing checkpointing-based approaches, then goes in detail on log-based approaches, with special interest in causal logging approaches, as it is a central component of this work.

2.2.1 Checkpointing-based Rollback Recovery

Checkpointing approaches[8, 30, 41] have the participating processes occasionally take *snapshots* of their state to stable storage. A consistent set of snapshots makes a global consistent checkpoint. When a process fails, normal execution is paused and a recovery algorithm is executed. When the failed process recovers, its previous state is lost, and it can only load one of its available snapshots from stable storage. However, to resume normal execution the participating processes must be in a consistent state. The job of the recovery algorithm is to calculate the latest consistent set of checkpoints available, called a *recovery line*. To conclude the recovery algorithm, each process will load their respective checkpoint of the recovery line, and execution resumes.

Checkpoints divide the execution of a process into intervals. The j'th checkpoint of process i is denoted $C_{i,j}$. Between two consecutive checkpoints $j-1$ and j of process i, an interval $I_{i,j}$ is defined.

2.2.1.1 Coordinated or Uncoordinated

Distributed checkpointing algorithms can be either coordinated[30] or uncoordinated[8]. Coordinated protocols immediately create a new recovery line by ensuring that processes take checkpoints at the same logical point in time. Uncoordinated protocols allow processes to take checkpoints at any time, and thus must track dependencies between checkpoints in order to compute the recovery line. Uncoordinated checkpointing allows each process to choose the time to checkpoint which is most convenient to it, such as when its state is smallest. Generally, uncoordinated checkpointing protocols incur overhead from tracking causal information during normal operation, while coordinated checkpointing incur coordination overhead, though the coordination overhead has been shown to be negligible[41]. Thus, coordinated checkpointing algorithms have a simple recovery algorithm, while uncoordinated algorithms must calculate the recovery line.

Uncoordinated protocols must be designed carefully to avoid the *domino effect*, which may happen if no recent recovery line can be computed, due to the fact that there is always

a message reception in a processes checkpoint which has not yet been sent in another processes checkpoint. This concept is illustrated in Figure 2.4, where due to a failure in P_1 the recovery line begins with the three most recent checkpoints $C_{0,3}, C_{1,2}, C_{2,2}$. because m_4 has been received in $C_{0,3}$, but not sent in $C_{1,2}$, we must rollback the 0'th process further, to checkpoint $C_{0,2}$. This rollback leads to the fact that m_3 is now received in $C_{2,2}$, but not yet sent in $C_{0,2}$, so we must rollback the second process to checkpoint $C_{2,1}$. This process may continue indefinitely, until we reach the start state of the system.

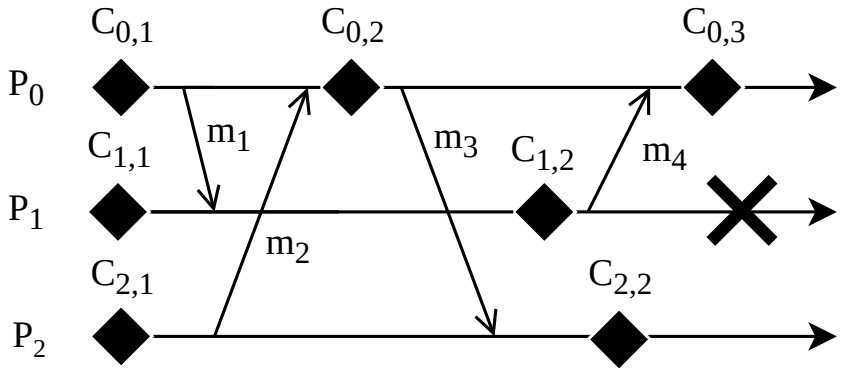


Figure 2.4: Example where uncoordinated checkpointing leads to domino-effect

Several variations on uncoordinated checkpointing have been developed, which aim to avoid the domino effect, such as communication induced checkpointing and model-based communication induced checkpointing[8]. These protocols piggyback even more information on their messages, which is used to decide when to force checkpoints to avoid heavy rollback.

2.2.1.2 Blocking or Non-Blocking

In coordinated checkpointing, one must ensure that the individual checkpoints happen in a consistent point in time. To be explicit, for a message m sent from i to j, if $C_{j,k}$ reflects the event $receive(m)$, then $C_{i,k}$ must reflect $send(m)$.

A simple checkpointing protocol[96] may behave like a two phase commit protocol[20], where an initiator first takes a checkpoint and then sends a checkpoint request to all other processes. When processes receive this request, they perform their own checkpoints (including the state of their channels), and acknowledge the initiator of their success or failure in performing the checkpoint. The initiator can then decide whether the global checkpoint is valid or should be discarded. When processes are informed of the decision they may resume processing. This is a *blocking* algorithm, because during the execution of the checkpoint, processes stop their computation.

The alternative is *non-blocking* algorithms, such as Chandy-Lamport's distributed snapshot algorithm[30]. This algorithm builds on the assumption that FIFO channels will order checkpoint markers with the application messages. The initiator begins by taking a checkpoint of its state and then sending a marker along each of its channels.

Upon receiving the first marker on a given channel, a process checkpoints its state and marks the state of that channel as empty. The process also sends a marker on each of its outgoing channels. It then records incoming messages on other input channels, up to the point where the markers arrive. The result is a consistent global checkpoint, including the state of the individual processes and their channels, without blocking the entire computation.

2.2.1.3 Synchronous or Asynchronous

Blocking and non-blocking checkpointing is distinguished from *synchronous checkpointing* and *asynchronous checkpointing*. Asynchronous checkpointing happens when the act of flushing the state to stable storage is not on the critical path of the system, such as when it is done by a separate thread. This can be achieved by taking an in-memory copy of the state, and spawning a thread to asynchronously flush it the copy, while simultaneously working with the original data. Nowadays Multi Version Concurrency Control[19, 37] key-value stores provide this capability with some ease.

2.2.1.4 Incremental Checkpointing

In order to further reduce the overhead of checkpointing the state of an application, certain systems choose to provide *incremental checkpointing*[22, 41]. With this feature the changes performed to state since the last checkpoint are collected and act as the next checkpoint. An incremental checkpoint is the difference between the current state and the previous checkpoint. This works best in an application with very large state, but comparatively low amounts of modifications to it. A compaction algorithm must be run occasionally to combine several incremental checkpoints into a full checkpoint. This reduces memory consumption and eases searches.

2.2.2 Log-based Rollback Recovery

The lifetime of a process can be seen as a sequence of intervals separated by nondeterministic events. By definition, inside a state interval a process evolves independently and deterministically. Sending a message from one process to another then creates a new state interval for the receiving process as receiving a message is a nondeterministic event. The sender's current state interval is then causally linked with the receiver's new state interval. If a failure of the sender were to happen, either the receiver must be rolled back or the sender must be rolled forward in a deterministic way to reach its pre-failure state. Log-based rollback recovery approaches reach a consistent state with the rest of the MPS by replaying the nondeterministic events generated pre-failure at the recovering process. In most cases this completely avoids rolling back the state of other processes. Instead only the replacement for the failed process is deterministically *rolled forward* to meet its peers. Log-based approaches can be combined with checkpointing approaches to allow limiting how far back the replacement process starts its recovery from.

These approaches rely on the PieceWise Deterministic (PWD) assumption[41]. This assumption states that all nondeterministic events can be identified and their determinants logged. To reproduce a nondeterministic event e, one must store the event and its *determinant*, represented as #e, which is a piece of data that removes the nondeterminism of an event. As an example, suppose the event is generating a random number, the determinant of that event is the number generated. If the nondeterministic event is receiving a message, then the determinant is the order of delivery. In other work from the same authors[7], the PWD assumption is stated differently as *"The only source of nondeterminism is the order of delivery of messages"*. This other definition is fundamentally different as in most realistic use-cases other forms of nondeterminism are present. In this thesis, we refer to this other assumption as the Ordered Delivery Deterministic (ODD) assumption. This name is chosen because if the order of delivery was fixed, the system would be deterministic.

Thus, if a process re-executes the same nondeterministic events, in the same order, it will evolve to the same state as before failure. If all determinants are available, other processes need not be rolled back, as the failed process will reach its pre-failure state. However, having the determinants alone is not enough to replay the nondeterministic events. To replay message reception events, it is required that the message contents be replayed as well. This can be done in one of two ways. Either the receiver can log the message contents together with the determinant or the sender can keep a log of the sent messages which are not yet stable called an *in-flight log*. The second case is more common as the first requires logging possibly large messages in stable storage. The in-flight log can instead be kept only in volatile memory.

2.2.2.1 Orphan Processes

An orphan process is defined as a process whose state depends on a nondeterministic event that cannot be reproduced during recovery[41]. If a nondeterministic event cannot be reproduced, then the state of all processes must be rolled back to before that event, in order to provide consistency.

The *always-no-orphans* property[6], is used to reason about the effects of a nondeterministic event e:

$$\forall e : \Box(\neg Stable(e) \implies Depend(e) \subseteq Log(e)) \tag{2.1}$$

Where $Depend(e)$ is the set of processes whose state was affected by e. This set can be built by induction starting with the process where e occurred and adding to the set any process that receives a message from a process already in the set. In other words, dependence propagates according to the *happens-before* relationship. $Log(e)$ is the set of processes that have logged e's determinant in volatile memory and $Stable(e)$ is a predicate which becomes true when e's determinant is logged in stable storage. Finally, the operator $\Box$ is the temporal always operator. It is thus apparent that this is a safety property of the system.

This condition states that to ensure no orphaned processes at all points in the execution of a distributed computation, for every nondeterministic event e that a process executes, if its determinant is not yet in stable storage, then all processes whose state depends on event e must have logged its determinant in volatile memory.

2.2.2.2 Pessimistic Logging

Pessimistic logging protocols[41], such as in the write-ahead logs of database literature [20], implement a stronger property than the always-no-orphans property[6]:

$$\forall e : \Box(\neg Stable(e) \implies |Depend(e)| \leq 1) \tag{2.2}$$

Essentially, for a process to be able to send a message to another process, it must first ensure that all determinants are stable. The cost of accessing stable storage is often prohibitively high, however these protocols come with several attractive advantages. First, any process may send messages to the OWP without coordination. Second, any state in which the system is observed is recoverable. Rollback of other processes is unnecessary and recovery is simple. However, in general, failure-free execution has very high overhead.

2.2.2.3 Optimistic Logging

Optimistic logging protocols[41, 94] have the lowest overhead during failure-free operation. They achieve this by allowing for the temporary creation of orphans, but guaranteeing that by the time recovery is finished, no orphans will exist. Thus they do not ensure the always-no-orphans property. The property they provide is shown in 2.3[6], where $\mathcal{F}$ is the set of processes assumed to fail concurrently during execution and $\diamond$ is the temporal eventually operator:

$$\forall e : \Box(\neg Stable(e) \implies (Log(e) \subseteq \mathcal{F} \implies \diamond(Depend(e) \subseteq \mathcal{F}))) \tag{2.3}$$

This property states that if a determinant is not stable, then if all the processes which have logged it fail, then eventually only the processes which have failed will depend on it. This essentially means that after a failure, eventually there will be no orphans, because processes will roll-back until their state does not depend on a lost determinant. These protocols are implemented by occasionally logging their determinants to stable storage, often asynchronously. During failure-free operation they must track causal dependencies between process's state intervals, such that during recovery, a consistent state may be achieved. Though they have low failure-free overhead, optimistic protocols come at the cost of slow output commit, which requires a lot of coordination and complex recovery.

2.2.2.4 Causal Logging

Causal logging[11, 40] attempts to get the best of both worlds. It offers low overhead, the ability for each process to independently commit output to the OWP and ensures

the always-no-orphans property, while also removing access to stable storage from the critical path, except when committing to the outside world.

While pessimistic logging ensures no orphans by ensuring the antecedent of property 2.1 is true, causal logging focuses on ensuring the consequent. That is, causal logging ensures that all processes that depend on an event have logged its determinant. If a set of processes $\mathcal{F}$ fails, then for all events e either $Depend(e) \subseteq Log(e) \subseteq \mathcal{F}$, in which case there is no orphans, or $Depend(e) \subseteq Log(e) \not\subseteq \mathcal{F}$ in which case at least one surviving process has the determinant of e, and can share it with the recovering processes. In order for a process to message the outside world, it must ensure that the determinants it depends on are stable, however this can be done with no coordination.

These protocols can be made optimal by ensuring that no unnecessary determinants are sent to processes that do not depend on them. This is done by strengthening the always-no-orphans property, as shown in property 2.4:

$$\forall e : \Box(\neg Stable(e) \implies ((Depend(e) \subseteq Log(e) \wedge \diamond(Depend(e) = Log(e))))) \tag{2.4}$$

This can be interpreted to mean that, while e is not stable, all processes dependent on e must have logged it, and eventually the ones who have logged it will be no more than those who depend on it. However, processes only depend on events of other processes if they receive application messages from those processes, because those events happened-before the delivery of the message. It should thus be evident that there is no need to send extra messages containing determinants, since the determinants a process needs can be *piggybacked* on the message that makes it causally dependent on those determinants.

Finally, if the number of possible concurrent failures is bound to not be greater than f, it is possible to implement stable storage while avoiding disk access by logging to $f+1$ processes. This observation forms the basis for Family-Based Logging (FBL)[9] protocols. The adapted correctness property for FBL is shown in 2.5. Of course, in this case, one process may avoid sending its determinants to processes which have not logged them if enough processes have already logged them, such that they are now considered stable. This class of protocols reduces overhead by avoiding stable storage entirely, even during output commit. Implementations for $f = 1$ have been shown to have very small overhead[6]. Another special case is when $f = N$, which is the protocol that Manetho[39, 40] implements. Manetho, like most other work in the area of causal logging, works on the ODD assumption, that is, the only source of nondeterminism is order. Since all processes may fail, determinants must be completely shared, which can be done by tracking message deliveries only. Manetho does this with the antecedence graph, an efficient data-structure that tracks the state intervals of processes. Manetho also uses checkpointing to be able to truncate the antecedence graph. Checkpoints are uncoordinated and include the in-flight messages, the antecedence graph and the state of the application. In [40], it is shown that Manetho and by extension causal logging has low failure free overhead experimentally.

$$\forall e : \Box((|Log(e)| \leq f) \implies ((Depend(e) \subseteq Log(e) \wedge \diamond(Depend(e) = Log(e))))) \tag{2.5}$$

In the following, a general protocol for causal logging is described from the perspective of process p. Process p maintains a determinant log L, which is a mapping from processes to the determinants of the events on which the state of p depends. Processes maintain additional data-structures where they record which determinants are stable and have been received by which processes. Additionally, processes piggyback determinants on application messages, sending the determinants they believe the receiving process to not yet possess.

1. When a process p receives a message m, before delivery, for all determinants piggybacked on m, which originate in process q, p appends to $L[q]$ those determinants.

2. When a process p delivers m, it records in $L[p]$ the delivery sequence number.

3. During p's processing of m, more determinants may be generated (random numbers, timestamps), which are appended to $L[p]$.

4. Whenever p wishes to send a message to a process q, it will consult its log and data-structures, to compute the set of determinants that q should receive. These determinants are piggybacked on the message sent.

Intuitively, this approach recovers failed process p, by having it send the non-stable determinants it has generated piggybacked on every message it sends. This means that other processes, if causally affected by p, will know how to guide its recovery. They can share with it, for example, the order in which p previously received messages or generated random numbers such that it can reach the same state as before.

Causal logging protocols track the causal dependencies between state intervals of processes. Each process logs the determinant of every non-stable event that causally affects it. Thus, each process maintains a determinant log, which acts as an insurance against failures of other processes. Figure 2.5 provides an example execution of causal logging combined with uncoordinated checkpointing. When a process takes a checkpoint, determinants it had previously logged become stable. This is why $P0$ does not have $\#m2$ in its log, since $P2$ has logged it in stable storage. $P2$ took a checkpoint so it could send a message to the OWP, this was done without coordination. As an example, if $P2$ failed, $P2'$ would first recover from its latest checkpoint. $P2'$ then requests and receives the replay of messages sent since its last checkpoint. At this point, $P1$ can guide its recovery, because it knows that m_5 was delivered before m_6.

Other ways in which causal logging protocols may vary are the way in which they track the number of processes that may have logged a given determinant[10], which we explore next in Section 2.2.2.5.

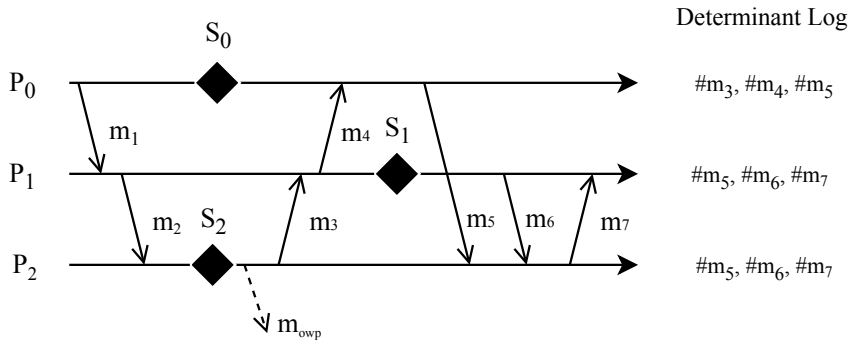


Figure 2.5: Causal logging example

2.2.2.5 Trade-offs in Membership Tracking

Whenever a distributed system follows the ODD assumption, which is that the only source of nondeterminism is message delivery order, the determinant of a message m, denoted #m, can be captured by the tuple $< m.src, m.ssn, m.dest, m.dsn >$. src and $dest$ are respectively the sender and receiver of the message, while ssn and rsn are the send sequence number and the delivery sequence number. This is of course assuming that the sender remembers the messages sent. These sequence numbers then allow a process to request replay from another process, starting at a given send sequence number. The deliver sequence number says the order of delivery, required for delivering in the same order.

In order to avoid redundant sharing of unnecessary determinants, processes may share some metadata regarding who knows about what determinants. This would allow processes to eventually stop sending determinants to processes that have already received them. But often, the amount of data to be sent is large, and to reduce the amount of determinants sent, more metadata must be sent. In [10], the authors explore this exact trade-off, presenting six causal logging protocols which share different amounts of data, under the ODD assumption:

- $\prod_{Det}$: A process p tracks who has received which determinants from itself only. If they have not yet received them, then they are piggybacked on the next message. No aditional metadata is sent.

- $\prod_{|Log|}$: A process p tracks the number of processes that have logged which determinants. When process p receives a determinant of a message m sent by process $m.src$ to process $m.dest$, and p is neither of those processes, then p knows that $|Log(m)| \geq 3$. A process which receives this metadata from p then knows that $|log(m)| \geq 4$. When $|log(m)| > f$, processes stop sending the determinant.

- $\prod_{Log}$: A process p tracks the identifiers of the processes that have received a given determinant. This way, a process will never send a determinant to a process it knows has already received it once. When receiving a determinant, the union of

the two sets of identifiers is taken. When $|Log(m)| > f$, processes stop sending the determinant.

Additionally, versions of the above protocols are specified, which additionally inform other processes of changes in the stability of storage of a determinant:

- $\prod^{+}_{Det}$: Additionally, informs other processes of which determinants have become stable from its perspective.
- $\prod^{+}_{|Log|}$: Additionally, informs other processes it had already told about the determinant of m of changes in $|log(m)|$.
- $\prod^{+}_{Log}$: Additionally, informs other processes it had already told about the determinant of m of changes in $log(m)$.

While $\prod_{Det}$ would piggyback much less information on each message, it runs the risk of sending determinants to processes that have already received them. Additionally, for $f > 3$ this protocol is not able to recognize that a determinant is stable, it must be explicitly informed. This is because, since no metadata is sent, a process receiving #m, can only assume that the holders of #m are itself, and $m.src$. On the other hand, $\prod^{+}_{Log}$ will rarely send redundant determinants, and will know a determinant is stable as soon as possible, but will piggyback a lot of metadata in order to achieve this. In general however, $\prod_{Det}$ is a good choice for $f < 3$, since only determinants need to be sent, and processes are able to recognize that they are stable.

Another good use-case is shown for when the channel graph - the graph of communication channels - is acyclic and shortcut-free. Then $\prod_{Det}$ is as efficient as $\prod^{+}_{Log}$, when $f = N$. If the channel graph is additionally a tree (meaning it is acyclic, shortcut-free and each node has only one parent, except for the root) then this is true for $f \leq n$.

In order to efficiently propagate the information of the protocols, a dependency vector may be used, first introduced in [94]. First note that in a PWD system, |Depend(e)| may be used to estimate |Log(e)|, since Depend(e) $\subseteq$ Log(e). Additionally, remember that processes are assumed to be deterministic, except in the order of delivery of messages. This means that each state interval is started by a deliver event. In [7], the approach is explained in detail, we urge the interested reader to refer to it, as it is heavily condensed here. By having each process i maintain a vector clock, called a dependency vector DV_i, which increments only on deliver events, it is possible to know which events causally precede a certain event e. For two messages m and m', delivered to processes p and q respectively, the following holds, because the DV are vector clocks:

$$deliver_p(m) \rightarrow deliver_q(m') \equiv DV_p(deliver_p(m))[p] \leq DV_q(deliver_q(m'))[p] \tag{2.6}$$

This means that $DV_p(deliver_p(m))[q]$ is the index of the latest state interval of q which affects p. To know whether a process depends on a determinant, one can use the following implication: $DV_q[m.dest] \geq m.dsn \Rightarrow q \in Depend(m)$.

To track dependencies between processes, each process maintains a $N * N$ matrix called DMat. Process p maintains in row p its dependency vector, and in all other rows, its estimate of the other processes dependency vectors. To keep it up to date, whenever process p receives a message m (with attached metadata) from q, it executes an update rule, different for each protocol.

As an example, to implement $\prod_{det}$, the metadata sent, which efficiently encodes all determinants, is simply an N long vector named PBC(m), where PBC(m)[p] is the maximum $m.dsn$ for all determinants #m piggybacked in the message, where $m.dest = p$. To update it, p first increments $DMat[p,p]$, then sets DMat's p^{th} row to the component-wise maximum of itself and v, and does the same for the q^{th} row and v. Finally, it updates the diagonal of the matrix to the maximum of itself and the corresponding entry in v.

To implement $\prod^{+}_{log}$, the full DMat matrix is piggybacked on each message sent from q to p. When p receives the message m it first increases $DMat[p,p]$, then takes the component-wise of the two matrixes $DMat_p$ and $DMat_q$.

A process p may estimate the size of Log(m) by counting the number of processes q such that $DMat[q, m.dest] \geq m.dsn$. More intuitively, by looking at the m.dest column and counting the entries equal to or above the original receive sequence number of the message, which is also the state interval initiated. When this value is above f, process p may consider m to be stable.

2.3 Dataflow Systems

Dataflow systems emerged in response to a need for scalable data-processing that traditional Database Management System (DBMS)s could not fulfill. Dataflow programs are structured as data-parallel flows of data between a Directed Acyclic Graph (DAG) of operators[4, 16, 45]. Each of these operators applies a transformation to a collection of data items. Structuring programs this way allows for easier scaling through the use of both data and task parallelism.

Users create their programs in a high-level API, imperative[28, 43] or declarative (SQL), using abstractions such as collections and transformations on those collections. This program is translated by the system into a high-level plan of the dataflow to be executed called the *logical execution plan*. The system can then parallelize and choose the algorithmic implementations of operators, pick the connection types between logical operators, choose a scheduling of operators to tasks and of tasks to hosts, and apply several optimizations thus producing a *physical execution plan*.

Figure 2.6 illustrates this process, using the common WordCount example. An input collection is first filtered, removing elements lacking punctuation, then a tokenizer function is applied, breaking sentences down into words. This collection is then partitioned, using the words as key and the results are summed. We see that the reduce operator is translated into a running reduce. This is an operator that simply maintains the state of

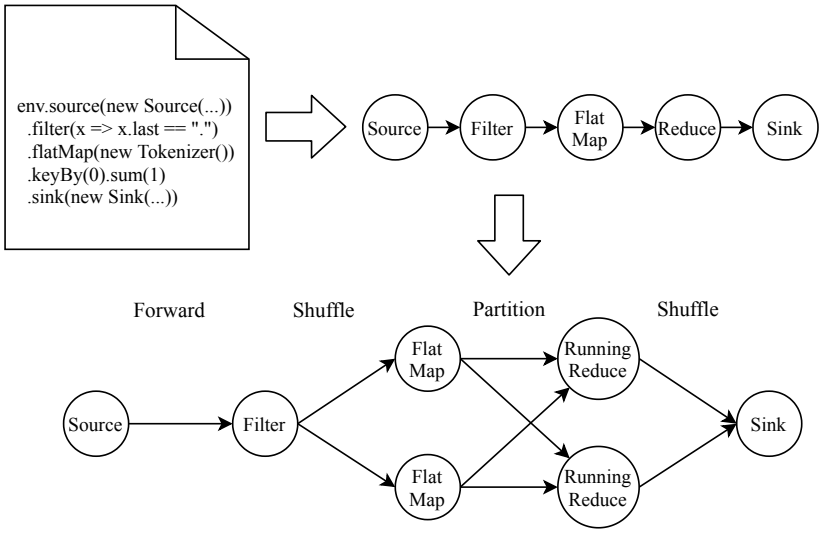


Figure 2.6: Dataflow programs, logical and physical execution plans.

the reduction as it runs, given that to compute a sum, one does not have to keep all past records, only the running sum.

In this work we are particularly interested in distributed dataflow systems due to their scalability and fault-tolerance. Two other important components that led to the popularity of these systems is that they generally offer transparent recovery mechanisms, alleviating users from implementing complex fault-tolerance logic, and also automatically schedule tasks with data locality awareness[55, 109]. Distributed dataflow computations fit the asynchronous MPS model, which allows us to pull from the rich literature on rollback recovery. To keep discussion clear, nodes (or hosts) and messages will refer to MPSs, while operators and records will refer to dataflow computations. In Figure 2.7, an example assignment of a dataflow computation to an MPS can be found. Importantly, dataflow systems must somehow receive data from an external system, and put results into an external system, both of which can be represented by the OWP (see Section 2.1). *Event sources* (sensors, Internet of Things (IoT) devices, other systems) generate events which are processed by the dataflow system. Event sources should not be confused with the *source operator* (shown as i in the Figure), the operator which translate events to records that can be processed by the system. Similarly, the place where data is put after processing is a *event sink*, which also has a corresponding *sink operator* (shown as o in the Figure). These may often be a database, message queue or even highly distributed filesystem. Another physical optimization is shown in Figure 2.7, where operators i and w are pipelined[51] into a single operator executing both. An operator w in a plan, that produces that to another operator x, is said to be *upstream* from x. Similarly, x is *downstream* from w. Source operators have no upstream, while sink operators have no downstream.

To describe the guarantees offered by the system, processing and delivery semantics [3, 4, 22] are typically used. These semantics are used to describe the fault-tolerance of a

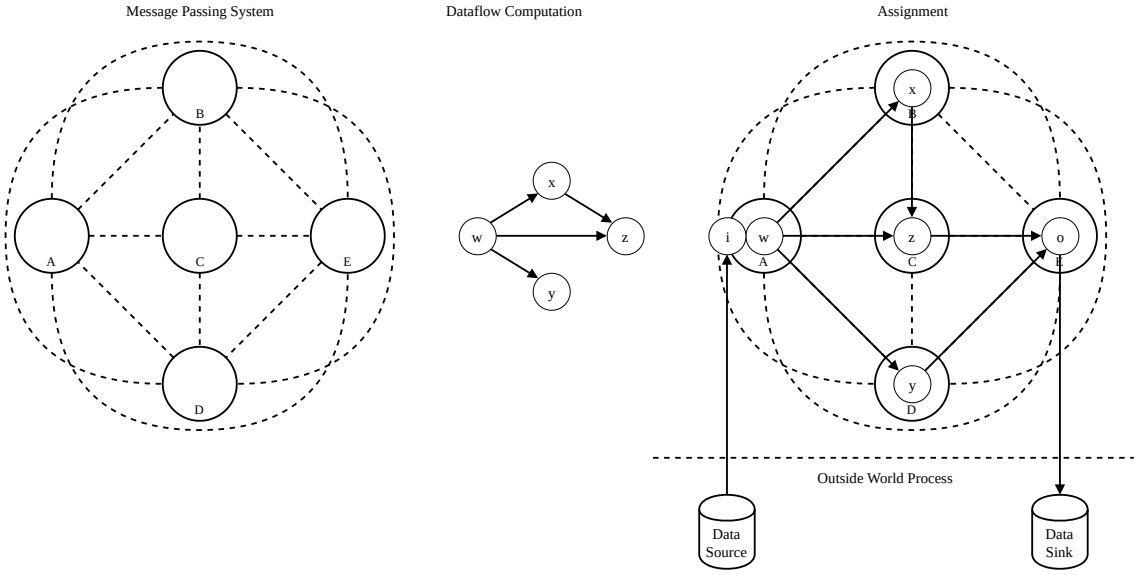


Figure 2.7: Example assignment of dataflow operators to message passing nodes.

dataflow system, as they refer to how many times a record is reflected in the distributed state of the system even in the face of failures, the possibilities being at-most-once, at-least-once, and exactly-once:

- **Processing semantics**: The guarantees offered by the system as to how many times the input records will affect the internal state of the system. At-least-once processing semantics means that each input record will be processed (and thus affect internal state) at least (but possibly more than) once by each vertex in the DAG.
- **Delivery semantics**: Often also called end-to-end processing semantics, these are the guarantees given across the system and its connections with the OWP. Exactly-once delivery would mean that each record affects the internal state of the system once and its corresponding results are sent once to the outside world data sink.

Delivery semantics can only be as strong as processing semantics as they extend them to the OWP. These semantics are important ways to classify the fault-tolerance provided by a dataflow system. During correct execution it is very easy for each operator to process its input exactly-once. However, in the presence of failures, one must be careful in the way one recovers otherwise records may be duplicated or lost. In the literature, these semantics are often reported, but formal definitions are lacking. It is however evident that these guarantees are heavily related to state consistency as defined in Section 2.1. If a system cannot guarantee a consistent global state after failure, then it cannot provide exactly-once processing. Guaranteeing at-least-once processing is similar, but allows some processes to send messages which are never received, which translates to some processes seeing some records multiple times, while others only see them once. Providing at-most-once processing allows breaking consistency entirely, as some processes may

process some records while others do not. In examining the fault-tolerance of dataflow systems, we will show what kinds of processing and delivery semantics they can offer.

From dataflow systems two categories are highlighted: *batch* processing systems, which operate on *bounded* collections of data, that is data which is finite and fully present such as a file or database table. *Stream* processing systems, on the other hand, operate on *unbounded* collections of data, which are continuously arriving and must be processed online. As we will show in Section 2.3.1, processing and delivery guarantees are much easier to offer in batch systems than in streaming systems. Before diving into the central topic of this work, streaming systems, batch systems are first reviewed in order to highlight the differences between them that make the fault-tolerance of streaming systems more difficult.

2.3.1 Batch Processing Systems

Batch processing systems specify computations on a bounded static dataset. This input dataset is generally stored in a DFS, which also hosts the batch processing system. The data is thus generally replicated and partitioned, prior to the computation starting. Connections between operators express data dependencies. Each operator in the DAG may only begin processing once all upstream operators have finished processing.

MapReduce[35] was the first majorly popular instance of this processing model. It allowed a computation to be specified in terms of a Map and a Reduce operation, with an implicit shuffle of data in between. Mappers act as source operators, reading data from the Google File System (GFS)[44], while reducers acted as sinks, writing to it. Intermediate results would be written to disk. Later Hadoop, an open-source implementation of MapReduce, became massively popular. Hadoop replaced GFS with the Hadoop Distributed File System (HDFS)[90], a scalable distributed file system. MapReduce was inflexible as a framework, allowing only one stage of Map and Reduce operations. As requirements became more diverse, users began chaining several MapReduce jobs to achieve more complex computations. Eventually this led to the appearance of frameworks such as Hive[98] and Pig[84], which allowed dataflows to be expressed in higher level, more expressive languages and automatically translated to several MapReduce stages. Nonetheless, intermediate stages of these chained computations were inefficient, as they had to fully complete before starting the next stage, while also writing and reading every intermediate result to and from HDFS. **Dryad**[55], a research system from Microsoft, improved on MapReduce by introducing native support for complex multi-stage dataflows. Any computational DAG could be expressed, with virtual input and output operators. Performance was improved by removing the need for materializing intermediate results and introducing operators other than maps and reduces. Due to its closed-source nature, Dryad never grew much in popularity. **Spark**[109] appeared a few years later, borrowing many ideas from Dryad, however building them on the open-source Hadoop ecosystem. It introduced many important performance improvements, such pipelining tasks. For

example, a map operator followed by a filter operator may be executed in a pipelined fashion, meaning the operators can be pipelined, as no intermediate materializations or data exchange to other hosts need happen between them. One important contribution that Spark made was introducing the Resilient Distributed Dataset (RDD). This abstraction allows users to write dataflow programs as simple sequential programs, while also providing fault-tolerance through automatic lineage tracking and recovery[108]. The RDD also improved performance over previous systems by allowing as much computation and data exchange as possible to happen in memory, removing intermediate materializations.

The most common fault tolerance mechanism in batch processing systems is lineage recovery. All three systems we have introduced use it either directly or indirectly. Lineage based approaches[104, 108] differ from general rollback-recovery approaches in that they only apply to dataflow systems with large coarse-grained and deterministic computations. When the computation is first submitted, a partitioned execution plan is generated, including the coarse-grained operations that tasks apply and data dependencies between tasks. This is recorded as a *lineage graph*. After a failure, lost tasks and partitions of data are identified and their inputs noted. To reconstitute the lost data, one simply needs to apply the same transformation to the inputs of the lost tasks. An example of this is shown if Figure 2.8. Here a stable input collection A is partitioned into to two, partition 1 and partition 2. Transformation T_1, transforms dataset A into dataset B, and similarly T_2 transforms it into C. The failure of task C_1, perhaps due to an out of memory error, requires node 1 to reapply T_2 to B_1. However a failure of the entire node, would require both T_1 and T_2 to be reapplied to A. This resembles a form of pessimistic checkpointing, as any given task must wait for its inputs to be stable before committing output.

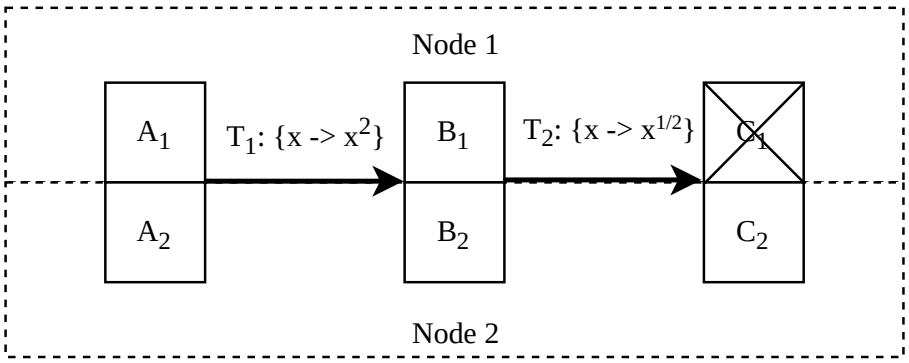


Figure 2.8: Lineage reconstruction

2.3.2 Stream Processing Systems

Streaming systems are designed for real-time low latency and high throughput data processing. As we have shown in Chapter 1, however, a growing trend has been their use for building reactive applications, which brings in the unanswered requirement for high-availability. We will start by describing the terminology used in SPSs. Though this work

is mostly concerned with the underlying distributed runtime of an SPS, it is important that we understand the operations executed, so that we may identify sources of nondeterminism. As such, we will also introduce common components of the stream processing model. Then, we will examine three kinds of systems for their fault-tolerance and high-availability capabilities.

2.3.2.1 Distributed Runtime Model

We base our model of an SPS in the one presented in [52], as it is general enough to allow most systems to be translated to it. Streaming systems also express their computations as a DAG of operators, connected by *streams*. A stream is a sequence of records generated in real-time which must be processed according to some notion of time. A record is typically a 3-tuple (K, TS, C) where K is a record key (which may be null), TS is a timestamp in the chosen notion of time (which may be null) and C is the record's contents. The K component is used to create logical *keyed streams*, streams of records that all share that same key. In such a case, the operator state is partitioned according to the keys as well, meaning that a record with a given key can only access and modify the state for that particular key.

Since the data processed is unbounded, meaning it is potentially infinite and continuously being created, all operators in the graph must be simultaneously deployed. This means that while batch processing exploits *task* and *data parallelism*, streaming additionally exploits *pipeline parallelism*. Operators process records one by one, receiving them from input streams and outputting them into output streams. Since they are continuously deployed, operators may need to be *stateful*, meaning they maintain some state. Modern SPSs typically offer *managed state*[3, 24, 57, 82], meaning that the streaming system can manage the state of operators, making it fault-tolerant, being capable of moving it between nodes, partitioning it and moving it between volatile memory and disk on demand. SPSs with managed state keep operator state local to the operator, instead of in a remote store, greatly improving performance by reducing the latency to state access.

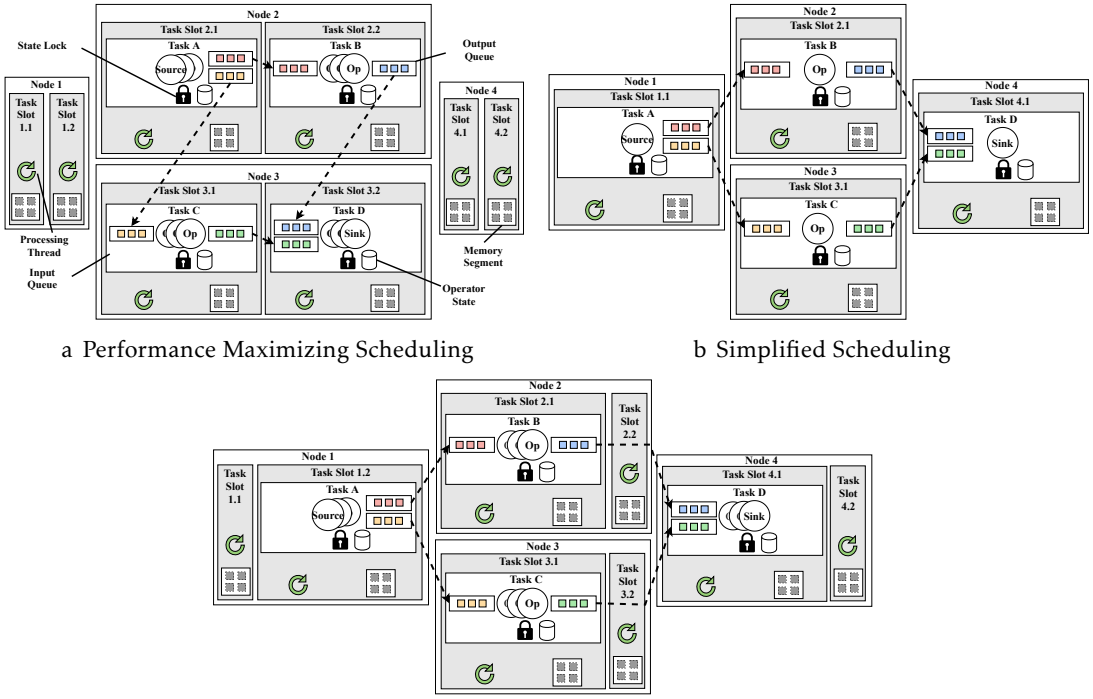
A streaming *task*, which executes some pipelined set of operators, has an *input queue* per input channel on which it receives a stream of records from upstream. These records are then delivered to the pipelined operators, allowing them to affect the operators' state. The streams of records produced as a result are similarly put into *output queues*, and eventually sent downstream by the underlying MPS. At any given point, the state of a streaming task, executing an operator, is the state that the operators hold, the state of the task's input queues and output queues. The way that a given operator o with parallelism π_1 shares their output with physical instances of the downstream operator d, ranging from d_1 to d_{π_2}, may vary. We show a few connection types in Figure 2.6. Five connection types are common:

- **Forward**: o_i will send its output to a single downstream physical operator d_j. Used when $\pi_1 = \pi_2$.

- **Shuffle**: o_i will round-robin among downstream physical operator instances. Used to rebalance when $\pi_1 \neq \pi_2$.
- **Random**: o_i will randomly choose the next downstream physical operator instance to send to. Used to balance load.
- **Hash-Partition**: o_i will use hash partitioning on the record key to select a downstream operator to send to. Used when it is important to key-by a certain field in the record for downstream aggregation.
- **Broadcast**: o_i will send the record to all downstream operator instances. Used when some data is to be shared with all downstream operator instances.

Streaming systems take advantage of data parallelism. Data parallel systems improve performance by partitioning data and processing it in parallel. They also improve performance by reducing the amount of shared state, and thus of synchronization between processing threads. In essence, data is partitioned either randomly or according to the K component of records, and each task processes a partition. A task has a single processing thread and needs not to synchronize with other tasks. However, in streaming systems there are often asynchronous actions (e.g. checkpoints) which may affect operator state, and thus, for each task there is often a *state lock*, used to synchronize access to operator state between the main processing thread and other asynchronous actions. Operator pipelining improves performance but cannot occur whenever a *pipeline breaker* exists in the execution plan. Pipeline breakers include partitioning connection types and shuffle connections that connect two operators with different parallelism levels, also known as a rebalance. When a pipeline breaker is present, a pipeline must be broken into two separate pipelines, and thus multiple tasks.

A node contains several *task slots*, where a task may execute. Each task slot provides a task with a memory segment and a processing thread. There are different scheduling strategies[97] for stream processing, some maximizing performance[24] and some maximizing availability[52] and yet others maximizing aspects like resource usage. Performance focused scheduling attempts to host as many connected tasks as possible in the same host, reducing the amount of connections over the network. Availability focused scheduling on the other hand does the opposite, placing as many tasks as possible in separate hosts, which increases the number of network connections, but reduces the number of failed tasks when a host fails. In Figure 2.9, we represent hosts, tasks, task slots, threads and state, and show examples of both scheduling strategies. We also show the simplifying assumption we make in this work, which is that a node hosts a single task which contains a single operator, allowing us to refer to failed operators, tasks or nodes equally. We ignore pipelined operators, as they are equivalent to a single more complex operator. This is only for ease of discussion, as Clonos as both an algorithm and a system prototype is capable of handling any kind of scheduling.



a Performance Maximizing Scheduling

b Simplified Scheduling

c Availability Maximizing Scheduling

Figure 2.9: Streaming Task Scheduling

2.3.2.2 Streaming Concepts

The unbounded nature of streaming data means that many operators would block forever without outputting results. Windowing [68] offers a solution to this problem, by allowing these operators to execute over a window (or view) of recent data. Windows can be described by a few aspects such as the function they execute over the window, how they are *triggered* and how records are *assigned* to windows. When triggered, window operators apply their function over the set of records in the window. Windowing is thus the backbone for many streaming operations such as joins, sorts, reductions and aggregations. Windows, and by extension all these mentioned operators are inherently stateful as they must maintain the state of their windows. Regarding window assignment, typically two properties, size and slide, defined on some notion of time, are used to break time into windows. Size defines how large the window is in units of the notion of time and slide defines by how much the window should advance in that notion of time, after each time it triggers. Input records are assigned to any window which covers their TS component. Users can define the notion of time used by operators of the streaming system and computed results change depending on the choice. Four notions of time are common[4], which are set by changing the way that the TS component of a record is assigned and thus change the results of the computation:

- **Processing-time**: Sets TS to the wall-clock timestamp at which the record is being processed in the current operator.

- **Ingestion-time**: Sets TS to the wall-clock timestamp at which the record was ingested by the system. This provides similar results to processing time, but reduces the overhead of assigning a new timestamp in each operator.
- **Event-time**: Sets TS to the time at which the record was produced at the event generator. This requires the event itself to carry a timestamp, which is then extracted.
- **Count-time**: Sets TS to the count of records ingested by the system. This is generally not used anymore as it is not sufficiently expressive.

Processing-time, ingestion-time and count-time can sometimes lead to semantically incorrect results due to *event-time skew*, the difference between the time at which an event happened and when it is processed. For example, suppose a mobile game owner wants to track the number of current players per region each hour. A user plays while disconnected from the network and generates some events. When he reconnects to the network a few hours later, old events are sent to the SPS, late. If the system performs windowing by processing-time, these old events will count towards the current windows and lead to incorrect results. To address this inherent *lateness*, event-time is used. We discuss the implementation of these different time notions in Section 2.3.3.2.

Another important concept is *backpressure*. Given that in stream processing tasks, which are data dependent on one another, are continuously and simultaneously in operation, it is possible that an upstream task could overload a downstream task. Backpressure is counteracting pressure that the downstream applies on the upstream to ensure this does not happen by signalling to the upstream that it should slow down. Older generation stream processors did not use backpressure, opting instead to implement *load-shedding*[18], where a task when overloaded, simply drops some records according to a random filter[52], introducing more nondeterminism. A related concept is that in keyed streams, the key distribution may be *skewed*, making certain keys may be "hot", meaning that they appear much more than others. This is addressed through load balancing techniques, which commonly utilize randomness to flatten the distribution[80] along with and partial aggregation to produce correct results. One final source of nondeterminism in stream processing is the use randomized algorithms[5], which apply randomness to deal with adversarial attacks or sample input streams, in order to reduce memory size.

2.3.2.3 Exactly-Once Semantics

We now examine state-of-the-art production-grade SPSs, that is systems that are used widely in industry and as such must be capable of supporting a variety of use-cases. To do so these systems must offer a large amount of functionality such as state management, different notions of time, windowing, user-defined functions and out-of-order processing. Given the simultaneous deployment of streaming operators, their inherent statefulness, the different kinds of nondeterminism present and the complex notions of time under which stream processors execute, it should be no surprise that exactly-once processing

semantics and consistent state are difficult to achieve. In our examination, four successful approaches were found.

The first such approach is *strong productions*, present in MillWheel[3]. In MillWheel, each input record is consumed by the sources, affects the state of the DAG and produces output records at the sinks atomically. In essence, a transaction is executed per input record, containing changes to the source offset, each operator's state and produced outputs. In the case of failures, any ongoing transactions are aborted, processing is interrupted to restore failed operators, and processing resumes from the latest atomically recorded state. Of course, in order to be scalable and performant, an extremely fast transactional state store is required. Thus, MillWheel depends on Spanner, a system that enables fast concurrent transaction execution through the use of specialized, highly synchronized clocks. This approach has two problems, the most evident being the reliance on non-commodity hardware for performance. Furthermore, the execution of per-record transactions introduces significant overhead, which leads to decreased performance.

A second kind of approach is the *micro-batching* approach. Micro-batching systems buffer small batches of records at the sources, before processing a batch in a manner similar to a batch processing system. Apache Storm[99] is widely used but provides only at-least-once semantics, however its Trident API[100] implements this micro-batching approach through the use of transactions. Each batch is given a unique and growing integer transaction identifier, and each operator processes these batches, storing the operator state in an external database together with the transaction identifier. If after a failure a batch is retried, then the transaction identifiers are compared with the ones in the database, and only applied if larger than the one stored. Discretized Streams[110], the processing model in Spark Streaming takes a different approach. Whenever a new batch is generated, tasks are scheduled into the Spark engine much like in a batch processing scenario. Due to its micro-batching nature, D-Streams incurs some overhead from the batching process, and latency is always at least as large as the micro-batch period plus the scheduling overhead. Stateful operators are supported, but are turned into stateless functions that accept a previous state RDD as input. The output of one of these operators is both the original output and the current state. As explained before, RDDs provide transparent lineage recovery, meaning that fault-tolerance is immediately provided by this system. Because they rely on lineage based recovery, D-Streams requires operators to be deterministic for consistency. To prevent recomputing an operator's state from the start, periodic asynchronous checkpoints of the state RDDs are taken, which may then be used to bound lineage recovery. Micro-batching can help improve the throughput of a system, but leads to second-level latencies due to the buffering time and task scheduling[59, 85].

Two systems, Samza[82] and Kafka Streams[57], implement a third approach to fault-tolerance, involving *changelog maintenance*. Both these systems are unique as they do not provide their own messaging layer. Instead, they rely on Kafka[62] to act as a persistent connection between processing tasks. For fault-tolerant state, they capture per-record

state changes in a *changelog*, implemented as a Kafka topic. In Samza, changes are batched and asynchronously flushed to the change log, after which operators inform the coordinator of the latest processed input offset. Upon failure the changelog is replayed to recover the previous state, after which processing resumes from the latest registered input offset. The asynchronous nature of the changelog updates and marking of the input offsets means in Samza it is possible that a failure happens after records are outputted but the corresponding changelog updates are not flushed, leading to duplicated effects. Thus, while Samza supports arbitrary operators, it achieves only inconsistent at-least-once processing.

Kafka Streams extends this to exactly-once processing, again through the use of transactions. However, unlike the previous approach, it still processes record-by-record, the difference is that records are streamed into open transaction in the Kafka output and changelog topics and only then atomically committed or aborted together with the input offsets. These systems are highly dependent on Kafka, which has been shown to often be a bottleneck for stream processing[59, 105]. Another downside of this approach is that downstream consumer tasks can only see the output of the upstream task after the transaction is committed. Because transaction intervals tend to be quite large, this leads to large accumulating end-to-end latency. Finally, there is simply some overhead derived from the use of Kafka as a communication channel, as opposed to an efficient low-level communication framework such as Netty[75], which is often employed by other systems. Since Kafka Streams is a fairly recent system, only one benchmarking work includes it[61], finding that it is less scalable than native SPSs, with latencies more on par with micro-batching systems and throughput below that of other systems.

The last approach is the use of *consistent checkpointing*, which we studied in depth in Section 2.2.1. Several variations have existed over the years. Yahoo S4[81] used uncoordinated checkpointing, but since it did not compute a recovery line, it achieves only at-most-once processing guarantees, as state may be lost. Due to its guarantee of progress and freedom from domino effect, coordinated checkpointing has gained popularity. Achieving exactly-once processing when there is a consistent checkpoint involves only resetting the computational DAG to the latest checkpoint. Input stream offsets are also checkpointed, ensuring that processing restarts from the point at which it stopped. The first implementations such as IBM Streams[56] and Naiad[78] blocked processing during a checkpoint and unnecessarily checkpointed output records. The approach was later refined in Flink[23], which removed these faults. Through the use RocksDB[37], an efficient Multiversion concurrency control (MVCC) Log-Structured Merge Tree (LSMT) key-value store, it is able to perform asynchronous and incremental checkpoints with ease. This approach has been adopted in most recent stream processing systems, as it provides an easy mechanism for exactly-once processing semantics with support for nondeterministic and user-defined operators, with low overhead and without relying on batching or transactions. Other systems that have adopted this approach include Heron[63] a rewrite of Storm, Trill[29] an in-memory embedded stream processing library and Hazelcast Jet[49],

a recent performance focused SPS, among others. When millisecond level latencies are necessary, this is the appropriate approach.

In general, achieving exactly-once processing semantics requires two properties. First, some segmentation (into transactions, batches or checkpoint epochs) of the input streams is necessary such that after the processing of a segment, either results are made permanent because no failures occurred (and as such each record is counted only once) or rolled back in case of failures. The segmentation also allows progress to be made in the input streams, as otherwise the rollback could extend to the start of the input streams. Second, the input streams to the SPS must be stored in stable storage, such that after the state is rolled back, they can be reprocessed. One advantage of the first three approaches is that they immediately provide exactly-once delivery, as they produce outputs atomically with respect to processing. To achieve exactly-once delivery semantics with checkpointing-based systems two approaches exist:

- **Idempotent data sinks**[22, 49]: Idempotent sinks are sinks which are not affected by duplicate messages. If the pipeline is deterministic (unlikely, as this requires no message ordering delivery nondeterminism) and the sink is idempotent, then to achieve exactly-once delivery the sink merely has to eagerly push its results. If the pipeline is not deterministic then there additionally must be a Write-Ahead Log (WAL) of the records to commit. When a checkpoint is completed, that epoch's operations may be flushed from the WAL to the data sink. The idempotency guarantees that even if the flush is interrupted, it can be simply repeated to achieve exactly-once delivery.

- **Transactional sinks**[13, 49]: By coordinating the output of the sinks with the checkpointing mechanisms, using an atomic commit algorithm like 2-phase commit [20], one can achieve exactly-once delivery. Apache Kafka is a popular data sink supporting such a 2-phase commit-like mechanism. The implementation of a transactional Kafka sink works as follows: on each epoch, each sink opens a transaction with Kafka, into which records are streamed; when a new epoch starts the previous one is considered pre-committed; when notified of the completion of the first checkpoint, the pre-committed checkpoint is committed. If a failure happens, both the transaction and the checkpoint are aborted, and state is rolled back.

In general, all existing approaches to exactly-once delivery semantics require some sort of latency inducing synchronization, which typically is the use of transactions to guarantee atomic output commit. In Section 5.2.2, we describe an extension to Clonos, which will allow for low latency exactly-once delivery, without resorting to transactions.

2.3.3 High-availability for Stream Processing

SPSs have two large components, the coordinator process and the streaming dataflows which actually compute the results of queries. High-availability of the coordinator is

simple to achieve through state machine replication[24, 63, 66] as it is not a performance-critical component. Work on high-availability in streaming dataflows has been mostly limited to the academic environment. This is because offering high-availability in stream processing has historically required foregoing either consistency or expressiveness, two sacrifices that production-grade SPSs are not open to making. In this Section we explore past work on high-availability and show why the choice of fault-tolerance mechanisms in stream processing is a fundamental choice that affects what capabilities the system can offer.

High-availability for stream processing is defined [52] as a mechanism that allows processing to continue in spite of process failures. In other words, even when a failure happens, overall progress should not be blocked. In order to avoid blocking during recovery, highly available SPSs apply *localized recovery*, meaning that only the failed operators are treated during recovery.

In Section 2.3.3.3, we will introduce methods that further speed up recovery by deploying standby operators, which are ready to take over in the case of failures. To support that discussion we begin by introducing a taxonomy that allows us to describe types of operators and localized recovery guarantees in Section 2.3.3.1, as well as presenting a discussion on nondeterminism in localized recovery settings in Section 2.3.3.2.

2.3.3.1 Describing Localized Recovery

In [52], the authors define a useful taxonomy for describing the recovery types offered by their highly-available streaming system. These recovery types relate the path of execution taken by the recovering operator, with that of the failed operator, not to the entire system as whole. Thus these recovery types must be combined with an analysis of the resulting consistency and processing semantics, to fully describe the recovery guarantees offered by a system. Three categories of recovery are defined, one of which then subdivides into three cases:

- **Gap recovery (GR):** Recovery methodology under which the recovering operator may miss some input records, thus failing to produce some output records.
- **Rollback recovery:** Recovery methodology which allows the operator to emit duplicate records, but guarantees that the recovering execution path is "equivalent" to some failure-free execution path. This is because sources of nondeterminism may affect the results computed by the recovering operator. Three sub-cases are identified:
  - **Repeating (RR - R):** Duplicate records are identical to those produced previously by the failed operator.
  - **Convergent (RR - C):** Duplicate records may be different, but the execution converges to the same state the failed operator was in.

  - **Divergent (RR - D):** Duplicate records may be different and the execution diverges from the state the failed operator was in.

- **Precise recovery (PR):** A recovery methodology under which failures are perfectly masked by the recovering operator.

Gap recovery directly implies at-most-once processing semantics, as input records may be missed by the recovering operator. However, implementing gap recovery is simple, a new operator simply needs to be instantiated in the place of the failed operator and processing is allowed to continue from that point. Rollback recovery, which is not to be confused with the rollback recovery field described in Section 2.2, guarantees at-least-once processing, but allows for the possibility that some messages may be duplicated. If not deduplicated, these duplicate records may propagate affecting the results of downstream operators. To provide rollback recovery, one must be able to replay the input streams of the failed operator. This is commonly achieved through in-flight logging. If recovery is repeating, then with careful deduplication exactly-once processing semantics can be achieved. However, convergent recovery will eventually provide correct results, which may useful for some use-cases[18]. Finally, because precise recovery perfectly masks the failure of the recovering operator, it allows for consistent global state and exactly-once processing semantics. No deduplication is required at downstream operators as the recovering operator does not emit duplicate records.

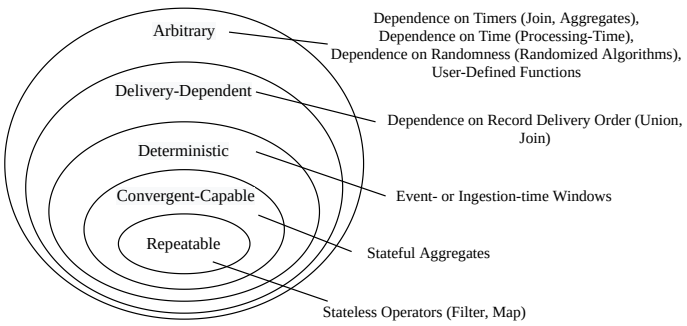


Figure 2.10: Streaming Operator Classification from Stonebraker et al.[52]

The authors also classify operators according to how they affect recovery types. The classes are, from most to least general, **arbitrary** (A), **delivery-dependent** (DD), **deterministic** (D), **convergent-capable** (CC) and **repeatable** (R). We show how the operator classes relate along with examples in Figure 2.10. *Arbitrary operators* have no constraints and may thus depend on many forms of nondeterminism such as wall-clock time, randomness, external systems or timers. All user-defined functions have to be assumed by the system to be arbitrary. Due to the large number of systems that support dependence only on delivery-order nondeterminism, we additionally separate out the class of *delivery-dependent* operators from the arbitrary operators. To rebuild their state, they must be restarted from a checkpoint and fed the same input streams in the same interleaving.

An example of such an operator is a stream join, which immediately outputs records it successfully joins from multiple input streams into a single output stream. *Deterministic operators* apply a deterministic function on their inputs and state. To rebuild their state, they must be restarted from a checkpoint and fed the same input streams, however these can be in any order and the operator will reach the prior state, creating the same output. Examples of such operators are event-time based window operators. A deterministic operator is *Convergent-capable* if it can rebuild its state starting from an empty state and eventually reach a valid state. For example, sort operators and certain binary aggregation functions (associative ones) are convergent-capable. This class is only relevant for systems targeting convergent rollback recovery. Finally *repeatable operators* are operators that produce their outputs based solely on a single input record. Examples of repeatable operators are stateless maps and stateless filters.

2.3.3.2 Expressiveness and Deterministic Replayability

In Section 2.3.2.3, we have shown how systems can provide exactly-once processing through global rollback mechanisms. These mechanisms also allow the systems to not concern themselves with nondeterminism, as any nondeterministic actions are either committed atomically with the global state they produce or rolled back and retried. In a setting which uses localized recovery, the same cannot be said, as downstream tasks may depend on nondeterministic actions that cannot be reproduced. In this Section, we will review several aspects of the runtime of SPSs that improve *expressiveness* at the cost of introducing nondeterminism. By expressiveness we mean the flexibility of the system in supporting a variety of use-cases and flexibility offered to users implementing computations. We will also review how past highly-available systems have dealt with this nondeterminism in their runtimes, and explain the advantages and drawbacks of each approach.

Systems must instead guarantee that the nondeterministic events executed at the failed operator can be reproduced at the recovering operator. We introduce the term *deterministic replayability* to abstract over different approaches to achieve this. Thus, systems that provide deterministic replayability have three choices when presented with a form of nondeterminism:

- They can forbid it. This is the most common approach in highly available systems. However it reduces the expressiveness of the system.
- They can replace it with a deterministic form which achieves the same effect. For example, nondeterministic delivery order is often replaced by a deterministic one by sorting the input streams.
- They can log the nondeterministic event at the failed operator, and use the logs to replay the events at the recovering operator.

User-defined functions are a key component of modern SPSs, allowing users to implement arbitrary operators for their use-cases. However, in doing so, systems expose themselves to the possibility that users introduce nondeterminism into their computations. While a window operator can blindly perform windowing on streams, a function must still be supplied to be executed over the window. Thus, highly available systems have often restricted users to supplying simple algebraic expressions which are evaluated[18, 32, 88]. Other systems offer imperative programming interfaces and trust that users will not introduce nondeterminism[27, 42, 71, 86], in which case their guarantees are broken.

Supporting different notions of time gives the user more flexibility on how to evaluate their streams. Count semantics are deterministic but hard to use as they do not express time directly. Event-time semantics often provide the most correct results but introduce higher latency[24]. Furthermore, they can lead to incorrect results when event sources do not have synchronized clocks or have different time-zones, such as end-users' mobile phones. Finally, it may be the case that event sources do not timestamp events, making it impossible to use event-time semantics. For those cases ingestion- or processing-time semantics are more desirable, but cannot be used by past highly-available SPSs as they depend on wallclock time. Hwang et al.[53] supports ingestion-time semantics but assumes that source operators do not fail. Similarly, SEEP[27] supports processing-time but does not provide recovery in such cases.

To deal with nondeterministic delivery order, a large majority of past highly-available SPSs[18, 50, 54, 73, 88] have used an In-Order Processing (IOP) architecture[69]. In this architecture, operators with multiple input streams first merge their input streams deterministically, producing an output stream which is then consumed by the operator. To do this, input streams are first buffered and then sorted according to some attribute, generally the event-time timestamp. This buffering introduces heavy latency, large memory overhead and does not scale well[4, 69]. Since in-order processing confuses stream progress with stream order, it leads to problems in the presence of idle streams. For example, a window operator with two input streams may only fire its windows for time t when all records up to time $t+1$ have arrived and been sorted on both streams. This means that the presence of a single idle stream, due to for example a failed event source, such as a sensor, can lead the system to block entirely. Because IOP orders input streams before processing, systems implementing this strategy can generally utilize delivery-dependent operators without issue.

An Out-of-Order Processing (OOP) architecture can alleviate the issues of resource overhead and latency by allowing operators to execute records in any order. To ensure deterministic replayability systems must either restrict themselves to deterministic operators at the most[27, 42, 46, 52, 65, 112] (i.e. operators that do not depend on record delivery order). Deterministic operators are hard to implement and require barrier synchronization logic[27, 112] which essentially adds back the overhead of serializing input streams. Alternatively, they may log the order in which records are processed, such

that the order can be replayed later[71, 86]. Furthermore, in an OOP architecture, watermarks[4, 69] are a necessary mechanism to trigger windows. Watermarks are implemented as streaming *punctuations*[102], elements embedded into a stream which contain a predicate. Their presence at a point in the stream states that no following records will match the predicate, which in the case of watermarks is that the current event-time is above a certain timestamp. When watermarks above a time t are received on all input streams, windows for time t may be triggered and a watermark produced on the output streams of the operator.

Watermarks are necessary for event- and ingestion-time semantics, and several mechanism for their generation exist. First, watermarks can be generated periodically at the event sources, however this requires them to be well-synchronized[53]. Furthermore, the event sources may be simple sensors, incapable of producing watermarks. They can also be generated deterministically at source operators[52, 71, 86], for example by emitting a new watermark every 500 records with the timestamp of the most recent record, but again idle streams can cause a watermark generation to stop and thus for processing to block for the entire DAG. Mechanisms for detecting and dealing with idle streams exist, but rely on nondeterministic timers[24, 69] and as such cannot be used by past work. Finally, watermarks can be generated through periodic timers at the source operators, which addresses the previous problems but introduces nondeterminism and as such previous systems cannot use this approach.

2.3.3.3 High-Availability Approaches

High-availability in stream processing dataflows is generally combined with a process-pairs[52] approach to speed up recovery. Process-pairs high-availability involves having a primary and a secondary copy of an operator running simultaneously in two different failure units. Failure units are simply the finest grained unit in the system that fails independently. Two modes of operation exist for operators in a process-pairs configuration, active standby and passive standby. In our review of past work, we also found two other approaches to implementing high-availability in SPSs, which are not based in process-pairs, and we now proceed to introduce all four.

One of the earliest approaches to high-availability in stream processing was *upstream backup* (UB)[32, 50, 52]. Intuitively, it simply tracks for each operator which of its outputted records have been fully processed downstream through backwards flowing acknowledgements, and ensures that those can be replayed through in-flight logging. No checkpointing is used, which at the time of its development was understood to be a source of performance overhead. However, upstream backup has several drawbacks. For example, stateful operators may depend on arbitrarily old records, meaning that in-flight records would have to be kept forever. Furthermore, because upstream backup defines no mechanisms for delivering tuples in order, it can only support fully deterministic operators (we expand on this in the following Sections). Upstream backup was introduced in

Aurora*[32], which synchronously logged in-flight tuples flowing towards operator o in K upstream hosts from o. The failed task can then be rescheduled to any of the K upstream hosts, where it will have local access to the tuples needed for replay. This approach is of course highly costly at runtime, due to the overhead of replicating the in-flight logs. Later implementations dropped this replication, opting instead to rebuild the in-flight state after failure.

A few systems take a very direct approach and upon failure simply reschedule the failed task, which is known as *localized recovery* (LR)[27, 71, 86]. In SEEP[27] uses an OOP architecture, but requires deterministic operators, which are difficult for users to implement. It also provides transparent scale-out and parallel recovery. TimeStream[86] and StreamScope[71] implement similar OOP architectures. To ensure deterministic replayability even with delivery-dependent operators, they apply logging of dependencies between input records and state. Since each record is logged this leads to a large amount of dependency data. The dependency information is asynchronously stored in batches external stable storage, for performance. This is reminiscing of optimistic logging, where determinants are asynchronously made stable. Of course, this means that if a failure occurs, some of the dependencies will be lost, which leads to other orphaned processes which must also be rolled back. Furthermore, because they use uncoordinated checkpointing, they are susceptible to the domino effect.

In *active standby* (AS)[18, 46, 52, 53, 73, 88, 95], also known as hot standby, the secondary simultaneously receives and processes all records the primary receives. The secondary thus consumes as many resources as the primary operator and some synchronization protocol is used to ensure that the two replicas do not diverge due to nondeterminism. In our review, we found that this protocol has generally been the use of a deterministic IOP architecture. The secondary also emits tuples downstream, and the downstream operators perform deduplication using sequence numbers, which increases bandwidth and CPU overhead further. Under failure, the secondary takes-over execution, usually at the same point at which the primary failed and little work has to be done to bring the system back to a stable state. This approach promises near-instantaneous recovery, but at the cost of a high failure-free overhead in both resource consumption and performance.

Flux[88] introduced the first implementation of active standby in stream processing, but it involved expensive acknowledgement tracking between the primary and secondary replicas for in-flight truncation and deduplication. Furthermore it required a total ordering of records, meaning a single source operator could exist, which assigned unique increasing identifiers to records. In Borealis[18] a single merge operator sorts input streams by timestamp and emits the same output streams to both replicas, which removes the acknowledgement overhead but introduces a bottleneck. StreamMine[73] removes this bottleneck by having both replicas order their input streams in the same way. Hwang et al.[53] reduce the overhead of ordering input streams by doing it only when strictly

necessary and introduce operator implementations which can produce deterministic results with disordered streams. Active standby techniques generally forego checkpointing, which limits the number of supported failures to the number of replicas maintained. Borealis[18] and PPA[95] address this through tentative tuples. When no replicas of an operator are found downstream computation continues using partial inputs. Borealis implements mechanisms for undoing the effects of tentative tuples, while PPA does not.

In contrast, the secondary may be in *passive standby* (PS)[46, 52, 54, 65, 81, 95], also known as cold standby. In this case, the secondary operator sits idly by during normal operation, receiving periodic operator state snapshot updates from the primary in systems that use checkpointing. When a failure is detected, the passive standby is activated and takes the place of the failed operator. Often this is combined with *in-flight logging*, a mechanism through which records sent downstream are kept by the sender in order to replay them in case the receiving operator fails. In-flight logging helps to ensure at-least-once processing in high-availability scenarios using passive standby. However, this is not sufficient to achieve consistent and exactly-once recovery, as nondeterminism and output deduplication must be considered respectively. This approach has a reduced resource footprint, requiring only that operator snapshots are maintained either in-memory or on-disk by some server. Moreover, the network bandwidth overhead can be reduced by utilizing incremental checkpoints. Since passive standby operators do not consume CPU resources, it is also possible to prepare multiple standby operators in one host, activating only the one that fails.

S4[81] simply restarts operators from the previous checkpoint, thus implementing gap recovery. Stonebraker et al.[52] use in-flight logging for the first time and carefully choose operator types to achieve repeating recovery. Like most other following works, they use unique record identifiers for deduplication. Gu et al.[46] add support for deterministic operators. PPA[95] extends this to delivery-dependent operators by deterministically sorting input records. Most work on passive standby is orthogonal to ours, building on previous systems and aiming instead to improve checkpointing mechanisms. Hwang et al.[54] builds on Borealis and improves checkpointing speeds by using incremental checkpoints, partitioning them and maintaining them in other failure units, however they protect only against single failures. SGuard[65] is the first system to utilize a DFS for checkpoints and studies how to reduce write contention. Gu et al.[46] reduces the amount of checkpointed data with the "sweeping checkpoint" technique, where checkpoints happen in a semi-uncoordinated fashion from sink to source, removing the need to checkpoint input queues. However, advancements in reliable SPSs[24, 57], particularly the asynchronous nonblocking and incremental coordinated checkpointing algorithms[23, 30] have rendered much of this work outdated, as they checkpoint only the operator state.

There are also works which implement *hybrids* (H)[50, 95, 112] of the previous approaches with a focus on achieving a balance between recovery time, runtime overhead and resource usage. Zhang et al.[112] switch passive operators to active when suspicion

of failure arises. Heinze et al.[50] estimate the recovery time of operators at runtime and switch between active standby and upstream backup in order to respect SLAs and minimize resource overhead. Finally, PPA[95] identifies operators with long recovery times and applies active standby to those, using passive standby for the remaining.

Simulation work[52] and experimental work[46] has validated the intuition behind the trade-offs of the different approaches. We compare the different properties of past highly-available systems in Table 2.3.3.3 using the taxonomy laid out in Section 2.3.3.1, where we additionally present the capabilities of Clonos.

Table 2.1: Comparison of the properties of prior work on high-availability in SPSs

System	Approach	Recovery Type	Operator Type	Architecture	Time Notions	Deduplication
Aurora*[32]	UB	RR-D	D	OOP	Count	No
Flux[88]	AS	RR-R	D	IOP	Event	Receiver
Borealis[18]	AS	RR-C	CC	IOP	Event	Undo
Stonebraker et al.[52]	AS/PS/UB	RR-R	R	OOP	-	Receiver
Hwang et al.[54]	PS	RR-C	CC	IOP	Event	Undo
SGuard[65]	PS	RR - R	D	OOP	Event	Receiver
Hwang et al.[53]	AS	RR-R	DD	IOP	Event/Ingestion	Receiver
Gu et al.[46]	AS/PS	RR - R	D	OOP	Event	Receiver
Yahoo S4[81]	PS	GR	A	OOP	Count/Processing	No
Zhang et al.[112]	H (AS, PS)	RR - R	D	IOP	Event	Receiver
StreamMine[73]	AS	RR - R	DD	IOP	-	Receiver
SEEP[27, 42]	LR	RR - R	D	OOP	Event	Receiver
Heinze et al.[50]	H (AS,UB)	RR-R	DD	IOP	Event	Receiver
PPA[95]	H (AS, PS)	RR-D	DD	IOP	Event	-
TimeStream[86]	LR	RR-R	DD	OOP	Event	Receiver
StreamScope[71]	LR	RR-R	DD	OOP	Event	Receiver
Clonos	LR/PS	PR	A	OOP	Processing/Ingestion/Event	Sender

2.3.4 Dataflow Systems Using Causal Logging

Other than the original Manetho system[39, 40], little work utilizing causal logging exists. In this Section we briefly introduce two recent dataflow system which have begun utilizing causal logging, at least partially. Like the systems presented in Section 2.3.3, these systems also follow the weaker ODD assumption, and as such do not support varied forms of nondeterminism.

Noria is a web application backend, serving the purpose of a database which automatically maintains the state of materialized views, accelerating reads. The maintenance is achieved through the use of a streaming dataflow graph which calculates and propagates changes to the materialized views. In [107], Noria is extended with a variation of causal logging, providing it with the ability to recover a single node while maintaining its materialized views intact. Noria achieves this through the use of *tree-clocks*, an elegant datastructure that tracks the provenance of records throughout the dataflow graph. In a sense, these are like vector clocks for dataflows. By associating a tree-clock with each message delivered, operators can track the order in which all messages in the system were

delivered. This information is efficiently shared with downstream operators by piggybacking only the relevant parts of the tree-clock, the neighbourhood. As explained in Section 2.2.2.4, downstream operators then have enough information to guide the recovery of failed upstream operators. However, this implementation presents a few limitations. It relies on the ODD assumption, that is, that the only source of nondeterminism is message delivery order. The tree-clock datastructure is only capable of tracking dependencies between input and output records, meaning that stateful operators are not supported. The approach cannot tolerate more than one concurrent failure. It also requires a central coordinator. Noria's recovery uses a central controller to collect all differentials and tree-clocks, these are then used to solve an algebraic constraint problem, which yields the order in which the failed operator should deliver their messages and which messages they should not emit.

Ray is a distributed application runtime targeting reinforcement learning and other AI tasks. Its programming model is based on tasks and actors, while the underlying execution engine is purely based on tasks bringing it closer to a batch processing system. To provide fault-tolerance, the lineage of these tasks is tracked. Since actors are stateful entities, lineage graphs are augmented with state edges, which order task executions on the same actor. Originally, lineage tracking was done synchronously, much like Dryad or Spark. In [104], Ray was extended with the lineage stash, a technique that allows lineage to be logged asynchronously, lowering latency. If only asynchronous writing of the lineage was done, then in the case of failures, orphan tasks would be created. This is why the system was also extended with a variation of causal logging. When a task invokes a second task, it piggybacks the delta of the lineage graph. This way, if the first task fails before logging the lineage in stable storage, the second task is able to provide it with the necessary lineage information. However, much like Noria, the authors assume deterministic tasks and actors, meaning that again the only source of nondeterminism is the order of execution of tasks.

2.4 Summary

In this chapter we begun by characterizing the model which we use for the remainder of the thesis, the MPS. We highlighted the difference between message reception and delivery, and defined events, nondeterminism, the happened-before relationship and consistent distributed state.

Armed with this model, we surveyed the somewhat limited but deep theoretical work on rollback recovery. Checkpointing-based approaches recover by computing and restoring a recovery line, a set of consistent state snapshots. However, this act of restoring a recovery line halts the computation, which is often not desirable. Log-based approaches aid in this by providing a way to restore only the failed process by deterministically replaying nondeterministic events and allowing processes to evolve naturally between nondeterministic events. Log-based approaches can also be combined with checkpointing-based

approaches to bound recovery time. Causal logging in specific strikes a good balance between pessimistic logging (which has a high runtime overhead) and optimistic logging (which fails to adhere to the no-orphans property).

Having understood the complexities of recovering the distributed state of a computation by handling nondeterminism, we move on to introduce dataflow systems. These systems can be largely classified into two categories: batch and streaming. Batch systems process bounded datasets using only deterministic operators, which facilitates their recovery. Streaming systems, on the other hand, process unbounded datasets, using concurrently deployed operators which greatly increases the difficulty of fault-tolerance. The fault-tolerance of SPSs is made more complex due to the presence of nondeterminism of different kinds, such as record delivery order, the use of wall-clock time for the implementation of different time notions, the use of timers to implement windows and ensure operators do not have infinitely growing state, and randomness in load balancing techniques. For each of the types of system, we surveyed the most popular and relevant examples for their features, differences and approaches to fault-tolerance. In review, production-grade SPSs have been converging on the use of coordinated checkpointing, which gives them the ability to support all forms of nondeterminism and achieve exactly-once processing. However, doing so prevents these systems from offering fast non-blocking recovery, as a recovery line must be restored for the whole DAG.

We continued our discussion of the related work with an analysis of past work on high-availability in stream processing. Approaches include active standby, which offers fast recovery at a high cost, passive standby which speeds up recovery by having prepared state snapshots but requires reprocessing part of the input streams and upstream backup which is no longer applicable to real systems. Here we saw that past implementations of highly-available SPSs have sacrificed either the expressiveness and usability of the system, or have sacrificed consistent recovery, meaning that they cannot achieve exactly-once processing. Time notions are generally restricted to event-time semantics, while allowed operators are very restricted. IOP architectures introduce large overhead while past systems supporting OOP architectures require users to implement operators with complex synchronization logic. Receiver-based deduplication is the common kind but wastes bandwidth and processing. Furthermore, because they are research systems for the most part, they lack many of the features of modern SPSs, such as managed state, clean APIs or windowing optimizations[68]. Due to this, current highly-available systems are not applicable to real-world critical use-cases that require both consistency and expressiveness. We concluded with a brief review of systems closest to the spirit of this work, that is, dataflow systems that apply some variation of causal logging, though none is quite satisfactory. In the following chapter we introduce Clonos, our antidote to these issues.

Chapter 3

Clonos

In this chapter we present Clonos, our streaming system providing high-availability with exactly-once processing guarantees. We begin by providing an overview of Clonos and motivating why causal logging is the most appropriate logging method for a stream processing setting.

Following that, we present the implementation of our prototype, split into two parts. The first details how we achieve high-availability for stream processing with Clonos. The second explains how we restore consistent distributed state after a failure, achieving exactly-once processing guarantees. To support that discussion, we first motivate why we chose Flink as the base on which to develop our solution. Following that, we present the necessary internal components of the system under modification, such that our modifications may be easily understood.

We conclude this chapter by reviewing how Clonos reacts to special failure cases, as well as a correctness sketch showing that we maintain exactly-once processing guarantees under any failure scenario.

3.1 Clonos' Overview

Having understood the underlying principles of rollback recovery protocols, the design of Clonos is simple to understand and can be applied to any production-grade Stream Processing System (SPS)[24, 29, 49, 63, 81, 99] which provides a clear separation between the underlying message passing system and the stream processing runtime. To enable high-availability we perform localized recovery. Systems using micro-batching[100, 110] or transactions[57, 82] are not targeted as they cannot be extended with localized recovery.

With Clonos, we extend the stream processing architecture with a deterministic replayability layer, as shown in Figure 3.1, which the stream processing layer uses to ensure

consistent recovery. Unlike the majority of previous work, we do not take the path of restricting the sources of nondeterminism, which would reduce expressiveness. Instead, with Clonos we aim to log all nondeterministic events (not just record delivery order) and utilize this log during recovery to achieve consistent precise failure recovery. This will allow users to freely implement arbitrary operators, without having to concern themselves with details of the runtime and whether the operator has a dependence on record delivery order.

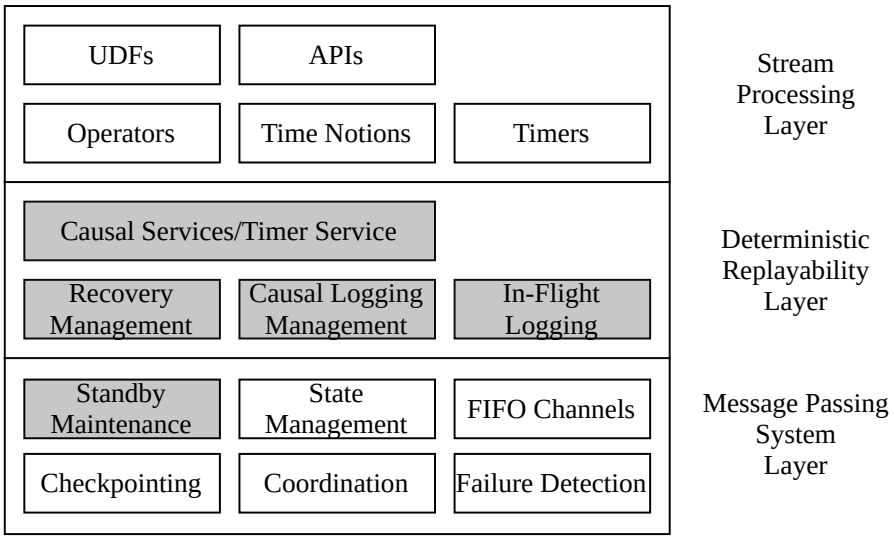


Figure 3.1: Clonos' Layered Architecture

Components which we introduce are shown in gray, though in implementing Clonos we also had to make some modifications to failure detection and checkpointing, and of course modify the stream processing layer to utilize the components of the deterministic replayability layer. We now describe several aspects of Clonos at a high level, before diving into the implementation.

Localized Recovery Regarding localized recovery management, whenever a failure is detected, a replacement task will start its execution with the latest state snapshot available for the failed task. The main idea behind Clonos is to allow the recovering task to progress deterministically, until it encounters a nondeterministic choice. At this point, the task consults the log of its previous execution so that it follows the same execution path. It then proceeds deterministically until the next nondeterministic event. When the log is exhausted, the task is guaranteed to be in a consistent state with the remaining Directed Acyclic Graph (DAG). By also logging the records that arrive at downstream tasks, we can perform deduplication at the recovering task, unlike previous works which waste processing and network bandwidth by deduplicating at the receiver. We provide details on how we do this in Section 3.2.3.6.

Passive Standby To speed-up recovery in the presence of large state we also introduce passive standby tasks, which are idle until a failure occurs. Passive standby tasks improve recovery time by being ready to fail over onto whenever a failure happens. They are scheduled, initialized and tend to have the latest state snapshot already loaded.

In-Flight Logging In-flight logging is the act of logging outputted records, so that they may be replayed later to aid in the recovery of a downstream task. It is necessary to guarantee at-least-once processing when performing localized recovery, as otherwise the failed task would not see some of the records generated upstream since the last completed checkpoint. In previous works, in-flight logs were maintained in-memory. However, modern reliable SPSs can emit several megabytes of records per second. As such, we make the design choice of allowing the in-flight log to be spillable to local disk, ensuring low resource usage.

Logging Protocol The choice of logging protocol can have large impacts on the system in both performance and recovery. SPS' tasks are in constant communication, which causes downstream tasks to almost always be dependent on nondeterministic events of upstream tasks. Thus, to avoid extensive rollback, we desire a logging method that respects the no-orphans condition. This immediately excludes optimistic logging from our list of candidates. Previous work[71, 86] which has used optimistic logging has shown that such systems often require rolling-back additional processes, the worst case being a failure of a source operator, in which case the entire DAG may be rolled back.

SPS' tasks have a highly efficient critical path that involves repeating three steps: deserializing a record, providing it to the operator for processing and serializing results to be outputted. As much work is performed in-memory as possible, for performance. While pessimistic logging respects the no-orphans condition, it requires synchronous access to stable storage. To worsen the situation, this stable storage cannot be the local disk in a high-availability setting, it must instead be an external system. Such a method is thus not applicable for performant stream processing.

Causal logging provides the best of the two alternatives above. Unlike pessimistic logging, it does not require synchronous access to stable storage, unless output is about to be committed to the Outside World Process (OWP). Thus, only sink operators require access to stable storage. This means that it can be freely combined with transactional sinks to achieve efficient exactly-once delivery, without accessing stable storage at all. Unlike optimistic logging, causal logging respects the no-orphans condition, by propagating non-stable determinants downstream according to the happens-before relationship, passing on the responsibility of ensuring their stability before output commit. We thus choose to employ causal logging for providing deterministic replayability of events.

Causal Logging Each task is equipped with a causal log for itself, and one for each upstream task. Conceptually, these causal logs are an infinite stream that is parallel to the

main dataflow. As records flow through the system, new determinants are piggybacked on the main dataflow. Tasks must log received determinants before allowing them to affect their internal state. After a failure, a recovering task will request its determinants from downstream tasks and use them in its recovery.

Stream processing dataflows tend to be shortcut-free DAGs[101]. This means that, in general, a $\prod_{Det}$ membership tracking protocol is sufficient to ensure optimal sharing of determinants[10]. In other words, more complex protocols will not lead to less duplicate determinant sharing, unless shortcuts are present in the DAG. We detail how we implement our causal logs and membership tracking in 3.2.3.3.

Stream processors are multi-threaded systems, while causal logging was designed for single-thread processes. We address this by following the methods laid out in [33], which studies multi-threaded optimistic logging. Timers affect operator state concurrently with the processing of streams, and as such synchronization can be used to ensure that causal log writes are serializable (see Section 3.2.3.5). Networking threads, on the other hand, share state (input and output queues) with the main processing thread. The state of input queues is recovered naturally by logging record delivery order, but the state of output queues is more complex as the in-flight log must also be rebuilt. We model the state of output queues as separate processes, which will have their own causal log, and depend on the main thread causal log (see Section 3.2.3.6). Output queue and in-flight log state is recovered concurrently with the main thread.

Causal Services One aim of Clonos is to be as transparent as possible to the user. However, one fact cannot be altered: nondeterministic events must be logged. Thus, in order to support user-defined functions, we must provide users with the capability of logging nondeterministic events as well as providing replay of those events during recovery. We do this as transparently as possible through the abstraction of causal services. In other words, causal services allow user-defined functions to log and replay synchronous nondeterministic events while being oblivious to them. We detail our implementations in Section 3.2.3.7. Similarly, a timer service is provided to ensure that timers are re-executed at the correct point in the input streams (see Section 3.2.3.5).

Checkpointing To utilize passive standby, primary tasks must checkpoint their state periodically, such that secondaries can download and prepare these snapshots. Checkpointing is also important as it places a bound on the recovery time of processes, by ensuring that they can restart from some point ahead of the start of their execution. It also allows for the truncation of in-flight logs and causal logs (whether they are pessimistic, optimistic or causal). Coordinated checkpointing allows for a simpler implementation with smaller checkpoints. However, with coordinated checkpointing, it is possible that one task, which runs out of memory for in-flight or causal logs, will block other tasks. In this case, uncoordinated checkpointed would be desirable, as it allows for that task to take a checkpoint, freeing up memory and enabling progress. In the end, we chose to continue

with coordinated checkpointing, both because it is the most common implementation in reliable SPSs, but also because we can rely on it to recover a consistent state if any issues are detected during recovery.

We design Clonos such that its determinant sharing has a configurable depth, allowing one to trade-off overhead for number of failures supported with fast recovery. In doing this, we subject ourselves to failure scenarios which cause determinants to be lost forever. We detect such failure cases and fallback on using slower recovery line restoration, thus maintaining exactly-once processing guarantees.

3.2 Clonos' Implementation

In this Section, we first motivate why we chose Apache Flink for our implementation, then introduce the large scale architecture that Flink provides in more detail. This then allows for the discussion of the two key modifications introduced by our solution. The first focuses on how we achieve localized recovery and standby tasks with at-least-once processing guarantees and divergent state, similar to previous work. Then we proceed to demonstrate how we implement causal logging during failure-free operation and how we use it during recovery to achieve deterministic replayability for any nondeterministic event. We also cover what causal services are and why we implement them.

3.2.1 System Under Modification

In Chapter 1, we motivated some use-cases for stream processing. These use-cases go from analytics, which requires *high throughput*, to critical applications which require *exactly-once processing*, *high-availability* and *low latency*. *Expressiveness*, such as the support for different notions of time and stateful, user-defined and nondeterministic operators, is a key requirement for modern-use cases as the implementation of deterministic operators hinders development and requires deep knowledge about the runtime. Finally, because operators may be stateful, the system must offer transparently *managed state*. Developers and researchers have long been aware of these requirements, as they have previously been laid out by Stonebraker et al.[93] and yet no system offering these properties can be found.

Few commercial systems are able to meet the performance demands that users look for[3, 24, 57, 81, 82, 99, 110]. Of those, fewer have been able to provide the managed fault-tolerant state that users desire[3, 24, 57, 82, 110]. Additionally, offering exactly-once processing is extremely nuanced which reduces our list further[3, 24, 57, 110]. Of the remaining systems, only Flink[24], Kafka Streams[57] and Spark Streaming[109] are open-source. Furthermore, MillWheel[3] requires specialized, non-commodity hardware for performance. Spark Streaming utilizes the micro-batch processing model, while Kafka Streams uses transactions and persistent connections, meaning neither is a proper setting for our work. Finally, experimental benchmarks have shown that Flink has orders

of magnitude lower latency[89] and higher throughput[59] when compared to similar systems.

We intend to extend a streaming system with logging rollback recovery. To do this, a strong checkpointing algorithm is necessary. Flink offers a clean implementation of non-blocking asynchronous incremental checkpoints[23], which is a good base for such work. Moreover, Flink also supports transactional exactly-once delivery[13], which will allow for future work comparing this approach to one based on causal logging. Flink offers all the desired capabilities of a modern SPS, such as larger-than-memory managed state, user-defined and arbitrary operators, several notions of time, usage through SQL or several programmatic APIs and many connectors to other systems. That is of course, without offering high-availability.

Non-technical reasons motivating the choice for Apache Flink are its popular use in large scale production deployments [22], ability to contribute through the FLIP system and having one of the largest Apache open-source communities.

Apache Flink is a large project, containing dozens of libraries and several different APIs for interaction. However, the changes made are to the underlying Message Passing System (MPS) of the system, which means that Clonos in general supports much of the work built on top of Flink with little modification.

3.2.1.1 Apache Flink Runtime

In Figure 3.2, we show a simplified version of Flink's distributed runtime. Flink has three large components. *TaskManagers* (or hosts) host dataflow tasks in their task slots. A dataflow constituted by two tasks containing source operators, one task containing a window operator and one task containing a sink operator is shown. We also show the network channels between TaskManagers, and the streams of records flowing between them. Records are serialized into network buffers for performance.

The *JobManager* (or coordinator) is in charge of scheduling, failure detection through heartbeats, and checkpoint coordination among other responsibilities. The JobManager has sent an Remote Procedure Call (RPC) to the TaskManagers hosting the source tasks telling them to initiate checkpoint $n+1$. This RPC causes the sources to momentarily block, emit a checkpoint barrier with the identifier of the checkpoint and then resume record emission. Checkpoints barriers are inserted into the stream in their own buffer, and due to the FIFO nature of the network channels cannot overtake records. Checkpoint barriers divide the infinite record stream into *epochs*. Tasks containing multiple inputs are forced to block their input channels whenever they receive a checkpoint barrier on one of them, in order to ensure stream alignment. Task C is shown processing the remainder of epoch n as its first input channel is already blocked.

The sink task is shown receiving a checkpoint barrier for checkpoint n, this causes it to snapshot its state and asynchronously upload it to stable storage, the third major component of the architecture. In the case of failures, all tasks in the dataflow restore

their most recent consistent global checkpoint from stable storage. The Figure shows that checkpoint n is complete, while $n+1$ has not yet been processed by two operators.

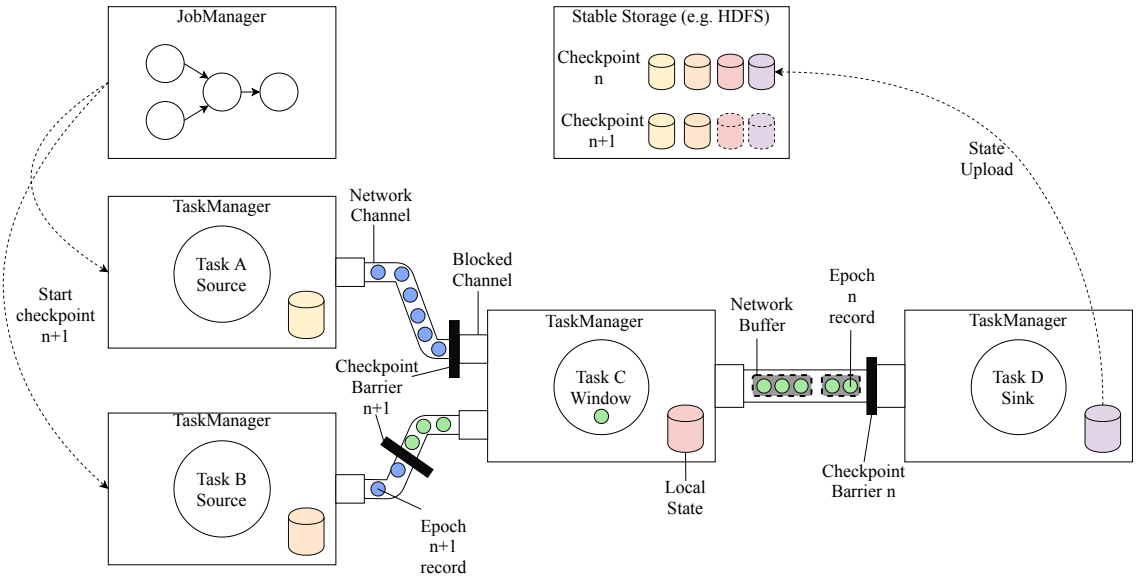


Figure 3.2: Flink architecture and distributed runtime components

Figure 3.3 zooms in on the TaskManager executing task C to show its internal architecture. A Flink task has a main thread (threads are marked with green circular arrows) of execution that starts with the *StreamInputProcessor* requesting the next buffer it should process from the *CheckpointBarrierHandler*. This request is passed on to the InputGate, which chooses an *Input Queue* from which to take a buffer. Flink uses the Netty[75] project for asynchronous event-driven communication over the network. This library uses a thread pool running an event loop to respond to networking events. It is these threads that deserialize messages from the network and place network buffers into the appropriate input queues. Netty is performance focused, providing an implementation of direct memory buffers on the JVM, and employing object pooling and reference-counting to allow the programmer to manage memory himself.

If one such buffer is a checkpoint barrier it is used in the CheckpointBarrierHandler to block that input channel. When the StreamInputProcessor receives the requested buffer, it is deserialized into the stream elements it is composed of. These can be regular records, watermarks and latency markers among others. Note that stream elements are serialized across buffers, meaning that a given stream element may be partly serialized on one buffer and partly serialized on the following buffer. The StreamInputProcessor then feeds the deserialized records to the *operator* one by one. When the buffer is exhausted, its reference count is decreased, leading to it being returned to the input buffer pool.

Whenever a record is processed by the operator, the *state lock* must be locked. The state lock is used to guarantee a serial order to operator state accesses. This is important because it is not only the main thread of execution that accesses operator state. Operators

like windows or other user-defined functions may register timers in the *SystemProcessingTimeService* by specifying a future point in time, which when reached *triggers* the execution of a *callback* function that may modify operator state. Windows use timers to register when window contents should be emitted. A timer may be a processing time timer (used in processing time or ingestion time notions) or event time timers (used in event time notions). Furthermore, the system itself registers timers for its own functioning, for example, watermarks are emitted at sources on the basis of a regular timer and similarly idle streams are detected through periodic polling.

As the operator processes the input records, it uses the *RecordWriter* to produce output records to one of more of its output queues. The RecordWriter has two responsibilities. First, it ensures that there is always a buffer in each output queue that it can write to. This means that when the latest buffer in a queue is full, the RecordWriter fetches an empty buffer from its *Buffer Pool* and places it into the queue. Second, it serializes records directly into the buffers in the output queues. This means that the Netty instance on the output side may pull half-filled buffers and send them downstream. This is done to ensure low latency even when there is a low volume of data. Whenever Netty sends a buffer it decreases its reference count, usually leading to it being recycled back into the output buffer pool. This is the normal lifecycle of output buffers in Flink, shown in red dashed line in Figure 3.4, which continues using Task C as an example. They exist in the output buffer pool, get requested by the record writer, put into the output queue and filled with data, later they are shipped by Netty and then retired back to the buffer pool.

Flink tasks inform their upstream tasks of how many buffers (or credits) are available in their input buffer pools through backchannels. This is done to ensure that downstream tasks have enough space to receive buffers. This provides a few advantages such as natural backpressure, improved checkpoint alignment times and improved memory utilization. This technique is known as credit-based flow control[12, 64].

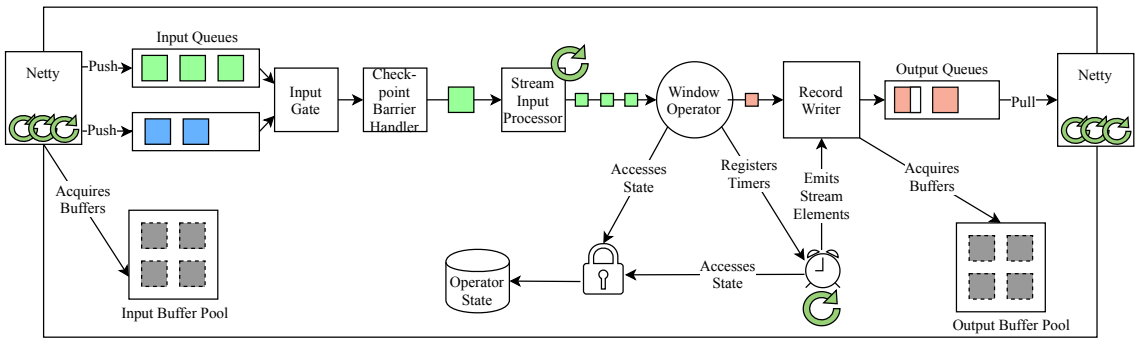


Figure 3.3: Overview of the components of a Flink TaskManager running a task.

This general design, using network buffers and thread pools to manage input and output to queues was popularized in Storm[99] and has since been used in many systems, meaning that many of the optimizations we are about to discuss are applicable to other systems as well.

3.2.2 Achieving High-Availability

Clonos has several components and its aspects are highly configurable, we introduce our approach in parts. We begin with the mechanisms we devise to achieve high-availability with at-least-once processing recovery (or in the taxonomy of Stonebraker et al.[52] Rollback Recovery - Divergent). For this, we require first, the ability to recover single tasks without resetting the entire computational graph, which we discuss in Section 3.2.2.1. Localized recovery is not however sufficient for high-availability if the preparation of the replacement task takes too long. To address this we introduce passive standbys in Section 3.2.2.2. With this configuration, we achieve at-most-once processing guarantees. In Section 3.2.2.3 we extend Clonos with in-flight logging and replay capabilities, allowing it to reach at-least-once processing capabilities.

3.2.2.1 Localized Recovery

Performing the modifications necessary for localized recovery involves first detecting the failure. Heartbeat-based failure detection can be rather slow, thus Clonos additionally employs network errors generated by the network stack when the connection is severed for failure detection. Upstream and downstream TaskManagers escalate these errors up to the JobManager, which triggers Clonos' recovery strategy. To avoid duplicate detection of failures, we implement a short window of time during which the JobManager ignores following failure signals for the same TaskManager.

Upon detecting one such failure, the JobManager marks the offending TaskManager as failed and reschedules the failed task in another TaskManager. If a TaskManager is incorrectly marked as failed, it will later rejoin the cluster in a reset state. The rescheduled task will go through its initialization and begin asynchronously downloading the latest stable checkpoint snapshot of the failed task from stable storage (typically a Distributed File System (DFS)). Then it will signal to the JobManager that it is ready and block until its network channels are reconfigured. The JobManager will then issue an RPC to the recovering task, with the addresses of the upstream and downstream TaskManagers to which it should connect.

As the channels are reconnected upstream, the upstream task will reset the state of its serializers. This is important because serialization spans across buffers and thus not resetting the serializers would lead to a corrupt stream when we later introduce the in-flight log and deduplication mechanisms. As soon as the channels are reconnected, upstream tasks can begin re-emitting records on that channel.

During this process, none of the remaining tasks in the graph is blocked. Upstream and downstream tasks from the failure continue processing data they receive on other channels and thus throughput is only lightly affected.

3.2.2.2 Passive Standbys

The number of standby tasks maintained per task is a configurable parameter of the JobManager. We treat localized recovery as a special case of our standby strategy, where the number of standby tasks to be maintained is configured to zero. This means that our passive standbys share much of the same logic with localized recovery. Standby tasks are deployed by the JobManager at job deployment time. A simple anti-affinity constraint is used to ensure that they are not scheduled on the same TaskManager as the original task, otherwise they would offer no protection. Similarly to the case of localized recovery, standby tasks block until their network channels are reconfigured. To ensure that standby tasks load the latest state snapshot of their corresponding task, whenever a global consistent checkpoint is completed, the JobManager sends an RPC to each standby task containing a handle to the state snapshot a standby needs to download, which the standby task immediately does. It is possible for a standby task to have its channels reconfigured before a state snapshot load is finished. In this case, the recovering task blocks until the snapshot finishes loading. In stream processing, it is important to take regular checkpoints, such that if a failure occurs, the rollback is not too excessive. However, this means that the DFS which hosts the checkpoints is under a lot of pressure, which is exacerbated by the presence of standby operators constantly downloading snapshots. As such, Clonos should be used in combination with incremental checkpointing, to reduce this pressure (Flink already provides such a capability). After the activation of a standby task, a new standby for the same task is scheduled, if a TaskManager with an available task slot exists.

3.2.2.3 In-Flight Logging

Our implementation of in-flight logging, whose buffer lifecycle is shown in blue dashed line in Figure 3.4, is unique in three ways. First, instead of logging individual records as they are placed in the output queue, we log the buffers as they are shipped downstream. We thus implement the in-flight log as a sorted map of epoch identifiers to a list of the epoch's buffers.

Second, we employ Netty's reference-counted buffers to avoid copying data. Instead, ownership of the outputted buffer is passed from Netty to the in-flight log. The in-flight log then fetches a buffer from the in-flight buffer pool and places it into the output buffer pool. We do this to ensure that the record writer does not run out of buffers as well as to ensure that credit-based flow control operates normally. We name this technique the *buffer exchange*. Initially, we attempted to simply increase the size of the output buffer pool as opposed to having an in-flight buffer pool and performing buffer exchange. However, this has the unintended consequence of breaking credit-based flow control, increasing latency and breaking the backpressure mechanisms of Flink.

The third way our solution is unique is in the ways that buffers can be replenished in the in-flight buffer pool. Given the sheer amount of data that Flink can process per second, the in-flight buffer pool may run out of buffers, in which case, processing would stall. One

way of replenishing buffers is in response to checkpoint complete notifications. When a checkpoint complete notification for checkpoint n is received by the task, it notifies the in-flight log, which can in turn prune all data for epochs lower than n. However, this is not enough to avoid processing stalls, unless the buffer pool is made large enough to accommodate all the data of one epoch. Since epoch length varies with both checkpoint interval and state size another method is required.

Thus, we extend our in-flight logger with asynchronous spilling capabilities. As the in-flight log accumulates buffers, it may choose to spill those buffers to disk according to a configurable *spill policy*. Since the spilling is performed asynchronously, it does not affect critical-path performance. We implement three such policies:

- **In-Memory**: This policy never spills to disk.
- **Eager**: As each buffer is added to the in-flight log, a write request is immediately submitted to the I/O Manager.
- **Availability**: A separate component, the BufferAvailabilityChecker, periodically checks the availability of buffers in the in-flight buffer pool. Upon hitting a configured threshold of availability, it triggers a spill of all unspilled buffers in the in-flight log. This policy is illustrated in use in Figure 3.4.

Intuitively, the eager policy would require a smaller in-flight log buffer pool, while the availability strategy would require a larger one but have better performance. The availability strategy also batches sequential writes, which reduces the number of disk seeks. As the write request for each buffer completes, a callback is executed marking the buffer as spilled in the in-flight log and reducing the reference count of the buffer, which in turn leads to its recycling back into the in-flight buffer pool.

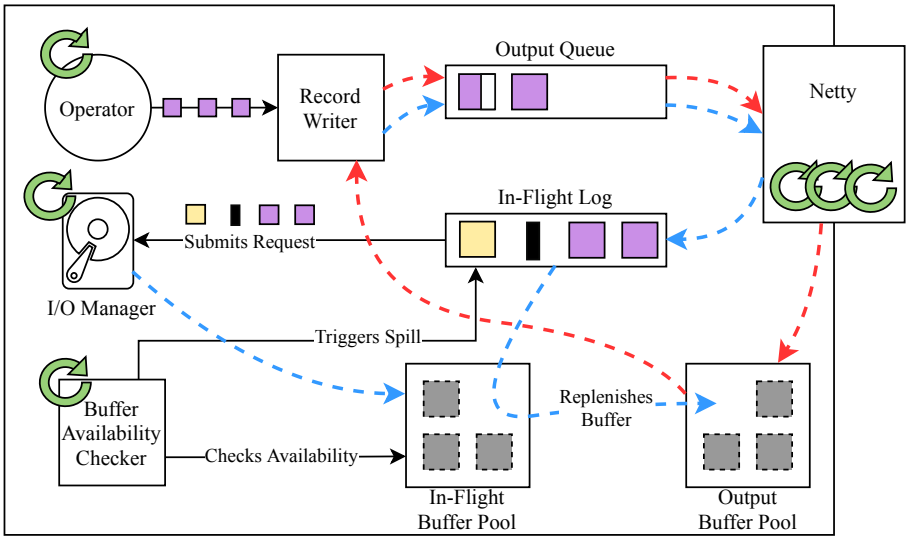


Figure 3.4: Buffer lifecycle and in-flight logging.

3.2.2.4 In-Flight Log Replay

When the in-flight log is combined with local recovery or passive standbys, a recovering task, upon having its network channels reconfigured, will emit to its upstream tasks an in-flight log replay request. Upstream, this request will be received by the corresponding output queue, which will obtain a *replay iterator* from the in-flight log. Until the iterator is exhausted, the output queue will provide Netty with buffers from the replay iterator. At the same time, the main thread of this task is able continue producing buffers into the output queue. These buffers are added to the back of the in-flight log and will be sent during the replay process.

The replay iterator hides away the complexity of differentiating between on-disk and in-memory buffers that need to be replayed. It maintains a *cursor* into the in-flight log, indicating which buffer is to be sent next. The in-flight log maintains buffer handles indicating whether the buffer is in-memory or on-disk. As next is called on the iterator, if the buffer is in-memory, it is returned immediately and the cursor advances.

If the next buffer is on-disk however, it must be read from disk into another buffer. In this case, a blocking queue is used. The iterator will attempt to return the next buffer in the blocking queue, blocking if none is present. A request to read the buffer is sent to the I/O Manager, which when complete places said buffer into the blocking queue.

As reading each buffer from disk as the requests arrive would incur heavy latency, the replay iterator is also capable of *pre-fetching*[91] a configurable number of buffers ahead of time and placing them into the blocking queue. Another cursor is used to indicate the position of the pre-fetch process in the in-flight log. A small *pre-fetch buffer pool* is used for this.

When the first cursor hits the end of the in-flight log, the iterator is destroyed and Netty is able to pull buffers from the output queue again.

3.2.3 Achieving Consistency

Having shown how we implement high-availability while ensuring that failed tasks reprocess lost epochs, we now turn to the arduous task of ensuring consistency. We first catalogue the sources of nondeterminism found in SPSs, with emphasis on those that are particular to Flink. Then, we describe our implementation of causal logging, with a detailed description of the datastructures used for a single thread's causal log. Following that, we describe the complete process of recovery and how we apply the determinants in it. Next, we describe how we make causal logging transparently usable for the end-user. We finish by addressing how Clonos handles special failure cases, a topic often left unspecified in competing solutions, as well as how it ensures correctness under any failure pattern.

3.2.3.1 Catalogue of Nondeterminism

Our implementation of causal logging relies on the PieceWise Deterministic (PWD) assumption, unlike previous work which relied only on the Ordered Delivery Deterministic (ODD) assumption (see Section 2.2.2). We must ensure that we do not break this assumption. To do so, we must identify and capture in determinants all nondeterministic events. Expressive stream processing systems present a significant number of nondeterministic events. We isolate them here in one place, along with the determinants necessary to make the event deterministic. As the system generates determinants, it encodes them into the causal log. Each determinant type is assigned a unique tag, used in deserialization.

The first and most obvious kind of such an event is the order in which records from different channels are delivered to the operator. In Flink, buffers are deserialized and their component records processed in sequence. We use this to our advantage by recording an *OrderDeterminant* only whenever a new buffer is pulled from an input queue. The determinant simply records the channel index from which the next buffer is to be pulled. A second, small optimization we do is to only record order determinants after the *CheckpointBarrierHandler*. This is because since barriers block channels from pulling further buffers, they do not introduce further nondeterminism.

The next most common type of nondeterminism is wall-clock time usage. The wall-clock time is accessed for every notions of time. In processing-time or ingestion-time it is accessed on every record to assign them a timestamp. However, it is accessed much less frequently in event-time, as only the sources access wall-clock time to generate periodic watermarks. User-defined operators may also choose to access the current time for any number of reasons. Accessing the wall-clock time to get a timestamp is nondeterministic as it is a function which, for the same inputs (no inputs in this case), returns different results at different times. A *TimestampDeterminant* holds only the timestamp returned.

Random numbers are generated for a few uses in stream processing. One use is in random connection types, as well as load balancing algorithms for skewed key distributions. The other is in user-defined functions or other randomized operators. There also exist a class of streaming algorithms which use randomness to be resistant to adversarial attacks or sample input streams. An *RNGDeterminant* is generated whenever a random number is created, containing only the value of that random number.

Clonos relies on the PWD assumption, as such it is not usable if users want to use a form of nondeterminism not currently supported in Clonos. We address this by introducing the *SerializableDeterminant*, which users can extend in order to encode their own determinants. This determinant is serialized using Java's own serialization mechanisms, and is thus less performant. We envision that this may be used, for example, to query external systems[99] and encode the responses received as a determinant, though our testing does not cover this use-case. The responses received in this way could change if queried again during recovery, and as such it is important that the operator utilizes the same response as received previously.

Timers are used throughout streaming systems, for example in the implementation of window operators or watermark generation. When a timer fires it executes a callback function providing as an argument the timestamp for which the timer was registered. Whenever a timer fires it generates a *TimerTriggeredDeterminant*, containing the callback identifier, timestamp of registration and the record count at which the timer triggered. We go into more detail on how timers are restored in Section 3.2.3.5.

Any RPC made to the a TaskManager that affects the state of the computation is nondeterministic as well. Each such RPC must be encoded in its own specific determinant. One example present in Flink is the RPC made from the JobManager to a source task whenever a checkpoint is supposed to be initiated. A *SourceCheckpointDeterminant* is generated, containing the checkpoint number, timestamp, storage reference where the checkpoint is to be stored and a single byte for checkpoint type. Similarly to the TimerTriggerDeterminant the record count must be stored as well.

We later found the need to introduce a new RPC which informs a TaskManager that it should ignore a checkpoint and unblock any blocked channels. This RPC must to be logged in a *IgnoreCheckpointDeterminant*, containing the checkpoint number and the record count at which the RPC was executed. We go into detail explaining the need for this RPC in Section 3.2.3.8.

The final kind of determinant that we log in Clonos is generated when Netty output threads pull a buffer from the output queues. The nondeterminism comes from the interaction of two different threads in the shared state that is the output queue. The main thread continuously writes to the most recent output buffer, which the Netty thread may pull at any moment, making the size of the buffer dispatched downstream nondeterministic. Thus, whenever Netty pulls a buffer we generate a *BufferBuiltDeterminant* containing the size of the buffer built.

Yet a few more sources of nondeterminism were found, but are easily addressable by small adjustments to the code, instead of adding to the list of determinants. One example is in the shuffle connection type. After a checkpoint, we must reset the position of the round-robin back to zero, to ensure that after localized recovery the order of channels chosen for output follows the order that was followed prior to failure.

These events can be split into three different categories, which require different treatment on the part of Clonos:

- **Synchronous nondeterministic events** are events generated by the main execution thread during its critical path. We must modify the system to ensure the state lock is held whenever such an event is generated.

- **Asynchronous nondeterministic events** are events generated by a thread that is not the main thread of execution, but that affect operator state and thus must acquire the state lock before proceeding. This means that a serial ordering between these events and synchronous main thread events exists. All determinants for events of

this kind must contain a record count field. The record count is important, as asynchronous events may happen not only in between buffers, but in between records of a buffer as well. During recovery we must execute the asynchronous event when the operator is at the same record count.

- **Output thread nondeterministic events** are events generated by output threads. Several Netty output threads are concurrently executing and generating determinants. A serial ordering between the events of those threads and those of the main thread cannot be established. This means that a separate causal log must be maintained for each output thread, to which only these events are appended.

Table 3.1 summarizes the determinants Clonos generates along with showing their encoding schemas and corresponding sizes. Where variable length parameters exist some bytes have to be added to encode their size, which are shown in parenthesis. Clonos is innovative in two ways with regard to the types of nondeterministic event that are supported. Previous work on causal logging systems[40, 104, 107] has only addressed synchronous main thread nondeterministic events (in fact, only order determinants). Clonos additionally supports asynchronous events, as well as, tracking nondeterminism for multiple interacting threads.

Table 3.1: Summary of sources of nondeterminism and determinants generated.

Name	Type	Schema	Encoded Size (Bytes)
Order	Synchronous	$\langle TAG \rangle \langle CHANNEL\_INDEX \rangle$	$1+1=2$
Timestamp	Synchronous	$\langle TAG \rangle \langle TS \rangle$	$1+8=9$
RNG	Synchronous	$\langle TAG \rangle \langle NUMBER \rangle$	$1+4=5$
Serializable	Synchronous	$\langle TAG \rangle \langle SERIALIZED \rangle$	$1+\lvert SERIALIZED \rvert$
Timer Triggered	Asynchronous	$\langle TAG \rangle \langle REC\_COUNT \rangle \langle CALLBACK \rangle \langle TS \rangle$	$1+4+(1+\lvert CALLBACK \rvert)+4 = 10+\lvert CALLBACK \rvert$
Source Checkpoint	Asynchronous	$\langle TAG \rangle \langle REC\_COUNT \rangle \langle CHK \rangle \langle TS \rangle \langle STORE\_REF \rangle \langle CHK\_TYPE \rangle$	$1+4+8+8+(2+\lvert STORE\_REF \rvert)+1 = 24+\lvert STORE\_REF \rvert$
Ignore Checkpoint	Asynchronous	$\langle TAG \rangle \langle REC\_COUNT \rangle \langle CHK \rangle$	$1+4+8=13$
Buffer Built	Output	$\langle TAG \rangle \langle BUFFER\_SIZE \rangle$	$1+4=5$

3.2.3.2 Causal Log Manager

A central component in Clonos is the causal log manager. Figure 3.5 shows a simplified causal log manager for the task shown before in Figure 3.3. This component manages all the *causal logs* of a TaskManager, both its own *local* causal logs and the causal logs of *upstream* tasks. Each causal log has a unique identifier composed of the task identifier, the type of causal log (main or output thread) and possibly an output channel identifier. In this case, task C is the local task, as such determinants are appended directly to its

local main thread causal log. Several components, shown in Figure 3.5, append to the main thread log. The CheckpointBarrierHandler appends order determinants, while the StreamInputProcessor may append timestamp determinants. The streaming operator may append any number of determinants, since a user-defined function may be provided. To ease the burden on the user, we provide *causal services* which abstract away this complexity from the user. More information on causal services is given in Section 3.2.3.7. The timer executor, the component that executes asynchronous timers, may also append determinants to the main thread, however it must synchronize with the main execution thread in doing so, by acquiring the state lock.

An output thread causal log is also maintained per output queue, task C has a single output queue and as such a single output thread causal log. The local output thread causal logs are written to by Netty threads whenever they pull a buffer from the output queue, as this possibly generates half-filled buffers with nondeterministic sizes. In essence, this approach models each thread as a separate process whose state must be recovered, an approach which has been taken before in optimistic logging[33].

Whenever a connection to a TaskManager is initiated or reconfigured, that connection is given a unique identifier. The connection then registers itself as either a consumer or producer with the causal log manager, depending on whether the task opening the connection is downstream or upstream respectively. The causal log manager then begins tracking the offsets of what determinants that consumer or producer has consumed or produced respectively.

A TaskManager must also track the nondeterminism of upstream tasks. In causal logging, when a process sends a message to another it piggybacks non-stable determinants with that message. Similarly in Clonos, when a Netty output thread is about to send a buffer downstream, it first passes the buffer to the causal log manager, who *enriches* it with the new determinants which that consumer has not yet seen. We call these updates of each causal log *determinant deltas*. Symmetrically, when a task receives a buffer, that buffer is first passed to the causal log manager which deserializes piggybacked deltas and appends any new information to the respective causal log.

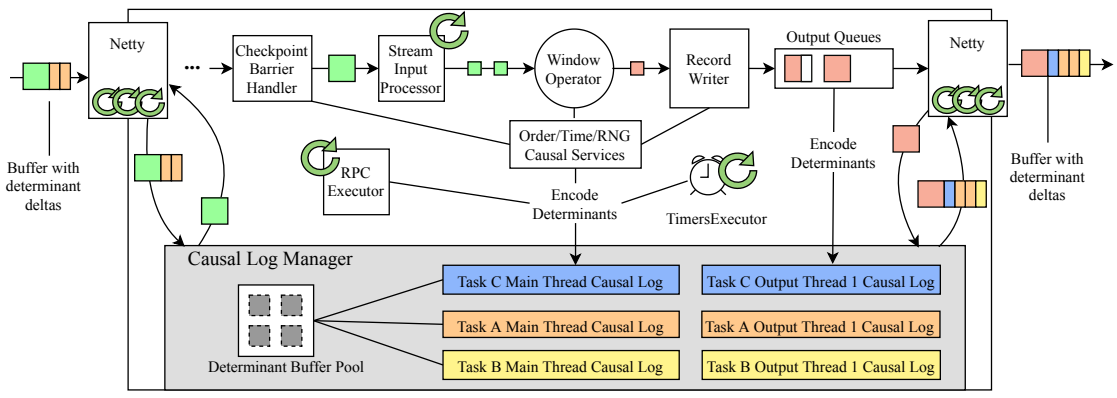


Figure 3.5: Causal log manager, delta piggybacking and determinant encoding.

As determinants are written into a task's causal log, they propagate downstream according to the happens-before relationship through piggybacking. Figure 3.6 shows a simple dataflow with a depth of four. Only operators, inputs and output queues are shown for simplicity. Focusing on the green line starting in the source operator, you can see how determinants of this task are propagated through the map 1 operator and then further downstream following causality. The causal log manager can also be provided a configurable determinant sharing depth. This parameter defines the depth up to which a causal log should be propagated. The causal log knows the distance in the execution graph from any task to any other task. It uses this information to decide whether deltas from a certain causal log should be shared. This feature allows us to balance causal logging overhead with the number of failures supported. Clonos can be configured to behave like a simple upstream backup approach, achieving only at-least-once processing semantics, by setting the determinant sharing depth to zero, which disables the causal log manager entirely. In Figure 3.6, black dashed line is used to mark the depth to which the determinants of the source operator are shared with different sharing depths. We go into more detail on the role of this in Section 3.2.3.9.

One optimization we perform in Clonos is avoiding sharing output thread causal logs of the local task with all downstream channels. Instead, a given output channel i consumes only the output thread causal log for thread i. This is also represented in Figure 3.6, by the two colored dashed lines. The green line shows the path through which determinants of the top output queue are shared, while the red line does the same for the bottom output queue. The reason we can perform this optimization is that determinants generated by other output channels do not causally affect the state of output channel i. Thus, on output channel i the only determinants shared are the ones generated by channel i of the local task, the main thread of the local task, and any other upstream thread causal logs (which respect the determinant sharing depth).

3.2.3.3 Thread Causal Log Implementation

Our initial design for the thread causal log was a circular buffer, however this design suffered from a few issues. First, if the buffer becomes full, it requires copying the entire causal log to another circular buffer with double the size, blocking processing. Second, it is wasteful, because as the buffer is grown, it eventually reaches a size that is roughly double the needed capacity. Finally, it introduced complex circular indexing logic, which was difficult to maintain.

Each thread causal log is instead built as an infinitely growable buffer. We show the main thread local causal log for Task C of Figure 3.5 in Figure 3.7. To avoid copying data when growing is required, we compose new buffers from the determinant buffer pool into a large composite buffer. Determinants are serialized across the several buffers. Determinant deltas are merely slices (or views) of this buffer, meaning that we perform no copies of determinant data for performance. Three types of indexes into the causal

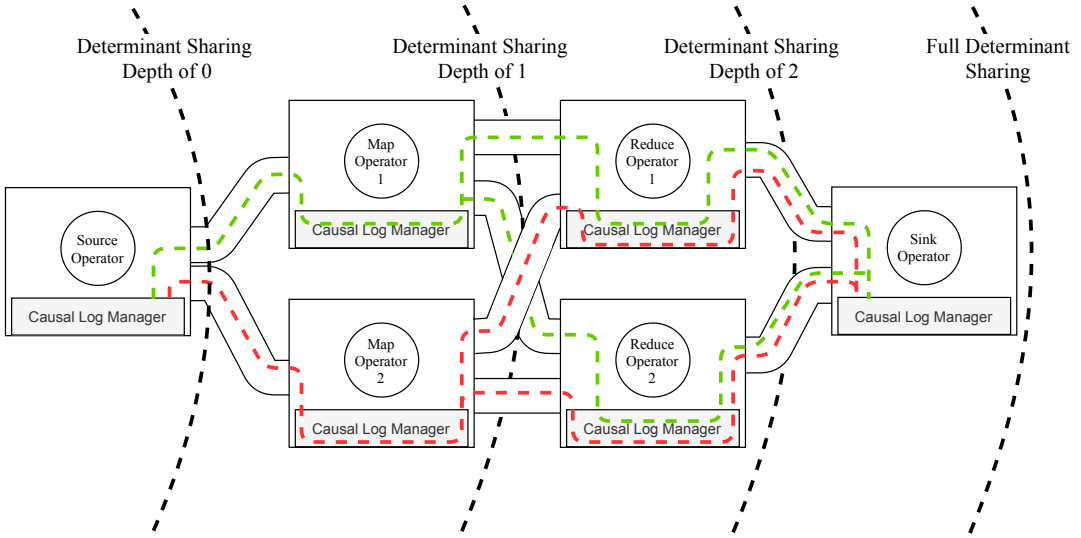


Figure 3.6: Causal paths followed by determinant propagation and the effect of determinant sharing depth.

log buffer are maintained: *consumer indexes*, *epoch indexes*, and the *visibility index*. Epoch indexes mark where new epochs begin in the determinant buffer. Whenever a checkpoint is completed, all previous epoch determinants may be discarded. We do this by discarding all buffers containing only old epochs and adjusting indexes accordingly. These are used to ensure that a buffer outputted for a certain epoch does not consume determinants of a following epoch which may already be present in the buffer. Consumer indexes mark up to where in the thread log a consumer has read. They are used to ensure that a consumer is not sent duplicate data. These indexes are logical with respect to a certain epoch. In other words, a consumer index is a tuple of an epoch identifier plus a logical offset from that epoch's index in the buffer. The visibility index, on the other hand, tells causal log consumers up to where they may consume when creating deltas.

We implemented the thread causal log as nonblocking as possible. Our implementation is a Single-Writer-Multiple-Reader data-structure. Three kinds of modification to the causal log are possible, local writes, upstream writes, and truncation. Regarding writes, local task logs already have their writes serialized through the state lock, meaning that a single writer is present at any time. Upstream task causal logs however, may have concurrent writes from multiple Netty input threads simultaneously processing deltas. In this case, we synchronize writes on the buffer with a critical section. Truncation of the causal log happens whenever a checkpoint complete notification is received. In this case, old data is deleted from the causal log and epoch indexes are updated. Truncation is the only event that can change the epoch indexes. Other accesses to the causal log must be isolated from changes to the epoch indexes. We use a Read-Write lock for this. All accesses (read or write) to the causal log must acquire the read lock, while a checkpoint complete notification acquires the write lock.

All three kinds of modification mentioned move the visibility index. Writes to the

causal log move the visibility index forward as their last action. This allows multiple readers to generate deltas concurrently with writes by sending the difference between a consumer index and the visibility index. Checkpoint complete notifications move it backwards by the same amount of bytes removed from the buffer.

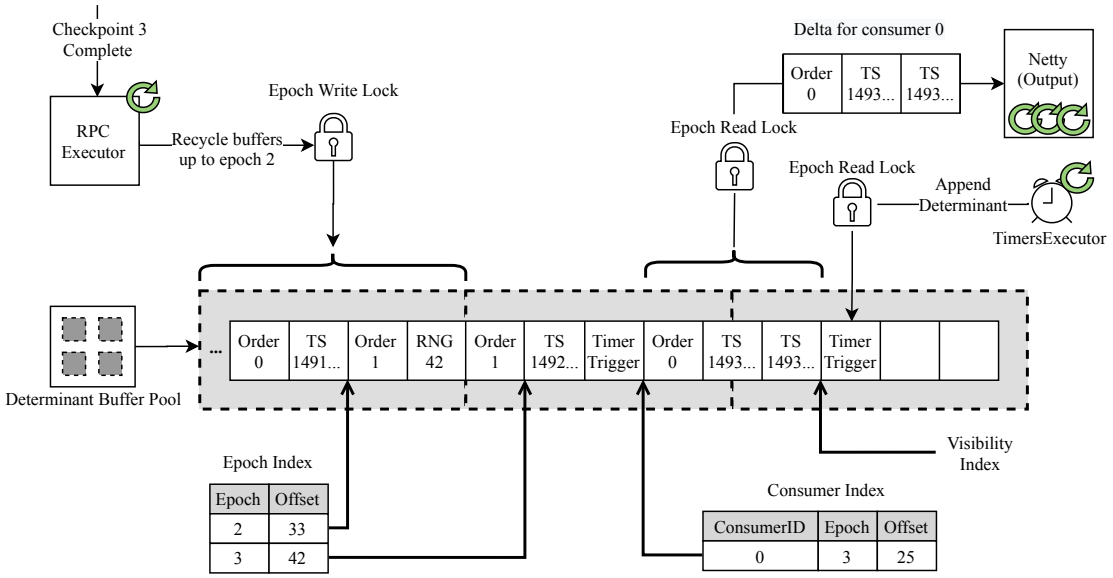


Figure 3.7: Causal log implementation.

Recalling from Section 2.2.2.5, our implementation of consumer indexes to avoid sending duplicate determinants to the immediate downstream operators is essentially an implementation of the $\prod_{det}$ protocol for membership tracking. We make it more abstract however, by supporting multiple types of determinants. This does not however mean that a downstream operator does not receive duplicate determinants. To see this we return to Figure 3.6. The green line indicates the causal path that the determinants generated by the top output queue follow. However, the determinants of the main thread of the source operator must also be shared. This is equivalent for the red path on which the determinants of the bottom ouput queue are shared. Notice that after the partition operation between the map and shuffle operators, these lines intersect. This means that duplicate determinants for the main thread of the source operator will be received, and to maintain the integrity of the causal log downstream they must be deduplicated. To do this, when a task shares a determinant delta downstream, it appends the logical offset from the epoch that the delta begins at. Downstream, the operators can use this offset to check if there are any new determinants in the delta. In Section 5.2.1 we detail a way of extending our causal log membership tracking to a $\prod_{log}$ protocol, which would further reduce determinant duplication.

3.2.3.4 Recovery

During normal operation, Clonos executes the dataflow graph the user requests while simultaneously propagating determinants generated in a given task with downstream tasks. When a failure happens Clonos restores the failed task either through local recovery (Section 3.2.2.1) or passive standby (Section 3.2.2.2). To recover consistency of the tasks state and its channels, Clonos executes a recovery algorithm which we illustrate in Figure 3.8. In this Figure, the several states a recovering task goes through are shown on the left, while a dataflow composed of a source, map, and sink (each with two parallelism) is shown on the right. The same color scheme is used to map events that happen in the dataflow to a given state of recovery. Each task is extended with a *Recovery Manager* component, responsible for recovering task state to the configured guarantees. The recovery manager is implemented as a state machine with 5 states. Initially, tasks in the dataflow start in *Running State*, unless they are recovering (local or standby), in which case they start in *Standby State*. Tasks in Running State cannot change to another state, they may only fail or finish successfully.

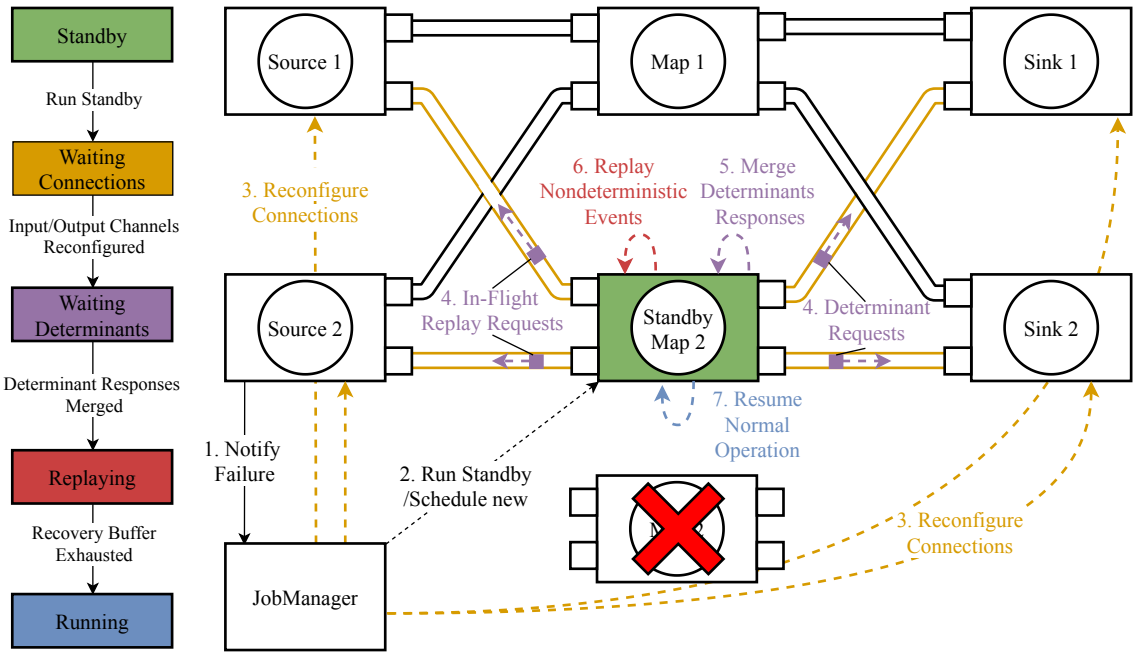


Figure 3.8: Causal recovery algorithm implemented by Recovery Manager.

Several types of notifications are signalled to the recovery manager, which may cause a state transition. In particular, after a failure is detected the JobManager may signal to a task in Standby State that it should run. This causes the task to switch to *Waiting Connections State*. In this state, the recovery manager receives several notifications regarding newly reconfigured input and output connections. The recovery manager tracks the completion of the network reconfiguration, switching to *Waiting Determinants State* when all channels are updated. Upon entering this state, the recovery manager emits an *in-flight replay request* per input channel as well as a *determinant request* per output

channel. The in-flight log replay requests, when received, cause the in-flight log of the receiving output queue of the receiving task to begin replaying previous buffers according the mechanism previously explained in Section 3.2.2.3. A determinant request, when received downstream, is propagated to the causal log manager of that task, who responds to the request with all available determinants for the failed task. As the recovery manager of the recovering task receives determinant request responses, it merges them, by choosing the longest buffer for each thread causal log in every determinant response. When all responses are received, the resulting buffers are called *recovery buffers*, as they will be used to recover the state of each thread and the state is changed to *Replaying State*.

Upon entering replaying state, the recovery manager first builds a cache of determinants which it reuses for deserialization during recovery (this helps with reducing garbage collection pressure). The recovery buffer of the main thread is loaded (we explain what happens to the output thread recovery buffers in Section 3.2.3.6) and the first determinant deserialized. Whenever a determinant is done being used, the recovery manager deserializes the next, staying one step ahead of the recovery. From here, the recovery manager unblocks the main processing thread, allowing it to begin processing input buffers. Whenever the main thread of execution is about to perform a nondeterministic event, for example obtaining the next buffer for processing, it will instead request replay of that event from the recovery manager, in this case by requesting the channel index from which to take a buffer. As stated in the introduction of Section 2.2.2, in between nondeterministic events, a process evolves independently and deterministically, and so it will always request the correct nondeterministic event at the same exact point of the execution. However, this is only true for synchronous nondeterministic events, see Section 3.2.3.5 for information on how we recover asynchronous nondeterministic events.

Finally, when the main thread determinant buffer is exhausted, the recovery manager switches to Running State, and normal operation is resumed. During the process of recovery, determinants are re-appended to the causal log as they are reprocessed. Similarly, a new consumer identifier is used in upstream causal logs, meaning that the recovering task receives all determinants that it had since lost. Note that, during this recovery process, other tasks in the dataflow are able to continue making progress.

In the event that a task fails during its recovery process, this process in once again initiated. Note that up to the point where the task is finished recovering, it has emitted no output, and as such has not affected other downstream tasks. In-flight replay requests and determinant requests similarly do not affect other tasks' state and can also be repeated.

We have described how the main thread recovers from a failure. This process begins filling output buffers with records, which both need to be deduplicated and added back into the in-flight log. In Section 3.2.3.6 we detail how we recover the state of the output queues, but first we introduce how Clonos deals with asynchronous nondeterministic events.

3.2.3.5 Recovering asynchronous nondeterministic events

For most of the recovery, it is the main execution thread that guides recovery by requesting replay of synchronous nondeterministic events from the recovery manager. However, for asynchronous nondeterministic events, the recovery manager must step in at the correct point in the re-processed stream and apply the callback function of either the RPC or the timer. Of course, during recovery, the timer executor is blocked, meaning that it cannot fire timers on its own, which would lead to the creation of nondeterministic events.

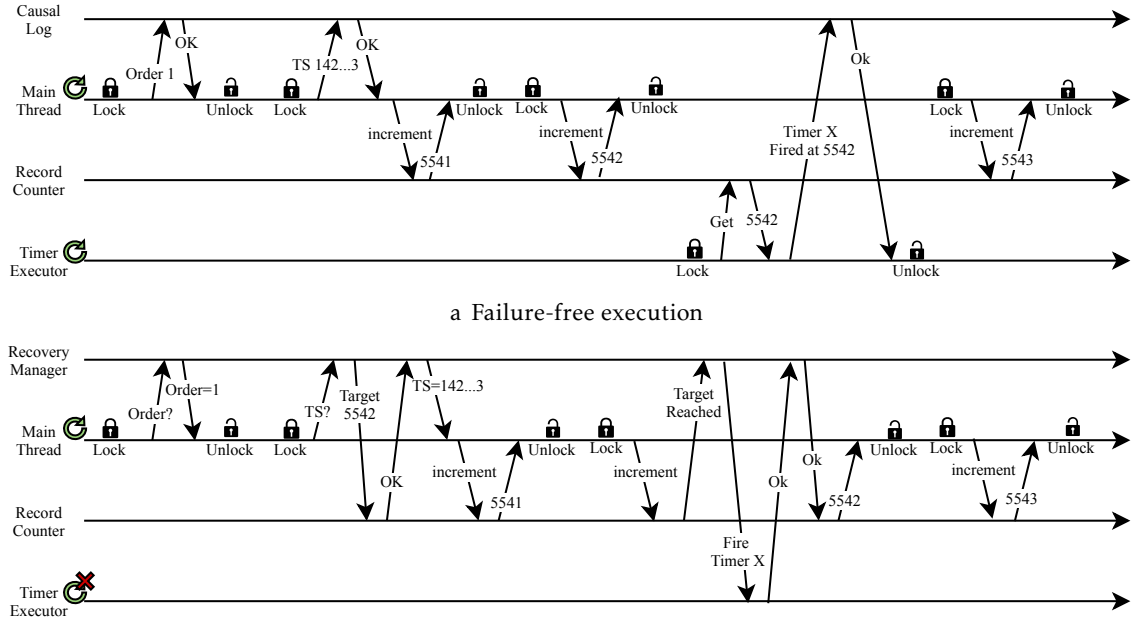
All asynchronous determinants contain a record count field which stores the number of input records processed since the start of the epoch in which they occurred. They get this record count from a small component called the *Record Counter*, which counts the number of input records processed by a task. Whenever the StreamInputProcessor sends an input record in for processing, it increments the record counter's count. The record counter is also responsible for notifying the recovery manager whenever a *record count target* is reached.

During recovery, the recovery manager deserializes determinants ahead of the main execution thread requesting them. Whenever the recovery manager deserializes an asynchronous determinant, it will not process it immediately, as the determinant's record count may not match the current record count. Instead, it will set the determinant's record count as the record count target in the record counter component. When the record counter hits that target number of records, it will notify the recovery manager that the target has been hit. This is done in the main thread of execution and allows the recovery manager to step-in and apply the callback function specified by the timer determinant or the RPC function specified by either a source checkpoint or ignore checkpoint determinant.

To obtain the correct callback function in the case of a timer determinant, the recovery manager maintains a *callback function registry*. Whenever a timer is registered in the timer executor, it's callback function is registered into this registry. Using the identifier present in the determinants it is possible to recover the correct function. After the recovery manager applies the correct function it prepares the next determinant and returns control to the record counter, who in turn, returns control to the StreamInputProcessor. The main thread then resumes guiding the recovery process by requesting the replay of synchronous nondeterministic events.

The concept that it is the main execution thread that guides recovery is often confusing when first explained. To better understand this, in Figure 3.9a we show the interaction of main thread and the timer executor with the causal log, and the record counter components for a slice of an execution. The figure also illustrates every time the state lock is locked and unlocked. The slice of the execution starts with the main thread obtaining a new buffer, from which it will deserialize records. This is nondeterministic and so the channel index is recorded into the causal log using an order determinant. Then, two input records are processed by the main thread and for each of these the record counter

is incremented. Note that on the first input record processed, the operator requested a timestamp, and so it also records in the causal log. At some point, the timer executor decides that it is time to fire a timer, and so it acquires the state lock, checks the current record count and before executing the timer callback function, records a timer fired determinant in the causal log. Finally, the main thread processes another input record.



a Failure-free execution

b Recovering execution

Figure 3.9: How the main thread guides the recovery of the operator.

This execution later fails and is recovered as outlined before, with the recovery manager receiving the recovery buffer of the main execution thread. In Figure 3.9b, we show the interactions of the same components, switching the causal log for a recovery manager for the recovering execution and with the timer executor thread blocked. Upon reaching the same point in the execution, the main thread knows that it must get a buffer, but since it is recovering, it asks the recovery manager for which channel to get the next buffer from. Then, the first input record is processed. Since inside state intervals processes progress deterministically (see Section 2.2.2), it is guaranteed that a timestamp will be requested by the main thread at the same point as in the previous execution. Instead of generating a new one, the main thread asks the recovery manager which timestamp it should use. The recovery manager obtains the response, but before returning, deserializes the next determinant and notices that it is an asynchronous one. The recovery manager then sets a target record count in the record counter that is equal to the record count in the asynchronous determinant. The recovery manager then returns control to the main thread. The next input record is processed and the record counter incremented. The record counter reaches the target value and signals to the recovery manager that the target has been reached. The recovery manager then reacts to the timer determinant by

finding the correct timer function in the timer executor and executing it, passing in the timestamp also included in the determinant as an argument. In this way, we guarantee that the state of the recovering execution follows the failed execution.

3.2.3.6 Recovering in-flight state and deduplication

Deduplicating messages outputted by the recovering task is important in order to achieve exactly-once processing. In Clonos, the process of deduplication is implicit in the process of recovering the state of the in-flight log. Output buffers that need to be rebuilt and re-added to the in-flight log are exactly those buffers that need to be deduplicated. We know that a given buffer needs to be deduplicated if there is a corresponding buffer built determinant in the recovery determinant buffer for that output queue.

Upon entering replaying state, a thread is created per output queue whose state needs to be recovered. This thread is given, among other things, the causal log of the corresponding output queue. The output queue recovery thread then blocks Netty from pulling buffers from the output queue until recovery is done. This prevents Netty from changing the sizes of the buffers which are built and sending a buffer that should be deduplicated. It then deserializes the buffer built determinants on demand, using each to tell the output queue to build a buffer with the exact size contained in the determinant. As the main thread goes through its process of recovery, it fills up the output queue buffers with records. The data that is filled into the buffers is guaranteed to be the same as prior to failure, as the main thread is recovering deterministically. When enough data is present in the active output queue buffer, the output queue recovery thread cuts the buffer and adds it to the in-flight, thus preventing records from being sent downstream in duplicate. When no more determinants remain, the output queue recovery thread unblocks Netty from pulling buffers and finishes. One advantage of our approach to deduplication is that it is done at the recovering process, instead of at downstream processes like previous works. This reduces the network bandwidth overhead, when it is most precious.

3.2.3.7 Causal Services

Causal services provide users with transparent interfaces to log and replay determinants. Causal services are exposed to the programmer of user-defined functions through the runtime context that every dataflow operator possesses in Flink. Currently, three causal services are provided to the user, a time service, a random number generator service, and an "other" service which may be used arbitrarily by users. The time service allows the user to obtain a reliable and recoverable timestamp of the wall-clock time. The random number generator service does the same, but for random numbers. We show some adapted code of the only function the time service provides in Listing 3.1. The timestamp service implements the function currentTimeMillis, to match the one provided in the System package of the Java language. It first checks with the recovery manager if the task is still recovering causally. If it is, it requests from the recovery manager the value of the

timestamp that should be returned and assigns it to toReturn. If the task is not recovering then the current time is assigned to toReturn. Before returning however, a determinant containing the returned value must be appended to the causal log, and then the value is returned to the function. The random service behaves much the same way, appending a determinant on every request.

Listing 3.1: Timestamp causal service implementation.

```
1  public long currentTimeMillis() {
2      long toReturn;
3
4      if (isRecovering && (isRecovering = recoveryManager.isRecovering())
5          toReturn = recoveryManager.replayNextTimestamp();
6      else
7          toReturn = System.currentTimeMillis();
8
9      threadCausalLog.appendDeterminant(reuseTimestampDeterminant.replace(toReturn));
10
11     return toReturn;
12 }
```

Causal services must be fast as they perform synchronous work in the main thread of execution. The time service demonstrates three important optimizations that are used throughout Clonos. The first is the use of short-circuiting logic in checking whether recovery is happening. Accessing the recovery manager is expensive during the critical path of execution and so a local boolean is maintained and reassigned during recovery. When the task finally finishes recovering, this boolean short-circuits the if condition and avoids having to query the recovery manager. The second optimization shown is the use of reusable determinants. Garbage collections can cause latency spikes, but a naive implementation of causal logging would create a lot of determinant objects. In Clonos, we reuse the same objects wherever possible to avoid frequent garbage collections. A third, smaller optimization is bypassing the causal log manager by maintaining direct references to the thread causal logs. This avoids looking up the correct causal log by identifier on every nondeterministic event.

We found experimentally (see Section 4.2.1.1) that these causal services were very inefficient when compared to the corresponding methods from the Java language. For example, in processing-time semantics a timestamp is assigned to each record, which must be recoverable after failure. If we write to the causal log on every timestamp, that generated a large amount of determinants per millisecond which has a high overhead. Thus, we later implemented more nuanced versions of our naive causal services.

The *deterministic random service* relies on the fact that random number generators are deterministic for a given seed. The seed sets some internal state for the generator and that state is used to generate the next random number. Thus, if we can ensure that the internal state of the generator is recoverable after a failed task is recovered, then we can ensure that the same random numbers are generated. We can do this by generating a new

random seed and storing the seed in the causal log after every checkpoint is taken. When recovering from a failure, the generator's seed is set to the seed recorded in the causal log. This way, we greatly reduce the amount of determinants stored to one per epoch.

The *periodic time service* is slightly more complex. Instead of checking the current wall-clock time on every request, it instead keeps an internal timestamp which it returns to the requester. Periodically, this timestamp is updated by employing our already causality logging infrastructure for timers. Thus, a periodic timer is registered with the timer executor whose callback function is to update the timestamp of the time service to the current time. The period with which the timer activates is configurable, allowing one to trade-off time tracking accuracy for overhead. By default, the period is set to one millisecond, as we have found that this does not affect performance. This method of time tracking does not, of course, provide incorrect results for processing-time or ingestion-time windows. It does however allow for the possibility of the returned timestamp to be a couple of milliseconds off from the real wall-clock time, but this could happen anyway for several reasons such as context switches or garbage collection pauses.

The "other" causal service has the same general structure as the causal service shown in Listing 3.1, but is meant to cover use-cases not directly supported in Clonos. It accepts a nondeterministic function which produces a serializable determinant. If the recovery manager is not in recovering state, then the function is executed and the serializable determinant serialized into the causal log, before returning the same determinant object to the user. If the recovery manager is in recovering state, then a serializable determinant is deserialized from the recovery buffer and returned to the user. We envision that this may be used, to for example, query external systems using a number of different REST frameworks, as it is sometimes done in stream processing[99]. If the query also affects the internal state of the external system, then it should be *idempotent*, as there is the possibility that the operator fails before propagating the determinant downstream.

A fourth kind of causal service exists, though it is not exposed to the user and is only used internally. It is a wrapper around the *CheckpointBarrierHandler* and provides deterministic delivery of buffers from multiple input channels. In other words, whenever the *StreamInputProcessor* requests a buffer to process, the order service provides a buffer from the same channel as was delivered prior to the failure in that point of the recovering execution. To achieve this, the order service reads order determinants, and pulls buffers from the input queue, until it obtains one from the correct input channel. Buffers from incorrect input channels are buffered internally, and delivered later. The order service relies on the assumption that the upstream processes replay their channels, which is why we implement the in-flight log. In the case that only one input channel exists, the order service does not track order events nor does it attempt to replay them during recovery, as they always concern the one and only input channel.

3.2.3.8 Checkpoints During Recovery

Clonos utilizes consistent checkpointing to bound recovery time and truncate the causal logs. If a failure happens then recovery of the failed task begins. Concurrently to the failure, the JobManager may initiate a checkpoint. This introduces a problem which we illustrate in Figure 3.10.

In this Figure, two epochs are currently being processed. Operator Map 2 has failed and been recovered, and is currently reprocessing and deduplicating the blue epoch. For this, Source 1 and 2 have had to replay their in-flight logs. In the replay stream, the checkpoint barriers dividing the blue and yellow epochs are still far from reaching Map 2. On the other hand, the blue epoch has been fully processed by Map 1, and so the channels connecting it to the sinks have been blocked by the checkpoint barriers that divide the blue and yellow epochs. This means that the sinks can now only receive data from Map 2, as they wait for checkpoint barrier alignment. Furthermore, Map 1 quickly exhausts the credits available to his downstream connections and thus stops processing the yellow epoch. Similarly, Source 1 and 2 will exhaust the credits of their connections with Map 1 soon after. This effect is known as backpressure and is caused by Flink's credit-based control flow. Thus, a single block in the dataflow cascades upstream blocking processing in the entire dataflow. In specific, the cause of this block is that Map 2 has to reprocess the blue epoch, which takes time, while downstream tasks wait for the checkpoint barriers but receive no data.

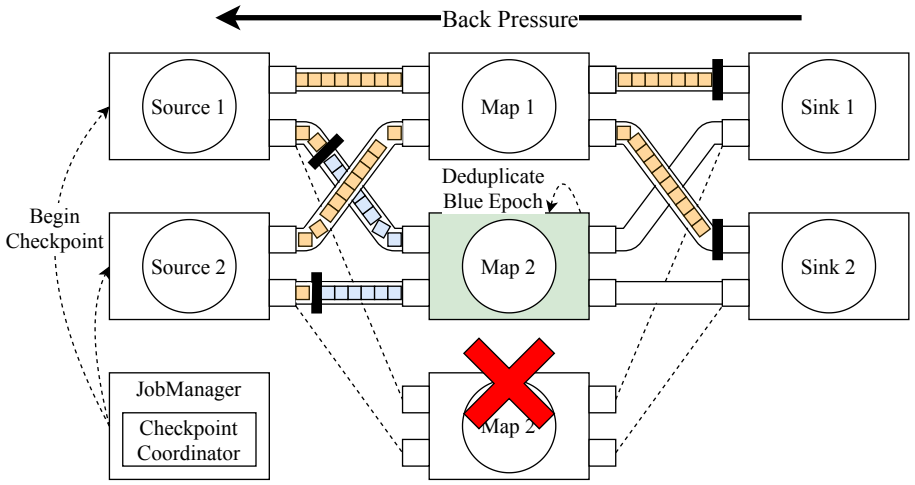


Figure 3.10: Checkpoint barriers block processing during recovery.

Clonos deals with this issue in two ways. First, after a failure is detected, the checkpoint coordinator is notified and this causes it to back-off from checkpointing for a configurable duration (by default we use three times the checkpointing interval). This gives the failed task enough time to finish its recovery, even if the task is slightly overloaded. Second, if there are already pending checkpoints, downstream tasks from the failed task are sent an RPC to ignore any pending checkpoints. This causes them to dissolve any

barriers that are currently blocking their input channels and ignore subsequent ones for the same checkpoint. Thus, in the example shown, the JobManager would RPC sink 1 and sink 2, leading to them unblocking their channels and allowing processing to resume. When the recovering task eventually emits the barriers, they are ignored downstream. Essentially, after a failure, a checkpoint is skipped in order to give the recovering task time to catch up to its parallel tasks. This RPC is an asynchronous nondeterministic event and generates an ignore checkpoint determinant. In the case of a failure downstream, the RPC would be replayed to ensure consistency.

3.2.3.9 Special Failure Cases

Several special failure cases exist in Clonos, all of which are dealt with in our solution and prototype. We highlight five such cases: failures of the source tasks, failures of the sink tasks, concurrent failures and sequential failures, and failures that exceed the determinant sharing depth. By sequential failures we mean failures of multiple tasks which are directly connected and share dataflows. Failures of the source tasks are simple to address. Since sources have no upstream tasks, they simply skip requesting replay from upstream. They instead rely on replayable sources such as Kafka, which are already a requirement for exactly-once processing. Sources are also the recipients of RPCs from the JobManager which instruct them to initiate a checkpoint. These RPCs create source checkpoint determinants, which must be replayed during recovery, causing the recovering source task to re-emit the same checkpoint barrier (which will be deduplicated). Additionally, any further RPCs that the source task receives must be ignored during replay, as these would lead the task to a divergent state.

Sink failures are not so simple. Sink tasks have no downstream tasks and so no other task from whom they can request determinants. Thus, replaying their execution exactly is impossible. However, sink tasks do not have any downstream tasks with whom they must remain consistent. This means that a sink task may recover in any way and we still retain state consistency and exactly-once processing guarantees. We do however produce duplicated results to the outside world, possibly with divergent values. To fix this duplication one may use a transaction sink with Clonos, in which case the sink will recover by aborting all open transactions and reprocessing the previous epoch. This solution introduces a lot of latency for downstream consumers, however. In Section 5.2 we describe an approach for transaction-less low latency output commit which would fix the duplication issue and lower latency when compared to current transactional approaches.

Concurrent and sequential failures are a concern as often process failures can cause cascading failures[95] or have a common cause. Clonos gracefully handles concurrent and sequential failures. If the failed processes have no dataflow in common, then recovery happens as described earlier in Section 3.8. If the failures share dataflows, then the upstream always recovers first. This is because before the upstream recovering task may replay for the downstream recovering task, it must first finish rebuilding its in-flight state.

Another important aspect is that the failed downstream task has lost its determinants for the failed upstream task. When it receives a determinant request from upstream it does not know how to respond. Our solution is to recur the determinant request further downstream. The downstream task will send the same determinant request to its downstream tasks and await the responses. The responses are then merged and sent back to the requesting task.

While a failed task is recovering, other failures upstream and downstream from it may happen. If this is the case, the recovering task's recovery manager may be in any of the outlined states. If it is in waiting connections state, it will simply mark the corresponding channels as not yet reconfigured. If it is in waiting determinants state it will either re-emit the corresponding in-flight replay request or the corresponding determinant request. If it is in replaying state, then no action needs to taken if the failure is downstream. If the failure is upstream, then there is the possibility that some buffers have already been received on that logical channel. We address this by sending the in-flight replay request again, however specifying that a number of buffers be *skipped* during replay. The upstream task will then pull and immediately recycle that number of buffers from the replay iterator, before sending them downstream.

Because the determinant sharing depth, that is the depth up to which a task's determinants are propagated, is configurable, in the case of sequenced failures, it would be possible that the determinants of a task are not recoverable. The most downstream tasks which receive recurred determinant requests will not be able to return the determinants, and thus escalate this to the JobManager, which proceeds to trigger a full rollback of the DAG, thus maintaining exactly-once processing semantics. Alternatively, one could also choose to continue with at-least-once semantics at this point, by recovering without determinants and thus without deduplication, by returning the empty determinant responses upstream.

3.3 Analysis

Clonos innovates on previous works that use some form of in-flight logging by making its in-flight log spillable according to configurable policies. Previous works have tracked input and output dependencies per record at a great cost, usually leading to diminished performance. In a sense, Clonos also does this, however, it digs deeper into the theory behind why this is necessary, allowing us to track input and output nondeterminism instead. Unlike previous works Clonos does so at the buffer level for performance while also supporting other kinds of nondeterminism. We identify three kinds of nondeterminism in stream processing and support causal logging for multiple threads unlike previous works on causal logging. Causal logging is implemented with efficiency and determinants are propagated only according to causal relations.

Clonos however has some downsides. Determinants must be logged synchronously,

which may introduce some processing overhead. As such, though Clonos handles nondeterminism, it is still advisable to attempt to reduce the amount of nondeterminism used. For this purpose, we have carefully designed causal services such that they can be used heavily, but generate a constant (per second) amount of determinants. Examples of this are preferring forward type connections to shuffle connections, and preferring shuffle to random connections whenever possible. Clonos consumes more memory than pure checkpointing approaches as it uses an in-flight log buffer pool, a determinant buffer pool and a pre-fetch buffer pool used for pre-fetching spilled buffers. However, the amount of determinants logged is small, especially if nondeterminism is avoided. Furthermore, the pre-fetch buffer pool is also kept minimal, as not much pre-fetching is needed. Our spillable in-flight log additionally helps keep the size of the in-flight buffer pool small. We evaluate the resource overhead of Clonos in more detail in Section 4.2.1.1.

$$RT = 3 * RTT + CI/2 * L \tag{3.1}$$

$$RT = S + SSD + 3 * RTT + CI/2 * L \tag{3.2}$$

Finally, while Clonos does not block processing during recovery, recovery involves the recovering operator reprocessing the last non-stable epoch. This is however an inescapable fact, either one pays the cost of active standby, or one must reprocess the latest epoch. The theoretical recovery time (RT) of Clonos is shown in Equation 3.1, where RTT is the average round-trip time between two nodes in the system, CI is the checkpoint interval (and thus $CI/2$ is the average epoch size) and L is a percentage representing the load on the system. If the system could handle two million records per second of input but only one million is provided then $L = 50\%$. The three round-trip times are for signalling that the standby should run, to reconfigure the network connections and to retrieve determinants. This is the same recovery time as presented in [52] for passive standby, only with the added round-trip for fetching determinants. The round trip time for local recovery (i.e. when passive standby is not used), shown in Equation 3.2, adds the time to schedule the replacement (S) and for state snapshot download (SSD). We conclude this chapter by arguing for the correctness of our solution.

3.3.1 Correctness

The correctness of causal logging as a rollback recovery approach has been formally proven in the past[6, 11]. However, because Clonos tracks nondeterminism for multiple threads (main processing thread and a variable number of output threads), we model each thread as process and recover them in unison. In this way, the proofs applicable to pure causal logging are extendable to Clonos. We instead aim to prove that Clonos guarantees exactly-once processing semantics, if configured to do so. Achieving exactly-once processing when performing localized recovery necessarily requires deduplication, which Clonos does at the buffer level by leveraging the buffer built determinants, which

encode the size of buffers which have been sent downstream. A recovering task will deduplicate x buffers of a given channel, if the recovery buffer of that channel has x buffer built determinants in it. This means that when all buffer built determinants in the recovery buffer are consumed, the next buffer created in that channel is safe to be sent forward, because if the receiver had already received its buffer it would also necessarily hold (and thus have returned upstream) a buffer built determinant for that buffer.

Assume a DAG composed of $\mathcal{N}$ tasks with a maximum depth of D (source tasks have a depth of zero, directly downstream tasks have a depth of one) and the failure of $\mathcal{F} \subseteq \mathcal{N}$ tasks happens. Clonos can be configured to use a determinant sharing depth (DSD) as large as the graph depth or smaller than the graph depth. We deal first with the case where $DSD = D$.

In this configuration, Clonos follows the condition stated in Equation 2.4. As such, any determinant for a nondeterministic event e whose effects have not yet been globally checkpointed are propagated downstream to any other processes whenever they become dependent on them. Determinants piggybacked on a buffer are logged by a task (processed by the causal log manager) before the operator state becomes dependent on them (before the operator processes the buffer's records), and as such at no moment do we break the condition that $Depend(e) \subseteq Log(e)$. Two failure cases can occur, either $Log(e) \subseteq \mathcal{F}$ or $Log(e) \not\subseteq \mathcal{F}$. In the latter case, at least one surviving process has the determinant of event e, in which case it guides the recovery, either by aiding in ensuring the main thread follows the correct execution path or by ensuring an output thread deduplicates a buffer and thus the records it contains. In the former case, because $Depend(e) \subseteq Log(e)$, then no surviving process depends on e, meaning that a different execution path may be taken without breaking consistency or the always no-orphans condition. Translating this to stream processing, this former case can only happen when for the failure of a given task, all downstream tasks also fail, as otherwise, downstream tasks will have the necessary determinants to bring the failed tasks into a consistent state with the surviving downstream tasks. The extreme case happens when $\mathcal{F} = \mathcal{N}$, in which case no task is dependent on any other and recovery is effectively equivalent to restoring a recovery line and beginning replay from the graph's input streams.

If however, $DSD < D$, then Clonos follows the condition of Equation 2.5 by not sharing e's determinant to a depth greater than DSD. In this case, there is the possibility that $Log(e) \subseteq \mathcal{F} \not\subseteq Depend(e)$, meaning that some orphaned process remains. When recovering tasks begin merging the responses to their determinant responses, they will find a determinant response marked as not found. Upon seeing this, Clonos will escalate this to the JobManager, which will trigger a full rollback of the DAG, thus achieving exactly-once processing guarantees. The alternative case is that $Log(e) \not\subseteq \mathcal{F}$, in which case at least one surviving task has the determinants of nondeterministic event e, and can guide the recovery of the failed tasks which depend on it.

Clonos is a highly configurable system in terms of its fault tolerance guarantees. By disabling in-flight logging and causal logging, failed operators are recovered with gap

recovery, leading to inconsistent state and as a whole at-most-once processing semantics are obtained, but incurring little overhead. Note that this approach is only supported conceptually, as due to the spanning serialization of records to buffers, Clonos cannot currently resume replay from any arbitrary point. By setting the determinant sharing depth to zero, only in-flight logging is enabled, and failed operators are recovered with divergent rollback recovery, achieving at-least-once processing semantics with very little overhead due to our no-copy in-flight log. Finally, by also enabling causal logging it is possible to perform precise recovery on failed operators, providing exactly-once processing semantics, again with little overhead due to our non-blocking causal log implementation, lack of synchronous access to stable storage and usage of piggybacking. If the overhead of causal logging is a concern, Clonos can also be used as an Family-Based Logging (FBL) causal logging protocol allowing one to trade-off determinant sharing overhead for safety. The determinant sharing depth is set to the depth of the graph by default, but by lowering it to another number f, the determinant sharing overhead is reduced in exchange for supporting at most f concurrent and sequenced failures. In this case, if a larger than f number of failures happens, Clonos can again be configured to favour either availability or consistency. If configured to favour availability, Clonos achieves divergent rollback recovery, otherwise it will fall-back on a recovery line restoration, using the latest global checkpoint, which will guarantee exactly-once processing but lead to longer, blocking recovery.

Chapter 4

Evaluation

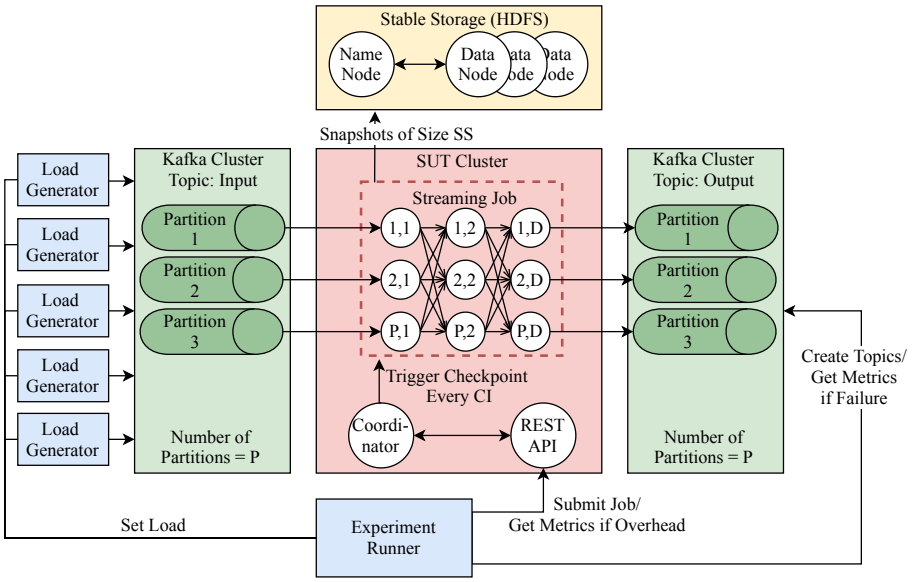
In this chapter we present our evaluation of Clonos. We aim to demonstrate both the applicability of our solution and validate its high-availability. We compare Clonos to checkpointing-based rollback recovery, by comparing it with the base Apache Flink version on which we applied our modifications. We also compare Clonos with itself in different configurations emulating approaches that use only localized recovery[71, 86] and in-flight logging by disabling standby operators or passive standby high-availability approaches[52, 54, 65] which apply only in-flight logging but do not achieve exactly-once processing, by setting the determinant sharing depth to zero, thus disabling the causal log manager.

To this end, we split our experimental work into two distinct Sections. The first explores simultaneously the resource overhead imposed by Clonos, the effect of its several configurable parameters, its scalability and the performance overhead introduced under failure-free scenarios. The second, explores failure scenarios in large-scale deployments, comparing throughput, latency, data duplication, and recovery time in configurations ranging a wide number of experimental variables. We begin, however, by detailing the experimental methodology used during the experiments.

4.1 Experimental Methodology

We aim to evaluate Clonos in a realistic deployment, to this end we deploy our infrastructure on a Kubernetes cluster which is itself hosted on an HPC Cloud environment. The bare metal on which we deploy several layers of virtualization use Intel Xeon Silver 4110 Processors, with a base clock speed of 2.10GHz which boosts up to 3.00GHz, memory supporting Intel UPI at 9.6 GT/s, a 10 Gbit/s connection to the network and SSD storage. We use Kafka supported by Zookeeper for stream storage when necessary, which

is the most commonly used stream storage platform. For checkpoint storage we use an Hadoop Distributed File System (HDFS) cluster, which we believe to be the most common type of checkpoint storage. Equally we deploy a cluster of the System-Under-Test (SUT), either Clonos or Flink. We compare Clonos to Flink only, as this offers the only apples-to-apples comparison possible, as other highly-available systems would not have the same expressiveness, and would offer similar recovery to Clonos, as those systems must also reprocess lost input records. We illustrate the layout of the infrastructure in Figure 4.1a and provide relevant Kubernetes deployment information in Table 4.1b. Since Flink uses heartbeats only for failure detection, we lower the failure detection bounds from the defaults to as low as was stable for Flink, without making them so frequent that they hinder performance. In particular, heartbeats are sent every 2 (default: 10) seconds, timing out after 3 (default: 60) seconds.



a Experimental Infrastructure Layout

Deployment	Replicas	CPU	Memory	Disk
Zookeeper	3	1	1GiB	5GiB
Kafka	5	5	5GiB	50GiB
HDFS NameNode	1	4	4GiB	5GiB
HDFS DataNode	3	2	4GiB	50GiB
Flink/Clonos JobManager	3	3	3GiB	5GiB
Flink/Clonos TaskManager	100	1	1GiB	5GiB

b Experimental Infrastructure Deployment Resources

Figure 4.1: Experimental Infrastructure

In each executed experiment, the SUT has 60 seconds of warm-up time before measurements are taken, and the SUT cluster is reset after each individual execution. These steps are important as the JVM will optimize the bytecode at runtime, and would otherwise lead to experimental noise in our results. Besides measuring overhead and failure

performance, the types of experiments we perform vary along another axis, that of realism. We conduct experiments using both a synthetic and a realistic workload. We perform experiments for the four combinations of these and proceed to introduce the specific methodologies used for each.

4.1.1 Workload

4.1.1.1 Synthetic workload

For the synthetic workload, we develop a configurable job, which allows us to range experimental parameters. This job disables slot-sharing and operator fusion optimizations. Each added operator thus simulates a task running a set of fused operators. To simulate a more complex chain of operators we increase the processing difficulty (PD) parameter. *PD* is important, as it helps to eliminate bottlenecks. We implement *PD* without yielding the thread (i.e. using sleep) by having a for loop increment a volatile integer. For every increase in depth or parallelism, we pay full network and serialization costs of determinants. Several parameters are adjustable in our benchmark (shown in Table 4.1) that are not in other benchmarks[42, 101, 105]. In particular, the amount of state held by each operator is configurable. Arrays of random data are used to reach the desired amount and a single array may be updated on each input record according to the *SA* parameter.

Table 4.1: Experimental parameters for synthetic experiments.

Parameter	Name	Default	Description
P	Parallelism	5	The number of parallel instances of each operator and Kafka partitions.
D	Depth	5	The depth of the graph including sources and sinks.
SS	State Size	100MB	The amount of state held by each parallel map operator.
CI	Checkpoint Interval	5s	The time the JobManager waits between checkpoints.
PD	Processing Difficulty	0	The difficulty of processing each record.
SA	State Access	0.0001%	Probability that operator will modify its state.
O	Operator	Map	The type of operator used."Window" or "Map"
TN	Time Notion	Processing	The notion of time used.
ST	Stream Type	Keyed	The type of stream used throughout the job. Keyed streams use hash-partitioning connections.
L	Load	50%	Percentage of the maximum load that system can handle.
KT	Kill Type	Single	The failure scenario used. "Single", "Concurrent" or "Frequent".
DSD	Determinant Sharing Depth	Full	How far in the graph to propagate determinants. Integer or "Full".
SB	Standby	1	The number of standbys maintained. 0 for local recovery.

The input data generated is very simple. The load generators generate records whose contents are a random key and a unique increasing value. The job can then extract the key if *ST* is keyed, in which case we also randomize the key deterministically on each operator, such that the records are sent to different downstream operators, simulating a realistic setting. The timestamp of the record is set depending on the *TN* used. They also contain rate control logic, allowing us to set a target throughput. By first measuring

the maximum throughput of a given configuration, we can then set the load on a specific experiment by using a percentage of the maximum throughput.

Jobs use mostly simple map operators, which read a record, apply the configured PD and output that same record with a deterministically randomized key. This is known as a *pass-through* query, as data simply passes through the dataflow, and is often used to evaluate the performance of the underlying distributed runtime[101]. This pass-through query is also important during failure experiments as it allows us to track the fine-grained recovery of the system, unlike in realistic jobs, where the effect may not be visible because the job only creates output every few seconds. If O is set to window, then a layer of window operators create windows of size 1 second and slide by 100 milliseconds (independent of the TN used) and when triggered simply emits a record with the concatenated contents, and map operators are used to pad the remaining depth.

4.1.1.2 Realistic Workload

For realistic workloads, we use Apache Beam's[58] implementation of NEXMark[101]. Apache Beam is a framework which allows for the implementation of dataflow programs in a way that is agnostic to the underlying execution engine. NEXmark is a set of benchmark queries specifically designed for evaluating Stream Processing System (SPS)s, unlike other benchmark workloads[79, 106] which have been adapted from Database Management System (DBMS)s to streaming. In order to use it, we build our own Apache Beam runner for Clonos. This was not too difficult, requiring only moderate adaptation of the Flink runner, and some work on the Apache Beam SDK such that we can inject and use our causal services. NEXMark has a simple schema, but a large and diverse number of different queries, using a diverse set of streaming operators. NEXMark simulates an online auction system such as EBay, with items which belong to categories (which are nested), auctions which may be open or closed and bids made by users. Queries range from the simple, such as currency conversion, to complex, for example finding the trending items by category. Queries in the NEXMark framework unfortunately only cover a notion of event-time, which allows them to guarantee correctness of the obtained results.

Besides the authenticity of the queries themselves, NEXMark introduces other features to simulate real user events. Events are consistent meaning that users must exist before they can bid on items, and must bid before they can win. The event rate fluctuates, putting more and less pressure on the system. Events can be randomly held back, simulating natural out-of-order streams. We apply the STRESS benchmark suite to both Flink and Clonos. This suite is a standard configuration for stress tests and produces one million events for processing.

4.1.2 Experiment Types

4.1.2.1 Overhead Experiments

In the overhead experiments we focus on measuring the SUT's maximum performance in different settings. Using a mediator (such as Kafka) to store input and output streams as well as measure throughput and latency is known to bottleneck the SUT's performance[59, 105]. To avoid such bottlenecks, during overhead experiments the load generators are placed directly into the streaming graph at the source operators. Sinks similarly discard any records instead of connecting to an external system. Because of this, our measurements must be done through the Metrics API offered by Flink and are thus more coarse-grained. This is not an issue however, as we only want to know the steady-state throughput and latency of the system.

Latency between operators is calculated using a punctuation mechanism[14]. We randomly pick a source and sink operator and sample their latency three times per second. Throughput is measured through the combined number of consumed input records per second across all sources and is measured every second. We then present the averaged values of these measurements.

We also execute some initial experiments to measure the resource overhead of Clonos (Section 4.2.1.1). To save on cluster resources we execute these locally using Docker Compose on an 8 core, 16 thread machine with 16 GiB of RAM. However, these experiments serve only to calculate the initial parameters, not to measure the overhead of Clonos.

4.1.2.2 Failure Experiments

For the failure experiments a more fine-grained mechanism for tracking throughput and latency is necessary, such that we can observe the behaviour of the system during recovery. To do so, we require a mediator system (Kafka in this case), which will allow us to measure *end-to-end latency*[105]. The number of Kafka partitions is set to be equal to the parallelism of the query, to maximize performance.

Unlike the latency measurements measured for overhead experiments, end-to-end latency is the real latency experienced by users. It measures the time between a record being available to consume at the input topic of the mediator system and it being processed and presented on the output topic. To achieve this, we modify both our synthetic and the realistic workloads such that the record generation timestamp is propagated through the graph towards the sinks. When written into the output side of the mediator, this latency timestamp is recorded, along with the timestamp at which the record was written into the output topic. These can be subtracted to calculate end-to-end latency for this record. Of course, records may be transformed or combined on their way to the sink, in which case we follow the rules presented in [59], by taking the maximum latency timestamp among any records being reduced into one.

Throughput is obtained through sampling the change in number of events in the Kafka

output topic for each partition and dividing by the elapsed time since the last sample. We are interested in the output side in this case because we want to observe the response of the system. Introducing a mediator (in our case Kafka), has been shown to be a considerable bottleneck[59, 105], and in our benchmark we have observed the same, with a bottleneck appearing at around 2 million records per second. In the case of synthetic failure experiments, we are forced to increase the *PD* parameter to 640, in order to bring the maximum throughput of the SUT well under the bottleneck at 0.5 million records per second. We then experiment with recovery scenarios using input streams at different load percentages (50%, 75% and 90%) of that maximum.

Three kinds of failure scenarios are tested, *single failure*, *concurrent failure*, and *frequent failures*. In the concurrent failure scenario we pick three sequenced (i.e. connected) tasks and kill them simultaneously. In the frequent failure scenario we pick pick three random tasks and kill one every five seconds. We define *recovery time* to be the time from which the first task is killed, to the time that end-to-end latency reaches the average value prior to failure. Intuitively, the recovery time is the time it takes for the system to catch-up to the real-time input stream. We do not compare the recovery performance for different levels of determinant sharing (1 versus full), as the recovery behaviour is identical independently of this. We use full determinant sharing except when comparing to approaches that use only in-flight logging.

For realistic failure experiments (which use the NEXMark framework) we pick queries 3 and 8. Our reasons for choosing these queries are that they are sufficiently complex (unlike queries 1 and 2 which are pass-through queries, and thus similar to our synthetic workload), perform windowing but produce regular output allowing us to track latency, produce large state sizes (around 100MiB) and have been used in previous works[36]. We show the CQL[15] (Continuous Query Language) syntax for both these queries in Listings 4.1 and 4.2.

Query 3 simulates a user looking for a certain product (category 10) in nearby states (OR, ID and CA). Its implementation involves filtering the Auctions input stream, finding only new auctions for category 10, and performing a key-by operation on the seller identifier. The persons stream is similarly filtered for new persons in the specified states and keyed-by the person identifier. Both keyed streams undergo an event-time windowed join with maximum waiting time of 600 seconds, and the desired data is projected. Of course, there are also operators for reading, deserializing, serializing and writing data from and to Kafka.

Query 8 monitors newly joined users who have created auctions. It filters the auction and person stream for only new users and auctions, then keys those streams using the person or seller identifiers. Each of those streams is partitioned into 10 second windows and each such window is treated as a static table. Whenever a new window is received, records are grouped by key and new matching elements are emitted. The same source and sink operators are present as well.

Listing 4.1: NEXMark Query 3 - Local Item Suggestion

```
SELECT Istream(P.name, P.city, P.state, A.id)
FROM Auction A [ROWS UNBOUNDED], Person P [ROWS UNBOUNDED]
WHERE A.seller = P.id AND (P.state = 'OR' OR P.state = 'ID' OR P.state = 'CA') AND A.
    ↪ category = 10;
```

Listing 4.2: NEXMark Query 8 - Monitor New Users

```
SELECT Rstream(P.id, P.name, A.reserve)
FROM Person [RANGE 10 SECONDS] P, Auction [RANGE 10 SECONDS] A
WHERE P.id = A.seller;
```

4.2 Overhead Experiments

Our overhead experiments are split into two separate groups. The first, using a synthetic workload, aims to compare Clonos to the system-under-modification, Flink, in a variety of settings and observe the overhead introduced in terms of throughput, latency and network bandwidth in pass-through queries that stress the systems. The second, using a realistic workload, serves to validate that the results transfer to real world use-cases.

4.2.1 Synthetic Workload

Before diving into comparisons between Flink and Clonos, we must first set reasonable parameters for Clonos and estimate the resource overhead that Clonos has.

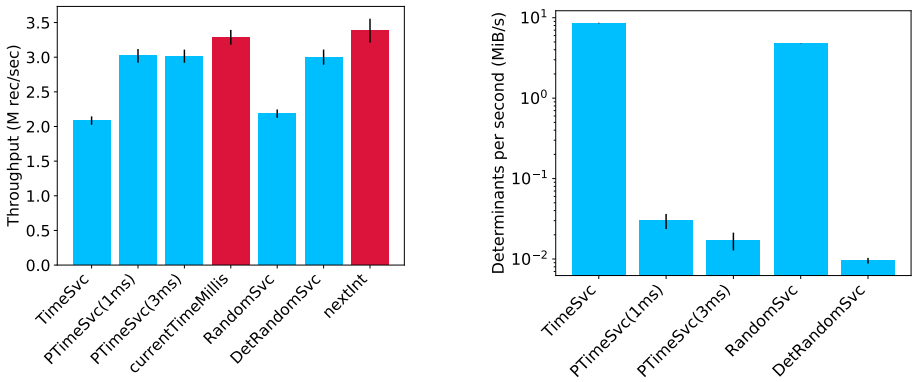
4.2.1.1 Resource Overhead

Clonos' in-flight logging and determinant logging both demand extra memory. We provide this through buffer pools of configurable size. We expect, through our design, that the size of determinant metadata collected is reasonably small. Because of this, we made the choice to have a fully volatile causal log, as opposed to a spillable one, like our in-flight log. We must choose a size for the causal log buffer pool such that it is never entirely used. The largest source of nondeterminism is by far the use of processing-time or ingestion-time. This is because they access the current time thousands of times per second. Other sources of determinants such as record delivery order, asynchronous determinants or output determinants occur at much lower frequencies either due to being periodic or by happening at the level of buffers instead of at the level of records. In ingestion time, a timestamp is assigned to every record as it leaves a source operator, while in processing time any operator that requires time to operate will access the current time on each incoming record.

If we execute a benchmark in Flink, measuring how many records a second it can process in processing time (without any processing logic), we can observe in Figure 4.2a that it reaches around 3 million records per second (currentTimeMillis). The basic time

service (TimeSvc) can reach slightly above 2 million records per second on the other hand. This is a still a positive result, and shows that even in a stress test, our causal service, causal log and determinant sharing implementations are well optimized. We can see in Figure 4.2b that the time service generates around 8 MiB of determinants per second at 1 million records per second. This amount of determinants per second would require a very large size for the causal logs. Furthermore, we would like the performance of our causal services to match the original implementations.

Thus, we designed the periodic time service (PTimeSvc), which uses timers to regularly update the current time to return. Observing the Figures again, we can see that the periodic time service is able to reach throughputs much closer to Flink, and generates determinants on the order of 0.03 MiB per second and 0.016 MiB per second for an update period of 1 and 3 respectively. Our deterministic random service (DetRandomSvc) also achieves much better performance than our naive implementation (RandomSvc), generating a single 5 byte determinant per epoch. With these implementations, and assuming a large epoch size of 30 seconds, we can calculate that to store the determinants of 50 tasks for an epoch of 10 seconds 8 MiB are necessary. For safety, we choose to use 16MiB (or 500 32KiB buffers) for the causal log manager buffer pool, ensuring we never run out of buffers. If multiple tasks execute on the same TaskManager they share this memory pool and determinants of the same upstream tasks are not kept twice.



a Throughput of causal services against JVM packages

b Determinant generation of causal services at 1M rec/sec

Figure 4.2: Causal Services

The in-flight log has two buffer pools, one for keeping in-flight buffers in memory and one for pre-fetching in-flight buffers from disk. In regards to the pre-fetching buffer pool, while too small a buffer pool may lead to decreased replay performance caused by a lack of pre-fetching, a large buffer pool offers no performance benefit. Thus, it is reasonable to select a value conservatively. We choose to use 10 buffers of 32KiB. Buffers are transmitted at variable rates, but these rates are limited by how fast the downstream operator can process them. It would be unreasonable to expect 10 buffers each with thousands of records be transmitted and processed faster that a single batch read from

local disk can be performed, and thus 10 buffers are sufficient to amortize the cost of disk reads.

We believe the size of the in-flight log buffer pool to depend on the other parameters of the in-flight log. To confirm this suspicion we locally benchmark the performance of the in-flight log while ranging its several parameters. This benchmark is similar to the previous one, but adds a small amount of PD to the operators, so as to simulate some computation. We show the results in Figure 4.3, where both strategies (eager and availability) are shown using different markers. The eager strategy, shown on the left using triangles, has a single configurable parameter, which is the buffer pool size. On the other hand, the availability strategy has two more parameters which are applied to the buffer availability checker. The checker interval defines the frequency with which the availability of buffers is checked, while the spill factor defines the availability level at which a spill is triggered. Thus, with a checker interval of 20 and a spill factor of 0.4, the availability checker will poll every 20 milliseconds for the current availability of buffers and if it is lower or equal to 0.4, it will trigger a spill. Performance of each configuration is indicated through a color map. The performance of Flink (the baseline) is also indicated on the color bar.

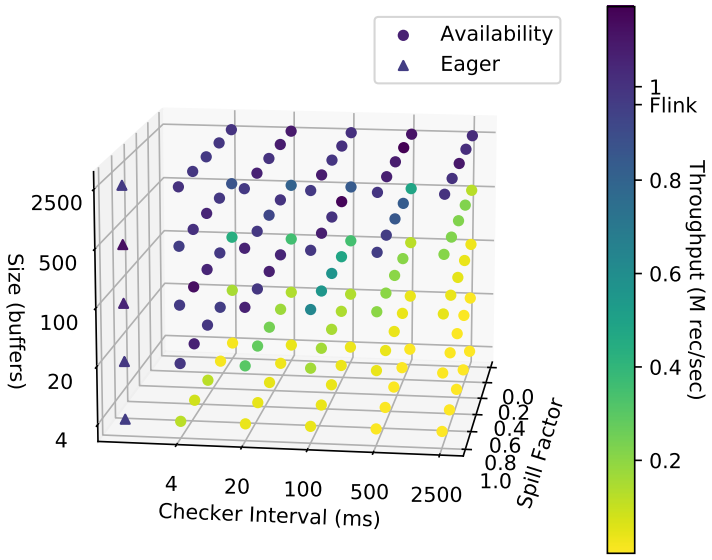


Figure 4.3: In-flight log performance grid

The eager strategy performs much better than expected. Our reasoning was that due to a lack of write batching and the synchronous work of submitting an asynchronous spill request, that this strategy would perform well, but below Flink for our benchmark. In hindsight, these benchmarks were performed on a system using solid-state drives, which may indeed benefit the eager strategy. The availability strategy, achieves slightly higher throughput, but has more nuanced results. First, large checker intervals lead to lower performance, as the pool runs out of buffers and a spill is not triggered until the interval has elapsed. This effect is hidden however, if the buffer pool size is large enough

to accommodate most of an epoch, approximating an in-memory in-flight log. For low in-flight log sizes, best performance is achieved with a small checker interval. This is most visible at size 100. Finally, the spill factor should be at least above 0.2, as under that value, there is a change that the buffer pool runs out in between availability checks.

When choosing the parameters for the in-flight log, other considerations must be taken in. First, we want to as much as possible avoid spilling to disk. If a failure occurs, having data in memory allows us to respond faster and furthermore, unnecessary and constant writes will quickly damage the underlying hardware. For this reason, we choose to proceed with the availability strategy. Good performance seems to be achieved with sizes above 100, and so we choose to go with 250 32KiB buffers for the buffer pool size. Spending CPU cycles on the availability checker is wasteful, so an interval of 50 milliseconds is used. Finally, we set the spill factor to 0.4, which keeps data in memory most of the time, but will spill when there is a risk that we may run out of buffers.

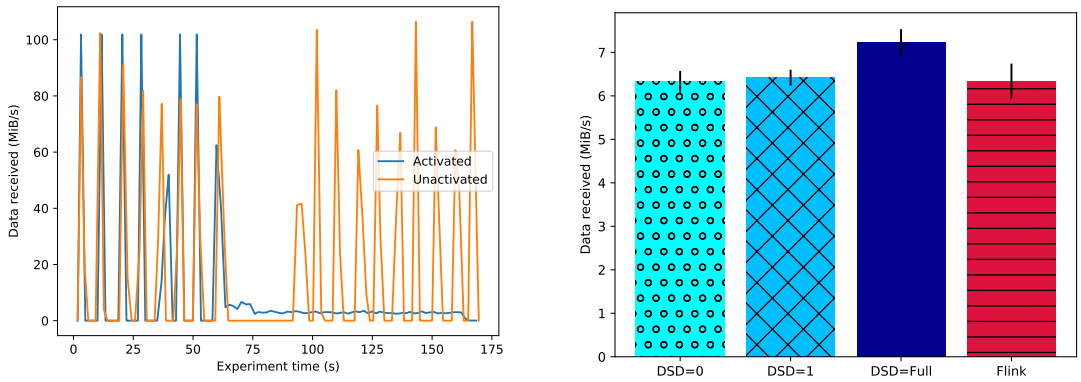
Overall, with this configuration Clonos has a volatile memory overhead of $(250 + 10 + 500) * 32KiB = 23.75MiB$ per job per TaskManager. If necessary, this can be reduced in several ways. The largest contributor is the size of the causal log buffer pool, which we set to a large value for safety. One way to reduce the necessary size of this buffer pool is to first adjust the queries to use a minimal amount of nondeterminism, such as using event-time and preferring forward type connections. Another way to limit the size of the causal log buffer pool is to reduce the determinant sharing depth. With a full sharing depth, the sink tasks are forced to keep determinants for all upstream tasks, and as such will be the first to run out of buffers. With a determinant sharing depth of one, they will only keep determinants for their direct upstream neighbours, decreasing the necessary size of the causal log substantially. Decreasing the epoch size by decreasing the checkpoint interval is another good way to bound the size of determinants kept, especially if incremental checkpointing is available. Decreasing epoch size will also allow us to have a smaller in-flight log buffer pool in the availability strategy. Furthermore, if memory overhead is a concern, the eager strategy also achieves good performance with a buffer pool size of four.

There is of also the overhead of disk storage, which is rarely used in stream processing for anything other than operator state. Clonos will spill in-flight buffers to disk when they can no longer fit in the in-flight log. This is done asynchronously to prevent a performance impact. In terms of space, the amount of spilled data is in the worst case equal to the epoch size multiplied by the average record size and the average throughput of the operator.

The use of passive standby also increases the amount of resources in two ways. First, it doubles the memory requirements, as standby operators should have the same capacity as their primaries. In future work, we plan to allow multiple standby operators share a host to alleviate this (see Section 5.2.1). Furthermore, state snapshots are downloaded frequently, utilizing network bandwidth. In Figure 4.4a, we present the amount of data received per second over time of both an activated and an unactivated standby operator

in our default setting. Note that before activation, standby operators do not receive network data, except when downloading a snapshot. Because incremental checkpointing is disabled, the operators download 100 MiB checkpoint roughly every 10 seconds. With incremental checkpointing enabled, this is however drastically lower as only the changes to state are downloaded. This Figure also shows the effect of our checkpoint backoff mechanism, which we described in Section 3.2.3.8.

We conclude our evaluation of resource overhead by measuring the amount of bandwith used by a sink operator in our default setting (5 depth, 5 parallelism) at different DSD. In this setting, the sink operator receives determinants for 5 tasks with a DSD of 1 and from 20 tasks for full (4) determinant sharing. We show the results in Figure 4.4b. With a DSD of 0, no increase in bandwidth used is measured, while with a DSD of 1 a very slight increase is present. Full determinant sharing leads to an increase in bandwidth used of slightly under 1MiB/s. In a cloud setting, such as the one in which we are evaluating Clonos, this is far from saturating the network links, however, this means that full determinant sharing could be infeasible for edge deployments. Finally, there is the concern of processing overhead, which we investigate in the following Section.



a Activated and unactivated standby tasks

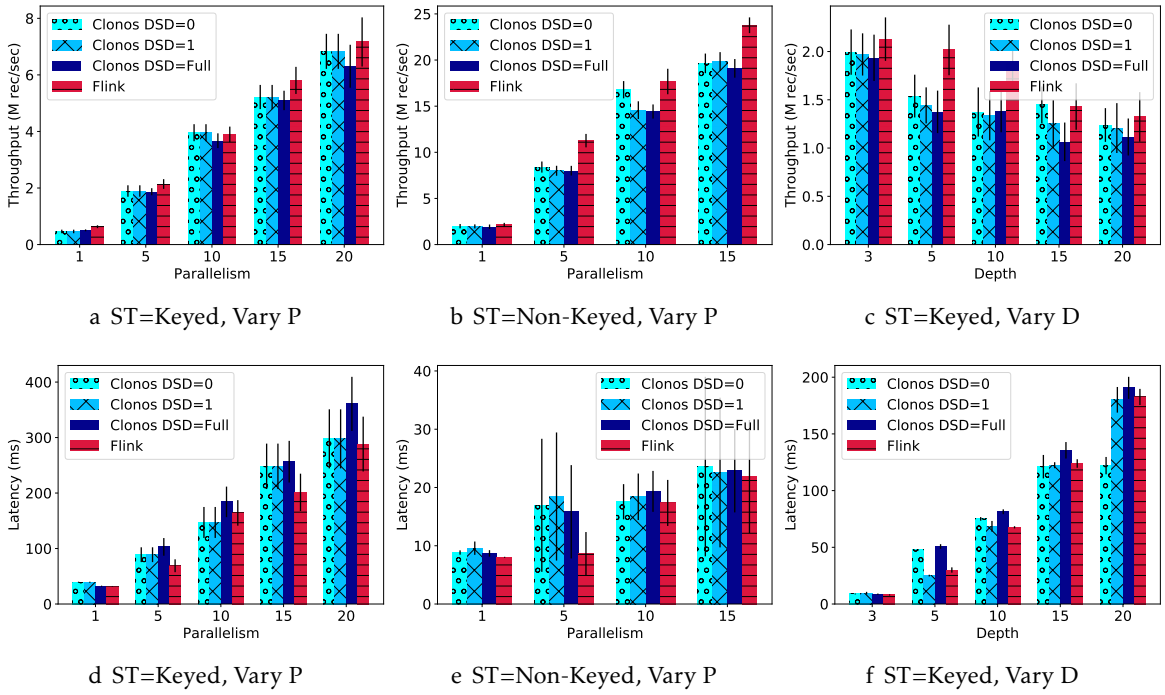
b Effect of DSD

Figure 4.4: Network plots

4.2.1.2 Performance Overhead

We now turn our attention to the performance overhead incurred by Clonos. We are interested in how the addition of causal logging and in-flight logging affect the failure-free performance of Clonos in terms of throughput and latency. Because the number of experimental parameters is quite large, creating an evaluation grid is infeasible. Instead, we vary a few parameters in our default configuration at a time and analyse the results.

In Figure 4.5 we show the performance of Clonos with DSD 0 (emulating passive standby works like [52, 54, 65]), 1 (which offers fast recovery for single failures), and full sharing of determinants.



a ST=Keyed, Vary P b ST=Non-Keyed, Vary P c ST=Keyed, Vary D

d ST=Keyed, Vary P e ST=Non-Keyed, Vary P f ST=Keyed, Vary D

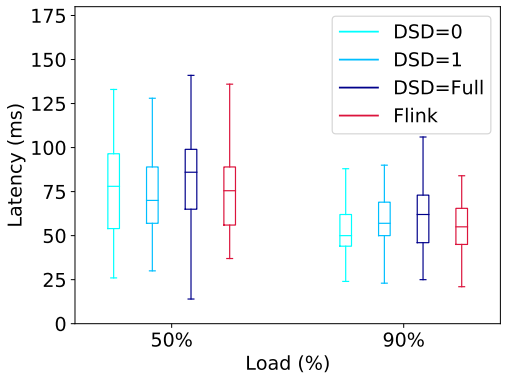
Figure 4.5: Performance overhead of Clonos in synthetic pass-through scenarios

Throughput is shown on the top row and the measured latency is shown in the row below. The first observation we would like to make is that in this setting, designed to stress Clonos by using a pass-through query and introducing network connections between each operator in which determinants must be shared, Clonos achieves impressive performance. In particular, in keyed streams, Clonos incurrs an average throughput overhead of 10% for determinant sharing depth of 1 and 13% for full determinant sharing. Using only in-flight logging also yields an average overhead of 10%, meaning that one layer of determinant sharing does not impact performance substantially. We also note that Clonos appears to be scalable, as the overhead does not increase substantially with parallelism. At a parallelism of 20 and a depth of 5, at the last layer of connections the determinants of 80 TaskManagers are being shared, without causing substantial overhead. However, we expect that at some higher levels of parallelism this effect would be evident for full determinant sharing, though we do not have the resources to test this. Non-Keyed streams achieve even higher throughputs, where the overhead of Clonos is even more visible, giving us an average overhead of 14%, 18%, and 19% for determinant sharing depths of 0, 1 and full respectively. Finally, the results of the effect of varying depth surprised us. We expected depth to have the largest impact on performance, and while it did have a high impact for depth 5 and 10, that overhead is quickly reduced at larger depths. This may be because at such an unrealistically high depth (all NEXMark queries have between 1 and 5 task depth), the bottleneck becomes not determinant sharing but serialization work.

We would also like to point out that these are the earliest results we gathered, and

were used to guide several improvements to Clonos. We did not manage to repeat these experiments due to resource constraints, as these are quite expensive to run, each run taking a few minutes and having to be repeated for different sharing depths In particular, improvements like reduced garbage collection pressure through determinant object reuse, maintaining direct pointers to thread causal logs as opposed to accessing them through hash maps and improved serialization routines through better encoding strategies were implemented. Smaller optimizations like avoiding recording order determinants when a single input channel is used also improved performance in non-keyed streams. In the meantime, we have also implemented our improved causal services, which should also reduce the amount of determinants generated.

Of course, since latency was not measured at a fixed throughput, it is not a reliable metric, but we can still observe that as a general trend Clonos has slightly higher latency, especially with full determinant sharing. We investigate this trend more in depth by running our default setting at a set throughput, for a longer period. We show the results in Figure 4.6a. At a load of 90% it is apparent that in-flight logging alone does not introduce significant latency overhead. A determinant sharing depth of 1 increases median latency slightly and full determinant sharing even more so. The lower bound for latency is similar for all deployments, but Clonos has significantly higher latency in the upper quantile for full determinant sharing. This is to be expected, as determinants must be serialized when they are about to be sent. It may seem surprising that at a lower load the average latency is higher, but this is by design. In Flink, records are not sent until buffers are filled. A safeguard exists which will send an incomplete buffer after 100ms, which explains the higher latency. At lower loads, the impact of determinant serialization is not as felt.

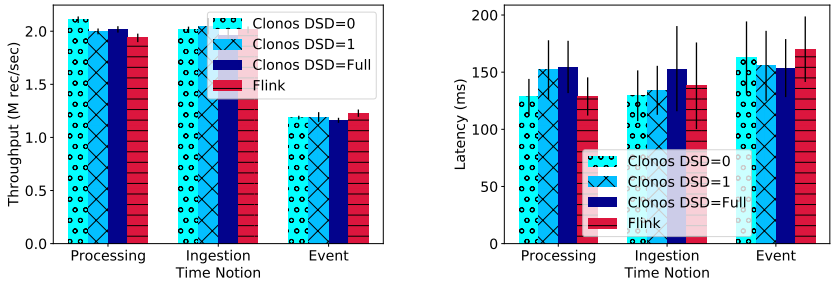


a Default Job

Figure 4.6: Latency at fixed throughput

Finally, we also report the values obtained in queries which contain a layer of window operators for different time notions in Figure 4.7. This is still our default setting (P=5, D=5), only with window operators in the middle layer. It is apparent that Clonos does not introduce discernible overhead in the presence of windows. In fact, the values obtained are so close, that we believe that most of what is observed is experimental noise.

This is most likely due to two facts. First, unlike the pass-through experiments, these experiments were completed after the improvements made to Clonos. Secondly, windows do not create as much output as a pass-through operator. As such there is less pressure on downstream operators and thus less determinants to be tracked.



a O=Window, Vary TN

b O=Window, Vary TN

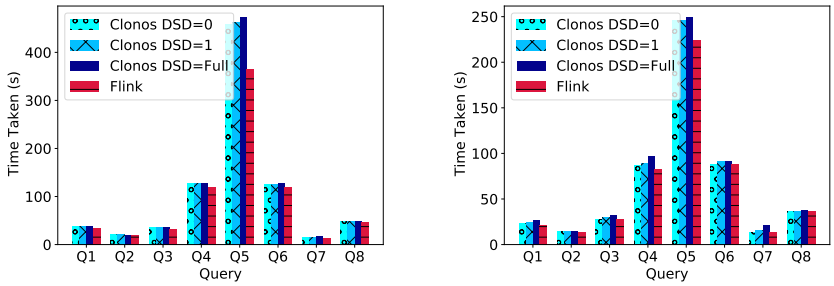
Figure 4.7: Performance overhead of Clonos in synthetic window scenarios

Next we will investigate the performance of Clonos in realistic stream processing conditions, where certain operators apply backpressure on upstream operators, filtering and windowing are interspersed in queries and not every connection is keyed, which should favour Clonos.

4.2.2 Realistic Workload

With reliable settings chosen, and some improvements made to Clonos, we proceeded to test Clonos on the NEXMark queries. We expect that because of said improvements and the more realistic nature of these jobs, Clonos will demonstrate performance much closer to Flink, independent of configuration. The queries in NEXMark have depths between 1 and 5, utilize timers for several purposes, utilize randomness for load-balancing and the current time for watermark generation. We show the obtained results in Figure 4.8. On average, for a parallelism of 5, Clonos takes 9%, 11% and 13% more time to complete the queries for determinant sharing depths of 0, 1, and full sharing. The difference in time taken is even lower for parallelism 10, though we believe this may be due to the low running times not allowing the overhead to accumulate. For the same reason, we note that it is in query 5 that the biggest difference in time taken is observed.

In conclusion, Clonos introduces a reasonably small amount of overhead in both throughput and latency. For realistic workloads, as opposed to pass-through queries, this overhead is smaller due to a reduced amount of communication between tasks. This overhead is to be expected as the features of Clonos do have to introduce an operational cost, however, we believe we have been able to substantially reduce the overhead in comparison to previous works, which rely on In-Order Processing (IOP) with deterministic sorting. We now turn to evaluating the performance of recovery, where we expect Clonos to truly surpass competing alternatives.



a P=5 b P=10

Figure 4.8: Nexmark Queries

4.3 Failure Experiments

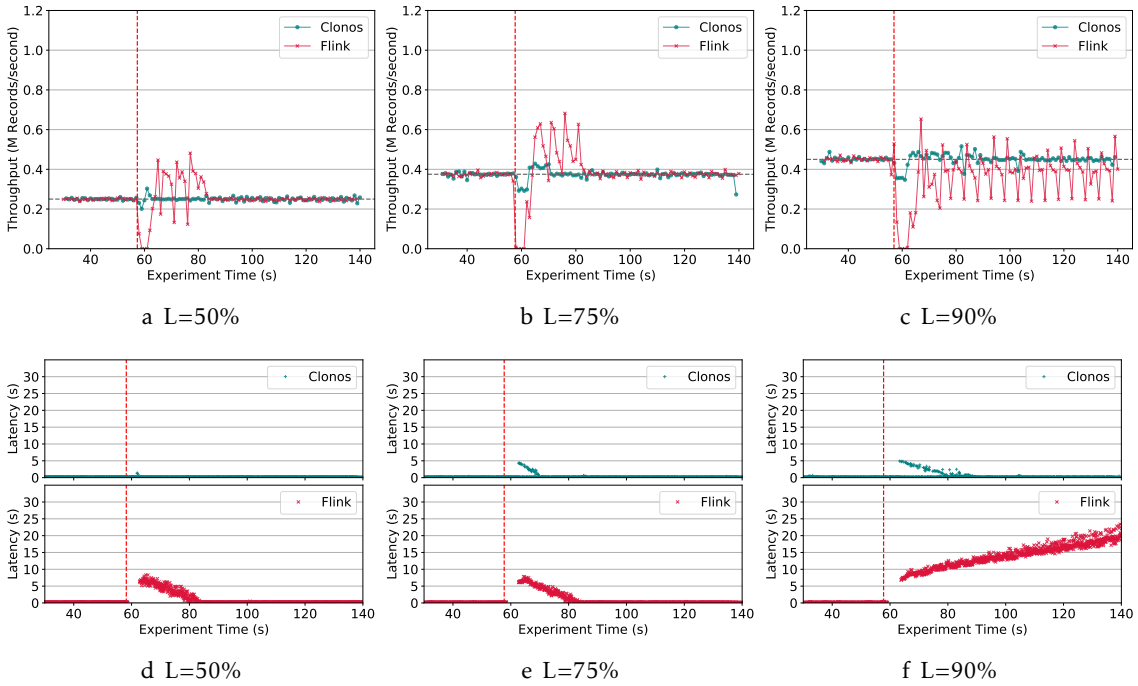
In this section we evaluate the recovery performance of Clonos, under several faulty scenarios. We begin with our synthetic workload, which will allow us to observe the effect of recovery more clearly and we validate these results with similar experiments using a realistic workload.

4.3.1 Synthetic Workload

Figure 4.9 compares Clonos and Flink in their recovery to a single failure at different loads using our default configuration, only with small state size (10MiB). The top row shows throughput over experiment time, while the bottom line of Figures shows latency as a scatterplot, where each marker represents a sampled record. The red vertical dashed line marks where the failure occurred, while the horizontal gray dashed line marks the load generator's throughput. Prior to failure, both systems are capable of following the load generator throughput with sub-second latency. After failure, Flink's throughput drops to 0 temporarily as the Directed Acyclic Graph (DAG) is reset, while Clonos's throughput decreases only slightly. Similarly in regards to latency, for Clonos a majority of records retains low latency, while only a small percentage reaches latencies under 5 seconds and this latency quickly decreases. Flink on the other hand has hits higher latencies and retains them for longer periods. Concerning recovery time, Clonos achieves 7 times faster recovery at 50% load (4 versus 28 seconds), but only 2 times faster recovery at 75% load (12 versus 28 seconds).

Examining the logs, we find that the cause of this is that while the Clonos task was killed near the end of an epoch, causing it to have to reprocess an entire epoch, the Flink task was killed in the middle of its epoch, meaning the entire graph had only to reprocess half an epoch. We attempted to control for this variable by triggering the failure after a fixed amount of time after submitting the job, but small differences in state upload times and processing make it hard to ensure that the failure happens in the middle of an epoch. Even still, more importantly, Clonos does not block processing in its recovery, the remaining tasks are able to continue making progress and consistency is later restored.

At 90% load, Clonos recovers fully in 30 seconds, while the equivalent experiment with Flink did not finish recovering by the end of the experiment. In fact, the growing latency that Flink shows indicates that recovery could take quite a while more, as the system struggles to catch up to the speed at which the input streams arrive.



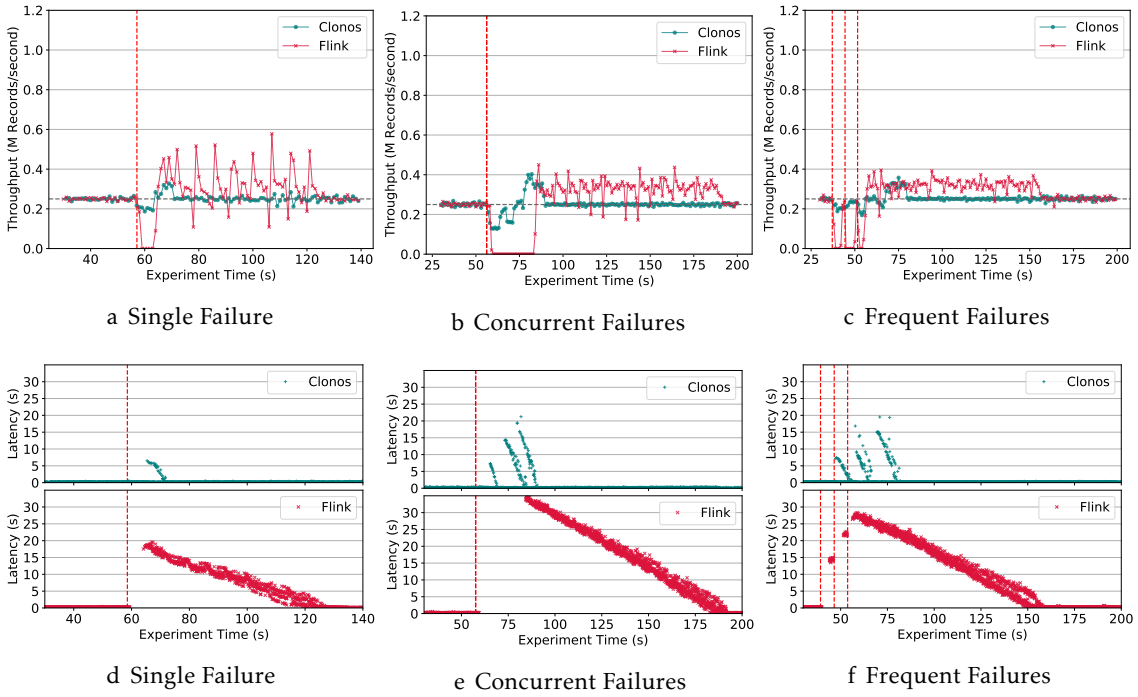
a L=50% b L=75% c L=90%

d L=50% e L=75% f L=90%

Figure 4.9: SS=10MiB Failure Experiments

In Figure 4.10 we present the results obtained for our default job configuration (similar to the previous one, only with 10 times as much state), at 50% load, but under different failure scenarios. With these experiments we show that Clonos can not only recover consistently but quickly even under complex failure scenarios. Clonos achieves 4 to 5 times faster recovery in these experiments, again without affecting the remainder of the DAG, meaning that average latency remains much lower. One may notice that Clonos's recovery in the concurrent and frequent failure scenarios is very similar. This is because when multiple failures happen, upstream tasks must finish recovery before downstream tasks may begin. Thus, in both these scenarios a similar behaviour occurs where first the most upstream task finishes its recovery, followed by the next downstream failure and only after that the last task may recover. Furthermore, we argue that it is this capacity to make progress under high failure rates that makes Clonos a possible contender for edge stream processing.

We now examine Clonos' recovery process in detail, using the experiment whose results are reported in Figure 4.10a as an example. Figure 4.11a zooms on to the recovery period of this task. Vertical dashed lines are used to indicate important moments in the recovery process. Right before failure, at moment 0, the standby task finishes downloading the latest state snapshot of the task that is about to fail. Only 200 milliseconds after



a Single Failure b Concurrent Failures c Frequent Failures

d Single Failure e Concurrent Failures f Frequent Failures

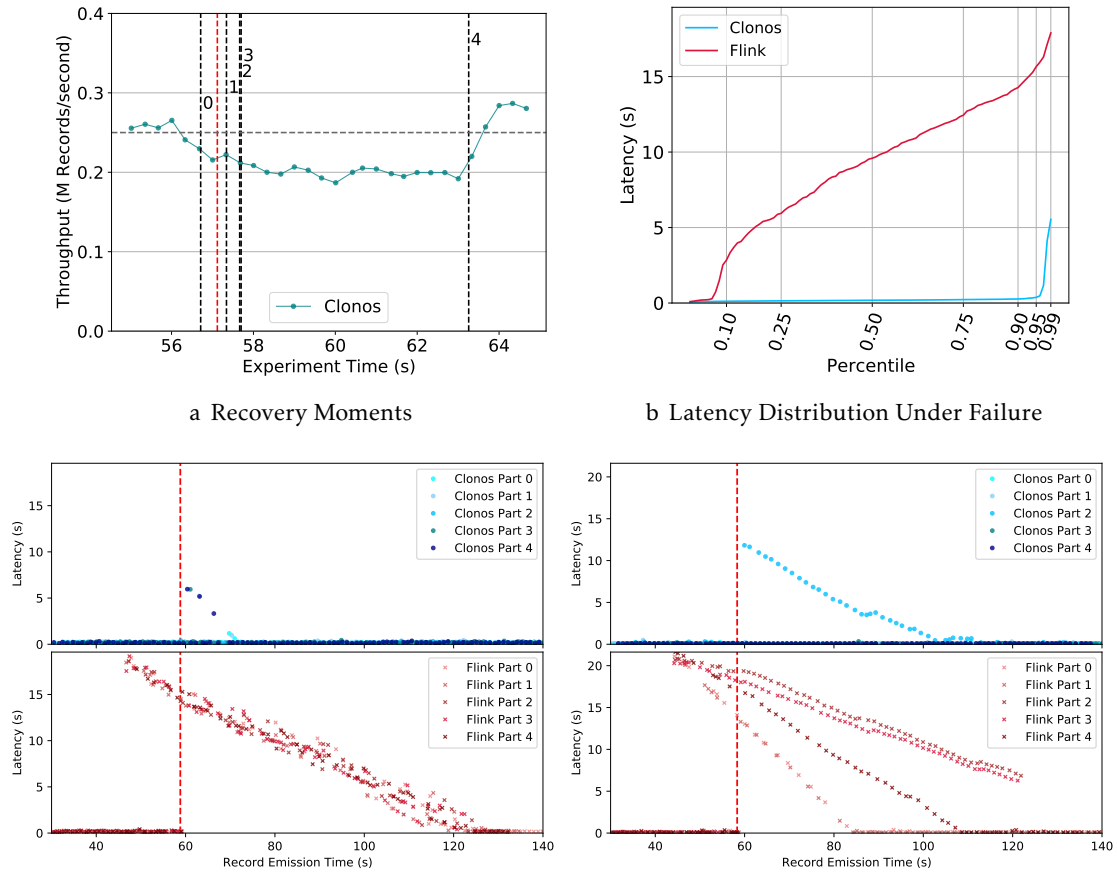
Figure 4.10: Default Job Failure Experiments

failure, the coordinator activates the standby task at moment 1, which begins reconfiguring its connections. This reconfiguration concludes at moment 2 after 320 milliseconds, at which point determinant and in-flight log requests are sent. Only 40 milliseconds after, in moment 3, all determinant responses are received and merged, at which point deduplication begins. Throughput remains low for this duration. The astute reader may have noticed that the loss of throughput shows an interesting behaviour in Clonos. In fact, with Clonos the loss of availability under failure is precisely $1/P$, where P is the parallelism of the particular logical operator in which a failure occurs. In this case, throughput decreases from 0.25 million records/second to 0.2 million records/second. If the load is not maximal, then the recovering task can reprocess the in-flight logs of upstream tasks at a higher throughput than during steady-state. After deduplication finishes (moment 4), throughput for the whole system temporarily increases as the recovering task catches-up to its input streams.

We additionally report in Figure 4.11b the percentile distribution of the end-to-end latency experienced by the systems for the 60 seconds after the failure. Not only does Clonos achieve much faster recovery, as even during recovery, only above the 95th percentile does latency rise sharply and only up to 5 seconds. Flink on the other hand, due to its recovery, only experiences latency under 5 seconds below the 25th percentile.

Flink does not perform deduplication of outputted records, as such consumers receive records processed prior to failure twice. This can be shown if we plot the latency graph using record emission time as the x-axis. We show this in Figure 4.11c. Flink demonstrates both high and low latency for records emitted prior to failure, indicating that these records

were emitted once prior to failure and then again after failure. Thus, for any failure that is not a sink failure, Clonos implicitly provides exactly-once delivery guarantees as well. In the same figure, we indicate the latency per output partition, and observe that every partition has elevated latency for both Clonos and Flink. Figure 4.11d, shows the same for a job that utilizes only non-keyed streams. While Flink continues to have elevated throughput on all partitions, Clonos now only has elevated throughput for the particular partition in which a failure occurred. This is because the partitions do not causally affect one another and thus in Clonos the recovery of one partition need not affect the other, something that current checkpointing-based rollback approaches do not take into account.



a Recovery Moments

b Latency Distribution Under Failure

c Duplication, ST=Keyed

d Duplication, ST=Non-Keyed

Figure 4.11: In-Depth Examination of Single Failure Recovery

We can emulate prior approaches[32, 46, 65] which utilized Out-of-Order Processing (OOP) architectures and passive standbys by disabling the causal log manager, which is done by setting DSD to zero. Of course, these past approaches could only utilize deterministic operators. The only aspect we do not capture of these past approaches is the deduplication performed at the receiver, however, this is convenient so that we can observe how fast replay begins. Figures 4.12a and 4.12c) present the results against Flink

in this setting. This approach skips the deterministic replay and deduplication period, as such throughput immediately increases as the recovering operator catches up to the input streams. Recovery is not faster than Clonos, as the operator must still reprocess the replay input streams. Furthermore, bandwidth is wasted sending the duplicates downstream. Finally, these approaches obtain incorrect results for any nondeterministic operators.

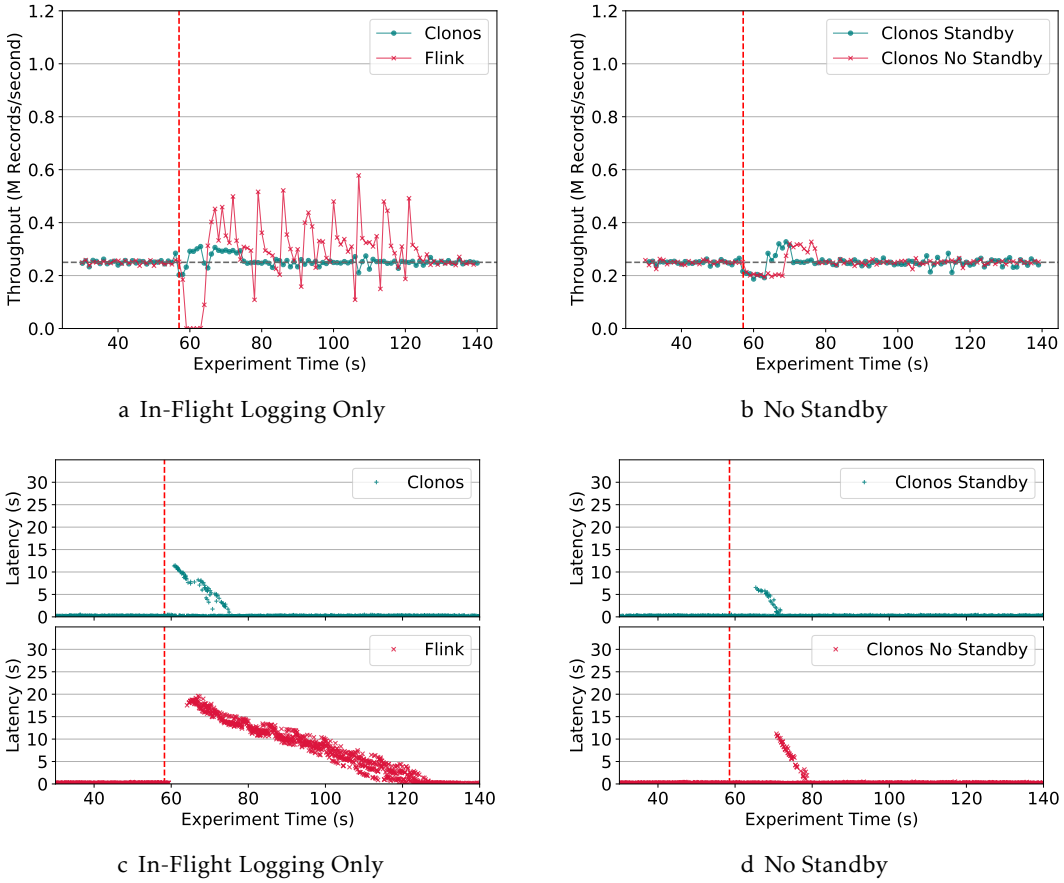
Similarly, we also test the recovery performance when not using standby tasks (Figures 4.12b and 4.12d). This also aims to emulate prior work which used only localized recovery[27, 71, 86], but is not a perfect replication, as prior work used optimistic logging of input and output dependencies only, while Clonos supports any form of nondeterminism and uses causal logging. Thus, in most cases, these prior works would suffer more rollback than Clonos without standby. In this experiment, the presence of standby task speeds up recovery by seven seconds. After which the deterministic replay process takes roughly the same time. However, with larger state this effect is exacerbated, as not using standby tasks leads to longer preparation times for the recovering task. The corollary of this is that for smaller state sizes, it may not be worth paying the resource cost of standby tasks for them to just remain idle. Instead, opting to use those resources to double the parallelism of the DAG may be best, thus lowering the load and speeding-up recovery by reducing epoch size.

The advantage of having standby tasks is more visible in the experiment reported in Figure 4.13, which use SS=0.5GiB for each operator. In this experiment, it took Flink a total of 15 seconds just to prepare the new task, before processing could begin. Clonos on the other hand begins reprocessing the epoch (which is quite large due to the large state size) immediately. The large state size causes epochs to be larger than the predefined checkpointing frequency, as the state upload time exceeds the checkpoint interval. This leads to larger replay periods for both Clonos and Flink.

4.3.2 Realistic Workload

Our realistic experimental results demonstrate perhaps even better results, but are slightly harder to interpret. We include them only to validate that our results transfer to real-world use-cases. Figure 4.14, shows the results for NEXMark queries 3 and 8, which we introduced in Section 4.1.2.2.

Query 3 is composed of a source task which performs some filtering, a stateless key-extracting and hash-partitioning task, a windowing task that performs the bulk of the computation, and a final task that performs some final windowed aggregation and also sinks the results. We fail the third task, which performs the bulk of the work and has the largest state. Clonos performed particularly well in query 3, where it recovered ten times faster than Flink. Because the third and fourth tasks are connected by forward type connections, we see this pattern in the Flink recovery that shows each partition recovering at its own pace. Because Flink resets the entire graph to maintain consistency, it is forced to go through the process of recreating and rebuilding all the windows where necessary.



a In-Flight Logging Only

b No Standby

c In-Flight Logging Only

d No Standby

Figure 4.12: Emulating prior approaches

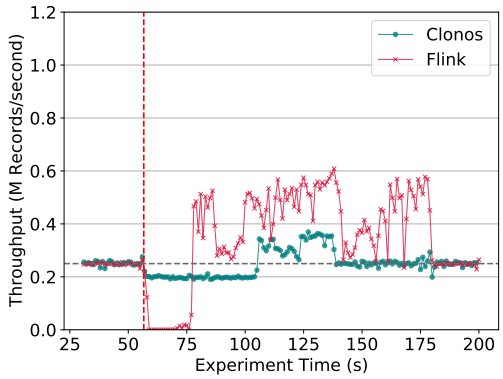
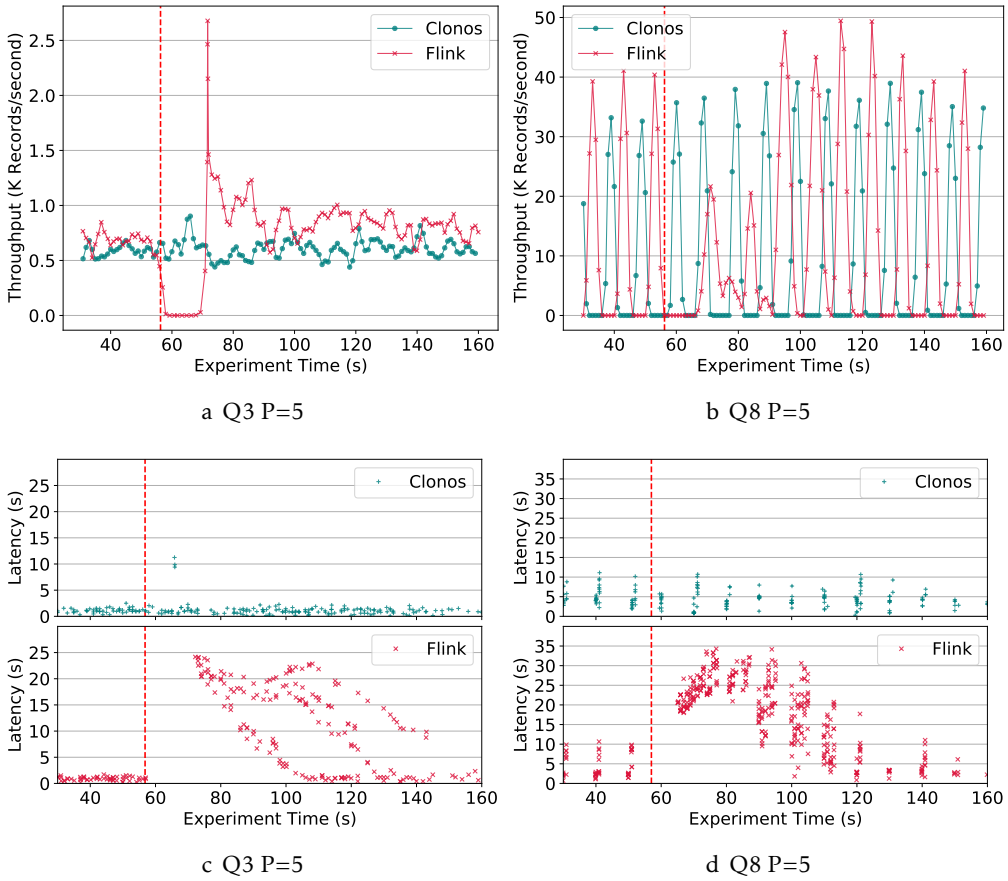


Figure 4.13: Recovery at 0.5GiB

In contrast, for Clonos, the four partitions where the failure did not occur are completely unaffected. The remaining failed task is able to quickly get up to speed as the upstream in-flight logs are already prepared. It is due to this that the failed task is able to recover so quickly.

Query 8 has a similar topology, but emits output much more infrequently. We again fail the third task, which performs the main windowing. It appears as though with Clonos,

the failure is not felt, as the standby recovers fast enough to trigger the window at the correct point. This is likely because there is a large amount of filtering occurring in the first task, and work is partitioned across the five windowing workers by key. However, Clonos was favoured in this test, as it started its recovery only a couple seconds after the start of a new epoch.



a Q3 P=5

b Q8 P=5

c Q3 P=5

d Q8 P=5

Figure 4.14: Failure experiments with realistic queries

Realistic streaming workloads perform heavy windowing, use a mixture of keyed and non-keyed streams and perform filtering of input. These characteristics favour Clonos as they mean that depending on where the failure occurs, the amount of in-flight data to be reprocessed can be very small, but they also make it difficult to present the recovery, which is why we hand-picked these queries.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

Stream Processing System (SPS)s are widespread in the industry and are utilized for a wide range of critical use-cases, from data analytics to event-driven applications. To support a variety of use-cases systems must be expressive, while to support critical applications they must be consistent and highly available. However, no solution in the state-of-the-art can be found which simultaneously provides all these requirements.

To provide a deeper understanding of the complexity of the issue at hand, we studied the field of rollback recovery, which provided us insight into how distributed applications can recover their state in a consistent fashion by dealing with nondeterminism. We then surveyed the field of stream processing and found that current systems can be split into two categories. Reliable production-grade SPSs provide consistency and expressiveness, but lack high-availability as they recover from failures through global rollback mechanisms, most commonly consistent checkpointing. On the other hand, highly available systems which achieve consistency have historically sacrificed expressiveness and performance to obtain deterministic replayability. Other approaches simply forego consistency and exactly-once processing guarantees completely. The closest approaches to our own solution utilized optimistic logging which we argued to be inappropriate for providing high-availability in a stream processing setting, as they lead to excessive rollback. Merging our knowledge of stream processing and rollback recovery, we found that the rarely utilized method of causal logging perfectly fit the stream processing paradigm, while providing a method towards achieving consistent high-availability.

Our solution, Clonos, embraces nondeterminism of all kinds and is able to recover arbitrary operators with exactly-once processing guarantees. As such, the solution proposed in this thesis is the only high-availability work which supports the full feature-set

of reliable modern SPSs such as user-defined functions, out-of-order processing, timers, effective watermark generation, as well as ingestion- and processing-time notions. The adaptation of causal logging to stream processing is not direct, and we make several contributions in this respect. The tracking of record delivery at the buffer level decreases the amount of determinants generated and thus shared. The implementation of causal services makes it simple for users to apply nondeterministic actions. We develop specialized causal services, such as the periodic time causal service, which heavily decrease the overhead of per-record access to nondeterministic actions over the naive version. We explain how we deal with the multi-threaded nature of an SPS using both locking discipline, record counting and multiple causal logs per task. Finally, we share our designs for efficient non-blocking and no-copy logging and sharing of determinants, as well as, logging and replay of in-flight records. Our in-flight log is novel in several respects, as it is able to spill to disk, limiting its resource consumption and uses pre-fetching to reduce the latency of replay.

We believe that with Clonos we have made significant progress towards consistent high-availability in high-performance streaming dataflows without sacrificing expressiveness. In realistic workloads, Clonos achieves a performance overhead of only 11 to 13% compared to checkpointing methods, while offering up to 10 times faster non-blocking recovery with much lower average latency. We also showed that Clonos does not introduce significant overhead over prior high-availability approaches when the determinant sharing depth is only 1, but gains immensely in expressiveness. Furthermore, Clonos does not require In-Order Processing (IOP) which would introduce latency and resource overhead. With further engineering, we believe it is possible to further reduce this overhead. We have also shown, through our synthetic experiments that Clonos is scalable, maintaining roughly the same overhead independent of graph size. Though our evaluation does not reach such large-scale deployments, at some scales the overhead of determinant sharing is bound to start introducing significant overhead, for which case we outline some solutions. In particular, resorting to a determinant sharing depth of 1 reduces the amount of determinants shared drastically, without sacrificing correctness, and still offers fast non-blocking recovery for single failures which is often regarded as sufficient[54].

5.2 Future Work

Clonos was an ambitious project, which attempted to implement a lot of functionality in a short time. As such there were still a few hypothesis and optimizations left untested. We will look at these first in Section 5.2.1. Furthermore, being the first stream processor implementing causal logging-based fault tolerance and thus having a considerably different runtime, Clonos allows for the exploration of many new topics, which we describe in 5.2.2.

5.2.1 Improvements and Optimizations

Clonos can still be improved in several ways. First, to reduce overhead of maintaining standby tasks, we propose two improvements. The first involves recognizing which tasks require a standby at runtime by tracking state sizes and deploying standby tasks only for those with large state. The second improvement in this venue is to allow multiple passive standby tasks to share the resources of a single task slot. Upon the activation of a standby, other standbys in the same slot would be disposed. These improvements could drastically reduce the cost of maintaining standby tasks.

Regarding in-flight log spill policies, we believe that smarter policies can be designed. Clonos currently spills the oldest in-flight records first, leaving fresh in-flight records in-memory. When a failure happens, replay of in-flight records is necessary and starts from the oldest data. We attempt to amortize the cost of reading this data by having a recovery buffer pool which we use to pre-fetch buffers ahead of the replay, but we still pay the full cost of the first read from disk. By simply having smarter spill policies, which maintain the oldest records in memory, we can reduce this cost.

To limit the complexity of this work, we chose not to implement support for cyclical graphs. Additionally, streaming systems research has not yet settled on a common implementation for supporting cycles, which makes this less desirable. However, use-cases necessitating cyclical graphs such as machine learning and graph processing have also been a growing trend. Causal logging supports cycles[11] quite naturally and as such it is likely that this is a feasible extension. This would however require improved membership tracking.

As we saw in Section 2.2.2.5, there is a large amount of work on reducing the amount of duplicate determinants shared. Clonos implements the simple $\prod_{det}$ protocol and our evaluation indicates that the amount of determinant data shared is small in comparison to the volumes of data that compose the main dataflow. However, this should not stop us from exploring a possible optimization. In concrete, we believe it is possible to implement a $\prod_{log}$ protocol leveraging the backchannels between tasks for credit-based control flow messages. Causal log consumer offsets can be propagated backwards through the graph in these channels, leading to reduced network bandwidth usage. This may be especially relevant in environments with constrained bandwidth.

5.2.2 Future Projects

Clonos' causal logging opens up a plethora of opportunities in stream processing. We explore the most interesting ones in this Section.

Achieving exactly-once delivery is generally only possible through the use of transactional sink operators. Transaction-based output commit causes increased latency[13, 57, 100]. In checkpointing based systems such as Flink, latency is at least as large as the checkpoint interval plus the time to upload the state. With causal logging, an alternative exists that drastically reduces latency. In essence, the external system to which events

are sinked participates in the recovery of the SPS, by also managing determinants. The system can be completely oblivious to what they mean, it is only required that they are maintained. Messages sent to the external system piggyback the determinants needed to ensure they are stable, and as such can be committed immediately. If a failure happens, the sink operator requests determinants from the outside world event sink and then recovers like any other operator in Clonos.

Clonos may be a solid foundation for building an Hybrid Transaction/Analytical Processing (HTAP) system. HTAP systems support both On-Line Analytical Processing (OLAP) workloads and On-Line Transaction Processing (OLTP) workloads. Such systems have historically been built by adding stream processing capabilities to a classic OLTP Database Management System (DBMS)[76]. Clonos enables building such a system starting with an SPS. As an example, consider an SPS executing a query over financial transactions. The state maintained as it processes streams of financial transactions is the account balance of each user. A user then goes to the bank and deposits a certain amount. Instead of complicating the query by adding a number of new operators and connections to it, a simple REST request can be made to the task containing the user's account. This request can be encoded as a determinant, and thus we can ensure that after a failure it is replayed. If only checkpointing is used, such point updates to operator state cannot be made, as they would be lost after the graph is reset in case of a failure.

Reconfiguration (changing parallelism, key partitionings, migrating state etc.) is still an unsolved problem in stream processing. The typical approach[24] involves waiting for a checkpoint to complete, and then redeploying the graph with the new configuration. More recent works have improved the latency of state migration[36] and the flexibility of the control plane[72]. However, reconfiguration suffers from the same problems that plague high-availability in stream processing, namely that one must choose between consistency and expressiveness (nondeterministic operations), as shown in [27], which solves scale-out and localized recovery simultaneously by constraining expressiveness. Several works also employ parallel recovery[27, 71, 110], where failed tasks are simultaneously scaled-out to speed up their replay processes. We believe that Clonos' approach to dealing with nondeterminism can aid in maintaining consistency and expressiveness while performing such reconfigurations without redeploying the entire graph. Furthermore, we believe we can also treat recovery as a case reconfiguration, and employ parallel recovery, which would reduce recovery times.

Our concurrent and frequent failure experiments show that Clonos is capable of making progress even under high failure rates. One question that arises is the applicability of Clonos in edge scenarios where high churn of member nodes is frequent. Edge stream processing is a topic of high interest in recent years, with projects such as NebulaStream[111] just starting to gain traction. In such a setting, the main issue is dealing with network partitions, which we believe can be addressed by providing processing guarantees only for the portion of the graph hosted in the cloud.

Bibliography

[1] G. A. Agha. *Actors: A model of concurrent computation in distributed systems*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[2] A. Akhter, M. Fragkoulis, and A. Katsifodimos. "Stateful functions as a service in action." In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1890–1893.

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. "MillWheel: fault-tolerant stream processing at internet scale." In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing." In: (2015).

[5] S. Albers. "Online algorithms: a survey." In: *Mathematical Programming* 97.1-2 (2003), pp. 3–26.

[6] L. Alvisi. *Understanding the message logging paradigm for masking process crashes*. Tech. rep. Cornell University, 1996.

[7] L. Alvisi, K. Bhatia, and K. Marzullo. "Causality tracking in causal message-logging protocols." In: *Distributed Computing* 15.1 (2002), pp. 1–15.

[8] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. De Mel. "An analysis of communication induced checkpointing." In: *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*. IEEE. 1999, pp. 242–249.

[9] L. Alvisi, B. Hoppe, and K. Marzullo. "Nonblocking and orphan-free message logging protocols." In: *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE. 1993, pp. 145–154.

[10] L. Alvisi and K. Marzullo. "Trade-offs in implementing causal message logging protocols." In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. Citeseer. 1996, pp. 58–67.

[11] L. Alvisi and K. Marzullo. "Message logging: Pessimistic, optimistic, causal, and optimal." In: *IEEE Transactions on Software Engineering* 24.2 (1998), pp. 149–159.

[12] *Apache Flink Credit-Based Flow Control*. https://flink.apache.org/2019/06/05/flink-network-stack.html#credit-based-flow-control. 2019.

[13] *Apache Flink exactly-once implementation*. https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html. 2018.

[14] *Apache Flink frontpage*. https://flink.apache.org/. 2020.

[15] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. "Stream: The stanford data stream management system." In: *Data Stream Management*. Springer, 2016, pp. 317–336.

[16] M. D. de Assuncao, A. da Silva Veith, and R. Buyya. "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions." In: *Journal of Network and Computer Applications* 103 (2018), pp. 1–17.

[17] L. Atzori, A. Iera, and G. Morabito. "The internet of things: A survey." In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[18] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. "Fault-tolerance in the Borealis distributed stream processing system." In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, pp. 13–24.

[19] P. A. Bernstein and N. Goodman. "Multiversion concurrency control—theory and algorithms." In: *ACM Transactions on Database Systems (TODS)* 8.4 (1983), pp. 465–483.

[20] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley New York, 1987.

[21] D. Borthakur. "The hadoop distributed file system: Architecture and design." In: *Hadoop Project Website* 11.2007 (2007), p. 21.

[22] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. "State management in Apache Flink®: consistent stateful distributed stream processing." In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1718–1729.

[23] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. "Lightweight asynchronous snapshots for distributed dataflows." In: *arXiv preprint arXiv:1506.08603* (2015).

[24] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. "Apache flink: Stream and batch processing in a single engine." In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[25] F. Carcillo, A. Dal Pozzolo, Y.-A. Le Borgne, O. Caelen, Y. Mazzer, and G. Bontempi. "Scarff: a scalable framework for streaming credit card fraud detection with spark." In: *Information fusion* 41 (2018), pp. 182–194.

[26] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. "Monitoring streams: a new class of data management applications." In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment. 2002, pp. 215–226.

[27] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. 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*. 2013, pp. 725–736.

[28] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. "FlumeJava: easy, efficient data-parallel pipelines." In: *ACM Sigplan Notices* 45.6 (2010), pp. 363–375.

[29] B. Chandramouli, J. Goldstein, M. Barnett, and J. F. Terwilliger. "Trill: Engineering a Library for Diverse Analytics." In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 51–60.

[30] K. M. Chandy and L. Lamport. "Distributed snapshots: Determining global states of distributed systems." In: *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985), pp. 63–75.

[31] H. Chen, R. H. Chiang, and V. C. Storey. "Business intelligence and analytics: From big data to big impact." In: *MIS quarterly* (2012), pp. 1165–1188.

[32] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. "Scalable Distributed Stream Processing." In: *CIDR*. Vol. 3. 2003, pp. 257–268.

[33] O. P. Damani, A. Tarafdar, and V. K. Garg. "Optimistic recovery in multi-threaded distributed systems." In: *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*. IEEE. 1999, pp. 234–243.

[34] J. Dean. "Handling large datasets at google: Current systems and future directions." In: *Data-Intensive Computing Symposium*. 2008.

[35] J. Dean and S. Ghemawat. "MapReduce: Simplified data processing on large clusters." In: (2004).

[36] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. "Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2471–2486.

[37] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. "Optimizing Space Amplification in RocksDB." In: *CIDR*. Vol. 3. 2017, p. 3.

[38] *Drivetribe's Flink Backend*. https://www.ververica.com/blog/drivetribe-cqrs-apache-flink. 2017.

[39] E. N. Elnozahy and W. Zwaenepoel. "Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit." In: *IEEE Transactions on Computers* 5 (1992), pp. 526–531.

[40] E. N. Elnozahy. "Manetho: fault tolerance in distributed systems using rollback-recovery and process replication." Doctoral dissertation. Rice University, 1994.

[41] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. "A survey of rollback-recovery protocols in message-passing systems." In: *ACM Computing Surveys (CSUR)* 34.3 (2002), pp. 375–408.

[42] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. "Making state explicit for imperative big data processing." In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 49–60.

[43] Y. Y. M. I. D. Fetterly, M. Budiu, Ú. Erlingsson, and P. K. G. J. Currey. "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language." In: *Proc. LSDS-IR* 8 (2009).

[44] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google file system." In: (2003).

[45] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. "Graphx: Graph processing in a distributed dataflow framework." In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 599–613.

[46] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. "An empirical study of high availability in stream processing systems." In: *Middleware (Companion)*. 2009, p. 23.

[47] M. Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[48] J. Hamilton. *The cost of latency*. 2009.

[49] *Hazelcast Jet*. https://hazelcast.com/products/jet/. 2019.

[50] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and C. Fetzer. "An adaptive replication scheme for elastic data stream processing systems." In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. 2015, pp. 150–161.

[51] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. "A catalog of stream processing optimizations." In: *ACM Computing Surveys (CSUR)* 46.4 (2014), pp. 1–34.

[52] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. "High-availability algorithms for distributed stream processing." In: *21st International Conference on Data Engineering (ICDE'05)*. IEEE. 2005, pp. 779–790.

[53] J.-H. Hwang, U. Cetintemel, and S. Zdonik. "Fast and highly-available stream processing over wide area networks." In: *2008 IEEE 24th International Conference on Data Engineering*. IEEE. 2008, pp. 804–813.

[54] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. "A cooperative, self-configuring high-availability solution for stream processing." In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 176–185.

[55] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks." In: *ACM SIGOPS operating systems review*. Vol. 41. 3. ACM. 2007, pp. 59–72.

[56] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce. "Consistent regions: Guaranteed tuple processing in ibm streams." In: *Proceedings of the VLDB Endowment* 9.13 (2016), pp. 1341–1352.

[57] *Kafka Streams Exactly-Once*. https://www.confluent.io/blog/enabling-exactly-once-kafka-streams/. 2017.

[58] H. Karau. "Unifying the open big data world: The possibilities* of apache BEAM." In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE Computer Society. 2017, pp. 3981–3981.

[59] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. "Benchmarking Distributed Stream Data Processing Systems." In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018). DOI: 10.1109/icde.2018.00169. URL: http://dx.doi.org/10.1109/ICDE.2018.00169.

[60] A. Katsifodimos and M. Fragkoulis. "Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications." In: 2019.

[61] B. P. Krasňan. "Benchmarking Big Data Streaming Platforms." In: ().

[62] J. Kreps, N. Narkhede, J. Rao, et al. "Kafka: A distributed messaging system for log processing." In: *Proceedings of the NetDB*. 2011, pp. 1–7.

[63] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. "Twitter heron: Stream processing at scale." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 239–250.

[64] N. Kung and R. Morris. "Credit-based flow control for ATM networks." In: *IEEE network* 9.2 (1995), pp. 40–48.

[65] Y. Kwon, M. Balazinska, and A. Greenberg. "Fault-tolerant stream processing using a distributed, replicated file system." In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 574–585.

[66] L. Lamport. "The implementation of reliable distributed multiprocess systems." In: *Computer Networks (1976)* 2.2 (1978), pp. 95–114.

[67] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[68] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. 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*. 2005, pp. 311–322.

[69] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. "Out-of-order processing: a new architecture for high-performance stream systems." In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 274–288.

[70] J. Lin. "The lambda and the kappa." In: *IEEE Internet Computing* 21.5 (2017), pp. 60–66.

[71] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. "Streamscope: continuous reliable distributed processing of big data streams." In: *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 2016, pp. 439–453.

[72] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, et al. "Chi: a scalable and programmable control plane for distributed stream processing systems." In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1303–1316.

[73] A. Martin, C. Fetzer, and A. Brito. "Active replication at (almost) no cost." In: *2011 IEEE 30th International Symposium on Reliable Distributed Systems*. IEEE. 2011, pp. 21–30.

[74] F. Mattern et al. *Virtual time and global states of distributed systems*. Citeseer, 1988.

[75] N. Maurer and M. Wolfthal. *Netty in Action*. Manning Publications New York, 2016.

[76] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, et al. "S-store: Streaming meets transaction processing." In: *arXiv preprint arXiv:1503.01143* (2015).

[77] A. Mukherjee, P. Diwan, P. Bhattacharjee, D. Mukherjee, and P. Misra. "Capital market surveillance using stream processing." In: *2010 2nd International Conference on Computer Technology and Development*. IEEE. 2010, pp. 577–582.

[78] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. "Naiad: a timely dataflow system." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 439–455.

[79] R. O. Nambiar and M. Poess. "The Making of TPC-DS." In: *VLDB*. Vol. 6. 2006, pp. 1049–1058.

[80] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini. "The power of both choices: Practical load balancing for distributed stream processing engines." In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, pp. 137–148.

[81] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. "S4: Distributed stream computing platform." In: *2010 IEEE International Conference on Data Mining Workshops*. IEEE. 2010, pp. 170–177.

[82] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. "Samza: stateful scalable stream processing at LinkedIn." In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1634–1645.

[83] S.-H. Oh, J.-S. Kang, Y.-C. Byun, G.-L. Park, and S.-Y. Byun. "Intrusion detection based on clustering a data stream." In: *Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05)*. IEEE. 2005, pp. 220–227.

[84] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. "Pig latin: a not-so-foreign language for data processing." In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1099–1110.

[85] S. Qian, G. Wu, J. Huang, and T. Das. "Benchmarking modern distributed streaming platforms." In: *2016 IEEE International Conference on Industrial Technology (ICIT)*. IEEE. 2016, pp. 592–598.

[86] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. "Timestream: Reliable stream computation in the cloud." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 1–14.

[87] R. Ramakrishnan, J. Gehrke, and J. Gehrke. *Database management systems*. Vol. 3. McGraw-Hill New York, 2003.

[88] M. A. Shah, J. M. Hellerstein, and E. Brewer. "Highly available, fault-tolerant, parallel dataflows." In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 2004, pp. 827–838.

[89] E. Shahverdi, A. Awad, and S. Sakr. "Big Stream Processing Systems: An Experimental Evaluation." In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2019, pp. 53–60.

[90] K. Shvachko, H. Kuang, S. Radia, R. Chansler, et al. "The hadoop distributed file system." In: *MSST*. Vol. 10. 2010, pp. 1–10.

[91] A. J. Smith. "Sequentiality and prefetching in database systems." In: *ACM Transactions on Database Systems (TODS)* 3.3 (1978), pp. 223–247.

[92] *Statefun*. http://statefun.io.

[93] M. Stonebraker, U. Çetintemel, and S. Zdonik. "The 8 requirements of real-time stream processing." In: *ACM Sigmod Record* 34.4 (2005), pp. 42–47.

[94] R. Strom and S. Yemini. "Optimistic recovery in distributed systems." In: *ACM Transactions on Computer Systems (TOCS)* 3.3 (1985), pp. 204–226.

[95] L. Su and Y. Zhou. "Tolerating correlated failures in massively parallel stream processing engines." In: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE. 2016, pp. 517–528.

[96] Y. Tamir and C. H. Séquin. "Error Recovery in Multicomputers Using Global Checkpoints." In: *In 1984 International Conference on Parallel Processing*. 1984, pp. 32–41.

[97] N. Tantalaki, S. Souravlas, and M. Roumeliotis. "A review on big data real-time stream processing and its scheduling techniques." In: *International Journal of Parallel, Emergent and Distributed Systems* 35.5 (2020), pp. 571–601.

[98] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. "Hive: a warehousing solution over a map-reduce framework." In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

[99] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. "Storm@ twitter." In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156.

[100] *Trident computational model*. http://storm.apache.org/releases/current/Trident-tutorial.html. 2012.

[101] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. *NEXMark–A Benchmark for Queries over Data Streams (DRAFT)*. Tech. rep. Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.

[102] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. "Exploiting punctuation semantics in continuous data streams." In: *IEEE Transactions on Knowledge and Data Engineering* 15.3 (2003), pp. 555–568.

[103] K. V. Vishwanath and N. Nagappan. "Characterizing cloud computing hardware reliability." In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 193–204.

[104] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. "Lineage stash: fault tolerance off the critical path." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM. 2019, pp. 338–352.

[105] Y. Wang et al. "Stream processing systems benchmark: Streambench." In: (2016).

[106] *Yahoo! Streaming benchmark*. https://github.com/yahoo/streaming-benchmarks. 2017.

[107] G. Yuan. "Scalable Fault Tolerance for High-Performance Streaming Dataflow." Doctoral dissertation. Massachusetts Institute of Technology, 2019.

[108] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[109] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[110] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. "Discretized streams: Fault-tolerant streaming computation at scale." In: *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM. 2013, pp. 423–438.

[111] S. Zeuch, A. Chaudhary, B. Del Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl. "The NebulaStream Platform: Data and application management for the internet of things." In: *arXiv preprint arXiv:1910.07867* (2019).

[112] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. "A hybrid approach to high availability in stream processing systems." In: *2010 IEEE 30th International Conference on Distributed Computing Systems*. IEEE. 2010, pp. 138–148.