DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Benchmarking checkpointing algorithms in Stream Processing Engines

Author:
Gianni WIEMERS

Supervisor:
Dr. Asterios KATSIFODIMOS

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

August 24, 2023

Declaration of Authorship

I, Gianni WIEMERS, declare that this thesis titled, "Benchmarking checkpointing algorithms in Stream Processing Engines" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 24-08-2023

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Benchmarking checkpointing algorithms in Stream Processing Engines

by Gianni WIEMERS

The use of data streams has increased a lot over the last two decades or so. and With this increase comes the need for fast and consistent fault recovery. Rollback recovery mechanisms from traditional distributed systems have been adapted successfully for stream engines. These mechanisms can be categorized into one of three different categories; uncoordinated, coordinated and communication induced protocols. While most well-known stream engines implement a variant of the coordinated Chandy-Lamport algorithm, there is no practical comparison available that actually confirms whether this is the optimal solution for data streams specifically. Compared to traditional distributed processing solutions, stream processing has a higher need for low latencies due to the continuously generated input. This paper aims to create more insight into the advantages and disadvantages of these solutions by implementing a checkpointing algorithm for each of these categories. These are then benchmarked using various workloads and evaluated using a number of metrics such as latency, throughput, recovery times and network overhead. From these results, it can be concluded that a coordinated approach indeed outperforms uncoordinated solutions across all of these metrics, most likely due to the need for message logging in both the uncoordinated and communication induced scenarios. Additionally the benchmarks indicate that the overhead of the communication induced approach does not outweigh its benefits, due to the rarity of the occurrence of the so-called domino effect.

Acknowledgements

This research has been done over a period of approximately 10 months and would not have been possible without the help and support of a lot of different people. First of all I would like to thank Dr. Asterios Katsifodimos for giving direction to my thesis and being my responsible professor, providing me with useful insights along the way. I want to thank Dr. Jérémie Decouchant for being part of my thesis committee and taking the time to take a look at my thesis. Throughout the entirety of the project I had regular meetings and worked together with my supervisor, PhD student Georgios Siachamis, whose input and help was greatly appreciated and I enjoyed the collaboration. The implementations for this thesis were done on top of a basic streaming engine provided by Kyriakos Psarakis, who also regularly provided us with technical support and feedback and we had several meetings with Dr. Marios Fragkoulis and Dr. Paris Carbone, who also gave us some feedback and interesting ideas. Besides all the people involved with the project itself, I would like to thank my family and friends, who were always supportive, but maybe even more importantly also made, not only my thesis, but my entire time at Delft university of technology a lot more fun.

Contents

Chapter 1

Introduction

Distributed computations have been used for a while now and a lot of research has been done on these types of systems. One important aspect of these systems, depending on the application, is fault recovery. Whenever a failure occurs in the system it can be quite time consuming, and thus inefficient, to completely start over from the beginning. To mediate this issue there are multiple possible solutions, such as replication or checkpointing. This paper focuses specifically on the latter. Checkpointing algorithms can be divided into three different categories; uncoordinated checkpointing, communication induced checkpointing and coordinated checkpointing. Fault recovery using any of these checkpointing techniques poses another quite interesting challenge which has to do with the consistency of the system. For example, in a transactional system it would be quite problematic if some of the transactions were lost or executed twice, since this could lead to huge financial losses. Therefore exactly once processing has to be guaranteed, to make sure this fault recovery is done in a correct way.

Over the past two or so decades the usage of distributed stream processing has increased tremendously and, while some of the checkpointing protocols from the distributed computations literature have been adapted effectively by stream engines such as Flink[6], IBM Streams[22], Hazelcast Jet[23] and Risingwave[12], there still is a lack of practical comparisons between the different checkpointing techniques within streaming systems specifically. Additionally, the protocol that forms the basis for most of the checkpointing methods used in those streaming systems, the Chandy Lamport algorithm, does not support any dataflows containing cycles. Due to the marker based checkpointing this protocol uses, it will deadlock if any cycles are present. Furthermore the choice of checkpointing algorithm can have a substantial effect on factors such as network usage, storage and throughput. To hopefully fill this research gap, for this paper one checkpointing algorithm for each of the three different types discussed in the previous paragraph has been implemented in a basic in-house stream processing engine. By benchmarking these algorithms under different workloads, the goal is to create a more conclusive practical comparison between these algorithms.

As explained, this thesis paper presents a practical evaluation of three different types of checkpointing algorithms in an in-house stream processing engine. The paper is structured as follows; first, related work will be discussed in chapter 2 to give a better impression of the current state regarding checkpointing in stream processing engines and the benchmarking thereof. In chapter 3 some theory and background knowledge necessary for understanding the algorithms and how fault recovery works will be presented. Afterwards, the specific implementation details will be discussed more in-depth in chapter 4. Lastly in chapter 5 and 6 an extensive explanation of the benchmarking setup can be found, followed by the results and a discussion thereof.

Chapter 2

Related Work

With the use of distributed computations comes the need for fault recovery. This subject has been researched extensively over the years [13][35][3]. The two most commonly used fault recovery techniques are replication [32] and rollback recovery using checkpoints (also referred to as snapshots) [26][7]. Originally, these protocols were designed for systems that execute a certain task on a set dataset using a set amount of machines for the entire execution. However, due to changes in the applications of distributed computations, such as mobile computing systems, scalable systems and data streams, new challenges were introduced.

Mobile Computing Systems – Various papers discuss why the limited bandwidth and long latencies of mobile computing systems often cause traditionally used protocols to be ineffective in this setting [24][1]. As stated by [1], the often used Chandy-Lamport algorithm [9] only works if FIFO channels are guaranteed between the workers, which is something that may not naturally occur between the mobile hosts. One solution would be to enforce FIFO messaging, however that could dramatically reduce system performance. To mediate this negative effect on system performance, the authors propose a alternative solution. By enforcing the FIFO property only between marker and application messages, the performance won't be impacted as much compared to when FIFO is enforced between any two messages. This guarantee is enough to enable support for algorithms such as Chandy-Lamport.

Scalable Systems – To ensure high performance in scalable systems, various improvements for existing protocols have been proposed. One paper tries to achieve this by reducing the checkpoint size [10]. For this purpose three commonly used methods are discussed. First of all, live variable analysis is applied to avoid storage of unused (dead) variables. Secondly, incremental checkpoints can be used in combination with full checkpoints. Incremental checkpoints only contain the changes compared to the full checkpoint in this case, instead of the entire state. Lastly data compression is used to get rid of any redundant information. Another paper proposed three different algorithms (grid-based, tree-based and centralized) to reduce both the amount of messages necessary to execute the snapshot protocol, as well as the amount of space that is necessary for these messages [16].

Streaming Systems – Long checkpointing and recovery times can become quite problematic when processing continuously generated data. Not every stream processor needs very strict recovery protocols however, since loss of information can be tolerated in some cases. One paper specifically defined and analyzed three different recovery guarantees with corresponding recovery protocols to compare their runtime overhead and recovery times [16], showing that each approach covers a complementary part of the solution space. The most challenging consistency guarantee to achieve is exactly-once processing, which therefore also yields the highest runtime overhead. Various performance improvements for streaming systems without compromising consistency, similar to the ones for scalable systems, have been

proposed. Most of them relying on some variation of incremental checkpointing or a reduction in the blocking behaviour of the checkpointing approach [33][25].

Benchmarking Stream Processors – Distributed stream processing, although similar to traditional distributed computing, might require some different performance guarantees. Quick recovery times, as well as low latency, are often of bigger importance, due to the never ending generation of data. This has sparked research about how to properly test the performance of these types of systems. A well-known, though relatively old benchmark is NEXMark[36]. NEXMark is an adaption of a benchmark used to measure performance of XML repositories (XMark). NEXMark now contains a total of 12 different queries that each try to test a different kind of behavior. Quite a lot of different benchmarking approaches have been proposed over the years [15][5][34][28], some focusing specifically on the streaming workloads, while others tend to focus more on the type of metrics that are being collected. Especially when dealing with a system with strong consistency guarantees, metrics that incorporate fault recovery behavior are of higher importance.

Chapter 3

Checkpointing algorithms & Guarantees

Fault recovery for distributed computations using checkpoints requires not only the checkpointing algorithms themselves but, depending on the algorithm choice, also some additional protocols, for example message logging or finding a consistent recovery line. This chapter provides an in-depth overview of the seminal checkpointing algorithms found in literature and of all the necessary additional concepts required.

To explain certain scenarios that can occur within the system, at various points in this chapter, a visual representation of a streaming system will be used. Figure 3.1 illustrates such a representation of a streaming system. Each process is represented by a timeline, denoted as a horizontal arrow (i.e. P_1). The arrows between the processes represent messages and the nodes on the processes represent the checkpoints (i.e. $C_{1,1}$). The interval between two checkpoints of the same process is referred to as the checkpointing interval (i.e. $I_{1,2}$).

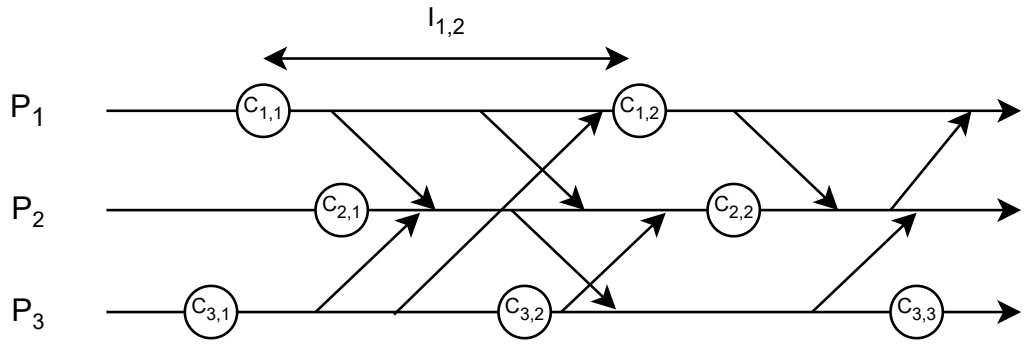

FIGURE 3.1: A basic distributed streaming system.

3.1 Fault recovery

When working with distributed computations, there is a lot that can go wrong and there are also a lot of different ways to deal with these issues when they happen. This section will briefly go over the different types of failures and the most commonly used recovery strategies.

3.1.1 Types of failures

There are three common types of failures that need to be considered when working with distributes systems. **Software crashes** can happen for all sorts of reasons. In most cases simply restarting a worker can resolve these issues. It is in this case important however that all the information necessary for a successful recovery is stored on persistent storage. **Hardware failures** do often not have an easy fix, the quickest way is to let another machine take over. Due to this reason, data should be replicated to prevent it from being lost. **Network failures** can consist of simple lost messages that need to be replayed or completely disconnected workers. In case of the latter, simply spinning up a new worker to replace the disconnected one is often the fastest solution.

3.1.2 Fault recovery strategies

In general, fault recovery can be divided into four different types [27];

Retry – Failures are handled by restarting the job from scratch. All state is discarded and the restarted job has to start by processing the data from the beginning. This can be quite time-consuming when the job a)employs heavy computations, b)runs for long periods of time (e.g., hours or days), and/or c)failures happen regularly. This recovery strategy requires that all data is stored and available until the job has finished. In reality, since streaming applications are usually long-running jobs, storing all data for the lifetime of the job is infeasible, and restarting processing from scratch leads to serious duplication of data and computations, as well as outdated replayed output.

Replicate – The entire worker state is replicated multiple times. Whenever a failure occurs one of these replicas can take over, while the failed worker recovers. Although in most cases very effective, this strategy has two major downsides. First of all a protocol needs to be in place that decides which replica takes over in case of a failure and how the recovery of the failed worker is handled. The second downside comes from the fact that every replica needs to be in exactly the same state at all times to prevent inconsistencies when a failure occurs. This means that not just one worker, but also every replica, needs to be updated before execution can continue, having quite an effect on throughput. This last point can become especially problematic when you need a high-throughput streaming system.

Checkpoints – The entire system state is stored as a checkpoint on stable storage. Whenever a failure occurs, the state of the system can be reset to the last available checkpoint and computations can continue from that point onward. This eliminates the need to start from scratch. There is some overhead however when the workers have to recover to older worker states and when these checkpoints need to be taken. Protocols also need to be in place to coordinate the creation of this global state and these protocols block execution in the system to guarantee consistency. Additionally a lot of storage might be needed when dealing with big worker states.

Message logs – Messages within the system are logged to a stable storage. In case of a failure these messages can be replayed in the same order to arrive at the same worker state as before the failure. This can be used in combination with checkpoints to achieve a consistent state, meaning that the checkpoint coordination for a consistent global state is no longer necessary. The trade off in this case is between a protocol that blocks execution to capture a global state and the logging of every message in the system.

In this work specifically checkpoints and message logs will be considered, since these are the methods used by most stream engines and this strategy offers a good balance between recovery times and the effect on the throughput. In the next section the different consistency guarantees will be elaborated on, after which message logs and checkpoints will be discussed further.

3.1.3 Consistency guarantees

One important thing to take into account when working with fault recovery is the consistency guarantees that are required. In general a distinction can be made between three different types of consistency; at least once, at most once and exactly once processing [14]. At least once processing means that messages might be processed multiple times. At most once, on the other hand, means that some messages might be lost in the process. When exactly once processing is guaranteed, no messages will be lost or duplicated. Important to note is that there are two ways to interpret exactly once processing [14]. Either the state or output could be considered for this purpose. To illustrate the importance of these consistency guarantees, consider a transactional system. Whenever at least once processing is guaranteed in such a system, money could be generated out of thin air, since whenever person A sends money to person B, person B might receive it multiple times. With an at most once guarantee on the other hand, there might be a chance that person B receives nothing at all, while person A did send the money. In this example both scenarios are extremely problematic, which shows why such a strong exactly once processing guarantee is necessary in some cases. To achieve this, a combination of log based and checkpoint based recovery can be used.

3.1.4 Log based recovery

Log based recovery, as mentioned shortly before, describes the process of logging messages when they are sent and replaying those from a certain point onward when a failure occurs. When combining log based recovery with checkpoints, the necessity of these logs depends on the protocol. For example, when a blocking version of the coordinated checkpointing approach is used, the checkpoints will be taken in such a way that no messages are in transit. Therefore there is no need for message logs. However, when checkpoints are taken in an uncoordinated manner, there is no way to guarantee that there are no messages in transit. To prevent those messages from being lost, logs are necessary. It is, however, not as simple as just replaying all the message logs from a certain point after recovery. To illustrate this, imagine the scenario as given in figure 3.2 where message m_2 follows as a result of processing message m_1 (scenario a). When both of these messages are logged and replayed upon failure (scenario b) message m_2 will now be processed twice, due to the fact that it is replayed from the logs and naturally follows after processing of the replayed message m_1. To prevent this from happening, it is important to somehow keep track of these causal messages. There are multiple ways to ensure that messages are only sent, and thus processed, once. How this is done in the proposed solution is discussed in section 4.3.

3.2 Checkpoint based recovery

Checkpoints can be described as intermediate results. They are written to stable storage, such that if a failure occurs a worker can continue from that intermediate

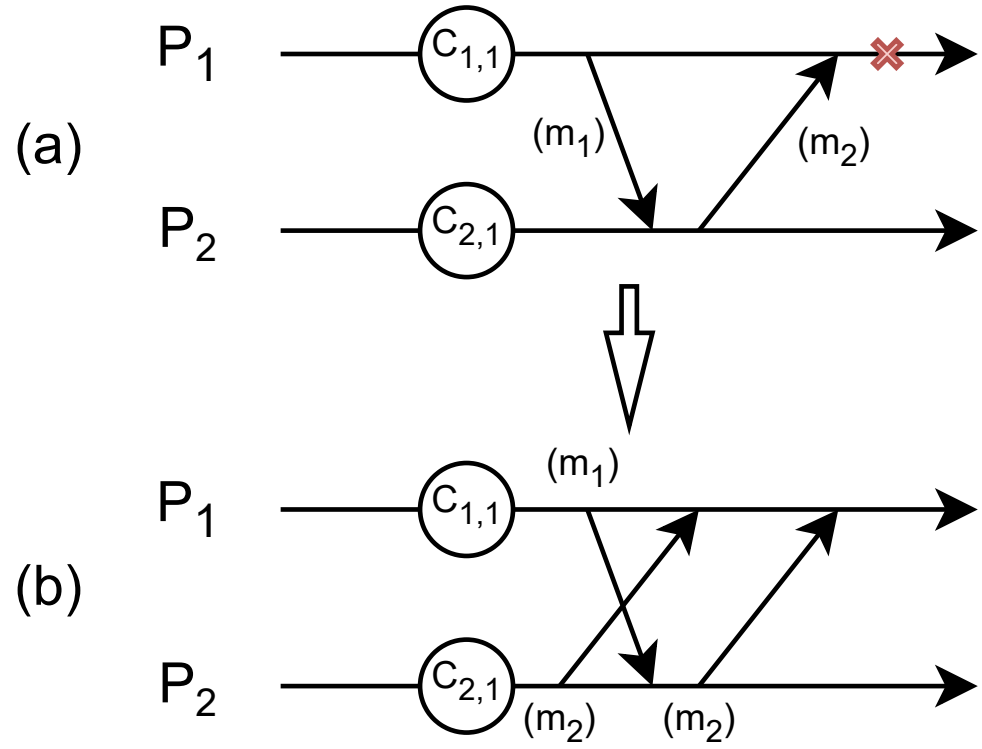

FIGURE 3.2: A scenario in which all logged messages are replayed on failure.

result onward instead of starting over from the beginning. There are quite a lot of checkpoint based recovery protocols, which can be divided into three different categories; uncoordinated, coordinated and communication induced protocols. The workings of these algorithms, their flaws and necessary additional protocols, will be discussed in this section.

3.2.1 Coordinated checkpointing

In case of the coordinated checkpointing approach workers explicitly communicate about the checkpoints and coordinate their creation. The most well-known and widely adapted coordinated algorithm is the Chandy-Lamport algorithm [9]. The algorithm consists of checkpointing rounds that each capture a consistent global state. In literature both a non-blocking, as well as a blocking variant of this algorithm can be found, each having their own advantages and disadvantages [11]. For the proposed solution a blocking variant [31] has been implemented, since it uses no messages logs. The rounds of this specific algorithm work as follows; one or more of the workers initiate the checkpointing round. Once they have finished creating their checkpoints, markers will be sent over all their outgoing channels. Whenever a worker has received a marker over all of the incoming channels, it will, again, take a checkpoint and propagate the marker to its own outgoing channels. The round is done once all of the workers have taken a checkpoint. Whenever a marker is received, all incoming messages on that channel will be buffered, until all markers

have been received and a checkpoint has been taken, after which execution continues.

Important to note is that this algorithm, due to its blocking nature, only works in acyclic execution graphs [8]. Cyclic dataflows would cause this algorithm to deadlock, since at least one of the incoming channels for one of the workers would depend on at least one of its outgoing channels. This would mean that a marker could never be received on this incoming channel, causing the worker to block execution forever.

Once a round of this algorithm has been completed, this set of checkpoints contains a consistent global state, without any need for message logs.

3.2.2 Uncoordinated checkpointing

On the other end of the spectrum there are uncoordinated checkpointing protocols, which can be seen as the simplest of the three protocols. Workers determine by themselves when checkpoints should be taken. This can be done in a variety of ways, most common is a periodic algorithm that takes a checkpoint on a set time interval, but it could also be done every certain number of operations, for example. One important thing to note is that there is no communication between the workers about the checkpoints whatsoever, in contrary to coordinated checkpointing.

Orphan messages

One way inconsistencies can be introduced upon recovery, when using an uncoordinated approach, is through orphan messages. Orphan messages are messages that, if the system was to be restored from a certain set of checkpoints, have been processed, but haven't yet been sent at the point of recovery. An example of such a scenario can be found in figure 3.3, if the system was restored from the latest possible checkpoints ($C_{1,2}$ and $C_{2,1}$). In this case, the message would be replayed, meaning that it would be processed twice, causing the consistency guarantee to loosen to at least once processing. Therefore it is important that these types of messages are detected, such that a set of checkpoints can be used that would maintain a consistent state (in case of the example; checkpoints $C_{1,1}$ and $C_{2,1}$).

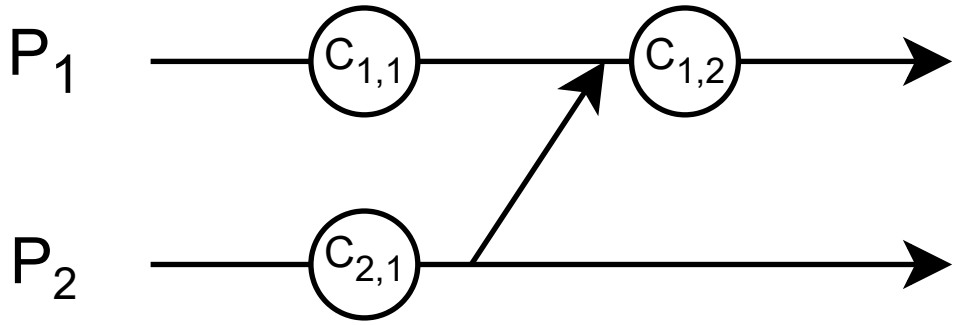

FIGURE 3.3: An orphan message from process P_2 to P_1.

Recovery graph and line

Finding a set of checkpoints that maintains a consistent state after recovery can be done through finding a recovery line within a recovery graph. Checkpoints that introduce orphan messages cause inconsistencies and can therefore never be part of a recovery line. These checkpoints are also referred to as useless checkpoints (checkpoint $C_{1,2}$ in figure 3.3). There are two different ways to build a recovery graph in which a recovery line can be found [13]. The first one is based on the rollback-dependency graph [4]. This graph can be built as follows:

- Let each checkpoint be represented by a node in the graph
- Draw an edge from $C_{i,x}$ to $C_{j,y}$, if $i \neq j$ and a message m is sent from $I_{i,x}$ and received in $I_{j,y}$
- Or draw an edge from $C_{i,x}$ to $C_{j,y}$, if $i \neq j$ and $y = x + 1$

An example of the rollback dependency graph can be seen in figure 3.4 (b). The name of the graph stems from the fact that if an edge is present between nodes $C_{i,x}$ and $C_{j,y}$ and $I_{i,x}$ has to be rolled back, $I_{j,y}$ should also be rolled back. The recovery line can easily be found by applying a reachability analysis on the resulting graph from the points of failure, marking all nodes encountered along the way. The recovery line then consists of the last unmarked nodes for every worker within the graph.

Another option would be the checkpoint graph [38], this graph is built in a slightly different way (note the small difference in the second step):

- Let each checkpoint be represented by a node in the graph
- Draw an edge from $C_{i,x-1}$ to $C_{j,y}$, if $i \neq j$ and a message m is sent from $I_{i,x}$ and received in $I_{j,y}$
- Or draw an edge from $C_{i,x}$ to $C_{j,y}$, if $i \neq j$ and $y = x + 1$

An example of the checkpoint graph, built using the same execution graph as used for the rollback dependency graph, can be found in figure 3.4 (c). For this graph the rollback propagation algorithm [38] should be used to find the recovery line. Note that both approaches are equivalent and will yield the same recovery line. This checkpoint graph is the recovery graph used in the proposed solution.

Domino effect

The domino effect [30] can be described as a chain of orphan messages that cause multiple consecutive checkpoints to become useless. An example of this effect can be seen in figure 3.5. The initial recovery line consists of the last available checkpoints for every process ($C_{1,2}$ and $C_{2,2}$). However, there is an orphan message present in this recovery line, meaning that checkpoint $C_{1,2}$ becomes useless. This shifts the recovery line to checkpoints $C_{1,1}$ and $C_{2,2}$, causing a similar situation. Therefore checkpoint $C_{2,2}$ now becomes useless as well. This continues all the way until the start of the processes. In this extreme case all of the checkpoints taken during the computation have become useless, which clearly illustrates why the domino effect can be quite problematic.

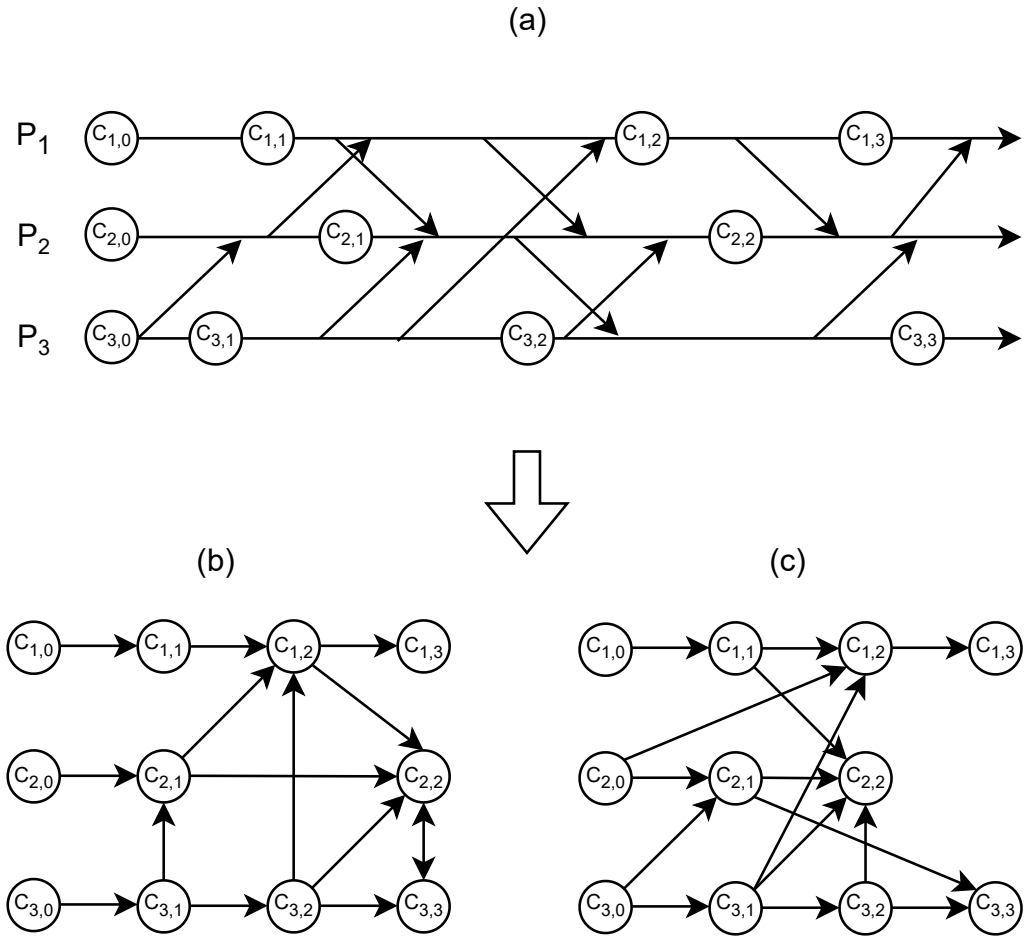

FIGURE 3.4: a) execution graph, b) rollback dependency graph, c) checkpoint graph.

3.2.3 Communication induced checkpointing (CIC)

Communication induced checkpointing approaches are used to prevent domino effects from occurring. Various implementations of the CIC approach can be found in the literature [37][21][18], however they all rely on the detection and prevention of Z-cycles [29] by taking forced checkpoints[17]. Z-paths, short for zigzag paths, are special sequences of messages that connect checkpoints. An example of such a path can be found in figure 3.6, where messages $\{m_3, m_4\}$ and $\{m_2, m_4\}$ connect checkpoints $C_{1,1}$ and $C_{3,3}$. Note that only messages originating from the same checkpointing interval as the previous one arrived in, may be considered to create this sequence. Z-cycles are defined as Z-paths that connects a checkpoint to itself, an example of this can be found in figure 3.7, for checkpoint $C_{2,1}$, where the Z-path consists of messages $\{m_2, m_1\}$. These Z-cycles in particular are very interesting for the CIC approach, since it has been proven that checkpoints can only be part of the recovery line if, and only if, they do not lie on such a cycle [29]. Therefore detecting and preventing these cycles can reduce the amount of useless checkpoints, and thereby prevent the domino effect.

Whenever a cycle is detected on reception of a message, a checkpoint is forced before processing that message. It is important to note that there is, in this case, a

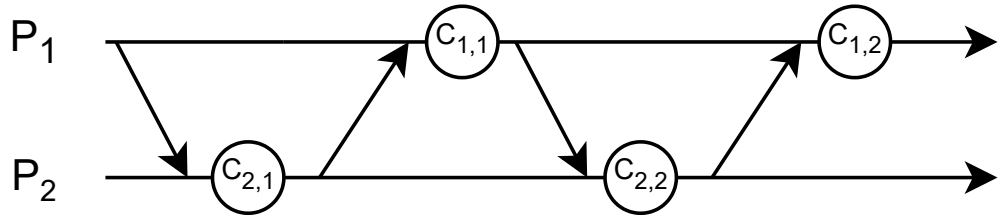

FIGURE 3.5: The domino effect.

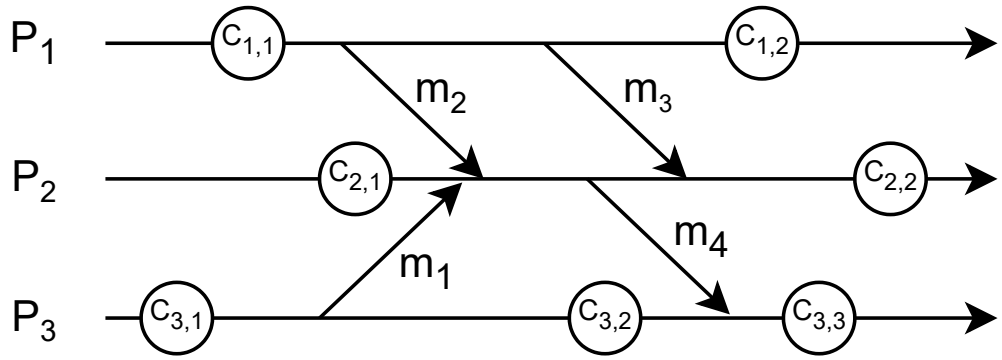

FIGURE 3.6: Execution graph containing multiple Z-paths.

clear distinction between the reception of a message and the so-called delivery of a message. The checkpointing happens in between of these two stages. An example can be found in figure 3.8.

Traditionally, CIC algorithms were categorized as either index-based or model-based protocols [13], which have been proven to be fundamentally equivalent [20]. Index-based protocols do however tend to create less forced checkpoints [2], which will yield better performance.

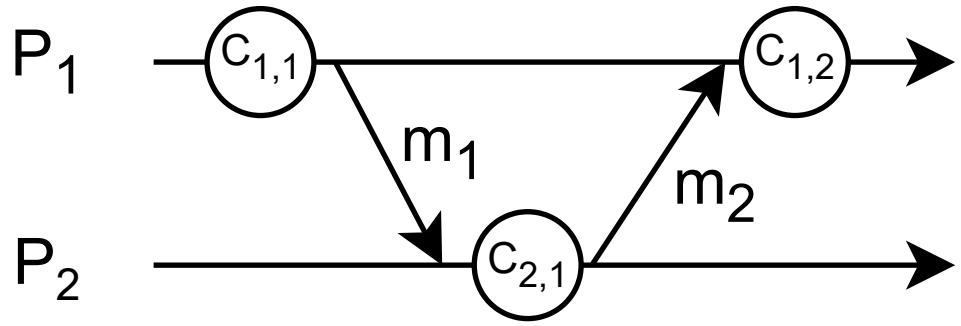

FIGURE 3.7: Execution graph containing a Z-cycle.

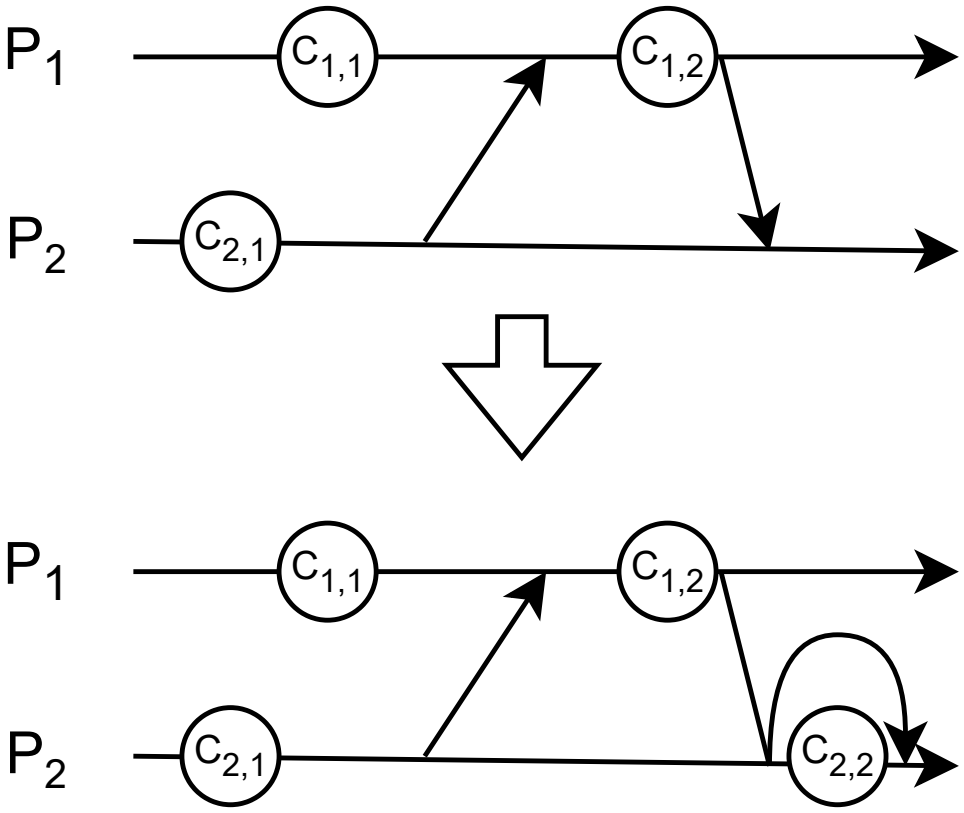

FIGURE 3.8: Whenever a cycle is detected on the reception of a message, a checkpoint will be taken first, before the message is processed.

Chapter 4

Implementation

To benchmark the checkpointing algorithms in a standardized way a basic streaming system without any checkpointing implementation should be used as a base. For this purpose an in-house streaming system developed by one of the PhD candidates had been provided to build on top off. This chapter will describe the working of this in-house system, as well as the implementation details of the proposed solution.

4.1 In-house streaming system

The in-house system provided by one of the PhD candidates was an early version of a transactional streaming system developed in Python. It contained only two basic operators; a filter and a map. Additional operators (join and aggregate), as well as windows (tumbling), had to be implemented for the benchmarking purposes. The system divides the operator workload into multiple partitions, which are distributed over the different workers in a round-robin fashion. For example, if there are 4 partition for a map operator on two different workers, worker one is assigned map 1 and 3, and worker two is assigned map 2 and 4. There is a coordinator that handles this workload distribution and this same coordinator is later also used for the fault recovery coordination, as well as finding recovery lines, and any additional protocols that need coordination. The workers start query execution by reading from a Kafka topic and the result of the query is written to a Kafka topic. Communication in the system is done through TCP and Minio buckets are used for checkpoint storage. Message logging is also done through Kafka, where each communication channel has its own topic partition.

4.2 Coordinated checkpointing

For the coordinated checkpointing a blocking marker-based Chandy-Lamport algorithm has been implemented [9]. This checkpointing mechanism makes use of rounds which are initiated by the coordinator. Whenever the workers receive a message from the coordinator that the round started, they will create a checkpoint of the source operator in the execution graph. To be able to know which operator is the source operator and where to pass or expect markers, the execution graph is defined manually before the execution starts. The workers use this execution graph to build a mapping of all incoming and outgoing channels and define the source and sink operators. Once the source operators have been checkpointed, a marker will be sent over all the outgoing channels to the next operator(s). Whenever a marker has been received the channel will be blocked, and when this is the case for every incoming channel (which is tracked by a boolean array), the worker will checkpoint that operator and again send markers on its corresponding outgoing channels. Any

messages received on a blocked channel will be buffered and processed directly after the checkpoint has been taken, before new incoming messages are processed. Once the sink operator has been checkpointed a message will be sent to the coordinator. The coordinator also keeps track of every worker status through a boolean array and once it received a message from every worker, the round is considered done. Now, whenever a failure is detected, the coordinator will simply send the last completed round number to every worker. Workers will then recover to the matching checkpoint. Note that in this case no message logs are necessary and therefore no messages are replayed on recovery either, making the recovery protocol a lot simpler then for the uncoordinated and CIC approach.

4.3 Uncoordinated Checkpointing

The uncoordinated checkpointing approach itself is relatively simple, since the workers only need to create a snapshot every set time interval. However, there is quite some information that needs to be stored within those snapshots, besides the actual worker state, to enable for consistent recovery. To achieve exactly once processing, the recovery protocol works as follows; The workers log their messages to a Kafka topic, partitioned per channel. The corresponding Kafka offsets are stored for both the incoming and outgoing channels (indicating the last offset processed and sent, respectively). Additionally, workers keep track of the last Kafka offset that they processed from the source (tasks assigned by the generators). Using this information, it can be determined which messages were sent but not yet received at the point of recovery. By simply replaying those messages and resetting the Kafka offset for the source to the last ones processed as stored in the checkpoints, exactly once processing can be guaranteed.

The workers, in this case, only store these offsets and log their messages. Whenever they are done checkpointing, this information is sent to the coordinator. The coordinator uses this information to build the checkpoint graph, as discussed in chapter 3. First, the new checkpoint is simply added as a node to the graph and an edge from the previous checkpoint on the same process is added to the new node. The offsets are stored and used to add the edges between the processes whenever a failure is detected. An implementation where these offset are processed immediately upon arrival could be faster, however it is a lot less trivial to implement, and since checking these overlaps is a relatively short task it should not affect performance a lot. These edges represent orphan messages, which can be found by looking at the interval overlap. Once all edges are added, nodes are marked using the rollback propagation algorithm [38] (which comes down to a reachability analysis). The set of every last node for every worker will be considered as the root set. If any of the nodes in the root set get marked through the reachability analysis, it is replaced with an preceding node and the analysis is redone. When no nodes get marked after the reachability analysis, the recovery line has been found. The coordinator then sends messages to all the workers, containing which checkpoint they should recover from and the offsets per channel that they should replay. Lastly the coordinator does some garbage collection, where the recovery line becomes the new recovery graph and all offsets are cleared. The recovery protocol for the workers restores the values as stored in the checkpoint, replays the offset intervals per channel as specified by the coordinator, and resets the Kafka offset from the source corresponding to the checkpoint.

4.4 Communication Induced Checkpointing

The Communication induced approach builds on top of the uncoordinated approach. A protocol for the Z-cycle detection and forced checkpointing were added and some information was piggybacked to the already existing messages between workers. Specifically the HMNR protocol [19] was chosen since it uses relatively small data structures and tries to minimize the amount of forced checkpoints. For this protocol to work some additional data structures are required. These include three different boolean arrays, a logical clock and a vector clock. Whenever a message is sent, the logical clock, vector clock and two of the boolean arrays are piggybacked. This information is used on message reception to do the cycle detection as described in the paper [19]. Whenever a potential cycle is detected a checkpoint will be forced and the values will be adjusted accordingly.

One thing to note is that, for the communication induced approach to work properly, there should not be a perfect overlap between checkpointing intervals, otherwise no cycles can be formed. When taking checkpoints every x amount of seconds, even though the workers aren't in sync, checkpoints might still be taken too close to each other to find any cycles. To create some more room for these cycles (and orphan messages) to occur, some randomness has been added to the uncoordinated (and CIC) intervals.

4.5 Metric collection

To be able to benchmark the proposed solution some implementations were necessary to collect certain metrics. Why these metrics were chosen specifically and what they aim to capture is explained in the benchmarking chapter. This section will discuss how these metrics were collected within the system.

Latency – The latency is measured in an end-to-end fashion, using the timestamps of the ingestion Kafka topic, as well as the Kafka output topic. These timestamps will be matched based on their corresponding keys, such that the difference in these timestamps represents the latency for the item with that key. This implementation however differs a bit depending on the query. When a join or an aggregate is used, there is a mapping from multiple inputs to one output. In this case the latest timestamp of all the contributing inputs is compared with the output timestamp. For the cyclic query one input can lead to multiple outputs, because of this all the different input/output matches are considered for the latency measurement.

Throughput – The throughput of the system is measured by the input rate specified in the generators. The highest sustainable rate is found manually by changing the rates based on the systems performance.

Checkpoint time – How to measure the checkpoint time differs per protocol. For the uncoordinated and CIC approach the time at the beginning and the end of the checkpoint can simply be compared, from which the average time can be derived. For the coordinated protocol this is a bit less straightforward. Ideally the effect of the channel blocking should also be captured in this measurement. Therefore the time is measured from the coordinator, with the starting timestamp being the beginning of the checkpointing round and the end timestamp being the moment when every worker confirmed that they are done with the round. By dividing this time over the total amount of checkpoints in a round, an average checkpointing time that includes the channel blocking can be calculated.

Recovery time – Recovery time is measured in the same way for every protocol. A timestamp of failure detection and a timestamp of confirmation of successful recovery for every worker are used to determine this time.

Rollback distance – The rollback distance is simply captured by the number of useless checkpoints. This number can be found by counting the amount of marked nodes during the recovery line algorithm. This does not give the actual time that was lost, however, since the checkpointing interval is known, it does give a worst-case estimate.

Network usage – The network usage is measured in the total amount of bytes sent through the network. This is done by simply accumulating all the sizes in bytes for every message that is sent out. Protocol specific message sizes are also counted separately to determine their relative effect on the total network usage.

Not all of these metrics can be collected outside of the execution, meaning that they could have an effect on the performance of the system, especially the network usage collected during the runs. Therefore, for every experiment, a run with and without network usage calculations is done.

4.6 Assumptions

In the interest of time some assumptions were made about the underlying streaming system, as well as the technologies used for the implementation of the algorithms. Although the performance and consistency strongly depend on correct performance of these factors, their fault free implementation lies outside of the scope of this research. This section will discuss these assumptions and simplifications.

First of all, it is assumed that all of the messaging in the system happens correctly. No messages are lost and they arrive in the correct order. This goes for the TCP channels, as well as all messages sent to Kafka, which includes the message logs and final results. The same assumption is made for the storage of the checkpoints in the Minio bucket.

Another challenge was that the in-house system did not come with any implementation that monitors heartbeats that can detect worker failures or disconnects. Therefore it was decided to implement some relatively simple artificial failures. These are messages from a worker to the coordinator, mentioning that it failed. The coordinator will then start all the necessary protocols as if it detected the failure itself and instruct the workers to recover. This means that workers do not actually fail, but this does not have an effect on the fault recovery.

For the coordinated approach, shuffling between operator partitions could have some effect on its performance, since this directly affects the amount of markers that each process should wait for. For the proposed solution it has been assumed that, if the amount of partitions differ per operator with a direct channel between them, markers simply need to be broadcasted to every partition of the downstream operator. This indirectly also means that the downstream operator expects a marker from every upstream operator partition as well. Additionally, for the coordinated approach, it is assumed that the execution tree is known beforehand, since it is passed to the workers to manage the marker passing.

4.7 Testing

Manual testing using system logs has been done for each of the three algorithms to specifically check their fault free execution, as well as consistency. To do this a simple execution graph was used, in which one of the operators simply incremented a count in the worker state. Upon correct fault recovery and message replaying, the sum of these counts should exactly equal the amount of queries created by the generator. If this count is too low it would indicate an at most once consistency guarantee, whereas if it was too high, it would indicate an at least once consistency guarantee. The implemented solutions also contain some more abstract logic that could be quite difficult to actually test through a manual method. Therefore some unit test have been created to confirm correct workings of these methods. This was done for some of the coordinator logic, such as building the recovery graph and finding the correct recovery line, as well as some of the CIC logic. An overview of the test methods and the specific behavior that each test tries to capture can be found in table 4.1. The methods that have been tested have been created with the tests in mind. By distributing the logic in the coordinator over smaller methods, a simple dummy coordinator object could be created for the testing purposes. In this dummy object only the necessary parameters had to be set to test each specific functionality. The CIC logic has also been decoupled from the messaging, such that method outputs could simply be passed as other method inputs, so no actual messaging was necessary for the unit tests. The coordinated and uncoordinated algorithms were simple enough to be able to test them through manual methods.

Method name	Protocol/algorithm	Behavior
test_find_reachable_nodes_empty	Finding reachable nodes in recovery graph	Checks if an empty set is returned if there are no reachable nodes in the graph
test_find_reachable_nodes_non_empty	Finding reachable nodes in recovery graph	Checks if a non-empty set is returned if there are reachable nodes in the graph
test_find_reachable_nodes_recursion	Finding reachable nodes in recovery graph	Check if all the reachable nodes are found recursively
test_find_recovery_line	Finding the recovery line	Check if the expected recovery line is found. given a simple graph with an orphan message
test_find_recovery_line_no_orphan	Finding the recovery line	Check if the expected recovery line is found when no orphan messages are present in the root set
test_find_recovery_line_domino	Finding the recovery line	Check if the expected recovery line is found when there is a domino effect in the graph
test_simple_edges	Adding edges to the recovery graph	Check is edges are added as expected given some overlapping offsets
test_same_interval_ends	Adding edges to the recovery graph	Check if edges are added correctly when offset intervals have the same value
test_clear_checkpoint_details	Clearing checkpoint information	Check if all the redundant information is cleared correctly (used after recovery)
test_find_channels_to_replay	Finding correct channels to replay	Checks if the correct subset of messages is found to replay for the workers, given a simple graph
test_cycle_detection	Detecting cycles	Check if no cycle is detected in a simple case where there is no cycle (note that in more complex scenarios this might not hold, due to it being a prediction)
test_cycle_detection	Detecting cycles	Check if a simple cycle is detected as it should be

TABLE 4.1: Testing methods and their behaviour.

Chapter 5

Benchmarking Setup

To extensively benchmark the proposed solution there are several factors that come into play. First of all different kinds of workloads and message patterns should be tested. Additionally there are several parameters that can be tweaked, such as the the amount of failures or the parallelism of the system. Lastly the evaluation metrics should be defined. This chapter will discuss the experimental design.

5.1 Queries

To ensure that the benchmarks cover different types of message patterns, a variety of queries should be used. Since the in-house system that the proposed solution is based on did not yet have support for all types of operators, these had to be implemented first, before meaningful benchmarks could be run. As discussed in the previous chapter, some basic operators such as a hash-join and an aggregate have been implemented, together with tumbling windows to enable for the benchmarking. With these additions to the in-house system, some queries based on the workings of the NEXMark queries[36] were implemented. NEXMark is a well-known benchmark for data streams. It has a total of 12 different queries that each aim to test a different type of behaviour. These queries will simply be referred to as NEXMark queries in this chapter, or simply a numbered query (e.g. query 1 when discussing the query based on NEXMark query 1). Workload generators for these queries from another thesis project were adapted to work with the in-house system. Between operators, shuffling can occur, meaning that messages from the same operator partition can be send to various receiving partitions. This has an effect on the amount of markers that have to be passed for the coordinated approach. Additionally, since the CIC algorithm relies on cyclic dataflows to force checkpoints, a custom cyclic query has been created to test the effect of this algorithm. A detailed description of each of these queries, along with some visual representations, can be found below.

5.1.1 NEXMark query 1

NEXMark query 1 (figure 5.1) represents a very simple currency conversion. It has only three operators, the first one converts the arguments to a Bid object, the second one converts the bid price from dollars to euro's and the last one outputs the results. There is no shuffling between the operators and no state is used. This query represents a workload with only a simple map.

5.1.2 NEXMark query 3

NEXMark query 3 (figure 5.2) creates some local item suggestions. The person source and auction source operators create the corresponding objects. The filter then

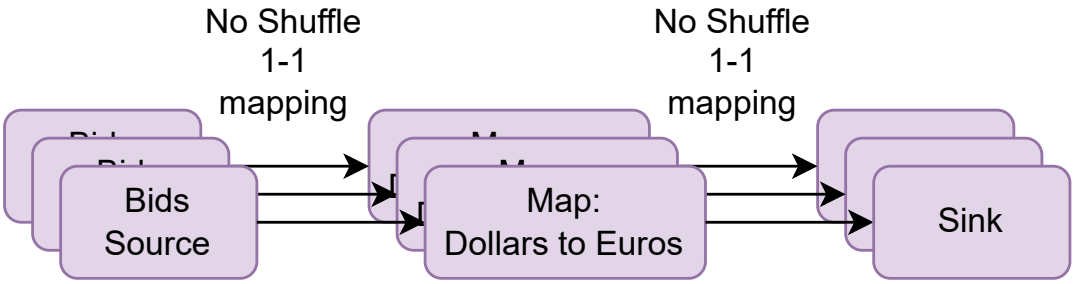

FIGURE 5.1: Execution graph of query 1.

filters the persons based on specific states. A flatmap is used to join this filtered result with the auction objects, which is then returned as output by the sink operator. This query tests stateful join and filter behavior, with some shuffling between the join and its input.

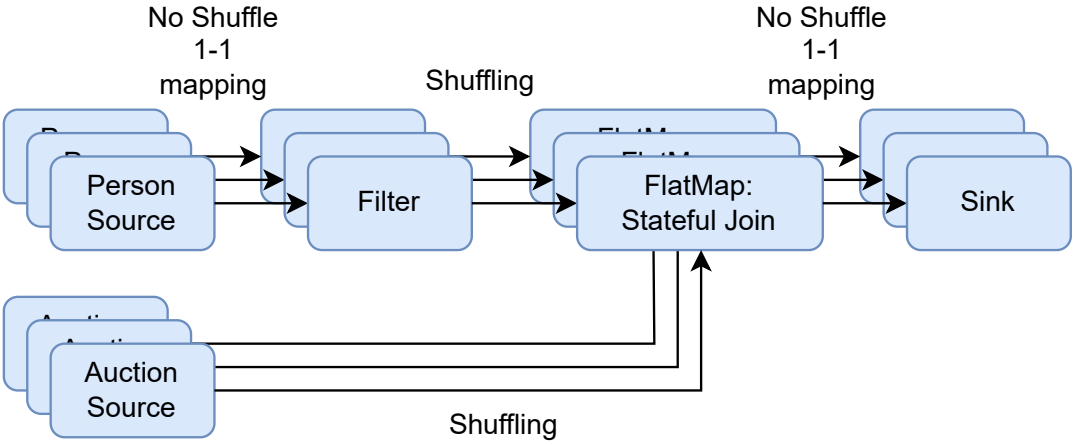

FIGURE 5.2: Execution graph of query 3.

5.1.3 NEXMark query 8

NEXMark query 8 (figure 5.3) makes use of a tumbling windows to monitor new users. It uses the items in this tumbling window to do a stateless join, of which the results are returned again as an output by the sink. Query 8, similarly as query 3, has the person and auction sources, and has shuffling between the inputs for the window (instead of the join in case of query 3). This query has window behaviour, as well as a stateless join.

5.1.4 NEXMark query 12

NEXMark query 12 (figure 5.4) is the last NEXMark based query, added to the augmented NEXMark suite. It basically counts the amount of bids a person makes within a fixed processing time window. Input for this window is again shuffled and the output is sent to an aggregate that counts the occurrences per person. The result of this is again sent to a sink that outputs the count. This query tests windowed aggregate behavior.

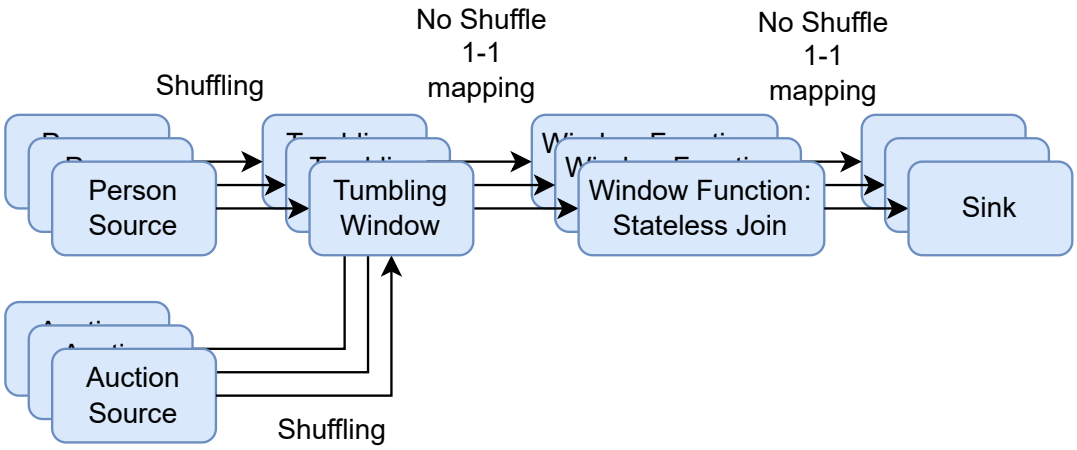

FIGURE 5.3: Execution graph of query 8.

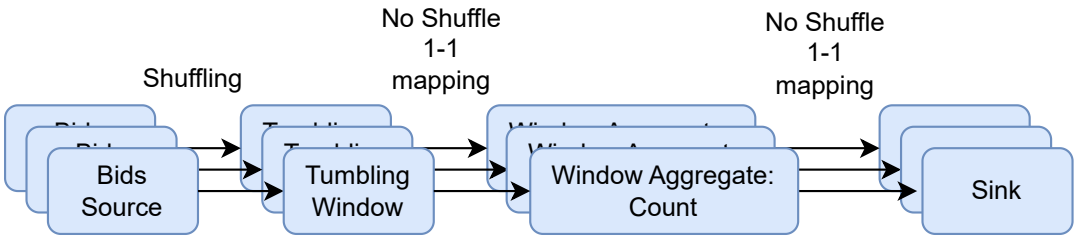

FIGURE 5.4: Execution graph of query 12.

5.1.5 N-hop approximation (cyclic)

For the purpose of testing the CIC workings, a relatively simple N-hop implementation has been chosen, which uses two operations. The execution graph for this query can be found in figure 5.5. The first operator simply receives two nodes and a count. This can either represent a directional edge with count 1 from the generator, or a route (constructed of multiple edges) with the hop count from its downstream operator (this is where the cyclic dataflow comes from). When the input is an edge it will be stored in a map in its local state. It will then (in any case) look for any edges starting from the second node (the end of the route or edge) in this map. Finally the operator forwards the edge/route received, along with all outgoing edges/routes and their counts, to the downstream operator. This second operator then simply adds the edges to the original edge/route, checks the new hop count, and stores this new route in memory if it is a new shortest route between two nodes. During this hop count check any of the following things can happen; The route between the nodes is longer than the specified n, in which case the route will be discarded. The route is exactly n, meaning it will be given as a valid output. Lastly, when the route is shorter than n hops, the new route will be sent back to the upstream operator, to find possible additional edges to add to the route. It is important to note here that this query will only work to test the uncoordinated and CIC approaches, since the coordinated approach will deadlock in this scenario. To improve the performance during the experiments, a time to live for the worker states of 5 seconds was chosen, meaning that the state would be cleared after that time. Additionally N was set to 3, to limit the amount of recursion such that a reasonable input rate could be

maintained.

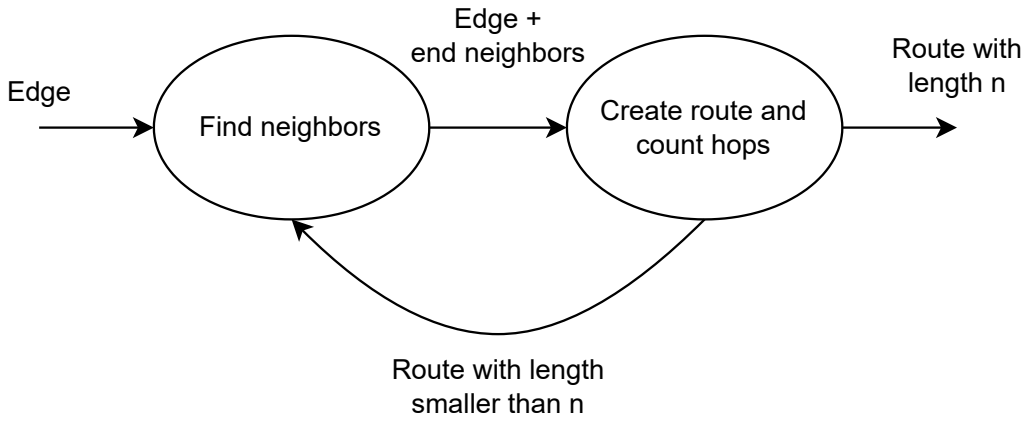

FIGURE 5.5: Execution graph of the n-hop approximation query.

5.2 Evaluation metrics

There are several metrics that will be used to evaluate the performance of the system. First of all, to test the general performance, the throughput and the latency of the system will be measured.

Latency – Latency is measured in an end to end manner, meaning that the output time is compared to the ingestion time. However, due to the differences between the queries, there are some small differences in how this measurement is taken. For query 1, both the output, as well as the input timestamps can be used for the same key. Query 3 and 8 contain a join, meaning that the input timestamp of the last item contributing to the join is compared to the output timestamp. Query 12 works in a similar way, where the timestamp of the last item that contributed to the aggregate (instead of the join) is used. For the cyclic query, all combinations of matching input and output will be considered for the latency calculation.

Throughput – For the throughput a sustainable generator rate will be searched for. A sustainable rate in this case indicates the highest possible rate that the system (and generators) can handle. This will be done in a manual fashion.

Due to the differences in the checkpointing approaches, there are also several other metrics that might yield some interesting results. The metrics considered in this case are the *recovery time, checkpointing time, network usage, and recovery distance.*

Recovery time – The recovery times can be measured by the coordinator, starting from the failure detection up until confirmation of successful recovery for all the workers. Since there is quite a difference in recovery protocols for the uncoordinated and CIC approaches compared to the coordinated approach, it is expected that there is quite some difference in terms of recovery time. However this also has to do with the fact that the checkpoints are taken in a different manner, meaning that a reverse effect is most likely to be found when looking at the checkpointing times.

Checkpointing times – This metric is a bit a less straightforward to measure. Since no rounds are used in the uncoordinated and CIC approaches, the time that it takes to create a checkpoint can simply be measured individually. However, for the coordinated approach only the time of a round can be used, since its channel blocking behavior should also be captured in this measurement. Although one round

does include multiple checkpoints, due to the system design, it can be determined how many checkpoint are exactly present. Therefore an average time per checkpoint can be determined using this round time, that includes this blocking behavior.

Network Usage – Network usage might also be an interesting metric to consider. Although the amount of messages for the CIC and uncoordinated approach should be similar, the amount of data for the CIC should be higher due to the piggybacked information. The coordinated approach on the other hand should see a lot more messages being passed within the system. To measure the effect on the network, both the size of all the messages in the system, as well as the protocol specific messages are captured. By doing this the effect per protocol on the network usage can be represented as a percentage increase.

Recovery Distance – Lastly the recovery distance should be measured for the uncoordinated and CIC approach. To do so the amount of useless checkpoints can be counted when searching for the recovery line. It would be expected that the CIC approach has, on average, a lower amount of useless checkpoints. Even thought this does not exactly capture the rollback distance in time, a worst case time indication can be provided by multiplying the amount of useless checkpoints with the checkpoint interval length.

5.3 Parameters

To highlight the strengths and weaknesses of each of the algorithms, various parameter configurations should be ran during the benchmarking process. The parameters considered for the this purpose are; amount of workers (parallelism), amount of failures and workload.

Amount of workers – With the increase of the amount of workers, the amount of channels increases as well. This has an effect on both the message logging in the uncoordinated and CIC approaches, as well as the marker passing in the coordinated approach.

Amount of failures – The amount of failures can be used to make the effects of fault recovery on the overall performance a lot more clear, especially if the performance differences turn out to be relatively small. In this case an effect on the throughput or latency might be difficult to capture if only a single failure would be used. Runs without failures should also be performed to see the effect of the recovery.

Workload – The workload can be changed to see in which scenarios the checkpointing overhead might be worth it and whether there is a big difference in terms of the performance between the approaches when presented with varying workloads.

Chapter 6

Results and Discussion

To test each of the three protocols, every query has been ran once per protocol using four different configurations. These configurations only differed in parallelism (either 10 or 50 workers) and the amount of failures (1 or 0). Sustainable data rates for each of these configurations were searched for manually. Each configuration ran for 90 seconds, with checkpoints being taken every 5 seconds and the failure occurring after approximately 45 seconds. For the latency calculation the first 30 seconds were not used, to get rid of any higher latencies possibly caused by starting up. The results can be found in tables 6.1, 6.2 and 6.3. Please note that, due to the deadlock of the coordinated protocol when running a cyclic query, no metrics could be collected. Similarly, recovery related metrics are left out when no failure occurs during the run, or when no checkpointing protocol is used.

Config	No protocol	UNC			CIC			COR		
(query, partitions)	Total	Total	Checkpointing	%	Total	Checkpointing	%	Total	Checkpointing	%
cyclic, p=10	97015421	95157016	82727	0,087	304380672	161225326	52,968	-	-	-
cyclic, p=50	254881793	265575479	431011	0,162	2940937744	1599080441	54,373	-	-	-
q1, p=10	576077875	581356019	122106	0,021	1671293911	876097998	52,420	579919823	63410	0,011
q1, p=50	2902863073	2909714974	607967	0,021	20360422336	12311421935	60,467	2893992110	317050	0,011
q3, p=10	903667741	910604090	802227	0,088	1935558744	870568994	44,978	905918547	410720	0,045
q3, p=50	3242220725	3324658797	35246899	1,060	19756258121	12090431984	61,198	3247878508	9057600	0,279
q8, p=10	1284685527	1289378584	1022570	0,079	2620113580	1114767525	42,547	1288544779	429590	0,033
q8, p=50	3904617241	3954408898	35251838	0,891	21683591386	12958178468	59,760	3916145642	9525950	0,243
q12, p=10	1462574856	1470178743	246524	0,017	2204160934	970152943	44,015	1465412692	239530	0,016
q12, p=50	4983564781	4995561581	3663748	0,073	16236451943	11946757207	73,580	4990270718	4767650	0,096

TABLE 6.1: Network usage in bytes.

	config	No Protocol		UNC		CIC		COR	
	partitions, rate, failures	50th percentile	99th percentile	50th percentile	99th percentile	50th percentile	99th percentile	50th percentile	99th percentile
cyclic	p=10, 4k, no failures	693	1193	770	1637	807	1843	-	-
	p=10, 4k, 1 failure	-	-	801	11178	815	3394	-	-
	p=50, 15k, no failures	117	289	151	489	168	564	-	-
	p=50, 15k, 1 failure	-	-	161	5804	193	5129	-	-
q1	p=10, 7k, no failures	33	48	69	103	73	110	33	48
	p=10, 7k, 1 failure	-	-	72	5282	21838	31614	34	2677
	p=50, 35k, no failures	30	45	73	139	78	4340	31	46
	p=50, 35k, 1 failure	-	-	1369	7662	26358	44037	37	2861
q3	p=10, 7k, no failures	23	31	58	210	64	196	24	98
	p=10, 7k, 1 failure	-	-	62	9716	71	11732	24	3137
	p=50, 25k, no failures	21	37	87	1852	2628	9492	21	131
	p=50, 25k, 1 failure	-	-	8403	27645	17329	41120	22	3488
q8	p=10, 5k, no failures	27	36	62	105	69	133	27	37
	p=10, 5k, 1 failure	-	-	72	10582	81	10467	28	2482
	p=50, 15k, no failures	28	44	90	1776	743	3209	24	40
	p=50, 15k, 1 failure	-	-	89	21521	9318	25370	24	2594
q12	p=10, 5k, no failures	24	37	68	114	74	148	32	44
	p=10, 5k, 1 failure	-	-	701	11426	2966	10644	33	2969
	p=50, 15k, no failures	29	43	93	977	434	3683	30	43
	p=50, 15k, 1 failure	-	-	1042	16351	12674	22295	31	2737

TABLE 6.2: Latency percentiles in ms.

Though maybe not as surprising, since most of the well-known stream engines use some version of the coordinated Chandy-Lamport algorithm, these results clearly show that the coordinated approach performs not only more consistently, but also more efficient for several reasons. The network overhead (table 6.1) in terms of size

	config	UNC			CIC			COR	
	partitions, rate, failures	ACT	ART	Useless CP's	ACT	ART	Useless CP's	ACT	ART
cyclic	p=10, 4k, no failures	71,54	-	-	107,82	-	-	-	-
	p=10, 4k, 1 failure	92,15	212	0	117,80	123	0	-	-
	p=50, 15k, no failures	8,68	-	-	40,95	-	-	-	-
	p=50, 15k, 1 failure	27,41	149	0	81,51	98	0	-	-
q1	p=10, 7k, no failures	4,39	-	-	14,68	-	-	9,25	-
	p=10, 7k, 1 failure	27,88	12	0	84,55	24	0	9	13
	p=50, 35k, no failures	9,94	-	-	5609,91	-	-	15,50	-
	p=50, 35k, 1 failure	222,42	376	0	4369,04	1887	0	30,13	18
q3	p=10, 7k, no failures	3,66	-	-	27,08	-	-	15,75	-
	p=10, 7k, 1 failure	28,54	1359	16	247,07	1331	12	58,5	274
	p=50, 25k, no failures	24,74	-	-	1670,82	-	-	90,81	-
	p=50, 25k, 1 failure	303,49	21079	98	1989,87	7661	105	92,63	869
q8	p=10, 5k, no failures	4,94	-	-	29,92	-	-	17,5	-
	p=10, 5k, 1 failure	66,84	799	9	491,86	432	10	19,56	27
	p=50, 15k, no failures	41,96	-	-	657,07	-	-	50,44	-
	p=50, 15k, 1 failure	143,20	5943	50	2436,09	6461	58	46,06	74
q12	p=10, 5k, no failures	6,50	-	-	66,59	-	-	17,75	-
	p=10, 5k, 1 failure	190,63	333	9	1381,89	345	0	22,44	20
	p=50, 15k, no failures	59,31	-	-	747,65	-	-	27,35	-
	p=50, 15k, 1 failure	202,05	4974	49	2672,06	5210	54	33,88	46

TABLE 6.3: Checkpointing metrics (average checkpointing and recovery times in ms and amount of useless checkpoints).

for the coordinated protocol is way lower than for the uncoordinated or communication induced approaches (a maximum of 0,3% compared to 1,1% and 73,6% respectively). In terms of latency (table 6.2) the 50th percentile (average) never exceeded 40ms for the coordinated approach, whereas the uncoordinated approach could reach up to roughly 8400ms and the CIC approach roughly 26400ms. When there are no failures the 99th percentile for the coordinated approach also stays a lot more stable with only some, still relatively low outliers around the times checkpoints are taken. The other two protocols do show way higher latencies, especially the CIC (figures 6.1, 6.2, 6.3). Taking checkpoints also seems to have a longer lasting effect on the latencies for the uncoordinated approach. The CIC approach does not show any huge outliers in this case, but the latencies are extremely high already. When failures do occur, naturally more latency is introduced due to recovery. This clearly shows in the 99th percentiles for every configuration (table 6.2 and figures 6.4, 6.5, 6.6). Again, the coordinated approach performs by far the best in this scenario as well. From figures 6.4 and 6.5 it becomes clear that the uncoordinated and CIC protocol cannot catch up quickly enough after a failure recovery, resulting in a slowly decreasing, yet extremely high latency. The coordinated approach on the other hand quickly returns to its behavior before the failure occurred (figure 6.6). Lastly when looking at the checkpoint specific metrics (table 6.3), it shows that the coordinated approach stays pretty consistent when it comes to checkpointing and recovery times, with small increases depending on the workload.

Since the coordinated approach has no need to transfer any information about its checkpoints, it logically follows that the network usage in bytes is quite a lot lower than for the other protocols. Additionally, the CIC needs to include several datastructures on every execution message to be able to perform the cycle detection, which is why its relative effect on the network is way higher than even the uncoordinated approach. The main difference in performance for these algorithms is most likely caused by the message logging necessary for both the uncoordinated and CIC approaches however. When checkpoints are taken, workers need to wait for their message logging buffers to be committed to ensure consistency. This has

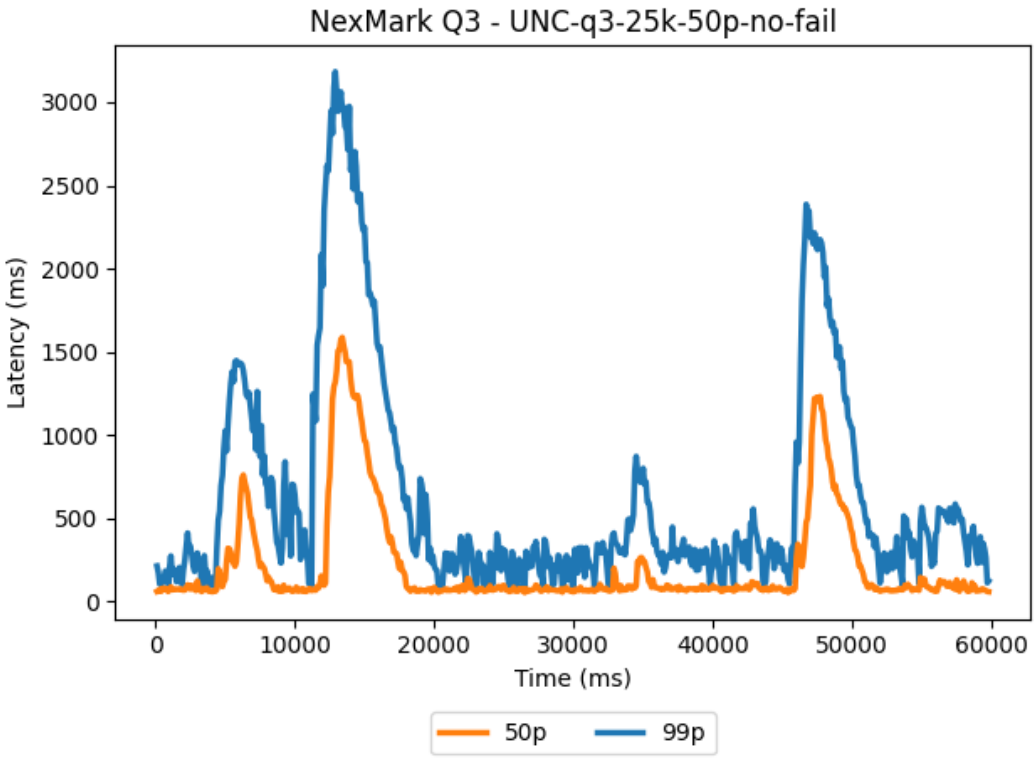

FIGURE 6.1: Uncoordinated approach, query 3, 50 workers, no failure.

an effect on the average checkpointing times, and therefore also the latency. Recovery times are also effected by this, since messages usually need to be replayed upon recovery, which isn't the case for the coordinated approach. Finding a recovery line also contributed to the higher recovery times of the uncoordinated and CIC approaches, since the coordinated approach can simply keep track of the last completed checkpointing round instead of having to build a graph and find a consistent global checkpoint.

A bit more unexpected is the performance of the CIC algorithm compared to the uncoordinated one. In terms of latency and checkpointing metrics the uncoordinated approach, in most cases, outperforms the CIC by a lot. An important thing to note here is that the cycle prediction is not perfectly accurate, meaning that sometimes cycles are predicted when there are none, even in non-cyclic queries. This increases the amount of checkpoints taken and, especially if checkpointing times are relatively long, can increase latency a lot. The cyclic query created specifically to generate cycles did not have as many orphan messages or domino effects as hoped, therefore yielding similar results for the CIC and uncoordinated approach. Their latencies also showed a very similar behaviour, both with and without failures (figure 6.7, 6.8, 6.9 and 6.10). Queries 3, 8 and 12 on the other hand did have quite some useless checkpoints (table 6.3). Since these queries do not contain any cyclic dataflows, these are caused by orphan messages and cannot be prevented accurately with the use of cycle detection. These orphan messages are more likely to occur the higher the amount of operators and the more shuffling occurs between them. Judging from these results, the occurance, and thus prevention by the CIC approach, of

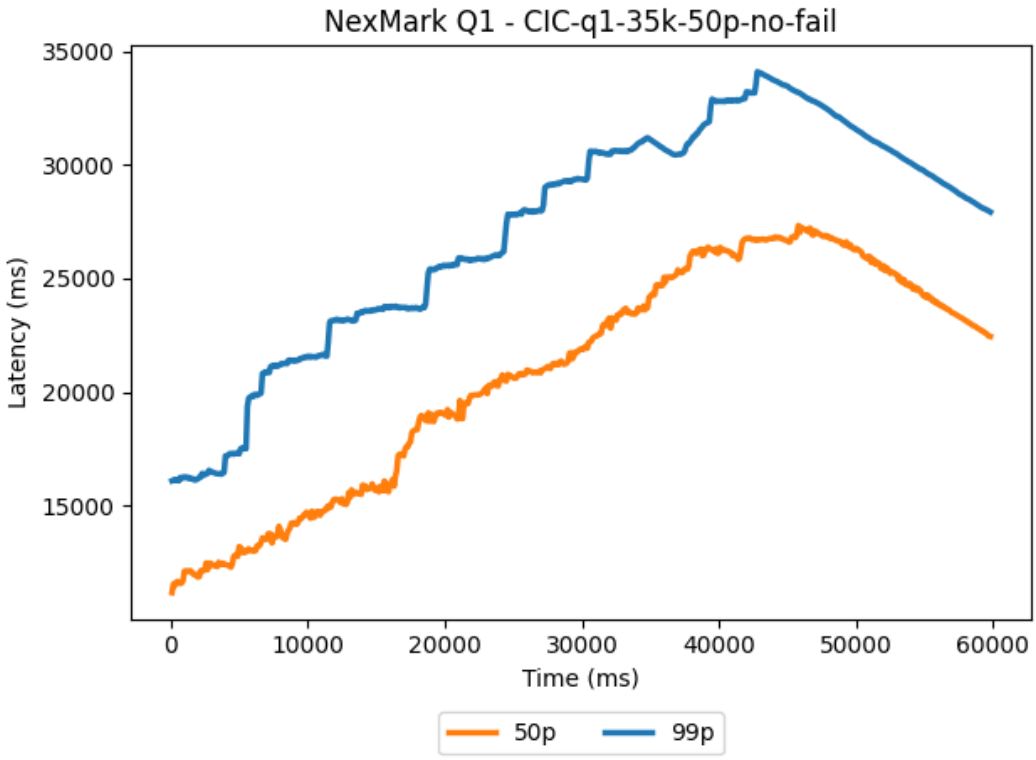

FIGURE 6.2: CIC approach, query 1, 50 workers, no failure.

the domino effect seems rare enough to not outweigh the negative effect on the performance. Which might be an interesting future direction to take this research in, since a blocking coordinated approach is not an option when dealing with cyclic dataflows.

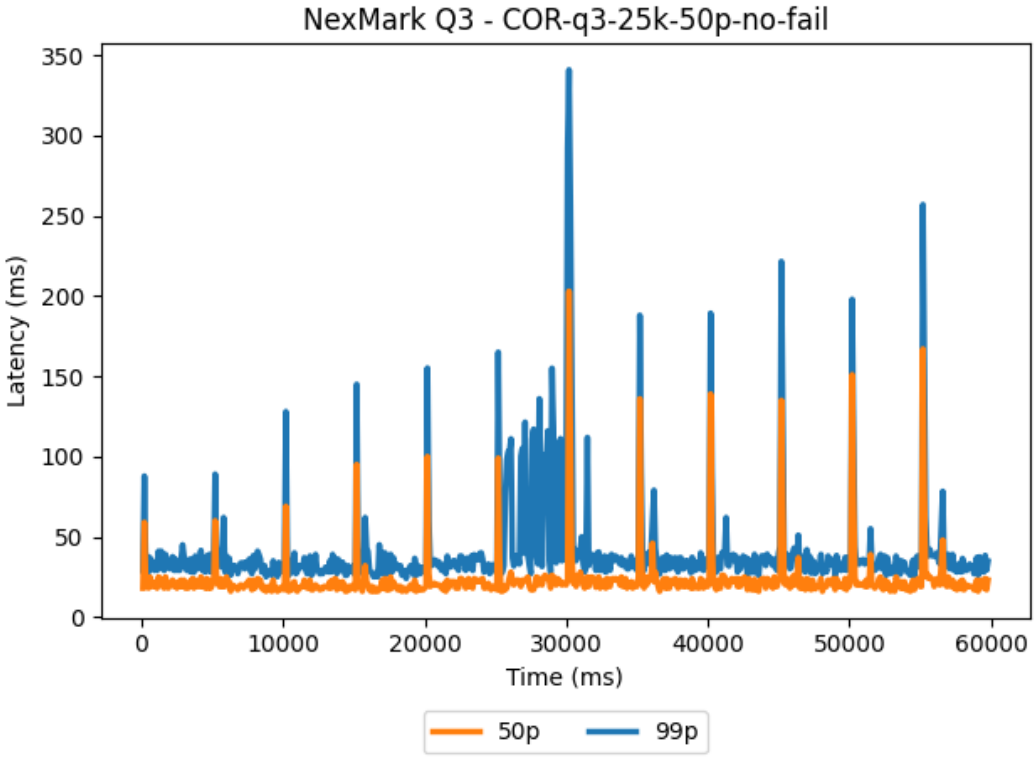

FIGURE 6.3: Coordinated approach, query 3, 50 workers, no failure.

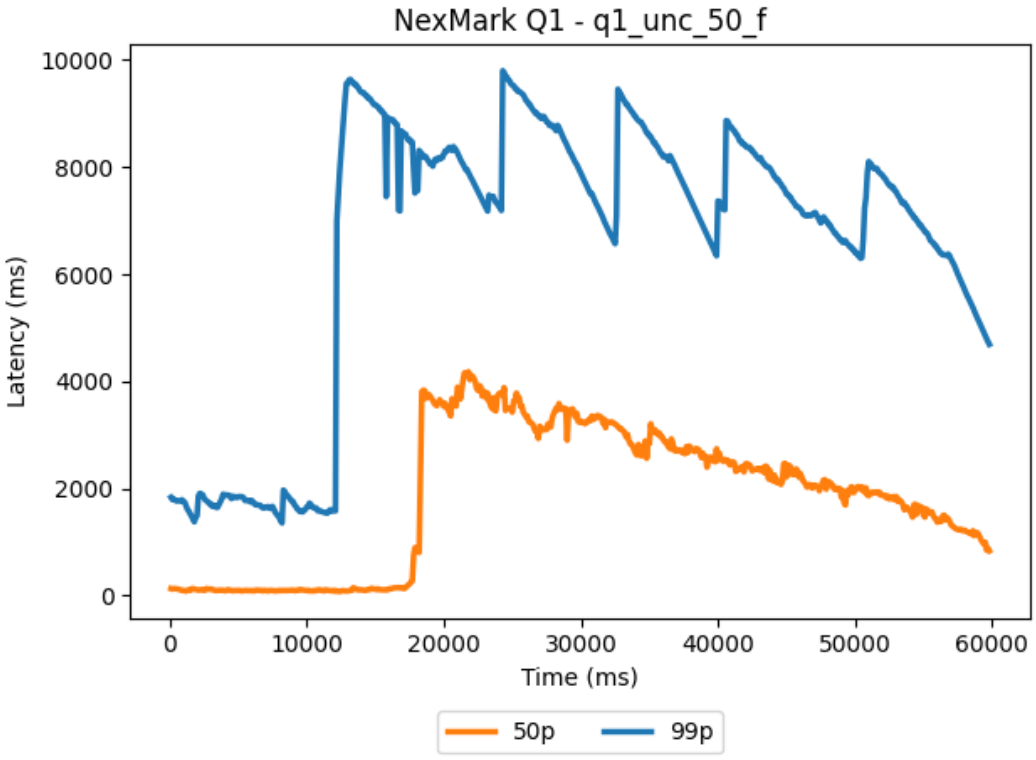

FIGURE 6.4: Uncoordinated approach, query 1, 50 workers, 1 failure.

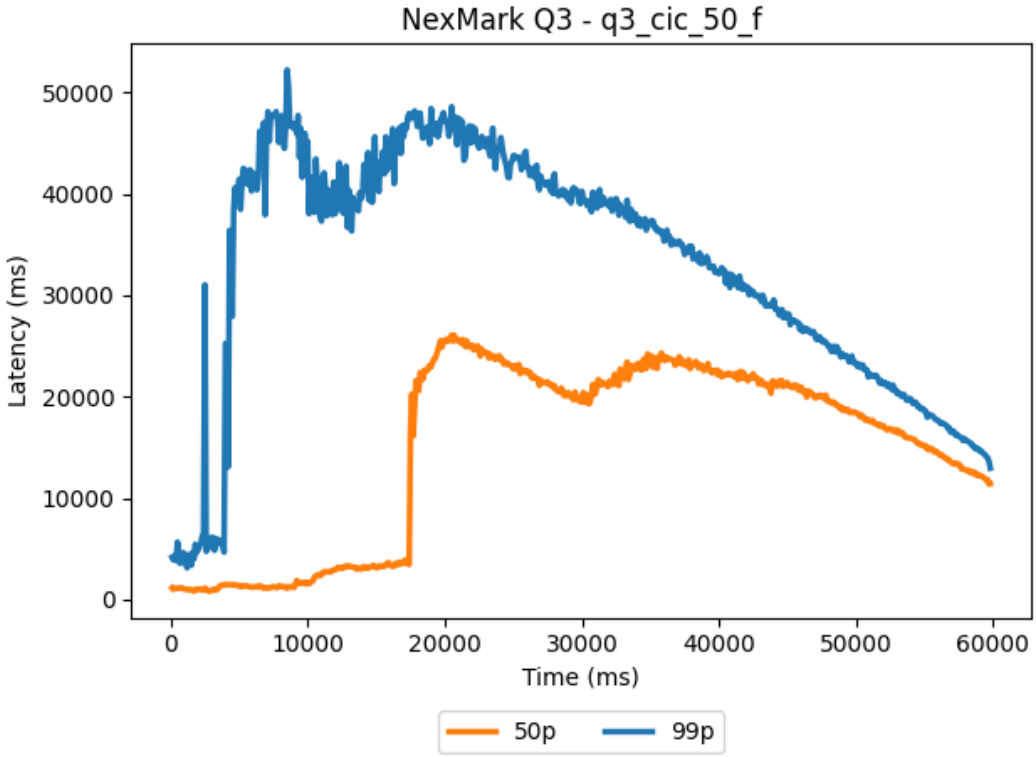

FIGURE 6.5: CIC approach, query 3, 50 workers, 1 failure.

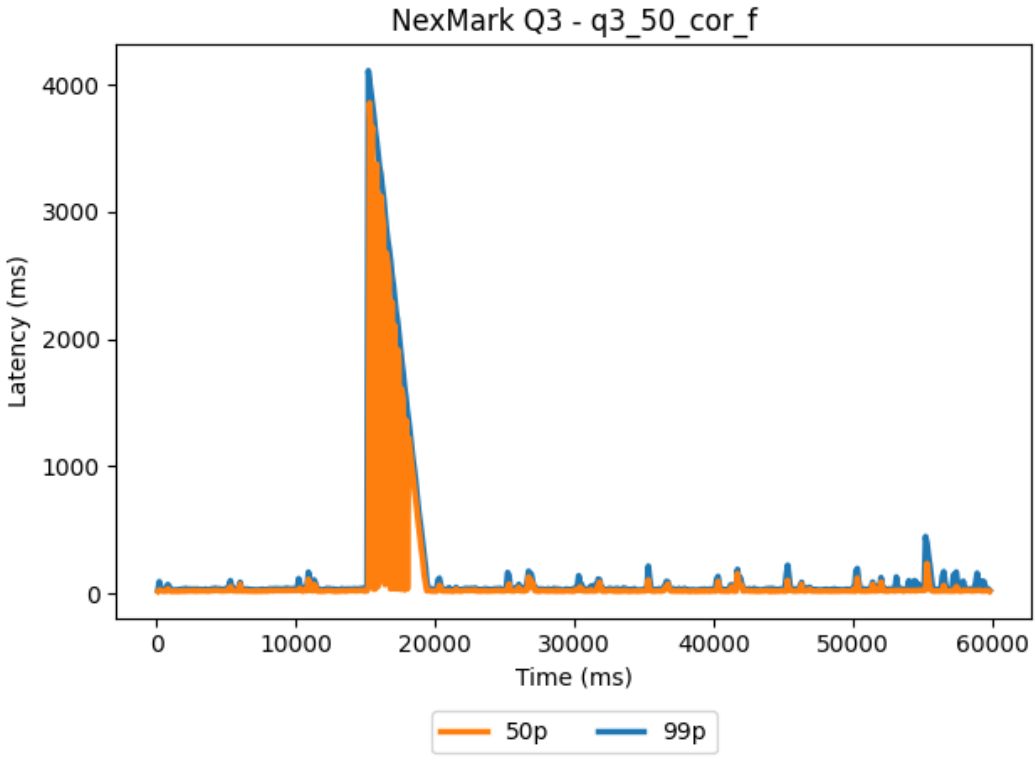

FIGURE 6.6: Coordinated approach, query 3, 50 workers, 1 failure.

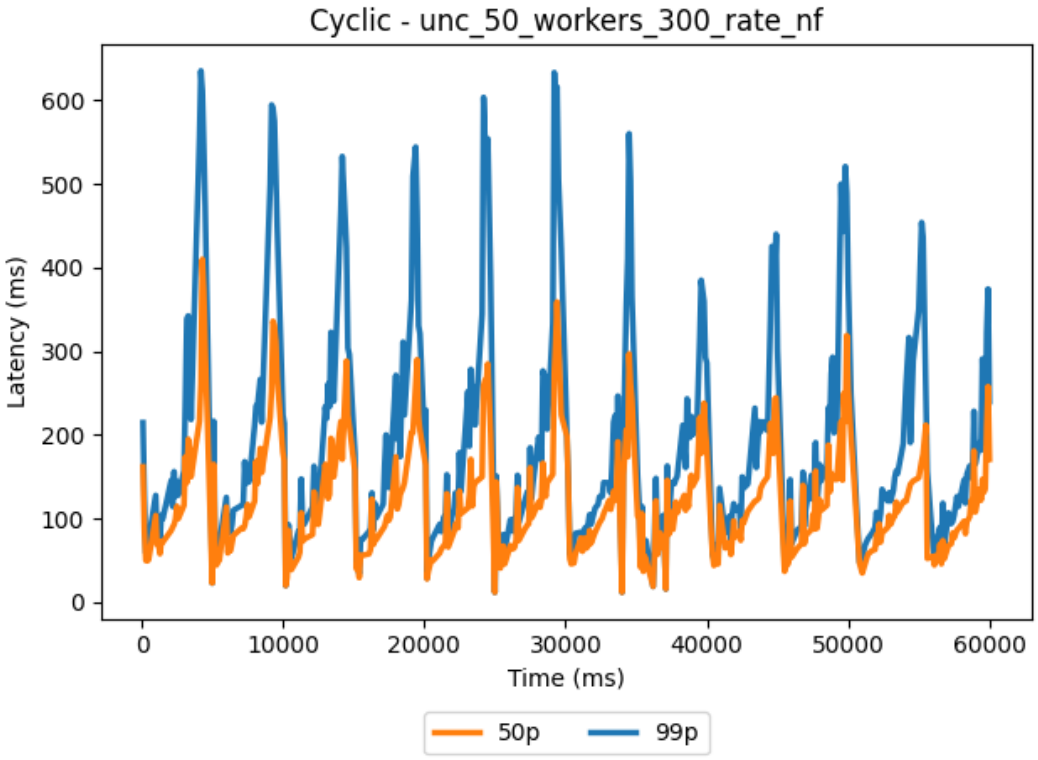

FIGURE 6.7: Uncoordinated approach, cyclic query, 50 workers, no failure.

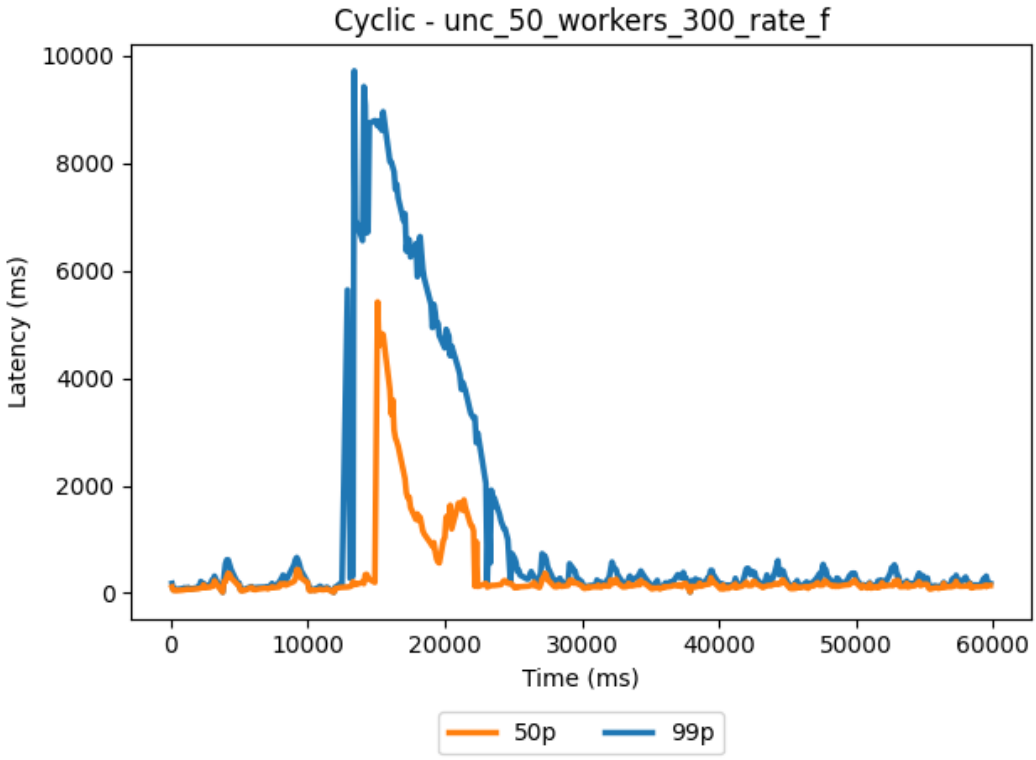

FIGURE 6.8: Uncoordinated approach, cyclic query, 50 workers, 1 failure.

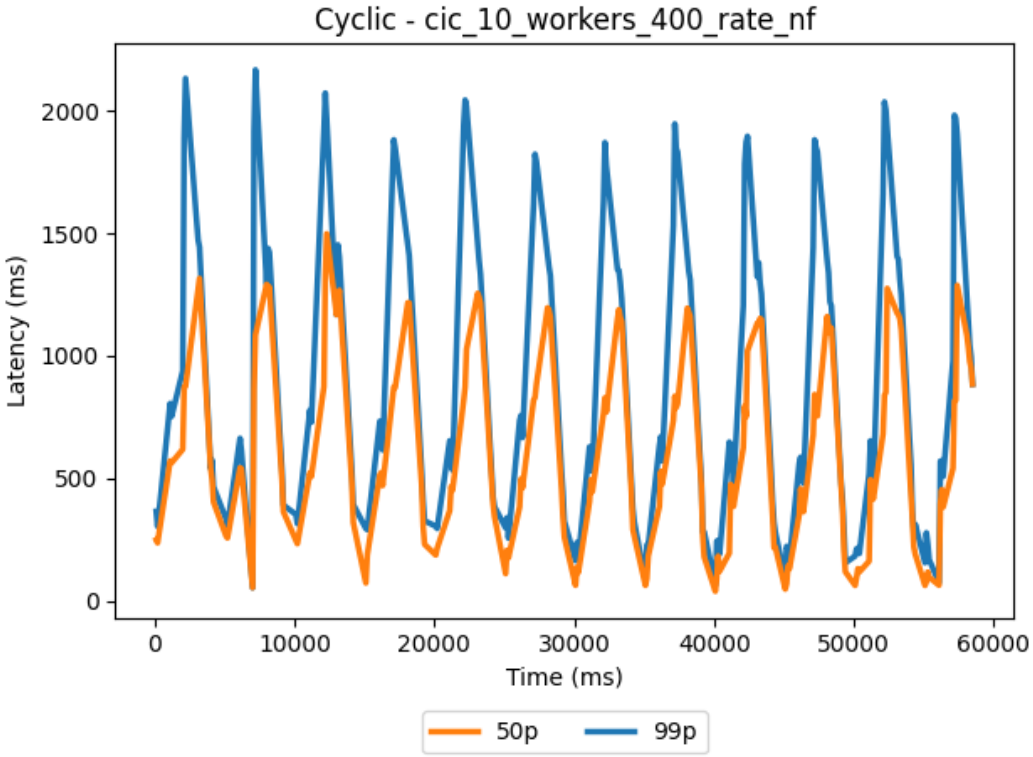

FIGURE 6.9: CIC approach, cyclic query, 10 workers, no failure.

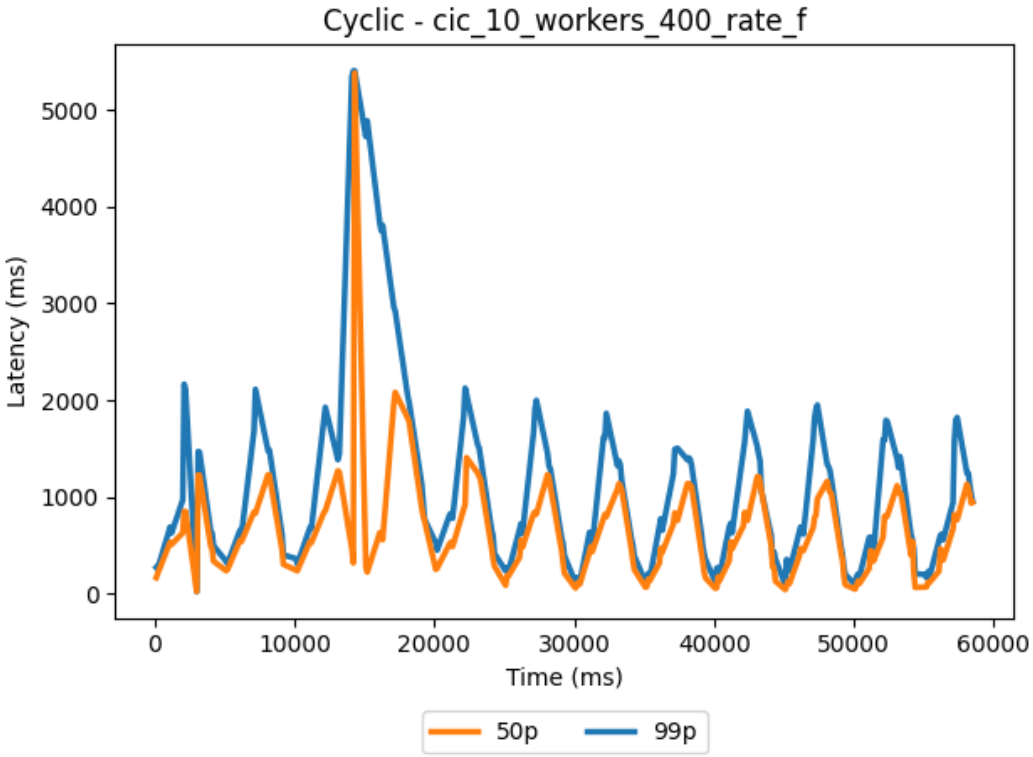

FIGURE 6.10: CIC approach, cyclic query, 10 workers, 1 failure.

Chapter 7

Conclusion

This paper compared and implemented the three different types of checkpointing algorithms (uncoordinated, communication induced and coordinated) in a basic streaming system. By analysing the results of various benchmarks on the network, latency and checkpointing metrics of these three approaches, it has become clear that the widely used coordinated blocking Chandy-Lamport solution indeed outperforms the other options. Additionally it has been shown that, due to the rare occurrence of (long) domino effects, the potential benefits of the communication induced approach do not outweigh its negative effects on the performance of the system. Based on this practical evaluation it is indeed highly recommended to go for a coordinated approach that omits the need for message logs. However, since these types of algorithms do not support cyclic dataflows due to their blocking nature, a different solution is necessary for these specific scenarios. The results presented in this paper suggest that an uncoordinated approach might actually be preferred in that case, however this is based on one very simple cyclic dataflow. Continuing this research by creating a more in-depth comparison of the uncoordinated and CIC approach, along with perhaps a non-blocking coordinated protocol, specifically for the purpose of running cyclic queries, might yield interesting results.

Bibliography

[1] A. Agbaria and W.H. Sanders. "Distributed snapshots for mobile computing systems". In: *Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the*. IEEE, 2004. DOI: 10.1109/percom.2004.1276856. URL: https://doi.org/10.1109/percom.2004.1276856.

[2] L. Alvisi et al. "An analysis of communication induced checkpointing". In: *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*. 1999, pp. 242–249. DOI: 10.1109/FTCS.1999.781058.

[3] Sanjay Bansal, Sanjeev Sharma, and Ishita Trivedi. "A Detailed Review of Fault-Tolerance Techniques in Distributed System". In: *International Journal on Internet and Distributed Computing Systems* 1.1 (2011), p. 33.

[4] B. Bhargava and Shu-Renn Lian. "Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach". In: *Proceedings [1988] Seventh Symposium on Reliable Distributed Systems*. 1988, pp. 3–12. DOI: 10.1109/RELDIS.1988.25775.

[5] Maycon Viana Bordin et al. "DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems". In: *IEEE Access* 8 (2020), pp. 222900–222917. DOI: 10.1109/ACCESS.2020.3043948.

[6] Paris Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Engineering Bulletin* 38 (Jan. 2015).

[7] Paris Carbone et al. *Lightweight Asynchronous Snapshots for Distributed Dataflows*. 2015. arXiv: 1506.08603 [cs.DC].

[8] Paris Carbone et al. "State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12 (2017), 1718–1729. ISSN: 2150-8097. DOI: 10.14778/3137765.3137777. URL: https://doi-org.tudelft.idm.oclc.org/10.14778/3137765.3137777.

[9] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (1985), 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/214451.214456.

[10] Iván Cores et al. "Improving Scalability of Application-Level Checkpoint-Recovery by Reducing Checkpoint Sizes". In: *New Generation Computing* 31.3 (July 2013), pp. 163–185. DOI: 10.1007/s00354-013-0302-4. URL: https://doi.org/10.1007/s00354-013-0302-4.

[11] Camille Coti et al. "Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: Association for Computing Machinery, 2006, 127–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188587. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1188455.1188587.

[12] RisingWave Database. *Fault tolerance*. 2023. URL: https://www.risingwave.dev/docs/current/fault-tolerance/.

[13] E. N. (Mootaz) Elnozahy et al. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems". In: *ACM Comput. Surv.* 34.3 (2002), 375–408. ISSN: 0360-0300. DOI: 10.1145/568522.568525. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/568522.568525.

[14] Marios Fragkoulis et al. *A Survey on the Evolution of Stream Processing Systems*. Aug. 2020.

[15] Adriano Marques Garcia et al. "SPBench: a framework for creating benchmarks of stream processing applications". In: *Computing* 105.5 (Jan. 2022), pp. 1077–1099. DOI: 10.1007/s00607-021-01025-6. URL: https://doi.org/10.1007/s00607-021-01025-6.

[16] Rahul Garg, Vijay K. Garg, and Yogish Sabharwal. "Scalable Algorithms for Global Snapshots in Distributed Systems". In: *Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. Cairns, Queensland, Australia: Association for Computing Machinery, 2006, 269–277. ISBN: 1595932828. DOI: 10.1145/1183401.1183439. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1183401.1183439.

[17] Richard C Gass and Bidyut Gupta. In: *AN EFFICIENT CHECKPOINTING SCHEME FOR MOBILE COMPUTING SYSTEMS* (2001).

[18] Bidyut Gupta, S.K. Banerjee, and B. Liu. "Design of new roll-forward recovery approach for distributed systems". In: *Computers and Digital Techniques, IEE Proceedings* - 149 (June 2002), pp. 105 –112. DOI: 10.1049/ip-cdt:20020410.

[19] J.-M. Hélary et al. "Communication-based prevention of useless checkpoints in distributed computations". In: *Distributed Computing* 13.1 (Jan. 2000), pp. 29–43. DOI: 10.1007/s004460050003. URL: https://doi.org/10.1007/s004460050003.

[20] Jean-Michel Hélary, Achour Mostéfaoui, and Michel Raynal. "Virtual precedence in asynchronous systems: Concept and applications". In: *Distributed Algorithms*. Ed. by Marios Mavronicolas and Philippas Tsigas. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 170–184. ISBN: 978-3-540-69600-1.

[21] Jean-michel Helary et al. *Communication-Based Prevention of Useless Checkpoints In Distributed Computations*. May 1997.

[22] Gabriela Jacques-Silva et al. "Consistent Regions: Guaranteed Tuple Processing in IBM Streams". In: *Proc. VLDB Endow.* 9.13 (2016), 1341–1352. ISSN: 2150-8097. DOI: 10.14778/3007263.3007272. URL: https://doi-org.tudelft.idm.oclc.org/10.14778/3007263.3007272.

[23] Hazelcast Jet. *Fault tolerance · Hazelcast Jet*. 2023. URL: https://jet-start.sh/docs/architecture/fault-tolerance.

[24] Vijaya Kapoor and Parveen Kumar. "Article: A Comparative Study on Snapshot Protocols for Mobile Distributed Systems". In: *International Journal of Computer Applications* 106.3 (2014). Full text available, pp. 27–30.

[25] Boris Koldehofe et al. "Rollback-Recovery without Checkpoints in Distributed Event Processing Systems". In: *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. DEBS '13. Arlington, Texas, USA: Association for Computing Machinery, 2013, 27–38. ISBN: 9781450317580. DOI: 10.1145/2488222.2488259. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2488222.2488259.

[26] A D Kshemkalyani, M Raynal, and M Singhal. "An introduction to snapshot algorithms in distributed computing". In: *Distributed Systems Engineering* 2.4 (1995), p. 224. DOI: 10.1088/0967-1846/2/4/005. URL: https://dx.doi.org/10.1088/0967-1846/2/4/005.

[27] Abdeldjalil Ledmi, Hakim Bendjenna, and Sofiane Mounine Hemam. "Fault Tolerance in Distributed Systems: A Survey". In: *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*. 2018, pp. 1–5. DOI: 10.1109/PAIS.2018.8598484.

[28] Ruirui Lu et al. "Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks". In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. 2014, pp. 69–78. DOI: 10.1109/UCC.2014.15.

[29] R.H.B. Netzer and Jian Xu. "Necessary and sufficient conditions for consistent global snapshots". In: *IEEE Transactions on Parallel and Distributed Systems* 6.2 (1995), pp. 165–169. DOI: 10.1109/71.342127.

[30] Brian Randell. "System structure for software fault tolerance". In: *IEEE Transactions on Software Engineering* SE-1.2 (1975), pp. 220–232. DOI: 10.1109/TSE.1975.6312842.

[31] Sriram Sankaran et al. "The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing". In: *The International Journal of High Performance Computing Applications* 19.4 (2005), pp. 479–493. DOI: 10.1177/1094342005056139. eprint: https://doi.org/10.1177/1094342005056139. URL: https://doi.org/10.1177/1094342005056139.

[32] Arif Sari and Murat Akkaya. "Fault Tolerance Mechanisms in Distributed Systems". In: *International Journal of Communications, Network and System Sciences* 08.12 (2015), pp. 471–482. DOI: 10.4236/ijcns.2015.812042. URL: https://doi.org/10.4236/ijcns.2015.812042.

[33] Zoe Sebepou and Kostas Magoutis. "CEC: Continuous eventual checkpointing for data stream processing operators". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 2011, pp. 145–156. DOI: 10.1109/DSN.2011.5958214.

[34] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. "RIoTBench: An IoT benchmark for distributed stream processing systems". In: *Concurrency and Computation: Practice and Experience* 29.21 (Oct. 2017), e4257. DOI: 10.1002/cpe.4257. URL: https://doi.org/10.1002/cpe.4257.

[35] Michael Treaster. *A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems*. 2005. arXiv: cs/0501002 [cs.DC].

[36] Pete Tucker et al. *NEXMark – a benchmark for queries over data streams draft*. URL: https://www.semanticscholar.org/paper/NEXMark-%E2%80%93-A-Benchmark-for-Queries-over-Data-Streams-Tucker-Tufte/086cde207a125816091c34a076b473a4

[37] Y.-M. Wang and W.K. Fuchs. "Lazy checkpoint coordination for bounding rollback propagation". In: *Proceedings of 1993 IEEE 12th Symposium on Reliable Distributed Systems*. 1993, pp. 78–85. DOI: 10.1109/RELDIS.1993.393471.

[38] Yi-Min Wang et al. "Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems". In: *IEEE Transactions on Parallel and Distributed Systems* 6.5 (1995), pp. 546–554. DOI: 10.1109/71.382324.