Benchmarking Checkpoint-based Fault Tolerance Algorithms in Stateful Stream Processing

Marc Zwart
Delft University of Technology
Delft, Netherlands
m.d.zwart@tudelft.nl

Marios Fragkoulis
Delft University of Technology
Delft, Netherlands
m.fragkoulis@tudelft.nl

Asterios Katsifodimos
Delft University of Technology
Delft, Netherlands
a.katsifodimos@tudelft.nl

ABSTRACT

Major advances in the fault tolerance of distributed stream processing systems provided the systems with the capacity to produce strictly consistent results under failures. Consistent fault tolerance has been one of the catalysts fueling the maturity of streaming systems and boosting their widespread adoption not only for analytics use cases, but also as a platform for running applications, such as microservices and stateful functions. The most common fault tolerance strategy that many modern streaming systems adopt is an adaptation of Chandy-Lamport's distributed snapshots protocol that is based on global coordinated checkpoints. Given the importance of fault tolerance and the impact of coordinated checkpoints in the stream processing space, it is surprising that no other fault tolerance algorithm has been considered as an alternative.

In this paper we bring together and benchmark three seminal fault tolerance algorithms from the distributed systems literature: the coordinated, uncoordinated, and communication-induced checkpoint algorithms. We evaluate their behavior in terms of runtime performance and failure recovery on a variety of streaming workloads including pipeline, scatter-gather, and cyclical topologies. We benchmark the algorithms on a novel streaming system, FERDiS, built from scratch as an extensible benchmarking framework.

The experiment results show that the coordinated approach that state of the art streaming systems adopt is in many cases optimal, but with some exceptions where it is outperformed in both runtime performance and failure recovery by the uncoordinated and communication-induced algorithms. The cause of the differences in performance, both at runtime and during recovery, was found to be strongly related to the characteristics of the input stream(s) and the query graph. On the tested cyclic topology, the coordinated checkpointing algorithm could not be applied as by its design it will deadlock on such query graphs. The natural cycle support of the uncoordinated and communication -induced proved advantageous here and we observed high susceptibility to the *unbounded domino effect* of the uncoordinated algorithm which the communication-induced was not affected by, indicating the unsuitability of uncoordinated checkpointing in cyclic stream processing.

1 INTRODUCTION

In the early days of stream processing, systems could only produce approximate results [1, 6, 17] forcing applications to introduce extra supporting patterns and systems in order to secure the correctness of results. One notable such example is the Lambda architecture [33]. The last decade saw a steep maturity in the fault tolerance of streaming systems with systems such as Apache Flink [14], Google Millwheel [3], SEEP [21], IBM Streams [29], Hazelcast Jet [24], and Microsoft Trill [16], achieving exactly-once processing guarantees [44] despite failures. This advancement in fault tolerance was critical in opening the way for new applications that could now trust the correctness of streaming systems. In fact, many novel stream processing applications depart from the typical analytics use cases and involve the serving of machine learning model pipelines [13] as well as the execution of web and cloud services, such as microservices [31] and stateful functions [2]. Fault tolerance is even more important for such applications, which have strict operational requirements.

The fault tolerance approach of many modern streaming systems, such as Flink, Streams, Trill, and Jet, converges towards an adaptation of Chandy-Lamport's distributed snapshots protocol [18] introduced by Carbone et al [12]. The distributed snapshots protocol is founded on periodic, global coordinated checkpoints that capture a consistent snapshot of the system's distributed state via special markers that flow through the data stream causing the system's operators to checkpoint their state using two-phase commit. On fault recovery, all operators of the system roll back to the latest snapshot of their state and resume input processing from the correct offset that is also tracked and restored.

Despite the success of porting the traditional protocol of Chandy-Lamport from the distributed systems literature in the stream processing domain, there is no evidence regarding the appropriateness and performance of other important checkpoint-based fault tolerance algorithms, namely the uncoordinated checkpoint and communication-induced checkpoint algorithms. In addition, the emergence of novel stream processing use cases may pose a new set of challenges for fault tolerance that can potentially be better addressed by alternative strategies.

In this paper we benchmark three fault tolerance algorithms for distributed stream processing based on coordinated checkpoints, uncoordinated checkpoints, and communication-induced checkpoints. We stress the algorithms on a variety of workloads including pipeline, scatter-gather, and cyclical topologies and measure a descriptive set of metrics on performance and recovery experiment configurations. We run the experiments on FERDiS (Framework for Experimental Recoverable Distributed Streaming), an extensible framework that we introduce for benchmarking streaming system aspects. FERDiS is an important engineering effort within our team encapsulating all the main components of a streaming system, such as operators, checkpointing, networking, IO, and serialization, without the particular optimizations and sophisticated feature set of full-blown streaming systems that can affect benchmarking. FERDiS is available for use and extension as open-source software.[1]

In short our work makes the following contributions to the state of the art.

[1] https://github.com/delftdata/FERDiS

- implementation of a prototype streaming system that can be leveraged as an extensible benchmarking framework for experimentation
- implementation of three fault tolerance approaches based on coordinated checkpoints, uncoordinated checkpoints, and communication-induced checkpoints in distributed stream processing
- benchmarking of the three fault tolerance algorithms in a total of 116 configurations spread over six streaming workloads which includes a cyclic query

The rest of the paper is structured as follows. In section 3 we discuss related work and in section 4 we provide necessary background knowledge with respect to our work. In section 5 we present the benchmarked fault tolerance approaches. In section 7 we describe the benchmarking system and section 8 elaborates the experiment setup. Then, section 9 presents the experiments and discusses the experiment results. Finally, we present future work plans in section 10 and conclude the paper in section 11.

2 RESEARCH QUESTIONS

Our work proposes to address the following research questions with respect to fault tolerance algorithms in distributed stream processing.

RQ1. *How can a stream processing system support each of the three classes of checkpoint-based fault tolerance algorithms: coordinated checkpointing, uncoordinated checkpointing, and communication-induced checkpointing?*

RQ1.1. *Which specific variants of the three classes of checkpoint-based fault tolerance algorithms are suitable to implement for stream processing?*

RQ2. *How does the choice of checkpoint-based recovery algorithm affect the*

RQ2.1. *consistency guarantees of a streaming system?*

RQ2.2. *runtime efficiency of a streaming system?*

RQ2.3. *recovery efficiency of the SPE?*

RQ2.4. *support of cyclic dataflows?*

3 RELATED WORK

Related to our work are stream processing benchmarks proposing a benchmarking system (subsection 3.1) and experimental evaluations of fault tolerance in the stream processing domain (subsection 3.2).

3.1 Stream Processing System Benchmarks

Several benchmarks have been published tailored to streaming systems. A recent survey details the benchmarks that are currently available for evaluating different layers of big data analytics infrastructures [32] including, but not limited to streaming systems.

Linear Road [7] is a benchmark which simulates a traffic monitoring application with definitions of input streams, queries presented in an abstract, language agnostic way, and a result validator which evaluates the correctness of results. Latency, throughput, and accuracy are the metrics considered. **RioTBench [43]** is a real-time analytics benchmark tailored to Internet of Things (IoT) data, which contains 27 microbenchmarks and measures latency, throughput, memory utilization and cpu utilization. **CityBench [4]** is another real-time analytics benchmark, which defines 13 queries over one or more of four data streams containing parking data, road congestion data, cultural event data and weather data of a real city. It measures latency, memory consumption and completeness of query results. **SparkBench [36]** is a benchmark tailored to Apache Spark and defines 10 workloads in 4 different categories of streaming applications. The metrics that are evaluated are CPU, memory, disk I/O, network I/O, job execution time, and throughput. The **NEXMark [47]** benchmark defines a comprehensible set of eight queries over streams containing auction data. NEXMark has been implemented for Apache Spark and Apache Flink[2], and has been adopted in Apache Beam.[3] NEXMark does not define any metrics.

3.2 Experimental evaluations of fault tolerance

Matar et al. [38] have compared Apache Spark and Apache Flink. Two types of faults are considered, transient faults such as network failures and permanent faults like a power/hardware failure in a node. The work evaluates the typical throughput and latency metrics, resource utilisation and data generation rate. Another experimental evaluation, which compares Samza, Spark, Hadoop, Kafka and Storm [40], considers faults in only one workload, which only consumes input and performs no operations on it. A similar evaluation [37] includes seven workloads instead of one. Both works present the same new metrics: thoughput penalty factor and latency penalty factor, which are based on expected throughput/latency vs actual throughput/latency under failure. This requires running a benchmark twice, once failure-free and once with failures to compare performance. Both are very coarse metrics.

Research Gap. There is no standardized and publicly available benchmark definition to evaluate fault-tolerant stream processing with fault injection. Nor has any work attempted to isolate the fault-tolerance mechanism from the rest of the system, thereby making existing experiments unsuitable to draw conclusions on fault-tolerance performance. The state of the art lacks an evaluation of classical fault-tolerance mechanisms in a distributed stream processing system. To the best of our knowledge, there is no practical evaluation of CIC protocols in an SPE context. One reason may be that acyclic workflows was the norm [22]. However with the recent interest in cyclic dataflow graphs, communication-induced checkpointing becomes a new venue to explore for fault tolerance in stream processing. In fact, with the advances in hardware over the past decades, message passing overhead may become affordable with modern hardware [20], making CIC protocols more interesting than they used to be.

4 PRELIMINARIES

This section provides background knowledge on the streaming model of computation and rollback recovery.

[2]https://github.com/nexmark/nexmark
[3]https://beam.apache.org/documentation/sdks/java/testing/nexmark/

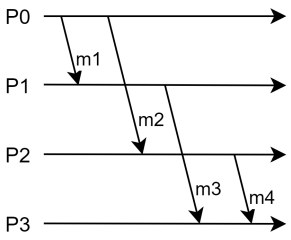
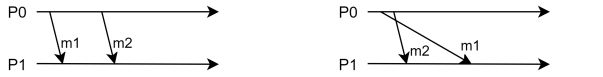


Figure 1: An example message pattern



(a) A message pattern with FIFO channel **(b) A message pattern without FIFO channel**

Figure 2: two message patterns each with two processes sending two messages over different channels

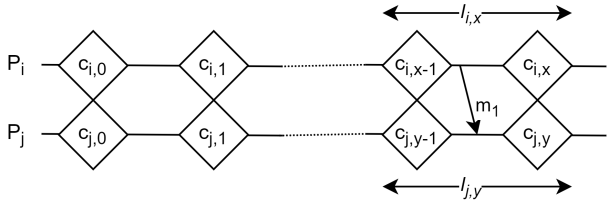


Figure 3: Checkpoint index and checkpoint interval

4.1 Streaming Model of Computation

A stream processing system can be modelled as a message passing system [20], where messages are tuples in streams. In this model processes communicate solely by sending and receiving messages over (un)reliable channels to perform a distributed computation. Processes are represented as horizontal lines and messages between them as diagonal lines. Figure 1 provides an example visualisation of this model containing four processes.

4.2 Checkpoint-based Rollback Recovery

Rollback-recovery protocols can be classified into *checkpoint-based* and *log-based* protocols. Checkpoint-based rollback recovery relies solely on state snapshots called *checkpoints* to restore system state and comes in three forms: coordinated, uncoordinated and communication-induced. Log-based protocols build on checkpoint-based protocols by logging non-deterministic events in an optimistic, pessimistic or causal fashion [20]. The protocols provide different processing semantics with varying levels of consistency. Rollback recovery protocols treat each process computation as a sequence of state intervals which can be recovered if there is stable recovery information available. On the boundaries of state intervals there are checkpoints. Each checkpoint has a *checkpoint index* and two checkpoints are separated by a *checkpoint interval*; these concepts are visualised in Figure 3.

There are two main type of events in message passing, *send* events and *delivery* events. This separation exists because there is latency between sending a message from process A and its delivery to process B. However, delivery can also be postponed by the receiving process. While a message has been sent, but not yet delivered, it is called *in-transit*. To reason about the order of events the popular *happened before* [35] ($\rightarrow$) relationship is applicable as long as message order is guaranteed. Some derivations using this relationship are depicted in Figure 2a: $send(m1) \rightarrow send(m2)$, $send(m2) \rightarrow deliver(m2)$ and by transitivity: $send(m1) \rightarrow deliver(m2)$. Without first in first out (FIFO) channels, ordering cannot be guaranteed by the network as visualised in Figure 2b and must be enforced by the receiving application before delivering.

Rollback-recovery protocols generally assume that network partitions do not occur to simplify protocol design [26]. Popular protocols such as Chandy-Lamport's distributed snapshots protocol [18] assume reliable FIFO channels while others assume that messages can be lost, re-ordered and even duplicated. It is worth noting that it is possible to utilize a reliable communication protocol, such as TCP, on an unreliable communication channel, effectively transforming it to a reliable FIFO communication channel.

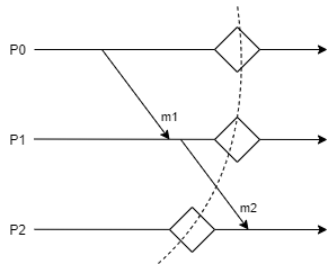
Processing Semantics convey a system's guarantees on data processing considering failures. Modern stream processing systems are able to produce correct results while the system faces failures, however the level of correctness may vary. The processing semantics terminology adopted in the stream processing literature includes *at-most-once*, *at-least-once* and *exactly-once* processing [22]. Under at-most-once processing, a system facing failures will lose input and as such it will process an incoming message either never or once, but never more than once. At-least-once processing ensures that arriving messages are processed one or more times, meaning that under failure the system can record the same changes to the state more than once, but it will never lose input. Lastly, exactly once processing ensures that each message is processed once, never less- and never more than once, resulting in no lost input nor any duplicate state changes. To achieve each of these semantics, different trade-offs must be made between performance at runtime or recovery time. An important observation is that for exactly-once processing there are two possible interpretations [22], exactly-once processing on *state* or on *output*. As many stream processing engines opt to implement exactly-once processing on state they still produce duplicate output on fault recovery. To understand how these semantics emerge we must take a deeper look at the notion of consistency in distributed systems.

Consistency is a property of a distributed system's state. A global state of a distributed system is a collection of states, one belonging to each participating process and optionally the states of the communication channels. While no failures occur in such a system, the global state is trivially consistent. *Consistent* meaning that no process's state reflects the reception of a message which does not reflect in the sender's state [20]. If a process's state does reflect a message receipt which does not reflect in the sender's state, that message is called an *orphan message* [11]. A global state containing orphan messages is called an *inconsistent* global state. If a process's state reflects a send, but the message reception is not present in the recipient's state, this is not considered inconsistent

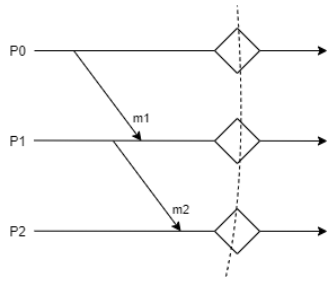
as the message would still be *in-transit*, also referred to as 'part of the *channel state*'.

In Figure 4 examples are given of a consistent and inconsistent global state in a distributed system containing three processes. A key observation is that an orphan message implies a happened-before relationship between the checkpoints of the sender and receiver. An example can be found in Figure 4d between the checkpoints of P1 and P2. From this example it becomes clear that a happened-before relationship between two checkpoints guarantees that the checkpoints are inconsistent with each other and any global state they may belong to. This observation forms the basis for the Recovery Line algorithms that will be introduced in subsection 5.4.

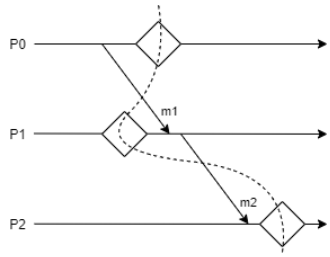
Notable Observations. Processing semantics emerge in a few different ways in distributed systems that employ rollback recovery. At-most-once processing can be guaranteed when a consistent global state is restored, but the messages that are part of the channel state are not replayed. This is also known as *gap recovery*. It is worth noting that when checkpoints are aligned, as in Figure 4b, which means that there is no channel state, then no messages are dropped and *exactly-once* processing can be guaranteed. When a consistent global state is restored and channel state is replayed, exactly-once processing can also be guaranteed as long as message deduplication is applied. When deduplication is not applied, messages may be delivered twice as a side-effect of message replay: the message will have been sent once prior to failure and once during replay. This results in *at-least-once* processing semantics, which naturally introduces inconsistency in the global system state. This is so because the messages will have been sent only once according to the system's state, but will have been received twice, making one of the duplicate receptions an orphan.



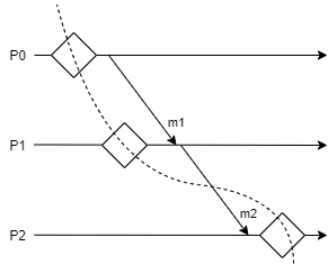
(a) A consistent global state with channel state (m2)



(b) A consistent global state without channel state



(c) An inconsistent global state with channel state (m1)



(d) An inconsistent global state without channel state

Figure 4: Examples of consistent and inconsistent global states in a distributed system containing three processes, diamonds indicate checkpoints

4.3 Log-Based Rollback Recovery

Log-based recovery has been extensively applied in stream processing, especially in its early days, under the term *upstream backup* [8, 19]. In upstream backup upstream operators keep a log of their sent messages and upon a downstream failure they replay the log. Stateful operators, other than windowed operators, may have their state depend on all tuples that have been processed thus far. For such stateful operators upstream operators are required to be able to replay all tuples, resulting in an endlessly growing log. This is very burdensome in stream processing systems due to the large volume of incoming tuples. Checkpointing was added to the fault tolerance protocol to mitigate this issue. After taking a checkpoint an operator can notify upstream operators which tuples have been persisted as part of a checkpoint, which allows upstream operators to remove (or *prune*) tuples from their log.

Relevance to Checkpoint-Based Algorithms. Log-based recovery can also be viewed as an enhancement of checkpoint-based recovery [46]. By keeping a message log besides checkpointing state, deduplication of already processed tuples downstream of a failure is feasible for deterministic computations. This approach can provide exactly-once processing semantics on output with the exception of failed sinks that have no downstream to enforce output deduplication. Recent stream processing systems tend to opt out of keeping a message log and only perform state checkpointing [22]. Message replay functionality is usually provided by message brokers such as Apache Kafka [45] that persistently store the input stream(s) allowing the input stream to be replayed without adding extra functionality to the stream processing system itself. If a streaming system were to implement this themselves, ensuring correct event order during replay for nondeterministic computations requires logging determinants [20, 44].

By keeping a message log, similarly to the upstream backup strategy, uncoordinated and communication-induced checkpointing approaches allow replaying lost channel state post-failure. Interestingly, this allows the checkpointing strategy to support exactly-once processing under the assumption that the message log will be available post-failure and deduplication is applied. Otherwise, if no deduplication is applied, at-least-once processing semantics is supported. To make a message log available post-failure it needs to be stored on a persistent local or remote disk, which introduces runtime overhead in the form of writing log entries to disk. Alternatively with a volatile message log, the impact on the runtime performance of a stream processor can be minimal, but as a trade-off only non-failed operators will be able to replay their message logs resulting in at-most-once processing semantics with a reduced amount of messages being lost as a result of failure. Using sequence numbers required for log replay, it can be determined how many messages were lost to quantify the amount of lost input, but with the added benefit of better runtime performance and not requiring twice the amount of computing resources.

5 FAULT TOLERANCE ALGORITHMS

This section presents the workings of coordinated, uncoordinated, and communication-induced checkpoint-based fault tolerance algorithms and the guarantees they provide regarding fault recovery.

5.1 Coordinated Checkpointing

In coordinated checkpointing participating processes synchronize and typically stop working in order to checkpoint their state at a moment when there is no channel state as Figure 4b shows. The individual checkpoints taken by the whole system using coordinated checkpointing always form a new most recent global state, this is referred to as a *recovery line*.

The most well-documented and implemented protocol in this class is Chandy-Lamport's marker-based algorithm [18], which has been adapted for acylic dataflow graphs [12]. Under the assumption of an acyclic operator graph checkpoint markers flow though the system and instruct operators to checkpoint their state.As long as each operator will not forward a marker until it has received one marker from each of its upstream operators, a consistent global checkpoint is taken in which there is no channel state. This algorithm guarantees exactly-once processing at the sacrifice of a blocking operation while waiting for all markers to arrive from all upstream operators. In Figure 5 an example execution of this algorithm is presented. A possible message pattern emerging from taking a coordinated checkpoint is visualised in Figure 6. Note how m_5's delivery is delayed because the channel from P_1 is blocked until a marker has been received from all upstream operators (P_1 and P_2).

A variant of this algorithm does not block input channels in order to improve performance at runtime [20]. As a result an operator can process messages from upstream operators which lay behind the marker before taking a checkpoint itself. Therefore inconsistency may be present in the global checkpoint as a downstream operator's checkpoint can now reflect the reception of a message which does not reflect being sent in the upstream operator's checkpoint. As a result this variant relaxes the exactly-once processing guarantee to at-least-once and the default recovery line algorithms (subsection 5.4) may not be applicable as they compute consistent global states.

While minimizing useless checkpoints and easing garbage collection, this approach either imposes overhead at runtime by blocking channels or guarantees lesser processing guarantees by not blocking channels. More importantly, without extending this approach with special constructs such as message logging, cyclic message patterns are not supported. In a cyclic message pattern at least one operator has an input channel that depends on one of its own output channels. Since an operator only advances checkpoint markers when it has received a marker on all of its input channels, but one of those channels requires a marker being sent out over its own output channel, then it will never receive a marker on that channel and as such, the checkpointing protocol causes a deadlock.

5.2 Uncoordinated Checkpointing

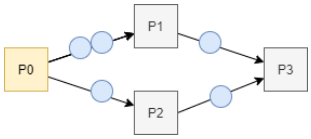
In uncoordinated checkpointing participating processes take checkpoints at will, without inter-process coordination. While this approach requires no communication or synchronization, there is a big drawback when facing cyclic communication patterns: the approach is susceptible to the *unbounded domino effect* [41] during the recovery phase. The unbounded domino effect is caused by every checkpoint having a dependency on an upstream checkpoint, thus potentially never advancing the recovery line due to the cyclic communication pattern. This results in the distributed system rolling back to its initial state, rendering many or all the taken checkpoints useless. In Figure 7 a cyclic message pattern is depicted in which the domino effect will manifest when trying to find the recovery line. As P_1 fails, it could roll back to $c_{1,2}$, however then P_2 would be inconsistent, so P_2 must roll back to $c_{2,1}$ which in turn requires P_0 to roll back to $c_{0,0}$ from which P_1 must roll back to $c_{1,0}$ and P_2 to $c_{2,0}$, making the recovery line the initial system state and all other checkpoints useless. This is the easiest type of checkpointing algorithm at a conceptual level, but introduces a optional new type of effort that is not part of the protocol specification which is largely related to the choice of when to checkpoint being deferred to the application itself, which then needs to define its own checkpointing decision-making process. This adds the advantage of a process being able to take a checkpoint when its state is small [5].

This type of checkpointing requires extra communication either at runtime or during recovery to determine which checkpoint dependencies exist among the taken checkpoints [20].

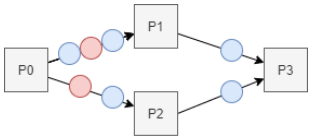
5.3 Communication-Induced Checkpointing (CIC)

Communication-induced checkpointing loosely coordinates checkpointing by piggybacking checkpoint-related metadata on application messages. There are two types of checkpoints, *local* checkpoints which are equivalent to uncoordinated checkpoints and *forced* checkpoints [23] which the algorithm inserts to prevent the domino effect. In addition, the communication-induced checkpoint algorithm comes in two flavours, model- and index-based, which have been proven to be equivalent [20].

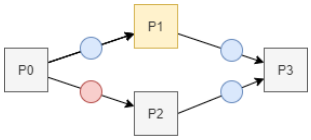
This class of checkpointing protocols is tightly coupled with Z-path/-cycle theory [20]. A Z-path is a special sequence of messages



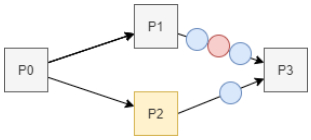
(a) P0 initiates by taking a checkpoint



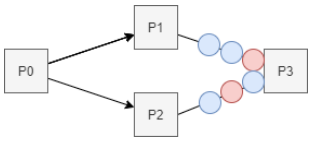
(b) P0 forwards markers downstream and resumes processing



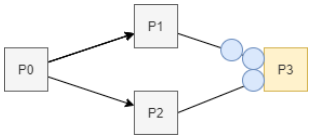
(c) P1 received a marker from all its upstream operators and takes a checkpoint



(d) P1 forwards the marker and resumes, P2 received markers and takes a checkpoint



(e) P2 forwards the marker and resumes, P3 blocks its channel to P1 until P2's marker is received



(f) P3 received both markers and takes a checkpoint, after which it can unblock its input channels and resume

Figure 5: Example execution of Chandy-Lamport's marker based coordinated checkpointing protocol. Boxes are operators, grey are computing, yellow are checkpointing. Circles are messages, blue are application messages, red are markers

connecting two checkpoints and a Z-cycle is a special sequence of messages connecting a checkpoint to itself. Given that it has been proven that a checkpoint lying on a Z-cycle will never be part of a recovery line and is therefore useless, the CIC algorithm's primary goal is to detect Z-cycles and break them by forcing a checkpoint before processing a message that will cause a Z-cycle to emerge.

In Figure 7 a Z-path is created by messages $[m_2, m_3]$ connecting checkpoint $c_{0,1}$ to the state of process P_2. A Z-cycle connects $c_{0,1}$

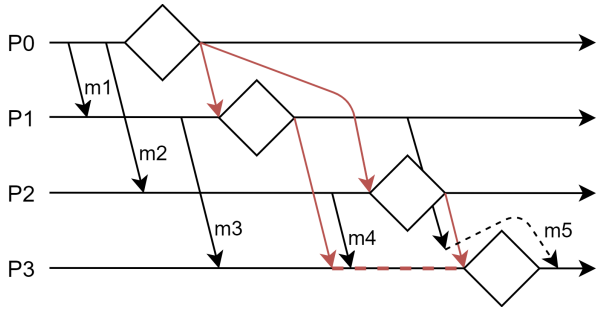


Figure 6: Message pattern resulting from taking a coordinated checkpoint with blocking channels, red lines indicate marker messages

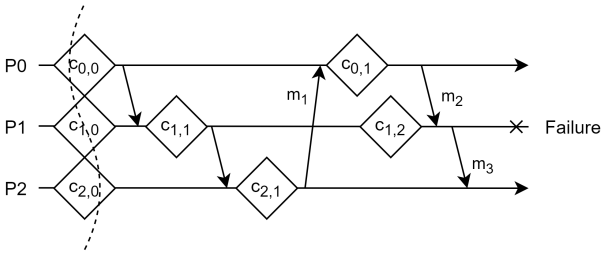


Figure 7: A message pattern which is susceptible to the domino effect

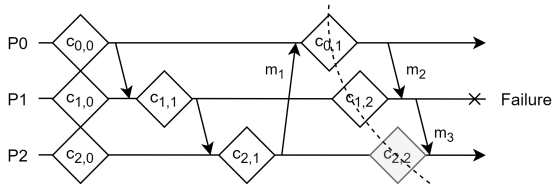


Figure 8: A message pattern where the domino effect was prevented by forcing $c_{2,2}$

to itself via messages $[m_2, m_3, m_1]$. When m_3 is delivered, the Z-cycle is formed and $c_{0,1}$ will be useless. Therefore, CIC algorithms will force a checkpoint before the cycle is formed and another checkpoint before the delivery of m_3. As shown in Figure 8, forcing a checkpoint in P_2 before the delivery of m_3 breaks the z-cycle and thus advances the recovery line and in doing so, avoids the possibility of the domino effect happening during recovery.

One CIC algorithm was presented by Gupta et al. [26]. The work does not provide any benchmarked implementation to evaluate its effectiveness or give insight in the imposed run- and recovery-time overhead of the algorithm. The authors later published an updated manuscript including comparisons with other similar algorithms [27], which shows the hypothetical effectiveness of their approach, without practical evaluation. Other proposed algorithms do not extend beyond this level of evaluation [20] but do adopt higher levels of sophistication, such as the HMNR protocol [28]. A

real practical analysis [5] concludes that insight in the communication pattern is required when devising efficient CIC protocols, which is usually known in streaming systems.

CIC protocols support cyclic communication patterns without risk of the domino effect but may take many checkpoints, potentially causing a lot of overhead [5]. This work observed that the local checkpointing policy that is part of a CIC protocol, and is usually timer-based, should not just consider when the last *local* checkpoint was taken but also when the last *forced* checkpoint was taken. The observation was that CIC protocols frequently took local checkpoints shortly after a checkpoint was forced, which in turn caused other processes to force checkpoints, resulting in "flurries" of checkpoints happening in the system. Eventually this would result in CIC protocols taking significantly more checkpoints than procotols that did not force checkpoints. The work suggests that to mitigate this behavior, CIC protocols should be implemented so that they reset their local checkpoint timer when a *local or forced* checkpoint is taken.

5.4 Recovery line algorithms

Once a distributed system that takes checkpoints runs into a failure, the checkpoint-based recovery protocol enters the roll-back phase in which it must determine which checkpoints to recover. The most recent consistent global state is called the *recovery line*, rolling the system back to the recovery line minimizes the amount of computation that gets lost. Checkpoints that will never be part of a recovery line are called *useless*.

There are two theories proposed in literature on recovery line calculation. The first employs a *rollback dependency graph* [20], in which nodes are checkpoints and a directed edge is drawn from $c_{i,x}$ to $c_{j,y}$ if either:

(1) $i \neq j$, and a message m is sent from $I_{i,x}$ and received in $I_{j,y}$, or
(2) $i = j$, and $y = x + 1$

If a checkpoint $c_{i,x}$ must be rolled back, all checkpoints reachable from $c_{i,x}$ must also be rolled back, by applying reachability analysis on this graph the recovery line can be found.

A second approach is employs a *checkpoint-graph* [20] in which nodes are again checkpoints and a directed edge is drawn from $c_{i,x}$ to $c_{j,y}$ if either:

(1) $i \neq j$, and a message m is sent from $I_{i,x-1}$ and received in $I_{j,y}$, or
(2) $i = j$, and $y = x + 1$

In this more intuitive graph an edge from $c_{i,x}$ to $c_{j,y}$ indicates at least one orphan message that was not sent when $c_{i,x}$ was taken but was received when $c_{j,y}$ was taken. To find a recovery line without inconsistency the *rollback propagation algorithm* can be applied [20]. These two graphs are equivalent in that they always produce the same (consistent) recovery line although the *checkpoint-graph* seems to be the most intuitive of the two. Moreover, the checkpoint-graph can be extended with *virtual checkpoints*, these represent the state that still exists within an operator. To insert these in the checkpoint-graph, knowledge of the inter-process communication pattern must be known, since this is typically known at runtime in an SPE due to their mostly fixed dataflow graphs this can easily be inferred without extra overhead. By considering virtual checkpoints as part of the recovery line calculation it becomes possible to preserve existing state in operators, resulting in less processes rolling back in face of failures.

6 BENCHMARKING FRAMEWORK REQUIREMENTS

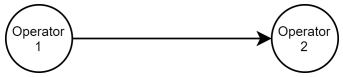
A distributed, stateful, and fault-tolerant stream processing system built as an extensible benchmarking framework poses stream processing, state management, fault model, fault tolerance, performance, experimental system, distributed execution, and infrastructure requirements.

Stream Processing System Requirements. The benchmarking framework should support the basic dataflow operators, such as map, filter, aggregate and join. To support the latter two, a time domain and matching window implementation(s) need to be provided. The minimal viable approach to achieve this is to implement the processing time domain in which out of order data gets reprocessed as part of new windows rather than updating older windows. While this reduces practical usability of the stream processing system as opposed to implementing event time, it does not reduce the experimental relevance of the system. Second, A key computational characteristic in stream processing is data parallelism or *sharding*. To present realistic query networks, the system should support sharding and the inherent communication patterns between sharded operators: *shuffle* and *pipeline*. These communication patterns are visualised in Figure 9. Third, to support correct recovery from failures, a replayable input source must be provided. This can be a dependency on an external system such as Apache Kafka, while preserving offsets in source operators as part of checkpoints. Lastly, to support user defined functions the system should have a user-facing API to implement the different operators and define query networks with them.

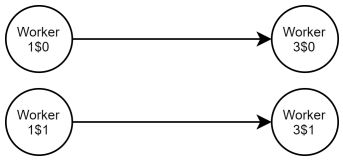
State Management Requirements. State management in a stream processing system can be performed in different ways, in main memory, on a local disk or in a remote store. Main memory state consists of an application's memory page or a dedicated main memory data structure store like Redis [4]. Local disk storage is typically a file system on a disk device that is connected to the same motherboard as the CPU on which the application runs. Lastly a remote store is a cluster of 1 or more machines providing a network-accessible store. This architecture trades read/write speed for tolerance to failures the further away the storage medium is. Main memory being high performance but low reliability, local disk being a bit of both and a remote store being low performance but higher reliability.

When considering the high-performance requirements this class of systems face, it makes sense to keep the data as close to the CPU as possible. In existing stream processing systems ephemeral state is most prevalent [22], which resides in main memory and CPU caches. The second most prevalent is embedded state where part of the state lives in main memory and part on a local disk. Storing data even further from disk (remote store) imposes significant overhead on the system and is, as expected, the least prevalent medium to store application state in practice.

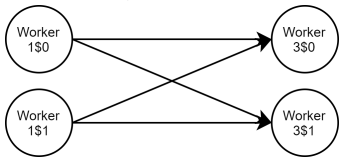
[4] https://redis.io/



(a) Two operators with a directed connection



(b) Two operators materialized with two shards in a pipeline configuration



(c) Two operators materialized with two shards in a shuffle configuration

Figure 9: An example of an operator connection and two possible materializations with different connections

There are two types of state that are present in stream processing systems: keyed state and operator state. Keyed streams are used to achieve data parallelism by using keys on stream elements to do partitioning. Keyed state is an extract from a keyed stream and resides in windows. Operator state is part of a user defined operator and is free to be shapen by the developer. For this work both state types are relevant to support. Keyed state is a hard requirement to support windows with fault tolerance. Operator state is not a hard requirement but nevertheless a *should have* requirement.

Fault Model Requirements. In a distributed system different types of faults can occur. These have been modelled into three primary fault models: Byzantine, Fail-stop and Fail-stutter [46]. The Byzantine fault model is the broadest of these three, where the model allows failed nodes to continue interacting with the system, failures can be arbitrary and importantly, failures cannot be observed by correctly operating nodes. The Fail-stop fault model is more simplistic. Any node may fail at any time and when it does it will stop producing output. In this model failures can be observed from other nodes. More subtle faults which do not cause a stop but hamper a program's ability to progress are not included in this model.

The Fail-stutter fault model stands in a middle ground between the Byzantine and Fail-stop models. It extends the Fail-stop model to include *performance faults*, which materialise as a correctly operating node with unexpectedly low performance. In stream processing nodes encountering a performance fault are known as *stragglers* [30]. How to optimally detect and migitate stragglers is an open research problem. As stragglers require no rollback recovery and neither do Byzantine faults, the Fail-stop model is the most fitting fault model within the scope of this work.

Fault Tolerance Requirements. Fault tolerance in stream processing systems can be achieved in three main ways, checkpointing, logging and lineage. As the existing body of stream processing systems is converging on checkpoint-based approaches and with the research goal in mind, this experimental system should provide fault tolerance through checkpointing.

The *creation of checkpoints* lies at the heart of checkpoint-based fault-tolerance. Ephemeral state requires checkpointing the application's main memory page, or at least, the stateful part of it. Main memory checkpoints can be taken from different levels, the underlying hardware [34], the operating system [25] or as part of the application itself, i.e. application level checkpointing [10]. Hardware checkpoints rely on checkpointing functionality provided by *checkpointable hardware*, which can take instructions to write main memory to a persistent storage and restore a checkpoint to main memory. This is a fine-grained type of checkpoint that can restore a process to resume at the exact instruction the checkpoint was taken. The operating system can provide checkpointing functionality at the kernel level by employing a kernel thread which can take a checkpoint of a process' memory page. This checkpoint can be restored by overwriting a process' memory page with the contents of the checkpoint. The last type of checkpointing happens at application level. Taking a checkpoint using this approach involves serializing and deserializing object values that form the application's state. These values can be stored and upon retrieval be deserialized and restored to the application state.

There are two *types of checkpoints* to choose from, the first and simplest one being a direct copy of the state. The second type is called incremental, which takes delta checkpoints, containing only the differences in state since last checkpoint, effectively reducing the size of the checkpoints. Incremental checkpoints must be supported by the state backend as the notion of a 'difference in state' must be defined by said backend. This notion can range from address values at kernel level till added/removed elements in a list at application level.

Storing checkpoints is most commonly done on a remote, resilient medium in existing stream processing systems [22]. This gives greater flexibility in terms of process relocation compared to local storage, which may no longer be available after a failure, for example when a process is restarted on a different machine after a failure. The experimental stream processing system should also store the taken checkpoints on a remote resilient store.

Given the observation that the recovery prodecure is a complex part of the implementation, a single recovery procedure that works for all checkpointing protocols would be a *could have* requirement. Based on the recovery line theory, a single protocol can be devised that can be used for recovery while being agnostic to the checkpointing protocol that was used during failure-free operation. This separation would decouple the checkpointing algorithm and recovery algorithm, which in turn creates opportunity to allow the configuration of checkpoint algorithm separately from the configuration of the recovery algorithm.

Performance Requirements. While high performance is not the primary goal of this research, it is worth considering it in the requirement analysis to ensure reasonable performance to make the stream processing engine comparable to existing frameworks. In distributed computing, (de)serialization has been observed to be a major performance concern [39], only seconded by load imbalance. As load balancing is not within the scope of this work, an as

efficient as possible serialization method should be applied in the implementation of the experimental stream processing engine.

Experimental distributed system requirements. To manually configure and run a distributed program on a cluster of machines can be a time consuming endeavour, particularly when considering frequently changing computations and configurations in an experimental setup. To allow rapid execution of experiments, the system should support an easy way to define, spin-up and teardown a distributed computation in an existing environment. This involves starting up the correct number of processes with unique identities and automatically establishing the correct connections between the processes to build the query network. The public API of the experimental stream processing system must be extended with the ability to configure which checkpointing algorithm to use at runtime.

Starting and stopping a distributed computation requires some form of coordination. The scope of this work also adds checkpointing and rollback-recovery as coordination requirements. Management of the execution of a distributed program can be done via either a centralised manager process, using a leader election protocol to elect a worker to act as both worker and manager or by sharing manager responsibilities among workers and negotiating manager actions through a consensus protocol. As workers can fail, the latter two options become significantly more complex due to re-election or re-negotiation steps required after a failure. A centralised manager process is significantly less complex to implement and as failures in such a component are not in the scope of this thesis, this approach is most desirable. This centralized manager can carry all the aforementioned responsibilities of starting and stopping computation, while also coordinating checkpointing and rollback recovery where necessary.

Another important aspect of the execution is the type of computation, specifically whether it is deterministic or non-deterministic. Deterministic programs will produce exactly the same output every time they are executed with the same arguments. Non-deterministic programs do not necessarily produce the same output every time. In rollback recovery, the nature of the program being executed is very important with respect to the guarantees that can be provided after recovery. If one or more operators in a dataflow graph behaves differently after a rollback, the computation will not converge towards the same state it was before the rollback. At-least-once processing can in such a case not be guaranteed as messages may not be sent again due to divergence from the pre-rollback state. To be able to reason about input processing, fault tolerance and processing semantics, the distributed computation must be deterministic in nature. Therefore for the remainder of this work any and all computations are assumed to be deterministic.

Infrastructure Requirements. Experiments using the benchmarking framework should be able to be executed on different cluster sizes supporting local and distributed execution and consisting of varying types of hardware. In addition, the framework should make it easy to spin up and tear down an environment. To satisfy this requirement some form of (non-)containerized automated infrastructure management software is required, such as Kubernetes[5], Red Hat's OpenShift[6], Consul[7], Rancher[8], or ZooKeeper.[9]

7 FERDIS: AN EXTENSIBLE BENCHMARKING FRAMEWORK

In subsection 7.1 we present the core system design and implementation aspects of FERDiS. In subsection 7.2 we elaborate separately the implementation of fault tolerance in FERDiS.

7.1 Core System Design & Implementation

FERDiS has been implemented using Microsoft's .NET stack combined with Microsoft's CRA software library [42], which is built on this same stack and integrates well with containerization technologies. These technologies provides the main functionality required for deploying a dataflow-style distributed system on a Kubernetes-managed cluster with restart and reconnect functionality out of the box. This alleviates a great deal of development effort on boilerplate functionality required for distributed computing. As a result of these technology choices, FERDiS has been developed in C# and is available as open-source software. Engineering FERDiS, which amounts to 20K lines of code, has been a large-scale effort within our team.

Message Processing. Each worker (which is an operator shard) consists of a message processor which consumes or generates input, processes it and dispatches it. Three interfaces were designed to adopt these responsibilities: *ISource* to generate or consume messages, *IPipeline* to process messages and *IDispatcher*. One message processor requires one or multiple *ISource* instances, an *IPipeline* and an *IDispatcher*.

ISource instances provide input messages for a processor through a *Take()* method. Their implementation can be backed by various sources, for example a Kafka source for a generating ISource or messages from an input stream for a consuming ISource. These sources can also be tied to local message generation logic, which is the case for some coordinator components where operator state transitions generate messages, this will be further introduced in subsubsection 7.2.4.

IPipeline instances perform processing on the content of messages, the interface allows the returning of zero to many messages, making it generic enough for any type of stream operator. The system currently has only one IPipeline implementation, which takes an ordered set of *IHandler* instances. An IHandler may check a message for payloads it can handle, an example being the OperatorHandler which checks for an *IEvent* payload and if present, hands it to the provided *IOperatorShell*. By taking an ordered set of these handlers, message handling logic can be configured by allowing the order of the handlers to be configured. The handlers are implemented in the Infrastructure module.

IDispatcher handles the dispatching of messages that were returned by the pipeline. To dispatch a message, it is serialized and assigned to a dispatch queue based on a partitioning defined by

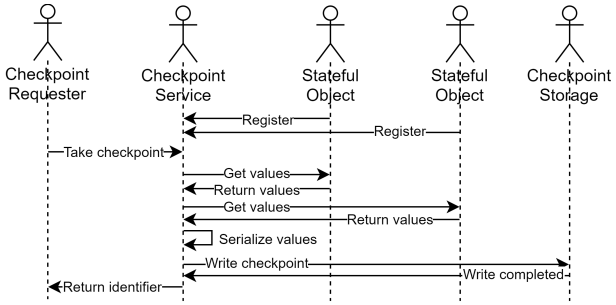
[5] https://kubernetes.io/
[6] https://www.openshift.com/
[7] https://www.consul.io/
[8] https://rancher.com/
[9] https://zookeeper.apache.org/

a provided *IPartitioner*. The current implementation has only one partitioner which performs hash partitioning on the message's partition key or broadcasts if the key is null.

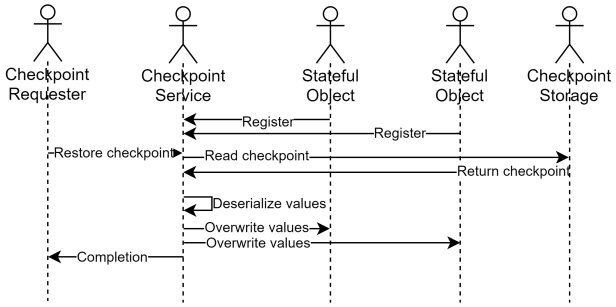
Network IO is required to support receiving and sending messages between operators, ie before and after a processor's window of responsibility. The *IInputEndpoint* and *IOutputEndpoint* interfaces were defined to handle network input and ouput. One endpoint considers the connection to one other vertex, meaning they must consider sharding. The endpoint types have *Ingress/Egress* methods respectively, which take streams and shard ids as arguments.

Input endpoints start a thread for each provided stream and attempt to read a message from it, deserialize it and hand it off to a provided *IReceiver*. The receiver collects messages from different input endpoints and queues them internally while exposing that queue via the *ISource* interface. The input endpoints and receiver are not necessarily required at runtime. An example where there are no input endpoints is in source operators, which receive input from elsewhere and therefore depend on an ISource implementation that depends on the input source (e.g. Kafka) and not on input endpoints.

Output endpoints start a thread for each provided stream, get a dispatch queue from the provided *IDispatcher* and whenever there is a message in the queue, write it to the stream. To improve performance, flushing the stream to the network happens after a configurable number of bytes has been written. Output endpoints and dispatchers are also not necessarily required at runtime. An example where no messages get dispatched is in sink operators where messages are written to an external system by a message handler in the pipeline.



(a) A procedure for taking a checkpoint



(b) A procedure for restoring a checkpoint

Figure 10: A high level overview of the procedures for taking and restoring checkpoints, object (hierarchies) are modelled as actors

7.2 Fault Tolerance Implementation

The implementation of fault tolerance is the core focus of this work. It regards the implementation of checkpoints (subsubsection 7.2.1), the three fault tolerance algorithms (subsubsection 7.2.2), the unification of the algorithms for easily configurable and extensible fault tolerance (subsubsection 7.2.3), the recovery orchestration (subsubsection 7.2.4), the recovery line calculation (subsubsection 7.2.5), the handling of channel state (subsubsection 7.2.6), and failure detection and message replay (subsubsection 7.2.8).

7.2.1 Checkpoints. FERDiS applies application-level checkpointing by presenting state as object properties. These property value(s) form the actual state, effectively making a checkpoint a collection of property values. The checkpoint is structured in such a way that it is derivable which property value belongs to which object. Finally, checkpoints are (de)serializable to support storage and restoration. FERDiS uses *Azure Storage*[10] as checkpoint store, which is both remote and resilient. Azure Storage provides two notable services, Tables and Blobs. Azure Storage Tables are already being used by the CRA software library. The Azure Storage Blobs[11] provide a scalable object store to which text or binary data can be written and read.

A high level overview of the procedure for taking and restoring checkpoints is presented in Figure 10. In this Figure, objects or object hierarchies are modelled as actors and can thus be interpreted as objects calling methods on one another. An important stage in this procedure is the registration phase which happens before any checkpoint is taken or restored. The CheckpointService can manipulate the state of objects in the process by maintaining references to these objects, which register themselves with the CheckpointService. Exactly the same types of stateful objects are registered with the CheckpointService after restarting a process, regardless if the restart happens on the same machine or not. Otherwise, missing objects may be encountered, which were part of a previous checkpoint or there could be new objects that are not part of that checkpoint. This would break the reliability of this checkpointing procedure. The registration phase is transparently implemented through the object initialisation hooks provided by the Dependency Injection framework AutoFac [12].

7.2.2 Fault Tolerance Algorithms. One implementation of each of the three classes of checkpointing algorithms are implemented and presented in this section.

Coordinated checkpointing. The chosen CC algorithm is the Chandy-Lamport marker-based protocol for distributed snapshots introduced in subsection 5.1. The implemented variant this the one found in most modern SPEs, which blocks input channels.

Uncoordinated checkpointing. The implemented UC algorithm is a simple timer based local checkpoint condition that is evaluated before every message reception. There is little more to discuss on its implementation given that it is so very simple.

[10] https://docs.microsoft.com/en-us/azure/storage/common/storage-introduction
[11] https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction
[12] https://autofac.org/

Communication Induced Checkpointing. The chosen CIC algorithm is the HMNR-protocol [28], which was named after its creators: Helary, Mostefaoui, Netzer and Raynal. This protocol was chosen over simpler protocols such as the BCS [9] protocol as HMNR applies the highest level of sophistication with respect to the decision whether to checkpoint. While some work indicates BCS to work equally well to HMNR [5] while being significantly easier to implement, initial tests indicated it forced far more checkpoints than HMNR, resulting in the decision to switch to the HMNR protocol.

The BCS protocol uses a single local, logical clock [35] (lc_i), which is kept in each process (P_i). lc_i is incremented only when a process takes a *local* checkpoint. The local clock is piggy-backed on every outgoing message m. This piggy-backed value is called $m.lc$. Before processing a message m, lc_i is compared to $m.lc$. If the message clock is strictly larger than the local clock, ie. $m.lc > lc_i$ is true, then lc_i is set to the value of $m.lc$ and a checkpoint is forced. This protocol guarantees there is always a recovery line belonging to checkpoints taken with the same lc_i. By extension this guarantees there is always a recovery line belonging to the lowest lc_i in the system.

The HMNR protocol extends on this concept by keeping a logical *vector* clock named *clock* with the length equal to the number of processes participating in the computation. Furthermore, two new logical vector clocks named *ckpt* and *min_to* and two boolean vectors *taken* and *sent_to* are introduced as part of the protocol. These data structures are in part local (*min_to*, *sent_to*) and in part sent as metadata piggybacked on outgoing messages (*clock*, *ckpt*, *taken*). An intricate description on how these data structures are used to decide when to force a checkpoint can be found in [28].

Communication-induced checkpointing is an extension of uncoordinated checkpointing, which requires some form of local checkpointing. If the forced checkpoint condition is not met, a local checkpointing condition is evaluated similar to uncoordinated checkpointing, only extended by incrementing the local clock values if a checkpoint is taken. This subroutine is implemented exactly the same as uncoordinated checkpointing, based on a configurable time-interval. This time interval should consider the time since the last *forced or local* checkpoint as this has been observed in previous work to improve the protocol's performance [5].

7.2.3 Unifying fault tolerance algorithms. To unify the three checkpointing protocols the specification of each is split in two stages: pre- and post-delivery. Delivery in this context is the invocation of the method that processes the message's content. Notably, this is different from reception, which is the event of receiving a message, after which it gets queued for delivery. Once at the head of a delivery queue, a message gets delivered. Important to notice is that until delivery, the state of the operator remains unaffected by the message.

In the model provided by CRA, reception happens at different levels. As there can be multiple input streams for an operator, messages initially reside in a network IO buffer. These are read and deserialized in parallel after which they must be added to a single (threadsafe) delivery queue. This concludes the reception stages, after which the head of the queue will get delivered to the local operator one message at a time. From there the output must be

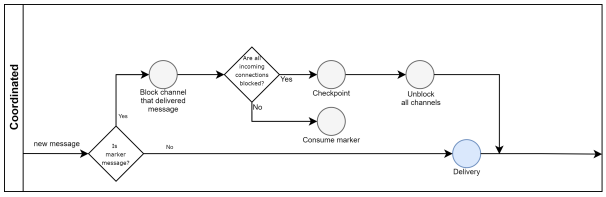


Figure 11: Coordinated checkpointing protocol logic surrounding the delivery event

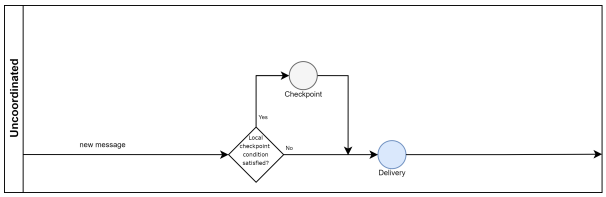


Figure 12: Uncoordinated checkpointing protocol logic surrounding the delivery event

serialized and handed off to a partitioner and written to an outgoing network IO buffer. The design of checkpointing protocols pre- and post-delivery stages lies completely within a CRA Vertex. Right after a message is taken from the delivery queue lies the pre-delivery stage and right before hand-off to the partitioner lies the post-delivery stage.

The arrangement of stages depends on the existence of a *checkpoint* method. The pre-delivery stage is executed right before a message is delivered to the worker; post-delivery is executed right after. By allowing the pre-delivery stage to consume the message or bypass delivery and skip to the post-delivery stage, the markers of the Chandy-Lamport algorithm are supported. All but the last marker that arrive at a worker must be consumed and the last one bypasses delivery to be forwarded. The implementation of this model must support state that is part of the checkpoints to ensure correct operation after a rollback. This particularly affects CIC protocols that have state in the form of local clocks.

Coordinated checkpointing is implemented in this unified model by implementing the pre-delivery stage to detect and handle marker messages and by leaving the post-delivery stage empty. One underlying assumption here is that the first marker arrives from the coordinator. A high level overview of the logic involved in this protocol is visualised in Figure 11. Uncoordinated checkpointing is implemented by simply checking a local checkpoint condition in the pre-delivery stage as shown in Figure 12. Communication-induced checkpointing in this model checks the checkpoint condition by comparing the local clock against the clock piggybacked on the message during pre-delivery, updating the local clock if the condition is met. The post-delivery step can piggyback the local clock on the outgoing message as Figure 13 depicts.

7.2.4 Recovery Orchestration: The coordinator. The coordinator carries the responsibility to coordinate startup, recovery and resumption of computation in the dataflow graph it manages. To simplify design without compromising the research goals, the coordinator is assumed to be fault-tolerant. This approach was chosen

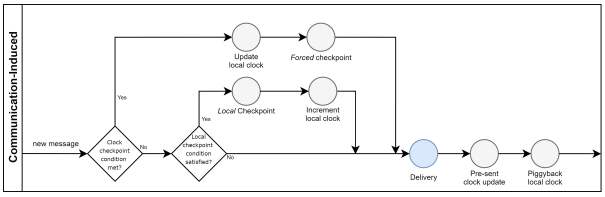


Figure 13: Communication-induced checkpointing protocol logic surrounding the delivery event

as it provides a unified way of orchestrating recovery, irrespective of the active checkpointing protocol.

The coordinator starts and stops data processing and instructs the recovery of a checkpoint. In this section compute instances are referred to as workers, these can be seen as the individual shards of an operator and as such encapsulate exactly one process.

For each worker in the system, the coordinator tracks a *worker state machine* as depicted in Figure 15. A worker can be in one of six states which are grouped in two categories: *Worker Healthy* and *Worker Unhealthy*. Via seven triggers, different state transitions are achieved. Triggers that have no outgoing edge from a state are ignored and in the implementation of FERDiS throw exceptions as they should not happen. Some states require receiving a confirmation from the worker, these are Halting and Recovering. When the worker responds the operation is completed, the *HaltCompleted* or *RestoreCompleted* triggers are fired on the state machine representing that worker. A worker can reach the Faulted state from any Healthy state by firing the *Failure* trigger.

The coordinator tracks a *system state machine* to track the state of the entire system, which is depicted in Figure 14. As with the worker state machines, from any healthy state, the faulted state can be reached. The only difference is in this machine the *WorkerUnhealthy* trigger is used for this transition. Whenever a worker transitions to a different state, a trigger is fired on the system state machine. The trigger can either be: *WorkerHealthy* if the transition ends in a state in the *Worker Healthy* category or *WorkerUnhealthy* if the transition ends in a state in the *Worker Unhealthy* category.

An important part ensuring correct transitions of the system state machine are the transition guards that ensure all workers are either running or halted before transitioning. This ensures that all workers a) are started up before starting the computation, b) have recovered from faults and those affected by the recovery line are halted and c) are done recovering before resuming computation.

7.2.5 Recovery Line Calculation. For the recovery line calculation the checkpoint dependency graph-based *rollback propagation algorithm* [48] is chosen due to being more intuitive while providing the same recovery lines as rollback dependency graph-based algorithms [20]. The rollback propagation algorithm presents the ability to consider existing state in operators as part of the recovery line, enabling preservation of existing state if it does not become inconsistent with a neighbouring operator that must roll back.

In the original design of this algorithm, a discovery phase was required to determine which processes' have communicated since taking their last checkpoint. This information was then used to append *virtual checkpoints* to the checkpoint dependency graph, where virtual checkpoints represent the volatile state that still exists in non-faulted workers. In stream processing systems, however, the dataflow graph is known exactly at run-time from which the communication pattern can easily be derived. Using this information it becomes possible to determine dependencies among virtual checkpoints instantly without discovery, under the assumption that at least one application message was send over every channel.

An important observation is that when a coordinated checkpointing protocol such as Chandy-Lamport's is used, the recovery line advances whenever an upstream worker takes a checkpoint, opening up the possibility of a partial global checkpoint to be restored if a failure caused the system unable to complete a global checkpoint. This can also happen when the calculation considers virtual checkpoints: the recovery line will only affect workers downstream of failed workers as upstream workers will not become inconsistent. Both scenarios result in the loss of channel-state messages on channels to failed workers, resulting in at-most-once instead of the expected exactly-once processing semantics. So this must be avoided, this can reliably be done for example by not committing checkpoints until a global checkpoint has completed or less reliably but simpler by not initiating a new global checkpoint upon the suspicion of a failure in the system. The latter approach was implemented, with high probability it ensures the avoidance of partial global checkpoints under the CC protocol.

When applying the recovery line algorithm to the known checkpoint dependency graph, without adding virtual checkpoints, the recovery line will be the last complete global checkpoint. This affects workers both upstream and downstream of a failure and results in the desired exactly-once processing semantics. From these observations we conclude that it must be configurable to include or exclude virtual checkpoints from the recovery line calculation.

7.2.6 Post-Rollback Channel State. Communication channels between operators can still contain messages after fault recovery. Under particular circumstances these may cause inconsistency upon being processed. It is therefore vital to perform a distributed rollback in a way that ensures no messages from pre-failure are still on the network, which will cause inconsistency upon being processed.

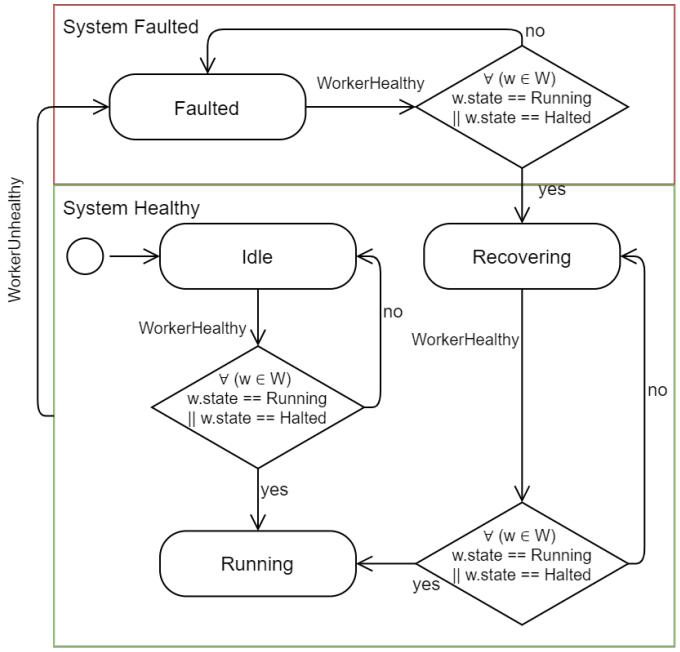
An initial observation is that an operator upstream of a failed instance must clear its downstream message queue to that operator, if it is known that the rollback propagation algorithm does not consider existing state because this guarantees a full operator graph rollback. Clearing the local output queue is enough as any channel state gets destroyed with the connection to the failed instance.

Consider the following three scenarios between two non-failed operators, named *upstream* and *downstream*, while having those positions in the dataflow graph with respect to each other.

A **Upstream rolls back, downstream rolls back**
This scenario incurs pre-rollback messages on the network moving downstream, therefore if downstream processes these messages they can be considered orphan messages due to the fact that upstream's state does not reflect sending them. From this it can be concluded that after rollback completed, all messages on the channel are orphan messages. Therefore as the recovery line by default guarantees consistency, and orphan messages cause inconsistency, the

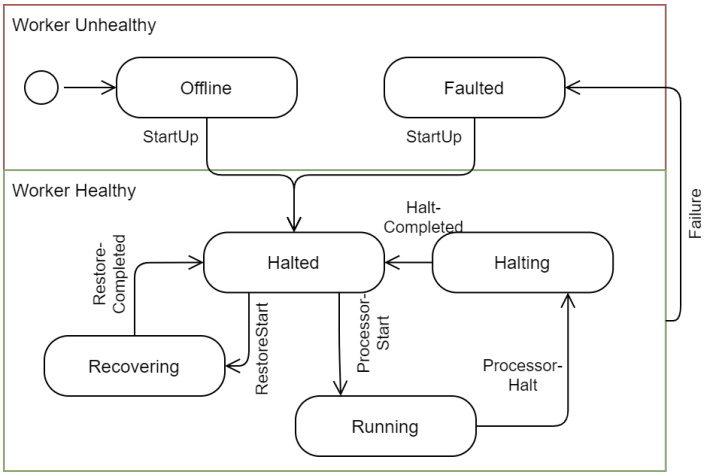
State	Entry action
Idle	n/a
Faulted	Calculate recovery line, fire ProcessorHalt trigger on worker state machines that are affected by the recovery line
Running	Fire ProcessorStart trigger on worker state machines that are in halted state
Recovering	Fire RestoreStart trigger on Worker state machines that are affected by the prepared recovery line, then garbage collect any checkpoint ahead of the recovery line

(a) Actions the coordinator takes upon state entry



(b) The state machine representing the state of the worker graph, W is the set of worker-state-machines in the system

Figure 14: The state machine representing the possible states of the entire distributed system and the state entry actions required for coordination purposes



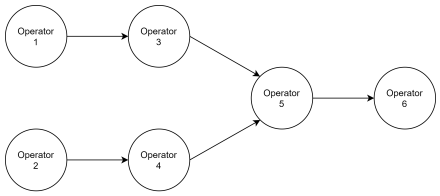
(a) The state machine representing an individual worker, states are categorised as healthy or unhealthy

State	Entry action
Offline	n/a
Halted	n/a
Halting	Send Halt message to worker
Recovering	Send Restore message to worker, including checkpoint index as specified in recovery line
Running	Send Start message to worker
Faulted	n/a

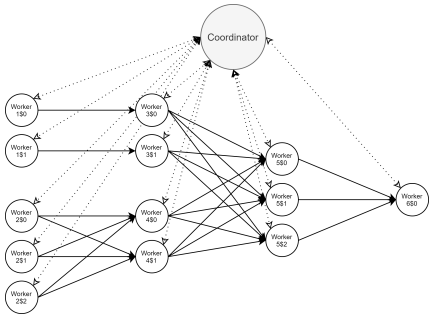
(b) Actions the coordinator takes upon worker state entry

Figure 15: The state machine representing the possible states of a worker which is part of the distributed system and the state entry actions required for coordination purposes

channel must be cleared. If the recovery line allows inconsistency, the messages that still exist on the channel will be sent again. If the remaining channel state is also processed downstream that means some orphan messages may be received downstream thrice (once before, twice after rollback), resulting in more inconsistency. Therefore in this scenario the channel state must also be cleared. This scenario is visualised in Figure 17a.



(a) An example operator vertex graph

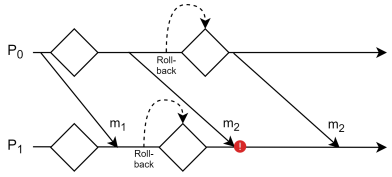


(b) A possible worker graph materialisation of the example operator vertex graph with the coordinator connected to each worker

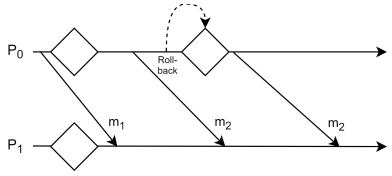
Figure 16: Visualisations how the coordinator is connected to workers

B **Upstream rolls back, downstream does not roll back**
This scenario incurs re-sending already sent messages downstream. It allows the existence of orphan messages as downstream operators may have received messages that are no longer part of the upstream's rolled-back state. If this scenario occurs that means the recovery line was calculated such that it allows inconsistency, thus permitting the resend of orphan messages. As the orphan messages may only be resent once, the processing of orphan messages will happen at most twice in total, resulting in at-least-once processing semantics at worst case. Thus, it can be concluded that the channel does not have to be cleared. This scenario is visualised in Figure 17b.

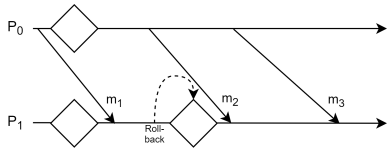
C **Upstream does not roll back, downstream rolls back**
This scenario incurs losing part of the already received messages in the downstream operator. These messages were sent and received before the rollback, but they are no longer part of the downstream state due to the rollback. This guarantees downstream is consistent with upstream, as orphan messages cannot exist downstream after the rollback. The channel will still contain messages but dropping them will only result in more lost messages than were already lost due to rollback. Therefore this scenario also does not require clearing the communication channel. This scenario is visualised in Figure 17c.



(a) Two rolling back instances becoming inconsistent due to m_2 lingering in channel state



(b) Upstream rolling back causing an expected double reception of m_2



(c) Downstream rolling back causing a lost message reception of m_1

Figure 17: Visualisations of different rollback scenarios and the effects of channel state

From this analysis it can be concluded that only channels between operators that both roll back have to be cleared to prevent inconsistency post-rollback. This can be implemented by making flushing channels part of the Halting procedure that workers can be instructed to perform. For this to work the coordinator must supply a list of adjacent workers that will roll back; then channels to those workers can be flushed. Since only workers that are affected by the recovery line receive the halt instruction, this approach covers the particular case where both operators will roll back.

7.2.7 Failure Detection. Using the coordinator/operator setup, combined with a remote resilient store gives the operators access to the store for checkpointing and restoration purposes, while the coordinator can fetch checkpoint dependency data to build the checkpoint dependency graph and calculate a recovery line when it detects an operator failure. Under the fail-stop model the assumption can be made that if an operator disconnects from the coordinator, it has failed and upon reconnection it has restarted.

7.2.8 Message Log Replay. When employing uncoordinated or communication-induced checkpointing, the ability to replay messages is a key requirement in minimizing or eliminating the amount of lost messages and thus guaranteeing or approaching exactly-once processing. Figure 18 shows a scenario in which a consistent recovery line is depicted and in the case where P_1 fails, m_2 is at risk of being lost if it is not replayed. The basic idea of this method

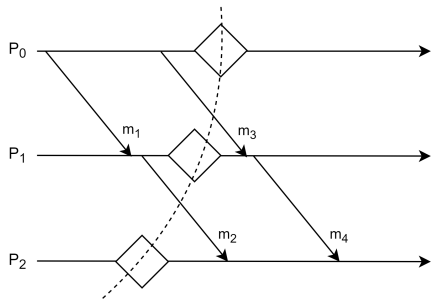


Figure 18: A message pattern in which m_2 can get lost if both P_1 and P_2 fail with a volatile message log

is that each operator keeps a log of messages it has sent out and when instructed to replay, it can replay the messages from this log, highly similar to the classical *upstream backup*.

The ability to replay messages relies heavily on sequence numbers being part of messages for the purpose of deduplication, identifying which messages must be replayed and which messages can be removed or *pruned* from the log.

At runtime the message log may grow endlessly if not occasionally pruned, which is undesirable as it may cost a lot of memory or disk space. It is known that whenever an operator takes a checkpoint, it persists the state manipulations made by messages it has received thus far. This means those messages will never have to be replayed, as long as the operator will never have to roll back to a checkpoint before that. When this happens, the upstream operators can prune these persisted messages from their logs as they will not be needed for replay any more. To accurately determine which messages can be pruned, each operator must keep track of which sequence numbers it has received from its upstream neighbours. The coordinator will be tasked with determining these pruning points based on worst case failure scenarios and must instruct workers to prune messages from their logs when necessary. To support this, the coordinator must be notified of operators taking checkpoints to determine if that checkpoint advances the worst case recovery line and in turn instruct log pruning to affected workers.

After recovery messages may have to be replayed, e.g. in the scenario depicted in figure 17c, m_1 through m_3 will have to be replayed. Which exact messages must be replayed is dictated by the last sequence number that was persisted as part of the recovered checkpoint. This makes storing sequence numbers in checkpoints a hard requirement for this feature to work.

A final important aspect is that the order of messages must be preserved pre and post replay. In figure 17c, if m_1 is replayed after P_0 has sent out m_2, their order may inverse or worse, m_2 may be delivered twice, resulting in at-least-once guarantees over the desired exactly-once. For this purpose the receiving process P_1 may deduplicate input based on the expected next sequence number, resulting in the first reception of m_2 to be *dropped*, after which the replayed m_1 and m_2 arrive, resulting in exactly-once processing in the correct order.

Log persistence is another factor that has influence on the processing semantics. If the message log is persisted in figure 17c, P_1 will be able to replay m_2 post failure, while if the log is kept volatile, the message cannot be replayed due to the failure having wiped the volatile state. For performance reasons the log is kept volatile and the amount of lost messages is tracked through sequence numbers for the purpose of quantifying how many messages are lost.

8 BENCHMARKING SETUP

8.1 Workloads

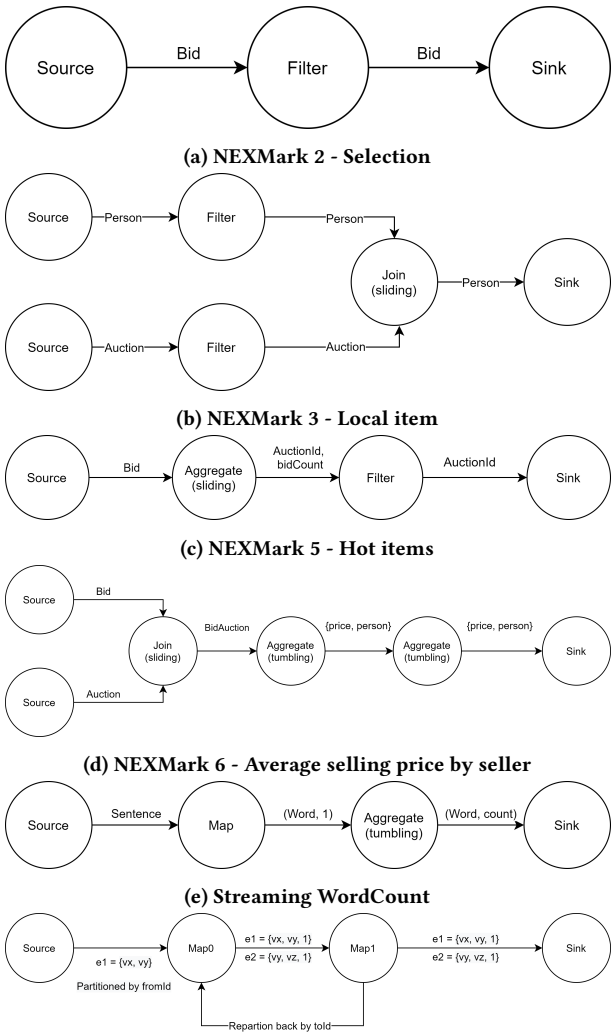
From existing work a set of queries is extracted to form a representative set of workloads for the evaluation of the stream processing system. From the NEXMark [47] benchmark, queries 2,3,5 and 6 are chosen. These cover most message patterns than can occur in acyclic dataflow graphs including all stream operators: map, filter, join and aggregate. In addition, we select a typical example in the map-reduce programming style: WordCount. This common example is intuitive and easy to validate for correctness. Lastly a cyclic query is designed as no such query appears in the literature. Each of the presented queries has been implemented as a FERDiS query. We describe the queries in the following paragraphs and visualise them in figure 19.

The NEXMark queries are designed to consume auction data from three streams: a person stream containing people that are buyers and/or sellers, an auction stream where each auction contains an item for sale and references the person selling it and lastly a bid stream where each bid is a numerical offer by a person on a particular auction. The engineers that implemented the NEXMark benchmark have graciously provided a public data generator. For this experiment the generator is embedded in a small program to feed these data streams into an Apache Kafka cluster. An entity diagram displaying the relations between these entities is presented in figure 20 and a SQL presentation of the queries can be found in the NEXMark paper [47].

NEXMark 2: Selection is a very simple query that consumes from the bid stream and filters the stream based on a certain condition. The filter condition presented in the paper is rather restrictive so here we implement it to filter out roughly half of the incoming events by filtering on even/uneven key values. Each operator is interconnected in a pipeline configuration.

NEXMark 3: Local item finds items being sold by a person residing at a specified location. The query consumes from two streams, the people and auctions streams. First each stream is filtered, the people stream by their location and the auctions stream by category. The remaining results are joined to find sellers which are the output of the query. The query performs a join using a window that according to the NEXMark specification should be unbounded, however we use 30-second windows that slide by 5 seconds (Large) and 15 second-windows that slide by 5 seconds (Small).

NEXMark 5: Hot items finds items on sale that receive the most bids. By aggregating the bids in a sliding fashion and counting the results per auction the stream is translated to (auctionId, bidCount) pairs. A stateful filter operator then adds these to its internal state and filters out the results that are not the highest seen so far. This results in the output stream continuously outputting the highest bid counts per auction id the system has observed so far. The NEXMark specification asks for windowing based on tuple counts, however as this has not been implemented in the stream processor, windows of



(a) NEXMark 2 - Selection

(b) NEXMark 3 - Local item

(c) NEXMark 5 - Hot items

(d) NEXMark 6 - Average selling price by seller

(e) Streaming WordCount

(f) NHop approximation with a cyclic edge in the query graph

Figure 19: The six different queries that have been selected for the system evaluation

size 10 (Small) or 30 (Large) seconds are chosen, which slide every 1 second.

NEXMark 6: Average selling price by seller consumes from the bids and auctions streams to find the average selling price per seller. This is a query that builds on a subquery that computes the selling price for each auction. This query adds one aggregate operator to that, which consumes the selling price per auction and averages over them grouped by the seller. As a result this query applies three windowed operators in a row making it a more complex query than the others. Like in the NEXMark 3 query, the join uses a window that according to the NEXMark specification should be unbounded, however we use 30-second windows that slide by 5 seconds (Large) and 15-second windows that slide by 5 seconds (Small). The aggregates are set-up to use tumbling windows of 5 or 15 seconds.

WordCount is a very common example in the map-reduce programming style. The streaming version is semantically slightly different from the original map-reduce paradigm. This is due to the fact that the reduce phase under map-reduce would require the entire input to be consumed, something that is not possible in the streaming paradigm. So the stream processing variant of WordCount is *Incremental WordCount*, where the reduce phase is executed through timed aggregate operators. For easy correctness validation, a single stateful sink operator is appended to the end of this dataflow graph, which keeps a map of words and their current counts and increments them whenever a new count is produced by aggregates. The aggregate in this query uses tumbling windows of 1 (Small) or 10 (Large) seconds.

Cyclic queries are not part of any existing benchmark definition. For CIC protocols this implies that the queries so far will never contain z-cycles and thus will never take forced checkpoints. CIC, which does not take forced checkpoints, effectively becomes uncoordinated checkpointing with some communication overhead. Beyond quantifying the communication overhead the queries do not provide insight in CIC-based fault tolerance.

Due to the lack of a cyclic dataflow query in the literature, a cyclic query is designed, which consumes graph data. The idea is to consume a stream of graph edges in order to find n-hop neighbours within the graph. The query graph can be observed in Figure 19f and forms a full pipeline downstream, with a shuffle back-channel between Map1 and Map0.

The main operator in this query is the first mapper, which tries to match input edges 'fromId' values to its internal states 'toId' values, when it finds matches with a new (lower) hop count these are forwarded. The input datastream is partitioned by 'fromId' of the edges but on the shuffle backchannel is partioned on 'toId'. This allows the different pipelines to feed eachother new n-hop neighbours they will be able to match their input stream to. If the hop count is larger than the configured n, the event is discarded, in the current implementation $n = 2$.

This approach allows effective state partitioning and will result in approximative results on n-hop neighbours as, particularly at the beginning, not many edges are known and so a neighbour may be missed. Furthermore, input from the back-channel is disallowed to generate output as this may cause a message explosion and as a result in a deadlock on the cycle. This deadlock can happen if two operators can not write output due to buffers being full and not being able to consume input because they are still writing output.

8.2 Metrics

To evaluate the performance of the algorithms, three types of metrics are collected, *performance-*, *checkpointing-* and *recovery metrics*. Each of these types are elaborated below and an overview is found in Table 1.

Performance Metrics are throughput and latency, two very typical metrics for evaluating the performance of stream processing systems. An important note with respect to these is that they should not be collected from within the stream processor as it may affect the performance of the system. For this reason the metrics get collected in an end-to-end fashion by having the stream processor write output to a special Kafka topic. These contain timestamps and

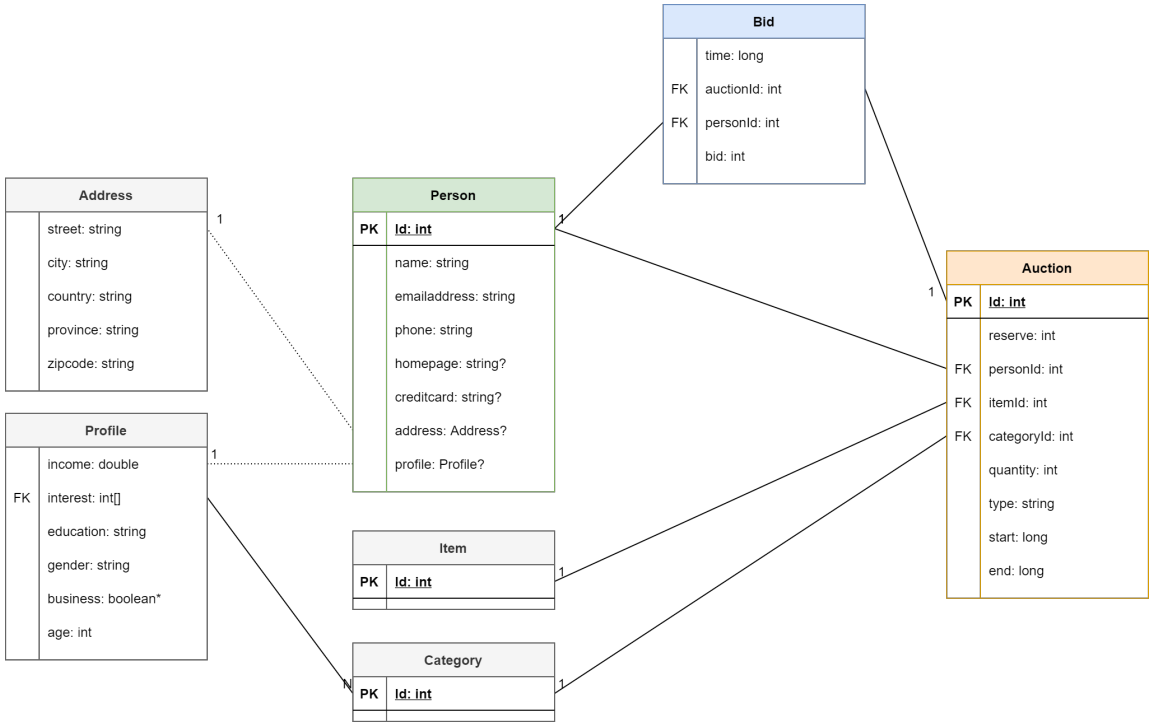


Figure 20: The entities defined in the NEXMark benchmark

through sampling the high watermarks of the output topic every 0.3 seconds, the throughput can be determined. The latencies are sampled slightly differently by asking from Kafka for each partition the last message at the next 0.3 second and using the message content together with the arrival timestamp in Kafka to determine the latency. Using these strategies allows collecting throughput and latency metrics accurately and without affecting the stream processor's performance.

Checkpointing Metrics measure three aspects: the size of the checkpoint, the amount of time the checkpoint halted the processing of records and lastly whether the checkpoint was forced. As the checkpointing library is part of the system's source code, writing these metrics to a log file is easy and, due to the operation not being on the applications hot path, has minimal impact on the system's performance.

Recovery Metrics are the last category and includes recovery time and rollback distance. The recovery time is defined as the amount of time it took from the insertion of a failure until the system is back within 10% of its pre-failure latency [44]. As this is a direct function of latency collection this requires no extra metric collection. Recovery time is also collected as the time it took for a process to restore a checkpoint and the rollback distance, that is the amount of time (ms) the process rolled back, combined with a potential non-zero amount of lost messages due to volatile message log being lost.

Metric	Type	Unit
Throughput	Performance	Events/second
Latency	Performance	Milliseconds
Size	Checkpointing	Bytes
Take-Time	Checkpointing	Milliseconds
Forced	Checkpointing	Boolean
Recovery-Time	Recovery	Milliseconds
Rollback distance	Recovery	Milliseconds
Restore-Time	Recovery	Milliseconds
#Lost Messages	Recovery	Integer

Table 1: The metrics used in the evaluation

8.3 Experiment Configurations

The cluster size has been fixed for the experiments. To maximize utilization of the resources, each query's dataflow-graph gets sharded to optimize for throughput. This on average results in about 24 operators per dataflow graph. Given the cluster capacity this results in roughly one core per operator as the Kafka instance in the cluster is allocated 8 cores.

The parameters that will change between experiment configurations are described in Table 2. Running all possible configurations of these parameters is infeasible within the scope of this work, for this reason some particular configurations are disregarded.

The window size parameter can be ignored for query 2 and 6, that is, NEXMark 2 and the cyclic NHop query as these queries do not contain windowed operators. Secondly, the coordinated checkpointing protocol can be disregarded in query 6 as the protocol

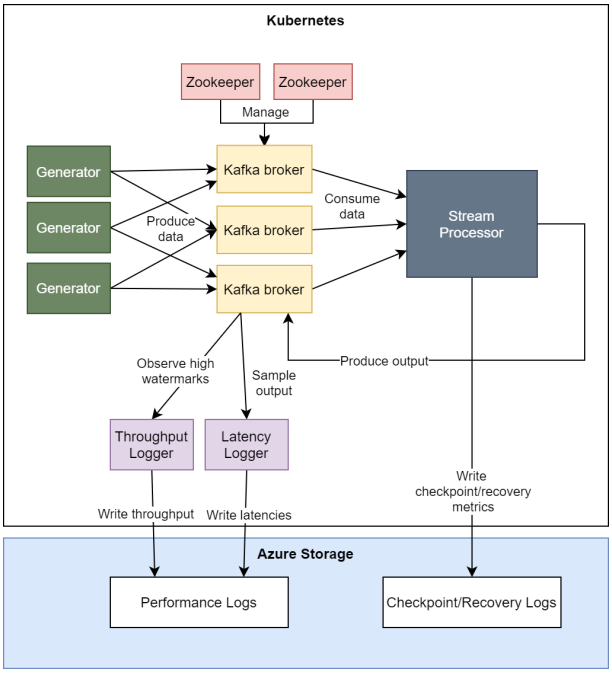


Figure 21: A high level overview of the components used in the experiment environment

does not support cycles. All queries will be executed in a failure-free manner to analyse runtime performance and with a single failure to analyse recovery.

Parameter	Options
Checkpointing protocols	Coordinated-, Uncoordinated- and Communication-Induced Checkpointing
Checkpointing interval	10 or 15 seconds
Window size	Small or Large [13]
Queries	1-6 of the 6 presented queries
Failure	None, One

Table 2: Experiment configuration parameters

8.4 Execution Environment

The experiments will be executed on a kubernetes cluster for easy management of the compute resources and easy deployment of the different programs required to execute the benchmark. The setup consists of a few components: the stream processor, Apache Kafka, Apache Zookeeper, query-specific data generators and lastly the metric loggers that observe the output topic in Kafka. An overview of the environment is described in figure 21. With the stream processor as a black box, its content is dictated by the query that it executes during an experiment. The possible contents of the black box can be found in figure 19.

[13]This changes per query but on average is 15 to 30 seconds

SurfSara HPC provides the hardware[14] on which the experiments are run. A total of five Virtual Machines (VMs) are provisioned and configured to host a Kubernetes [15] cluster. In this cluster the master is a small VM with 4 cores and 4Gb memory while the 4 remaining workers each contain 8 cores and 16Gb memory. This brings the usable cluster capacity to 32 cores and 64Gb memory, these resources must be used by all of the components described in figure 21.

Fault Injection in a Kubernetes cluster is relatively easy to simulate, given the fail-stop assumption. Faults are injected by sending a SIGKILL to processes within containers in Kubernetes pods. This in turn causes Kubernetes to restart the process within the same container and the same pod and as such, no relocation has to be performed by Kubernetes. Alternatively a pod could be deleted to achieve the same effect except Kubernetes will need to provision a new pod which introduces extra downtime after a failure. By killing the process within the pod, metrics like recovery time remain untainted by the overhead of allocating a new Kubernetes pod while still completely wiping the application's runtime and its memory.

9 EXPERIMENTS

The following section describes the results from 116 experiment configurations, each executed three times for the purpose of securing statistical significance, resulting in 348 experiment executions, which are worth roughly 35 hours of computation. A subset of statistics computed over the collected metrics is presented in Tables 3, 4, 5 and 6. Interesting latency and throughput experiment configurations have been selected, visualized, and discussed.

An important note on the interpretation of this data is that each of the queries has been calibrated to run at 80% of the throughput in UC configuration with a 10 second checkpoint interval. This means that the system under coordinated checkpoint (CC) configuration runs at a lower percentage of its maximum throughput compared to uncoordinated checkpoint (UC) and CIC configurations, which run at a higher percentage of its maximum throughput compared to UC. Each of the queries with windows also have been calibrated for both small and large window sizes separately as window size was observed to strongly impact the performance of the system.

9.1 Discussion on checkpointing the message log

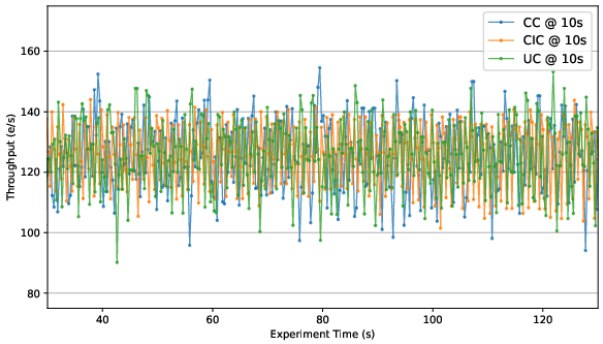
During the testing phase of the benchmarking setup, the performance of the system was analysed when including the message log in checkpoints. When including the message log in the checkpoints taken by processes, UC and CIC protocols needed to take significantly larger and thus longer checkpoints. During these long checkpoints no input could be processed, causing heavy input buffering and as a result, strong spikes in throughput and latency. When lowering checkpoint interval under 20 seconds the system would actually become unable to catch up with its input between checkpoints, causing a continuously increasing spikes in throughput and latency as time progresses.

The result of this effect is extreme destabilisation of the system and as a result very poor performance even under failure-free

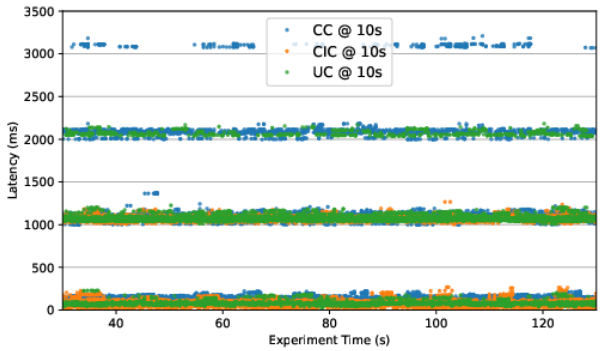
[14]https://userinfo.surfsara.nl/systems/hpc-cloud
[15]https://kubernetes.io/

Query	protocol	mean	95th	99th	std. dev.
NEX 2	UC @ 10s	39.45	74.00	117.00	21.32
NEX 2	UC @ 15s	42.82	80.00	120.00	22.81
NEX 2	CC @ 10s	41.69	71.00	134.00	23.06
NEX 2	CC @ 15s	44.45	73.00	172.00	25.36
NEX 2	CIC @ 10s	45.25	100.00	214.00	37.22
NEX 2	CIC @ 15s	43.85	87.00	229.00	35.91
WC L	UC @ 10s	1180.92	1771.00	2798.00	507.28
WC L	UC @ 15s	1235.92	1911.00	2808.00	498.06
WC L	CC @ 10s	980.28	1674.00	1757.00	467.73
WC L	CC @ 15s	1103.94	2040.00	2126.00	490.24
WC L	CIC @ 10s	771.57	2021.00	2962.00	663.43
WC L	CIC @ 15s	849.71	2158.00	2806.00	678.36
WC S	UC @ 10s	817.45	2053.00	2115.00	590.60
WC S	UC @ 15s	820.25	2046.00	2109.45	585.74
WC S	CC @ 10s	832.70	2118.00	3107.00	842.91
WC S	CC @ 15s	865.92	2107.00	3098.00	827.47
WC S	CIC @ 10s	422.84	1117.00	1162.00	469.34
WC S	CIC @ 15s	381.82	1108.00	1160.00	460.50
NHop	UC @ 10s	168.37	254.85	356.00	69.84
NHop	UC @ 15s	170.82	270.05	325.00	64.31
NHop	CIC @ 10s	99.81	144.00	431.00	69.64
NHop	CIC @ 15s	94.83	132.00	366.01	62.18

Table 3: Latency statistics under failure-free conditions



(a) WordCount (S) throughput



(b) WordCount (S) latency

Figure 22: WordCount (S) query performance under different checkpoint protocols

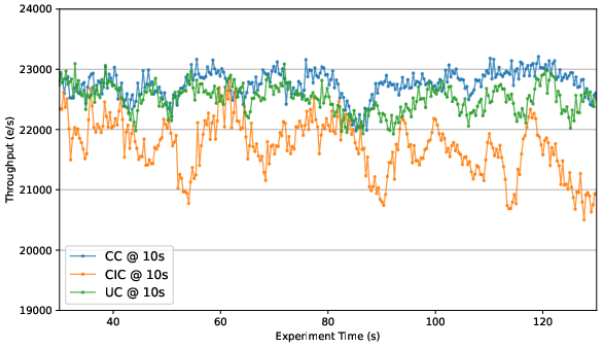
conditions. When checkpointing only once every 30 seconds, the system was able to keep up with the input stream but not without introducing 2-4 second latencies, even in the absence of windowed operators. Furthermore, if the system's implementation were to be optimized for higher throughput the effects of this destabilisation will only increase as the checkpoints will increase in size proportional to the increase in throughput. This would result in relatively even lower throughput and higher latency and a more rapid destabilisation of the systems performance. This is particularly obvious to observe in comparison to coordinated checkpointing which is not subject to this behavior.

At this point we made the choice to relax the delivery guarantees for the UC and CIC protocols to at-most-once, where only the failed operator would lose its message log. To quantify the effects of this choice total number of messages that was lost was logged to a separate file and made part of the evaluation. The suspicion here was that it may still be the case that including the message log in checkpoints is feasible for windowed operators as they produce far less events than a mapper or filter operator would. The amount of lost messages collected in the rest of the experiments may give insight in that.

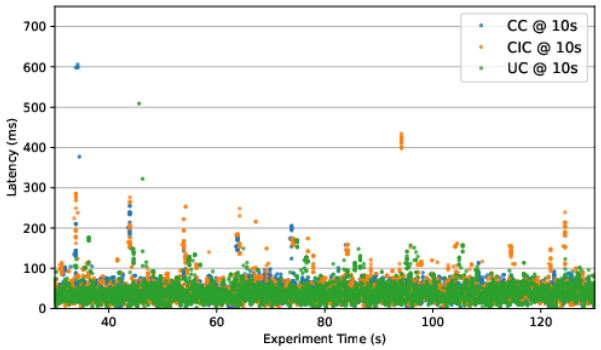
9.2 Performance experiments

In this section we discuss the results of failure-free experiments where each configuration is evaluated in terms of performance and checkpointing metrics.

The continuous query. (NEXMark 2), which has no windowing in the query graph, shows a relatively high performance impact across the different checkpointing protocols which is as expected. Coordinated checkpointing has the least impact on this query graph, resulting in the highest possible throughput and stable, low latencies, which may also be observed in Table 3. Uncoordinated checkpointing performs slightly worse on the throughput spectrum but has slightly more stable latencies with lower 99th percentile latency. This is likely attributable to this algorithm not performing system-wide input buffering during checkpointing. The impact of the extra work needed per message under the CIC protocol shows on both the latency spectrum and throughput spectrum, where mean latency is only elevated by a 10% but the 99th percentile is almost double that of the experiments under UC configuration. Throughput for the CIC protocol can be observed in figure 23 and is both lower and less stable than that of the other two algorithms. Notably, on this query graph the amount and size of checkpoints taken varies extremely little between the algorithms, CIC does force a few checkpoints which is attributable to the chosen CIC algorithm's attribute of forcing checkpoints when a z-cycle is suspected, which effectively becomes a means for advancing the recovery line when possible even in an acyclic graph. The high impact of performance on the CIC protocol may be attributed to the current serialization implementation, while likely to be slightly worse than CC, optimizations in the implementation may make this difference less pronounced.

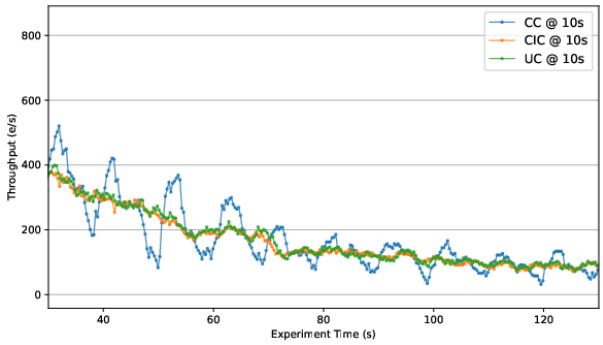


(a) NEXMark 2 throughput

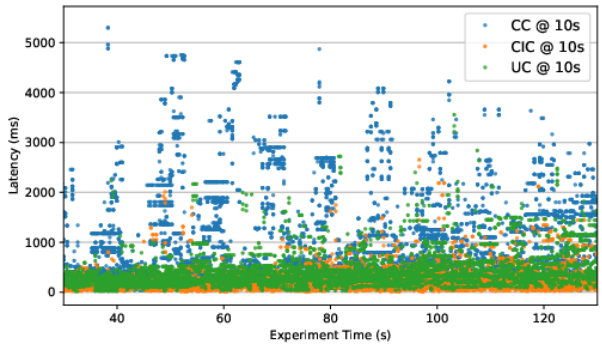


(b) NEXMark 2 latency

Figure 23: NEXMark 2 query performance under different checkpoint protocols



(a) NEXMark 3 (L) throughput



(b) NEXMark 3 (L) latency

Figure 24: NEXMark 3 (L) query performance under different checkpoint protocols

When considering aggregates. the distinct differences observed in the data collected from the NEXMark2 query dissapear. Figure 22 shows that in the WordCount query with small window size, the performance of the different protocols on the throughput spectrum is highly similar. This same behavior is observed in the WordCount configuration with a large window size and the NEXMark 5 query in both small and large window configurations.

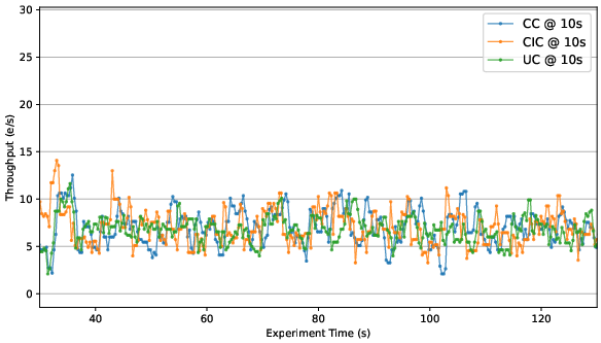
In general these queries exhibit the same statistical differences on the latency spectrum as well, with the notable exception of the WordCount configuration with small window sizes where the CC algorithm has a negative impact on the latency spectrum, resulting in similar mean latency as uncoordinated checkpointing but far higher 99th percentile latency and higher standard deviation. These results are observed consistently and show a case in which the CIC algorithm yields better performance on the latency spectrum than the UC and CC algorithms, these statistics can be found in Table 3. The elevated latencies under CC configurations may be attributed to system-wide buffering during checkpoints, especially on shuffle connections due to the higher amount of markers that must be received opposed to pipeline connections.

The amount of checkpoints. that are being taken is for all queries highly similar, where one may expect increased number of checkpoints particularly from the CIC algorithm as it forced extra checkpoints. The effort of considering the last forced checkpoint in the local checkpoint algorithm in the CIC implementation shows its merits: using CIC does not cause significantly more checkpoints to be taken.

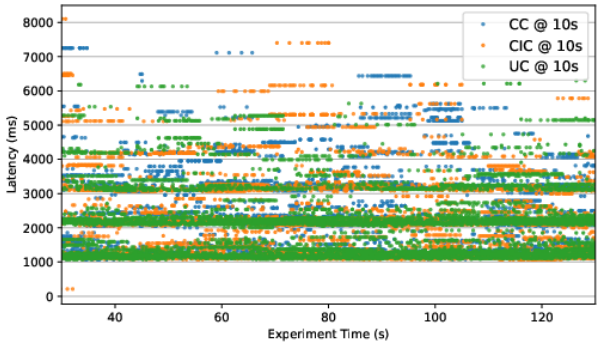
For the UC and CIC algorithms, under the WordCount and NEXMark 5 query configurations, slightly less checkpoints are taken than for the CC algorithm. This may be attributable to the way the checkpointing decision is implemented, as that condition is only evaluated when a message will be processed, meaning operators which lie behind aggregates are less likely to take checkpoints especially if the aggregates have a large window slide-size. A subset of the experiment's checkpointing statistics is presented in Table 4.

With multiple aggregates. in place, like in the NEXMark 6 query, no significant patterns emerge in the latency or checkpointing data. The query contains three windowed operators in sequence and the variation in output rates combined with the observed poor performance of the existing aggregate operator implementations cause a variation in the data that can not be attributed to the checkpointing algorithms but rather the nature of the query graph and the nature of the input streams generated by the NEXMark generator.

In the cyclic query. the first observation is that the CC algorithm cannot run on this query graph. Upon inserting barriers the entire dataflow will come to a halt due to the blocking mechanism causing a deadlock on the cycle. The UC and CIC algorithms however do support execution on this query graph and show an unexpected

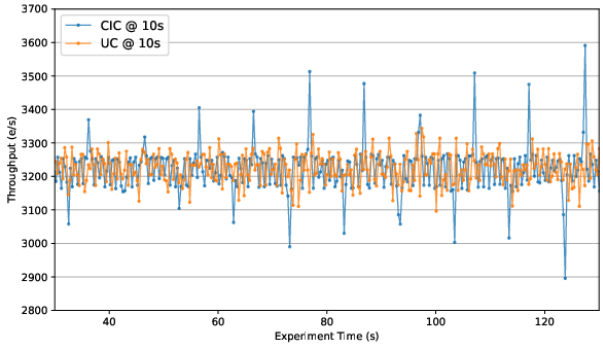


(a) NEXMark 5 (S) throughput

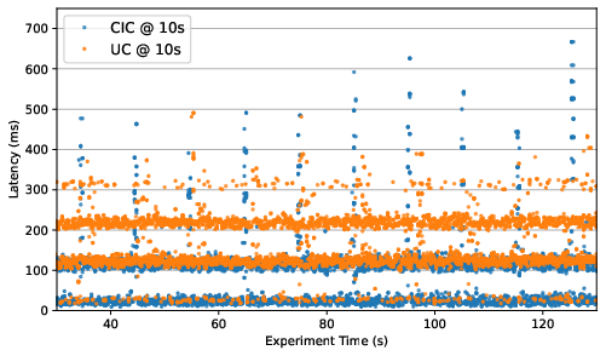


(b) NEXMark 5 (S) latency

Figure 25: NEXMark 5 (S) query performance under different checkpoint protocols



(a) NHop throughput



(b) NHop latency

Figure 26: NHop query performance under different checkpoint protocols

pattern on the latency spectrum. The CIC algorithm performs more work per message and yields far better latency results on the mean till 95th percentile latency but with highly similar standard deviation to the UC algorithm. However, the UC algorithm has a lower 99th percentile latency meaning it performs slightly better with respect to the worst case runtime latency. The CIC algorithm's effort to advance the recovery line when possible is visible on the throughput and latency spectrum in this query when observing figure 26. More input buffering happens under the CIC algorithm because two sequential operators may take checkpoints right after one another, a case highly unlikely to happen under the UC algorithm. This results in the low-then-high throughput spikes and the accompanying spikes in latency at those same points in time.

An unexpectedly high impact of barrier blocking. has been observed in the NEXMark 3 query under large window size configuration. The size and time required for the checkpoints are highly similar to that of other algorithms so this can not be the cause. In turn the conclusion is that it must be the barrier-blocking mechanism that strongly affects the throughput spectrum of this query. The impact was less observable under the small window configuration, which is hard to attribute to the window size parameter itself, as the window size also affects the rate at which the query can consume input.

The observation was made that the different streams being joined are differing in throughput, this makes sense as the amount of persons being generated by the nexmark generator is larger than the amount of auctions. This results in one of the join operators input's (on the auctions side) is being blocked while the other stream (people) still has a reasonable amount of events to process before the checkpoint marker reaches the operator. This results in temporarily less events being joined, thus the sudden lower throughput, which are processed right after the checkpoint was taken, thus the subsequently higher throughput.

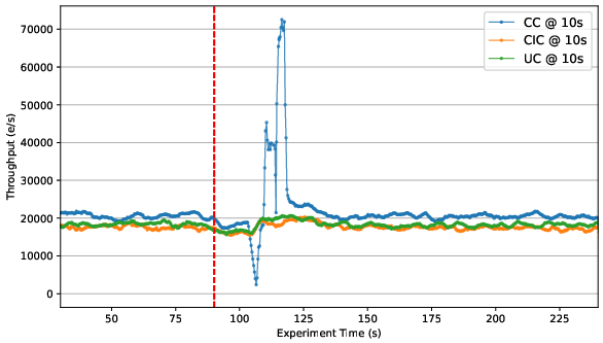
In the configuration for the small window size the difference in input rate was likely less pronounced and thus the observed performance deterioration was not observed. From this experimental result the intuition rises that operators with different input rates behave poorly under the coordinated checkpointing algorithm, this can go beyond join operators as during data parallel processing the rate of processing may also differ depending on the partitioning strategy and characteristics of the input stream.

9.3 Failure experiments

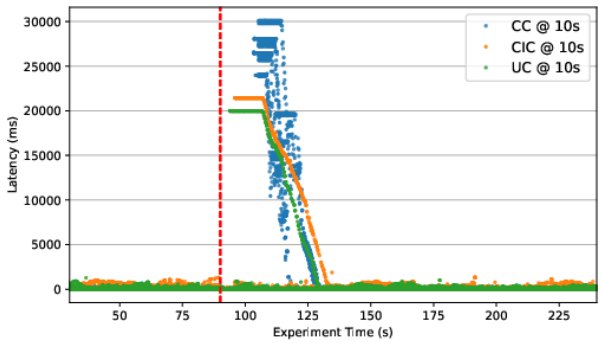
In this section we discuss the results of single failure experiments where each configuration was evaluated in terms of recovery metrics and performance metrics specifically surrounding the failure. The recovery statistics of a subset of the executed experiment configurations are presented in Table 5, The NEXMark 5 query has been omitted entirely as there was no meaningful difference in the observed metrics between the protocol configurations, its query

Query	protocol	#regular	#forced	time (ms)
NEX 2	UC @ 10s	240	0	56 ± 32
NEX 2	UC @ 15s	168	0	61 ± 19
NEX 2	CC @ 10s	240	0	52 ± 24
NEX 2	CC @ 15s	168	0	57 ± 31
NEX 2	CIC @ 10s	222	18	68 ± 20
NEX 2	CIC @ 15s	153	15	73 ± 19
WC S	UC @ 10s	235	0	57 ± 21
WC S	UC @ 15s	164	0	57 ± 17
WC S	CC @ 10s	240	0	52 ± 15
WC S	CC @ 15s	168	0	59 ± 39
WC S	CIC @ 10s	217	19	70 ± 22
WC S	CIC @ 15s	137	27	75 ± 48
NEX 3 L	UC @ 10s	238	0	85 ± 83
NEX 3 L	UC @ 15s	165	0	88 ± 79
NEX 3 L	CC @ 10s	239	0	82 ± 78
NEX 3 L	CC @ 15s	164	0	87 ± 75
NEX 3 L	CIC @ 10s	189	51	97 ± 77
NEX 3 L	CIC @ 15s	134	34	107 ± 80
NEX 5 L	UC @ 10s	223	0	188 ± 249
NEX 5 L	UC @ 15s	152	0	204 ± 263
NEX 5 L	CC @ 10s	240	0	195 ± 265
NEX 5 L	CC @ 15s	152	0	205 ± 263
NEX 5 L	CIC @ 10s	120	113	197 ± 230
NEX 5 L	CIC @ 15s	82	80	221 ± 266
NHop	UC @ 10s	240	0	100 ± 65
NHop	UC @ 15s	168	0	105 ± 63
NHop	CIC @ 10s	156	84	104 ± 58
NHop	CIC @ 15s	121	46	112 ± 61

Table 4: Checkpoint statistics under failure-free conditions



(a) NEXMark 2 throughput under failure



(b) NEXMark 2 latency under failure

Figure 27: NEXMark 2 query failure recovery under different checkpoint protocols

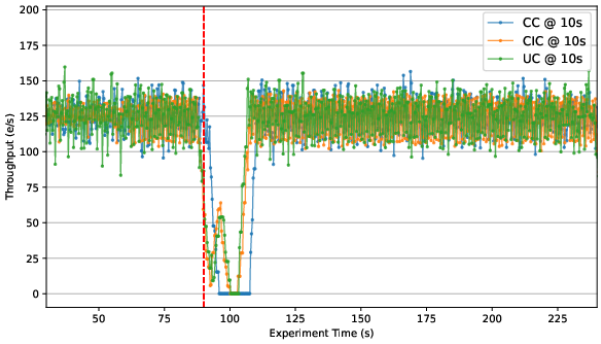
graph would make failures largely invisible in the collected metrics. We start of with a discussion on the lost messages per protocol, then look into recovery performance and lastly discuss recovery in the cyclic query.

9.3.1 Lost messages. The amount of lost messages varies significantly between queries and checkpointing configurations. A bug in the replay mechanism was observed after experiment execution affecting the NHop and NEXMark6 configurations due to the location of the inserted failure. This invalidated the reported number of lost messages in these experiments as more messages were dropped downstream of the inserted failure than strictly necessary, therefore these numbers were left out of the resulting data. The extent of the bug's effect is limited, as it only results in a bigger gap in recovery than is strictly necessary for these specific experiments. Therefore other data points with respect to these experiments are still discussed in the rest of this section.

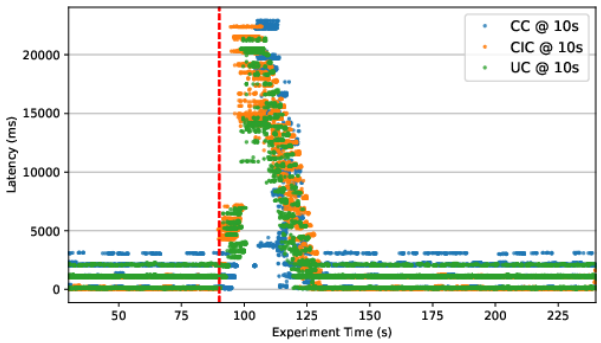
In Table 6 an overview of amounts of lost messages is displayed for a subset of the experiment configuration. The CC configurations are omitted as the amount of lost messages is alway 0. Overall a pattern is observed that a surprisingly low amount of output is lost. This behavior may very well be acceptable in applications which tolerate small amounts of output being lost in favour of more rapidly resuming the processing of new data.

Lower deviation in the amount of lost messages for the CIC algorithm is observed very consistently over the different experiment configurations. This can be attributed to the HMNR protocol forcing checkpoints to ensure the recovery line advances, resulting in generally fewer messages being part of the channel state between checkpoints. The actual amount of lost messages is not necessarily lower than in UC configurations and appears correlated to the rollback distance. Rollback distance statistics can be found in Table 5 and contain no observable pattern attributable to the checkpointing protocol beyond CC configurations having less deviation in rollback distance than the UC and CIC protocols. This makes sense as the checkpoints were taken coordinatedly, so the rollback distance of workers should be very similar.

If we compare mean throughput under failure-free conditions to the mean amount of lost messages, we observe that the amount of lost throughput in seconds ranges from 0.2s to 5.5s. This is remarkably low given the checkpoint intervals of 10 and 15 seconds. For the 10 second checkpoint intervals, the amount of lost throughput is consistently lower compared to the 15 second interval configurations. Based on this observation, checkpointing more frequently may reduce this number even further. Under the assumption failures happen only once in a few minutes, one may consider these protocols to effectively yield 99% consistency.



(a) WordCount S throughput under failure

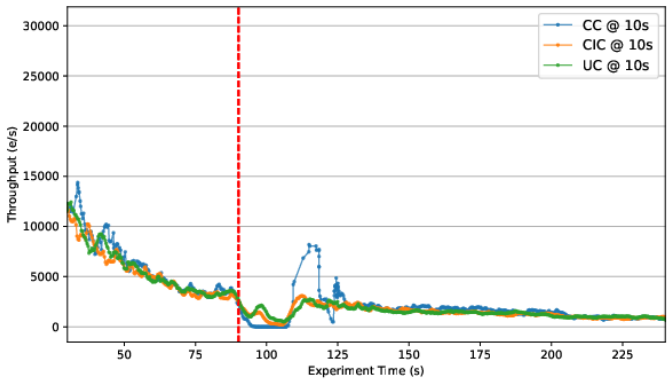


(b) WordCount S latency under failure

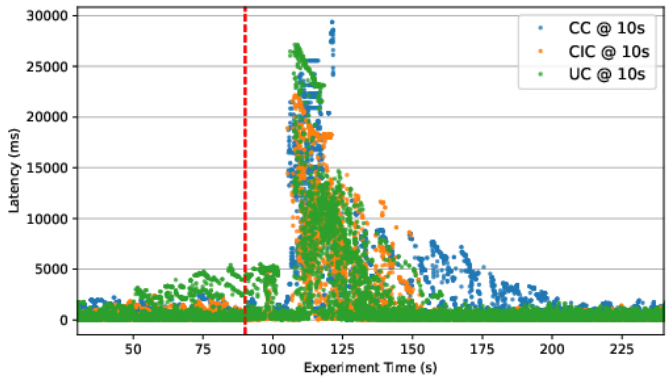
Figure 28: WordCount S query failure recovery under different checkpoint protocols

Perhaps more interestingly though, is that the observed number of lost messages is considerably lower for join and aggregate operator failures, based on this observation, aggregate operators and join operators may be able to checkpoint their message log without the extreme performance penalty that map and filter operators are subject to. This would allow quicker recovery compared to CC but with the same exactly-once processing semantics.

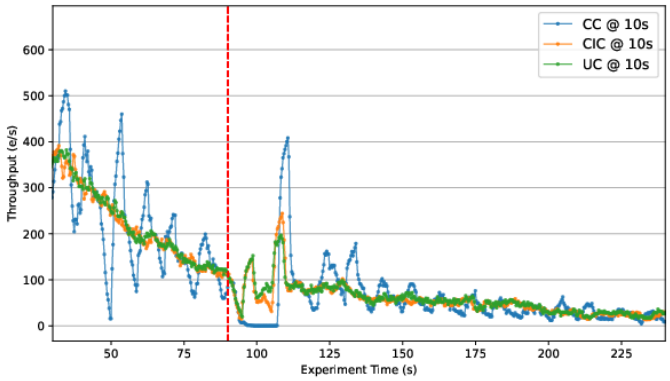
9.3.2 Recovery performance. The amount of reprocessing after recovering from a failure is observed to be highest for CC configurations. This reprocessing becomes visible in the form of a throughput spike after recovery from a failure, this behaviour can be observed in Figure 29 and is particularly evident in Figure 27. In the experiments that have an aggregate in the query graph, this spike is not observed as the aggregate buffers the reprocessed input, this may be observed in Figure 28. While aggregates may obscure the throughput spike that is caused by reprocessing, the effects are still visible on the latency spectrum and exhibit similar recovery patterns to the other queries. This recovery pattern is visible as a sudden increase in latency roughly as high as the rollback distance plus the time the system needed to perform the recovery followed by a relatively steep drop-off back to the pre-failure latencies. The steepness of this drop-off is expected to be proportional to the processing headroom the system has available during normal operation, this means that at 60% resource usage (40% headroom) the



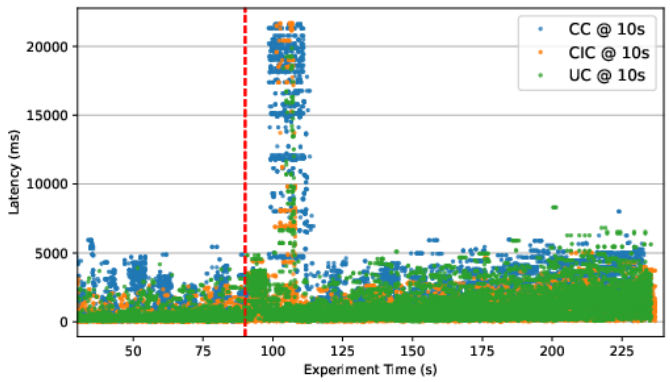
(a) NEXMark 3 S throughput under failure



(b) NEXMark 3 S latency under failure

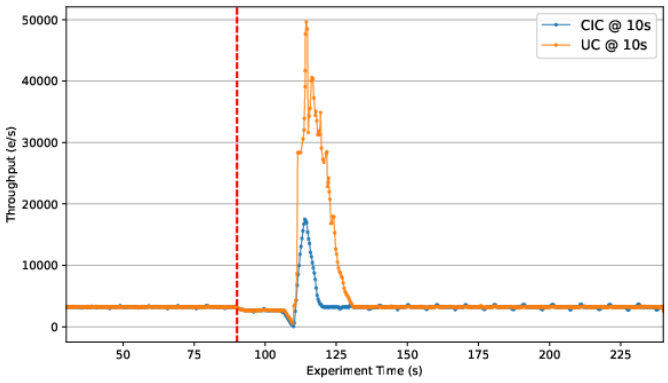


(c) NEXMark 3 L throughput under failure

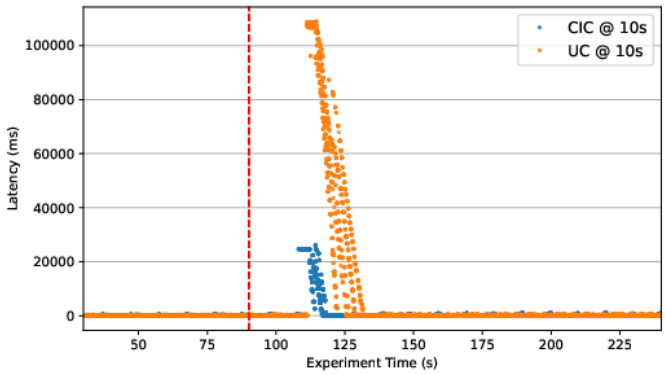


(d) NEXMark 3 L latency under failure

Figure 29: NEXMark S & L query failure recovery under different checkpoint protocols



(a) NHop throughput under failure



(b) NHop latency under failure

Figure 30: NHop query failure recovery under different checkpoint protocols

Query	protocol	recovery (s)	rollback (s)
NEX 2	UC @ 10s	44 ± 9.14	25.8 ± 7.72
NEX 2	UC @ 15s	43 ± 4.03	31.06 ± 7.74
NEX 2	CC @ 10s	35 ± 0.71	24.07 ± 0.36
NEX 2	CC @ 15s	35 ± 1.89	29.04 ± 0.36
NEX 2	CIC @ 10s	42 ± 3.29	25.9 ± 5.37
NEX 2	CIC @ 15s	48 ± 7.71	34.56 ± 8.29
NEX 3 L	UC @ 10s	23 ± 9.85	18.73 ± 7.05
NEX 3 L	UC @ 15s	28 ± 3.72	26.83 ± 7.59
NEX 3 L	CC @ 10s	28 ± 2.39	19.51 ± 2.44
NEX 3 L	CC @ 15s	30 ± 5.77	23.75 ± 1.52
NEX 3 L	CIC @ 10s	23 ± 3.22	24.5 ± 4.4
NEX 3 L	CIC @ 15s	28 ± 7.71	28.97 ± 11.43
NEX 6 S	UC @ 10s	49 ± 7.72	15.23 ± 7.67
NEX 6 S	UC @ 15s	40 ± 8.72	22.99 ± 10.72
NEX 6 S	CC @ 10s	65 ± 18.84	30.09 ± 12.56
NEX 6 S	CC @ 15s	61 ± 1.26	21.43 ± 2.06
NEX 6 S	CIC @ 10s	44 ± 4.57	17.45 ± 5.16
NEX 6 S	CIC @ 15s	46 ± 9.19	19.37 ± 8.06
WC S	UC @ 10s	41 ± 1.79	21.51 ± 2.29
WC S	UC @ 15s	36 ± 4.59	26.33 ± 6.34
WC S	CC @ 10s	35 ± 0.64	20.58 ± 0.74
WC S	CC @ 15s	40 ± 0.75	25.42 ± 1.22
WC S	CIC @ 10s	53 ± 16.74	23.09 ± 2.95
WC S	CIC @ 15s	44 ± 0.52	30.83 ± 5.98
NHop	UC @ 10s	41 ± 3.33	118.62 ± 25.82
NHop	UC @ 15s	36 ± 2.06	113.82 ± 37.36
NHop	CIC @ 10s	30 ± 1.23	20.68 ± 5.62
NHop	CIC @ 15s	27 ± 1.1	22.0 ± 8.64

Table 5: Recovery statistics under single failure conditions

drop-off will be steeper compared to 90% resource usage (10% headroom). Essentially this means that having more headroom results in a faster ability to catch up with the input stream.

In most tested configurations the rollback distance per worker is relatively similar between checkpointing protocols with a higher variance for UC and CIC configurations as their checkpoints are less aligned than for CC configurations. Based of this and the aforementioned observations, the expected behaviour post-recovery is that CC configurations will take most time to recover (i.e. have the highest recovery times). The intuition behind this is that the whole system is being rolled back and has to reprocess input since the restored global checkpoint, where the UC and CIC configurations only roll-back and replay input to workers affected by the failure, which theoretically is less work.

However the recovery time is highly variant between different experiment executions and even within repetitions of the same configurations. Especially UC/CIC configurations appear susceptible to differences between repetitions due to uncoordinated nature of taking checkpoints resulting in rolling back further in some executions than others. Nevertheless, some general observations can be made with respect to this metric.

First of all, for the continuous NEXMark 2 query, the CC configurations perform better than the UC and CIC configurations in terms of recovery time. This difference in performance was also observed in the performance experiments and this strengthens the intuition that the CC protocol yields the best performance on window-free query graphs. It may however also be attributed to the difference in performance headroom that the protocols are subjected to mentioned at the beginning of this section. To confirm this, an extra experiment should be ran where each protocol is tuned to run at 80% of *its own* maximum level of throughput and observe the effects on the recovery time under those configurations.

In the NEXMark 3 and WordCount queries, the difference in recovery time between the checkpointing protocols is negligible, making the choice of checkpointing protocol hardly observable in the collected data. Based of that, CC would be the best choice as there is no message loss and thus provides exactly-once processing which the UC and CIC protocols can not.

Lastly, for NEXMark 6 with small windows shows the best recovery times for UC and CIC protocols. The intuition for why this happens is that the multiple aggregates in the query graph take a reasonably large amount of time to reprocess the input from the beginning. As opposed to the UC and CIC approaches that in this query graph replay messages which are rapidly processed by being buffered in the aggregates. What should be noted is that the aforementioned message replay bug did affect this configuration and therefore a bigger gap in recovery was present than theoretically

Query	protocol	#lost messages	failed type	input rate (Ke/s)	mean throughput (Ke/s)	lost throughput (s)
NEX 2	UC @ 10s	14304 ± 9421	filter	45	22.6 ± 1.3	0.6
NEX 2	UC @ 15s	35948 ± 4512	filter	45	22.5 ± 1.4	1.6
NEX 2	CIC @ 10s	20547 ± 659	filter	45	21.6 ± 2.3	0.9
NEX 2	CIC @ 15s	32905 ± 586	filter	45	21.8 ± 2.4	1.2
NEX 3 L	UC @ 10s	173 ± 126	join	5.4	0.16 ± 0.1	1.1
NEX 3 L	UC @ 15s	368 ± 251	join	5.4	0.16 ± 0.1	2.3
NEX 3 L	CIC @ 10s	288 ± 25	join	5.4	0.16 ± 0.1	1.8
NEX 3 L	CIC @ 15s	529 ± 14	join	5.4	0.16 ± 0.1	3.3
NEX 3 S	UC @ 10s	915 ± 1411	join	25	4.9 ± 3.1	0.2
NEX 3 S	UC @ 15s	4345 ± 4874	join	25	4.8 ± 2.9	0.9
NEX 3 S	CIC @ 10s	6565 ± 1887	join	25	4.6 ± 3.3	1.4
NEX 3 S	CIC @ 15s	12241 ± 440	join	25	4.5 ± 2.8	2.7
WC L	UC @ 10s	34 ± 28	aggregate	6	0.013 ± 0.03	2.6
WC L	UC @ 15s	2 ± 0	aggregate	6	0.013 ± 0.03	0.2
WC L	CIC @ 10s	13 ± 0	aggregate	6	0.013 ± 0.03	1.0
WC L	CIC @ 15s	58 ± 0	aggregate	6	0.013 ± 0.03	4.5
WC S	UC @ 10s	331 ± 63	aggregate	17	0.13 ± 0.11	2.5
WC S	UC @ 15s	692 ± 88	aggregate	17	0.13 ± 0.11	5.3
WC S	CIC @ 10s	193 ± 0	aggregate	17	0.13 ± 0.13	1.5
WC S	CIC @ 15s	713 ± 62	aggregate	17	0.13 ± 0.14	5.5

Table 6: Lost message statistics under single failure conditions

necessary. Based off this the difference in recovery time may be less pronounced when reevaluating this configuration with the bug fixed, however due to the big difference in recovery time the intuition is that the same difference in recovery time will emerge, it will be less but likely still significant.

High recovery time deviation. is visible under 2 configurations in Table 5 and is caused by two outliers. The first is found in the WordCount S query under CIC @ 10s configuration where one of the executions suffered from a higher rollback distance, this caused more reprocessing to happen and thus cause bigger deviation in the recovery time. Secondly the NEXMark 6 S query under CC @ 10s configuration has a similar outlier that rolled back further than the other executions, again resulting in a bigger deviation in recovery time.

Other results also have some degree of variance in the recovery time and can be influenced by many factors. First and foremost, the time it takes to detect the failure, as FERDiS currently implements a timeout-based failure detection, the detection of the failure is not constant between experiment executions. Furthermore the resource allocation (or pod placement) performed by Kubernetes may differ and affect the performance of the system or noisy neighbours [15] in the virtualised environment may hog resources affecting the performance of individual processes in the distributed system. Such factors can affect the processing headroom the system has available, potentially affecting aforementioned steepness of the latency drop-off after recovery, which finally can reflect in the recovery time metric. To validate this intuition requires running more experiments, potentially even on an isolated private cloud, to see the effects on the recovery time metrics. In any case it is important to keep such factors in mind while interpreting these results.

Discussing the recovery time metric. What the recovery time metric does not encompass, is that while in the NEXMark 2 query UC and CIC have higher recovery times per its definition, they do present the ability to continue producing low latency results after the failure and during recovery. The CC configurations on the other hand abruptly produces *only* high latency results until the system has caught up with the input stream. This means during the entire recovery period, CC configurations produce high latency results, where UC and CIC configurations produce mixed high and low latency results. Based of this observation, one may consider that the UC and CIC protocols provide better results on the latency spectrum during failures. So while the recovery time may be higher in some configurations for the UC and CIC protocols, this does not mean that they present the same degree of reprocessing during this period.

A last note with respect to the recovery time metric, is that for the NEXMark 5 query, almost all the experiment executions would result in a 0 second recovery time. As per the definition, the recovery time is the first moment after a failure where the system's mean latency falls back within 10% of its pre-failure mean latency. Since the NEXMark 5 query has naturally high latencies due to the many buffering layers (3 windowed operators in sequence), the failures would often not even increase the mean latencies by 10%. This means the definition of the metric is best suited for continuous, low latency queries such as the NEXMark 2 and NHop queries, but does not seem to fit very well with higher latency queries such as NEXMark 5. With this in mind, a redefinition of the metric may need to be considered which is adaptive to the base levels of latency a query exhibits.

9.3.3 Recovery in a cyclic query. In the tested cyclic query a failure has significantly different effects on the two tested protocols, UC

and CIC. That is, throughout all executed runs, the UC protocol was affected by the domino effect, for both tested checkpoint intervals. The domino effect was consistently observed throughout repeated experiment executions, rising the intuition that either this particular query graph or cyclic stream processing in general is highly susceptible to this effect, which would make uncoordinated checkpointing unfeasible to use in cyclic stream processing in general. CIC avoids this effect completely as the protocol is designed to do so. The effects on the recovery behavior of the system are much as expected with the aforementioned observations in mind, UC has post-failure latencies as high as the total length of the warm-up period, that is roughly 130 seconds, where the post-failure latencies of CIC are in line with the operator's rollback distance, which is roughly 30 seconds. A higher spike in throughput is observed in UC, as under that configuration many more messages have to be reprocessed as opposed to CIC. This behaviour is visualised in Figure 30.

10 FUTURE WORK

Most effort in this work was devoted in the design and development of FERDiS and while it supports the basic required functionality to achieve the research goals, it requires further development cycles to produce results more akin to state-of-the-art stream processing systems. This can be interpreted from different perspectives. For example, more advanced stream processing functionality may be included in the processor, a prime example would be event-time processing and handling out-of-order input elements in the stream. From a performance perspective the system currently performs relatively poorly and needs to be optimized to be competitive with existing stream processing engines, this is especially true for windowed operators which have shown to sustain relatively low levels of throughput, but also serialisation was identified as a big contributor to the relative low performance of the system. Lastly from a stability perspective, some system components require improvement to be more reliable. One such example is the implementation of post-failure reinitialisation which currently can cause a deadlock within the coordinator.

While these first results indicate some potentially interesting patterns in behavior, executing and analysing more experiments including more or different metrics would yield more insight in the trade-offs between the protocols. With event-time processing implemented, event-time latency may be considered over processing-time latency. This would allow gaining deeper insight in the effects of these checkpointing protocols with respect to the characteristics of the output stream. Following up on the unexpected patterns in the results would allow verifying or debunking the observed patterns. These include the lower latencies for CIC in the WordCount query and the unstable throughput of CC in the NEXMark 3 query with large windows.

Lastly, after solidifying the observation that relatively low number of messages are lost in aggregate and join operator failures and the input buffering effects observed in the NEXMark 3 query with more data, a new checkpointing protocol could potentially be devised tailored specifically to stream processing engines. With the topology of the query graph and characteristics of the input streams as input, the protocol could decide on a hybrid CC and CIC approach that leverages the benefits of both approaches, that is to a) avoid the input buffering effects observed in the NEXMark 3 query, b) support the fastest possible recovery c) naturally support cycles and most importantly d) still support exactly-once semantics.

11 CONCLUSIONS

With the design and implementation of FERDiS, which we elaborate in section 7 we answer **RQ1** *How can a stream processing system support each of the three classes of checkpointing algorithms: coordinated checkpointing, uncoordinated checkpointing and communication-induced checkpointing?*) and **RQ1.1** (*Which specific variants of the three classes of fault tolerance algorithms are suitable to implement in a stream processing system?*).

The answer to **RQ2** (*How does the choice of checkpoint-based recovery algorithm affect the consistency gurantees, runtime efficiency, recovery efficiency, and support of cyclic dataflows?*) is highly nuanced.

Regarding consistency guarantees (**RQ2.1**), theoretically all protocols can support exactly-once processing, however practically this is unfeasible due to the requirement of checkpointing the entire message log, resulting in extremely expensive checkpoints, which deteriorates the runtime performance of the system to the point of uselessness. So practically CC supports exactly once while UC and CIC support gap recovery or at-most-once, where the observed gap has been quantified as part of the recovery statistics and is seemingly low. Meaning one may consider these protocols to yield 99% consistency.

Regarding runtime efficiency (**RQ2.2**), the more meta-data needs to be passed along with messages, the lower the runtime performance of the system becomes. Given that CC requires no metadata, UC only sequence numbers and CIC a package of metadata, the performance deteriorates in this order. CC results in the highest runtime performance, followed by UC, then CIC. Notably, this only considers query graphs that do not contain window operators. In the tested queries, aggregates cause the performance differences to become negligible and in one query CC performed far less stable than UC and CIC would. This means that there is no single answer for the most efficient checkpointing protocol as the query graph is of heavy influence on it. For queries that do not contain windowed operators however, CC is clearly the most efficient choice.

Regarding recovery efficiency (**RQ2.3**), in a reasonable amount of query graphs, the differences in recovery time between the protocols were not substantial. However in window-free query graphs, CC seems to perform best. Then again, with multiple aggregates in place UC/CIC seems to perform better, at the cost of some inconsistency, which has been quantified and, especially with windows in place, is surprisingly small. Overall the data indicates that the query graph is of influence on the performance of the recovery protocol.

Regarding the support of cyclic dataflows (**RQ2.4**), notably, CC protocols do not support cycles at all, while UC and CIC protocols do so. While some systems, such as Flink, have applied tricks to add cycle support to coordinated checkpointing protocols, their mechanisms are entirely or highly similar to those used by UC and CIC that allow the use of cycles, that is, via logging messages.

The conclusion with respect to (**RQ2**) is that the effects of the different checkpointing protocols are versatile and both influence the query graph they can be employed on and are influenced by the query graph they are employed on. Delivery guarantees were lower for UC and CIC approaches in our experiments however while log checkpointing does not seem practically feasible it may still be feasible for joins and aggregates given their generally low throughput and lost-message counts. Furthermore, the classic upstream backup approach may also be applied by having replicas of operators to take over when a failure occurs, resulting in better delivery guarantees at the cost of increased resource usage but with the added benefit of high availability.

The results show better performance of UC and CIC approaches under specific conditions in the query graph. This gives rise to the idea that a hybrid approach between CC and UC/CIC may perform better than CC alone under the right circumstances. This may be a very interesting direction for future research.

Lastly, while not visible in the data, the implementation of the CC protocol was easier than the UC and CIC approaches as these required distributed mechanisms such as sequence number tracking and message log pruning based on these sequence numbers. These extra complexities introduced extra margin for implementation errors, which were less present in the CC protocol. While the CC protocol had its own complexities, these largely revolved around the input blocking mechanism, which was isolated within individual operators making it easier to check for correctness. This made the CC protocol easier to develop and maintain than the UC protocol and particularly the CIC protocol.

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The Design of the Borealis Stream Processing Engine.. In *CIDR*, Vol. 5. 277–289.

[2] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful functions as a service in action. In *VLDB*. 1890–1893.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. In *VLDB*. 1033–1044.

[4] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. 2015. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *International Semantic Web Conference*. Springer, 374–389.

[5] Lorenzo Alvisi, Elmootazbellah Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. 1999. An analysis of communication induced checkpointing. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*. IEEE, 242–249.

[6] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2004. Stream: The stanford data stream management system. *Book chapter* (2004).

[7] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 480–491.

[8] Magdalena Balazinska, Jeong-Hyon Hwang, and Mehul A Shah. 2009. Fault-Tolerance and High Availability in Data Stream Management Systems. *Encyclopedia of Database Systems* 11 (2009), 57.

[9] Daniele Briatico, Augusto Ciuffoletti, and Luca Simoncini. 1984. A Distributed Domino-Effect free recovery Algorithm.. In *Symposium on Reliability in Distributed Software and Database Systems*, Vol. 84. 207–215.

[10] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2003. Automated application-level checkpointing of MPI programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 84–94.

[11] Guohong Cao and Mukesh Singhal. 1998. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 9, 12 (1998), 1213–1225.

[12] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. In *VLDB*. 1718–1729.

[13] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *SIGMOD*. 2651–2658.

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink TM: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[15] Eddy Caron and Jonathan Rouzaud Cornabas. 2014. Improving users' isolation in iaas: Virtual machine placement with security constraints. In *2014 IEEE 7th International Conference on Cloud Computing*. IEEE, 64–71.

[16] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, and James F. Terwilliger. 2015. Trill: Engineering a Library for Diverse Analytics. *IEEE Data Eng. Bull.* 38 (2015), 51–60.

[17] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, et al. 2003. Telegraphcq: Continuous dataflow processing for an Uncertain world.. In *Cidr*, Vol. 2. 4.

[18] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[19] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. 2003. Scalable Distributed Stream Processing. In *CIDR*.

[20] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* 34, 3 (2002), 375–408.

[21] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *SIGMOD*. 725–736.

[22] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. A Survey on the Evolution of Stream Processing Systems. arXiv:2008.00842 [cs.DC]

[23] Richard Gass and Bidyut Gupta. 2001. AN EFFICIENT CHECKPOINTING SCHEME FOR MOBILE COMPUTING SYSTEMS.

[24] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Hazelcast Jet: Low-latency Stream Processing at the 99.99th Percentile. arXiv:2103.10169 [cs.DC]

[25] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. 2005. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE, 9–9.

[26] B Gupta, SK Banerjee, and B Liu. 2002. Design of new roll-forward recovery approach for distributed systems. *IEE Proceedings-Computers and Digital Techniques* 149, 3 (2002), 105–112.

[27] Bidyut Gupta, Shahram Rahimi, and Ziping Liu. 2007. Novel low-overhead roll-forward recovery scheme for distributed systems. *IET Computers & Digital Techniques* 1, 4 (2007), 397–404.

[28] J-M Hélary, Achour Mostefaoui, Robert HB Netzer, and Michel Raynal. 2000. Communication-based prevention of useless checkpoints in distributed computations. *Distributed Computing* 13, 1 (2000), 29–43.

[29] Gabriela Jacques-Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. 2016. Consistent Regions: Guaranteed Tuple Processing in IBM Streams. In *VLDB*. 1341–1352.

[30] Hai Jin, Fei Chen, Song Wu, Yin Yao, Zhiyi Liu, Lin Gu, and Yongluan Zhou. 2018. Towards low-latency batched stream processing by pre-scheduling. *IEEE Transactions on Parallel and Distributed Systems* 30, 3 (2018), 710–722.

[31] Asterios Katsifodimos and Marios Fragkoulis. 2019. Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications.. In *EDBT*. 682–685.

[32] Athanasios Kiatipis, Alvaro Brandon, Rizkallah Touma, Pierre Matri, Michal Zasadzinski, Linh Thuy Nhuyen, Adrien Lebre, and Alexandru Costan. 2019. A Survey of Benchmarks to Evaluate Data Analytics for Smart-* Applications. *arXiv preprint arXiv:1910.02004* (2019).

[33] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. 2015. Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*. 2785–2792. https://doi.org/10.1109/BigData.2015.7364082

[34] Dirk Koch, Christian Haubelt, and Jürgen Teich. 2007. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. 188–196.

[35] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications* (1978).

[36] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM international conference on computing frontiers*. 1–8.

[37] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. 2014. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE, 69–78.

[38] Diana Matar. 2016. Benchmarking Fault-Tolerance in Stream Processing Systems. *Master's thesis. TU-Berlin* (2016).

[39] Radu Prodan and Thomas Fahringer. 2008. Overhead analysis of scientific workflows in grid environments. *IEEE Transactions on Parallel and Distributed Systems* 19, 3 (2008), 378–393.

[40] Shilei Qian, Gang Wu, Jie Huang, and Tathagata Das. 2016. Benchmarking modern distributed streaming platforms. In *2016 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 592–598.

[41] Brian Randell. 1975. System structure for software fault tolerance. *Ieee transactions on software engineering* 2 (1975), 220–232.

[42] Ibrahim Sabek, Badrish Chandramouli, and Umar Farooq Minhas. 2019. CRA: Enabling Data-Intensive Applications in Containerized Environments. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1762–1765.

[43] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. 2017. Riotbench: a real-time iot benchmark for distributed stream processing platforms. *arXiv preprint arXiv:1701.08530* (2017).

[44] Pedro F Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*.

[45] Khin Me Me Thein. 2014. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research* 3, 47 (2014), 9478–9483.

[46] Michael Treaster. 2005. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *arXiv preprint cs/0501002* (2005).

[47] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark–A Benchmark for Queries over Data Streams (DRAFT)*. Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers.

[48] Y-M Wang and W Kent Fuchs. 1993. Lazy checkpoint coordination for bounding rollback propagation. In *Proceedings of 1993 IEEE 12th Symposium on Reliable Distributed Systems*. IEEE, 78–85.