Haute École Léonard de Vinci

ECAM
Brussels Engineering School

Institut Supérieur Industriel

Hybrid ordering protocol for partially-replicated state machine

Travail de fin d'études présenté par

Charles VANDEVOORDE

En vue de l'obtention du diplôme de

Master en Sciences de l'Ingénieur Industriel

finalité Informatique

Année académique 2017 - 2018

Abstract

State machine replication (SMR) is a technique used to provide fault-tolerant services by executing transactions in the same order on multiple nodes. SMR are inherently not scalable because every replica executes all commands and stores the full state. Recent works introduce the concept of Partially Replicated State Machine (PRSM) which shards the state into multiple partitions and replicates each partition for fault-tolerance. For multi-partition replications, the PRSM's ordering protocol needs to orchestrate the ordering across multiple partitions to ensure the global ordering.

Unfortunately, in real workloads, multiple partition operations received by a partition may communicate more with some partitions than others because, for example, the data in the two partitions is related. Thus, multiple partition operations are introducing skewness in their partition access patterns.

This master thesis provides a new adaptive protocol *CaMu* to improve ordering latency by combining two communication patterns. One pattern provides an optimal latency with periodic communication while the other is slower but optimized for rare communication patterns. By following the workload characteristics, the proposed protocol can adjust the communication pattern to improve latency.

The proposed protocol has been implemented and tested as the ordering layer of a research database named Calvin. Calvin is an implementation of a PRSM which provides a scalable transactional datastore with full ACID-compliance.

When the workload is skewed, results show that *CaMu* provides on average a better latency compared to other ordering protocol. In case the workload is highly skewed, *CaMu* shows a latency improved up to three times compared to other protocols.

Acknowledgements

My thesis has been carried out in the Computer Science and Engineering Department (INGI), which is part of the ICTEAM institute of Université Catholique de Louvain (UCL). More precisely, I got the opportunity to work under the supervision of Prof. Van Roy and its PhD student Zhongmiao Li. They provided me with great insight along my work and allowed me to work and learn a field of computer science that I was not familiar with.

I would also like to thank my ECAM supervisor, Dr Sébastien Combéfis, who made this thesis possible in the first place. During my curriculum at the ECAM, he taught me invaluable knowledge in computer science and always pushed me further to improve myself.

Next, I would like to thank my family for the support and help they gave me during my study and this thesis.

Finally, I would like to thank my long-time friends who support me in all occasion; Céline de Thibault, Jonathan Petit, Justine Metzmacker, Maxime Hulet and Quentin de Hemptinne. During the course of my study I also had the chance to meet great people who enriched my five years at the ECAM; Antoine Vander Meiren, Gaetano Giordano, Grégoire Soete, Hadrien Cools, Isaline Roelens, Lorenzo Riga, Renaud Laurent, Rémy Guyaux, Sylvain Alonso, Yannick Berckmans.

Contents

Introduction

In recent years, *NoSQL* databases have emerge to solve some limitations of relational databases such as availability and scalability. Unfortunately, most *NoSQL* databases do not support transactions to ensure those properties.

Most recently, a new type of databases has arisen known as *NewSQL* [8, 21, 35]. *NewSQL* databases are providing scalable, fault-tolerant databases with support of transactions. Therefore, this new type of databases is providing the consistency of relational databases with the scalability of *NoSQL* datastores [15].

NewSQL databases provide scalability and consistency using different techniques developed through years of research in distributed systems. Partially Replicated State Machine (PRSM) is one of those techniques. The idea is to replicate and shard data inside a cluster to provide fault tolerance and horizontal scalability. The consistency is maintained by ordering transactions in a coherent way across the cluster. When a transaction is ordered, each node can execute it independently as the consistent order of transactions in the system provides a serializability constraint. This total order of transactions is a big challenge as it generally requires an agreement between multiple partitions containing them self multiple replicas.

In this thesis, a new ordering scheme for PRSM is presented, called *CaMu*. This new protocol combines two state-of-the-art ordering protocols to reduce the overall latency by switching protocols depending on the workload. Those protocols are known as Periodic Broadcast and TO-Multicast. In Periodic Broadcast, each partition sends periodic messages to all partitions in the system. When a partition received all messages of partitions in the system, it can order the operations received in a deterministic way. With TO-Multicast, each transaction is ordered independently from each others and only requires involvement of partitions that must be involved in the transaction but the protocol requires communication between partitions. TO-Multicast uses a timestamping system to totally order transactions in the system.

Therefore, Periodic Broadcast has a better latency than TO-Multicast when the number of partitions is small because the periodic synchronization between partitions of Periodic Broadcast can quickly introduce an overhead when the number of partitions is increasing. On the other hand, TO-Multicast latency stays constant when the number of partitions increases as TO-Multicast is said to be genuine (i.e. involves only partitions required by a transaction).

When data is partitioned, a transaction may require access to multiple partitions to get or update data. In a real workload, those accesses are defined by how data is sharded and which transactions are received. Therefore, we can assume that those accesses are skewed as multiple partitions operations will not access partitions evenly.

The idea behind *CaMu* is to improve Periodic Broadcast scalability problem by using the TO-Multicast protocol for partitions which are not communicating much together. This way, partitions having a high relation will use Periodic Broadcast providing an optimum latency. Partitions which are not involved much together will use TO-Multicast because TO-Multicast, unlike Periodic Broadcast, involves partitions only when required.

The Calvin PRSM paper [41], which uses Periodic Broadcast protocol as its ordering layer, only focuses on the throughput of PRSM without discussing latency. This thesis aims to fix that by focusing our work on latency in PRSM while keeping the same throughput as Calvin.

The latency of *CaMu*, Periodic Broadcast and TO-Multicast with both a synthetic and a real world workload simulation benchmark has been measured in a large scale deployment. With normal skewness, the *CaMu* protocol has the best average case compared to Periodic Broadcast and TO-Multicast with different cluster size. When the skewness is high, our protocol shows an improved latency up to 3 times the latency of Periodic Broadcast. Unfortunately, when the system is not skewed, the latency of *CaMu* is a bit worse than Periodic Broadcast due to some overhead introduced by the combination of the two protocols.

The evaluation also shows that the protocol developed for partitions to switch between Periodic Broadcast and TO-Multicast introduces almost no overhead compared to a static setup.

This thesis is organized as follows:

Chapter 1 gives some basic knowledge about distributed systems concepts and primitives.

Chapter 2 discusses existing ordering scheme for PRSM.

Chapter 3 presents the *CaMu* protocol and the switching protocol in detail.

Chapter 4 gives information about the implementation and the initial codebase of the database.

Chapter 5 studies the results obtained by benchmarking our protocol and compares those to other ordering implementation benchmarks.

Chapter 6 shows improvements that could be done to improve the performance of our protocol.

Chapter 1

Background

In this chapter, some distributed system concepts will be introduced and explained for readers who aren't familiar with this field of computer science. Due to the space limits, our discussion will stay brief.

Before diving into specific topics, let's first define what a distributed system is. Leslie Lamport, which is considered the father of distributed systems, defines the concept [22] as: *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."*

Wikipedia definition [42] is more formal: *"A distributed system is a model in which components located on networked computers communicate and co-ordinate their actions by passing messages. The components interact with each other in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components."*

As those definitions imply, distributed systems create some challenges for researchers and software developers. In the following section, some of those challenges will be explained and analyzed. Next, some distributed primitives to manage those problems will be studied.

1.1 Faults

When a system is distributed, several components work together on a common goal. Some nodes in the system may not behave correctly, introducing a fault. A fault in distributed system occurs when a node is not working as intended [32]. Different sorts of faults exists:

Transient faults Occur once and disappear

Intermittent faults Occur, disappear and re-occur after some time

Permanent faults External manipulation is required (either automatically or manually)

A fault is not necessarily a hardware problem like a dead disk. Garbage collection pause can be defined as an intermittent fault by other nodes. A deadlock or an infinite loop is an example of permanent fault.

A distributed system also has different failure modes [39] (ordered by ascending severity):

Crash/fail-stop failure A node is stopped but it was working correctly before stopping.

Fail-recover failure Same as fail-stop failure but the node is restarted after a failure.

Omission failure A node fails to respond to an incoming request because the request is never received by the node or the response is never received.

Timing failure A request timeouts.

Response failure The response received is incorrect.

Byzantine/arbitrary failure A node generates arbitrary messages at any time.

In a distributed system, faults can be handled by using redundancy. Redundancy can be added at different levels:

Data replication when the same data is saved on different storage nodes.

Computation replication when the same task is repeated on multiple nodes.

Thus, redundancy is achieved by adding replicated actions in the system. As data replication does not provide availability for most use-cases, only computational replication will be analyzed next. Computation replication can be achieved in different ways using an active or a passive replication technique. Active replication is when each replica is executing the task, while passive replication only replicates the result of the calculation. State machine replication, multi-master are example of active replication when master-slave is an example of passive replication.

Each failure mode comes with requirements to handle a failure. Fail-stop failure only requires $n + 1$ nodes to handle n failures because this mode assumes that at least one running node is always correct. For byzantine failure where any node could send adversarial messages at any time, an agreement between nodes needs to happen before processing any messages. In the Byzantine general problem [25] where each node is considered to be a war general where there are n traitors (i.e. faults), the problem is solvable only if we have at least $3n + 1$ generals[1].

In the case where the number of faults exceed the number of faults supported by the system, the system will either send an error or continue to operate in an inconsistent manner. This trade-off is known as the CAP theorem [3]. CAP stands for consistency, availability and partition tolerance.

[1]Other requirements are needed but not explained here for brevity. [25] contains the detailed information.

Consistency is the fact that every response to any clients is always correct which usually means that the return value is the latest written value. Availability is the fact that a service will always respond to a request. The response may be incorrect. Finally, a partition, in this context, is a network partition where some nodes are isolated from the rest of the cluster. A fault can be seen as a network partition isolating only one node. Partition tolerance is the system tolerance to any network partition.

As said above, the CAP theorem is a trade-off, a system can only have two of the three properties at any point in time. In case a network partition occurs, the system architect will need to choose between keeping availability by decreasing consistency or staying consistent by decreasing availability.

1.2 Asynchronous systems and clock

Synchronous systems [29] have an upper bound on the relative process execution, message delays and the maximal local clock drift. Those systems have very interesting properties such as clock synchronization, timed failure detection, . . . Unfortunately, in practice, those synchronous systems are very hard to achieve as it requires real-time OS and a specific network stack.

In contrast, asynchronous systems [29] does not have any global clock. Each node is totally independent from each other. Asynchronous systems do not give any guarantee on:

- Process speed
- Message transmission delay
- Clock

Asynchronous system is a realistic model of wide area networks but the model suffers from different limitations. For example, it is impossible to have a failure detector inside a fully asynchronous systems because the detector does not know if a node is down or the message is taking a very long time to arrive.

Another model can be used to solve some drawbacks introduced by asynchronous systems, partial synchrony systems. Partial synchrony systems are assuming a maximal time bound for a message to be received, after this time bound, the other node is considered to be faulty. This assumption may, in practice, be false as a node could have a garbage collection pause, be slowed down by other processes, . . . This model makes possible to implement failure detector, even if those are not perfect.

Clock and time are generally used a lot to describe causality between events. Unfortunately, most of the time, physical clock should be avoided in asynchronous or partially synchronous systems because all clocks can't be synchronized perfectly.

Even if perfect clock synchronization is impossible, various protocols try to synchronize clock as precisely as possible. This type of clock synchronization is generally used to synchronize workstations, unrelated servers, . . . The Network Time Protocol (NTP) [28] is an example of clock synchronization protocol. NTP uses different sources of time to improve its accuracy like atomic clocks, GPS, . . . but the protocol's precision is still dependant on network variations.

In special case, physical clock can still be used. Google's Spanner database is using NTP with atomic clock to globally order events based on an absolute clock error interval. If the maximal error in the system is lower than the accepted interval, Spanner stays consistent. The main drawback of this approach is the high cost of atomic clock and high precision GPS which are required in every datacenter.

1.3 Ordering

In distributed systems, total order is an ordering property defined as:

Assume two messages m_a and m_b. $m_a < m_b$ relation implies m_a is ordered before m_b on every correct process involved in m_a and m_b. The "$<$" relation is the total ordering property.

Other ordering properties exist which restrict the total order: First-In First-Out (FIFO) and causal order [11]. FIFO order guarantees to deliver messages in the same order as they were sent on top of a total order guarantee to order messages at the destination. Causal order [23] expresses an ordering constraints on events which are related to each other. Causal order is defined more formally in Leslie Lamport's paper [23] about causality in distributed systems. Causal order also restricts total order by adding a constraints on the relation between events.

1.4 Atomic broadcast and multicast

In a high-level way, a broadcast protocol allows a sender process to send a message to every process in the system while a multicast protocol allows a sender process to send a message to some selected processes. As a distributed system has faults, those primitives should provide some guarantees about messages' delivery by introducing more complex protocols [30].

A best-effort broadcast or multicast does not provide any guarantee in case of failure. Best-effort protocol only ensures non-faulty process will receive the message and this message will only be delivered once.

Some systems may want to deliver the message only if every process delivers it, those type of protocols are called reliable. Reliable broadcast or multicast has the same property as a best-effort protocol plus the failure handling.

When total order is required, total order broadcast and multicast provides on top of a reliable protocol a total order. Total order protocols are also called atomic ones.

Those broadcast and multicast guarantees can be formalized into properties:

Validity If a correct process sends a message m, then all involved correct processes will deliver m

Uniform integrity Message m is delivered at most once and, only if the message was previously broadcasted

Uniform agreement If a process delivers m, then all involved correct processes eventually deliver m

Uniform total order Messages are totally ordered

Best-effort protocols are ensuring validity and uniform integrity properties while reliable protocols add uniform agreement. Finally, atomic protocols are reliable protocols with the uniform total order.

1.5 Consensus

As we have seen in section 1.1 about faults, distributed systems need to operate even in case of faults. One of the big challenge in distributed systems is the agreement of multiple nodes where failures can occur. This agreement on a value or a string of value between multiple nodes is called a consensus. A consensus which supports only fail-recover failures must follow some properties [20]:

Agreement Every node decides the same value

Integrity No node decides twice

Validity If a node decides a value v, v was proposed by some node

Termination Every node that does not crash will eventually decide some value

Consensus primitives are highly used in distributed systems to elect a leader, commit a transaction, in a distributed database or locking service such as Google's Chubby [5]. Bitcoin's proof-of-work is also considered as a consensus designed for byzantine failures and a high node count [31].

Paxos [24] is a well known consensus protocol created by Leslie Lamport which supports fail-recover failures. Paxos can be resumed in three steps [4]. First, nodes can send *propose* messages which contain a proposed value and a sequence number. This sequence number is a logical clock giving a time relationship between messages. Second, nodes will send a *promise* message to other nodes stating that it will no longer accept *propose* message with a smaller logical clock than the maximal *propose* sequence number received. If a proposing node receives a majority of the *promise* messages, it means that a progress is possible in the algorithm. Finally, the proposing node having a majority can send an *accept* message to every node in the system. Each node will commit the value as being decided.

Consensus properties are also very similar to the atomic broadcast properties seen in section 1.4. This similarity is actually an equivalence [6] where an atomic multicast provides the same guarantee as a series of consensus.

1.6 State machine replication

A state machine is an abstraction which stores some information and accepts commands to modify its internal state. The information stored is called *state variables* and *commands* allow modification of those variables [36].

In a state machine, every command must be deterministic such as given an initial state S_i and a command C, the final state S_f must always be the same at any time or space such as $S_i \xrightarrow{C} S_f$. A command C is always deterministic if it only uses data contained in S_i and its own parameters.

A common example of non-determinism inside state machines is the local time. If a command uses the local time, S_f at t_i and S_f at $t_{i+\lambda}$ will be different. Most of the time, a non-deterministic command can easily be transformed into a deterministic one by adding the non-deterministic parameter inside the command's parameters.

Various software can map well to the state machine abstraction. We can cite databases, stateful services, etc.

The main problem with state machine is the fault tolerance. State machines are implemented as a single node which means that, even if the state is stored on persistent storage, the availability will be poor. For any process, the availability can only be really achieved by duplicating the state machine on multiple nodes. If done right, in a multi-node setup, one or multiple nodes could fail without compromising the availability of the service.

State machine replication (SMR) is a way to provide high-availability without compromising the consistency of the state machine [36]. State machine replication relies on the fact that a state machine command is deterministic. Given some commands that are deterministic, multiple nodes can end up in the same state if one condition is respected: each node must execute commands in the same order. This order is required because commands are not supposed to be commutative but can be dependant on each other.

State machine replication also improve the read throughput compared to a state machine as a read command can be issued from only one node.

State machine replication is composed of two layers: an ordering layer and the state machine itself. The ordering layer receives commands, orders them and finally dispatches commands to every state machine. Finally, each state machine executes commands received in the same order as received from the ordering layer.

The ordering layer can be implemented in many ways [36]. We can use logical clock ordering, physical clock (with some constraints), a consensus protocol, etc.

1.7 Partially replicated state machine

Replicated state machine holds the complete state on a single node which creates scalability problems. SMR can only scale vertically which costs more and is less performant than horizontal scaling. Horizontal scaling could be achieved by splitting the state into multiple partitions.

Partially Replicated State Machine (PRSM) is a scheme to partition a replicated state machine allowing horizontal scaling. Each partition holds a subset of the complete state and has multiple replicas to ensure fault-tolerance. A PRSM works like a SMR where every command is ordered. Even if the state is partitioned, the global state stays consistent as if the system was only one node.

For example, assume a key-value store storing session information about users. The fault-tolerance is already handled by using active replication.

Thus, the key-value store can be considered as a SMR. As the service is popular, the number of users starts to rise and one server cannot handle the load. One solution would be to shard the data in multiple partitions. Each partition is now handling a subset of the data. A sharding strategy would be to store session information of European citizens on servers in Europe while Americans session information would be in the US. On top of decreased latency for both Americans and Europeans, the load is also split across the two partitions. Of course, each partition is replicated to make sure the datastore is available even when a fault occurs.

Partitioning a state machine creates one challenge: commands must have the same order in every partition of the system to ensure consistency across partitions and replicas. Thus, partitions must guarantee a total order of commands.

With partitions, ordering is much more difficult because every replica in every partitions must agree on a common ordering. The ordering requirements differ from a command requiring only a single partition or a command requiring multiple partitions. When an operation requires only data inside one partition, the operation is called Single Partition Operation (SPO). Those are the easiest to order as they only requires an agreement of the partition replicas on the order of the operation.

When an operation requires data from multiple operations, the operation is called Multiple Partitions Operation (MPO). Those operations requires involvement of multiple partitions containing each multiple replicas to ensure transactions are ordered the same way across nodes.

Chapter 2

Related work

As seen in section 1.7, a total order in a partially replicated state machine is not a straightforward problem because it requires involvement of multiple nodes. In this chapter, different types of ordering protocols for PRSM will be studied. The ordering generally involves all nodes in the system using either a consensus or an atomic broadcast/multicast. One of the protocol presented is using only one server to totally order transactions.

The protocol that *CaMu* is based on are TO-Multicast and Periodic Broadcast which are described in section 2.3.2 and 2.4 respectively.

The protocols presented follow some common properties. First, every protocol ensures a total order of commands. They are assuming an asynchronous system with crash failure. Fault tolerance is handled by replicas.

2.1 Consensus based protocols

Consensus are a base primitive to agree on some value in a distributed systems with presence of faults. In partially replicated state machine's papers [1, 26], the protocol used is a modified consensus algorithm called Ring-Paxos [27]. The protocol creates a suite of consensus problem to broadcast and order transactions. As a consensus is an agreement between every node in the system, each transaction will have a total order because the suite of consensus is consistent system-wide.

The consensus used by RingPaxos for the agreement differs a bit from Paxos seen in section 1.5. In general, Paxos implementation uses a *coordinator* among proposers and acceptors to handle messages [27]. This process will decide which messages should be committed.

To avoid the overhead of sending a lot of unicast to the coordinator, RingPaxos uses a logical ring bus to reduce the network overhead introduced by unicast. Instead of sending messages into a single point, each node sends its messages to the next node which will send the messages it received and its messages to the following node until the coordinator receives the message. This method decreases the number of incoming messages at the coordinator level.

2.2 Sequencer

Unlike the consensus based protocol where each node participate in the order, the sequencer is only using one node which makes the protocol very easy to reason about. This node is called the *sequencer*. The sequencer will receive every command in the system. When received, a command will be assigned a sequence number and then dispatched to the involved partitions. As the sequencer is the only one responsible for the sequence number, the ordering will be consistent inside the system. The sequencer can be implemented with a multicast or a broadcast pattern.

The sequencer can be implemented in different manners. The sequencer can be fixed where one process is elected sequencer or it can be moving where the sequencer responsibility is distributed across the system to distribute the load. The moving sequencer is more complex but is more performant than a fixed sequencer.

Sequencer based ordering has the advantage to be easy to understand and reason about. The latency is also fairly good as the protocol only requires one communication step but this simplicity comes at a cost. First, if we need fault-tolerance, the sequence number needs to be replicated to multiple replicas which increase latency and reduce throughput. Second, the sequencer can only scale by adding a more performant server (i.e. vertical scaling) which is not scalable for a high performance application.

2.3 Timestamp based protocol

As TO-Multicast is based on the Skeen's algorithm idea, Skeen's algorithm will be first presented. Skeen's algorithm is a total order multicast. Next, TO-Multicast will be analyzed based on the explanation of the algorithm. TO-Multicast differs from Skeen's algorithm by supporting replicas for fault-tolerance.

2.3.1 Skeen's algorithm

As explained in section 1.2, physical clock in distributed systems cannot be used in an asynchronous system. Other types of clocks have been introduced such as logical clock [23] to solve synchronization problems. Many protocols use logical clock as a timestamp to provide an ordering of events. A logical clock is an always increasing counter which can be assigned to events requiring causality in the system. When a node receives or sends an event, the logical clock is updated to always be the maximum value. One of the most known protocol using logical clock is Skeen's algorithm introduced by [2]. Skeen's algorithm works by assigning a logical clock to messages. When messages are assigned a consistent logical clock across nodes, messages can be ordered using their logical clock values.

Skeen's algorithm works in steps. Given a message m requiring dispatching to nodes $m.dest = \{n_i, n_j, ...\}$, message m is dispatched to $m.dest$ using a reliable multicast protocol. Reliable multicast protocol ensures every destination node receives the message at most once in a bounded amount of time. When each node in $m.dest$ receives the message m, nodes in $m.dest$ will assign their local logical clock value to the message as a temporary timestamp. The node will then share this temporary timestamp to every node of $m.dest$. The local logical clock shared with the other nodes acts as a vote to decide the final logical clock. When a node receives the vote of every node in $m.dest$, nodes can decide a final timestamp by taking the maximum of logical clock received. As every node received the vote from every node in $m.dest$, the final timestamp is consistent in $m.dest$. Each node updates their local logical clock by taking the maximal vote received plus one.

A message is delivered when it follows two conditions. The message must have a final timestamp and the message timestamp is the smaller than any non-delivered messages timestamp. If those conditions are respected, the total order is consistent because no message m_x will ever exists such as m_x logical clock is smaller than any decided message.

Skeen's algorithm requires a lot of communication steps, one atomic multicast and $n * n$ messages where n is the number of nodes involved in the message. Even if this kind of protocols introduces a higher latency compared to other solutions, it has a good scalability.

Skeen's like protocols are called genuine. A genuine protocol will only involve nodes which are required by the message. Genuine protocols are therefore more scalable compared to non-genuine protocols.

2.3.2 TO-Multicast

TO-Multicast [13] is a total order multicast protocol based on the Skeen's algorithm, while improving fault tolerance by replicating the state of nodes. TO-Multicast stands for Total Order Multicast. Just like partially replicated state machine, the system is composed of partitions[1], themselves composed of multiple nodes acting as replicas.

Assume a message m which requires dispatching to partitions $m.dest = p_i, p_j, ...$ with TO-Multicast ordering protocol. Like Skeen's algorithm, message m will be dispatched to $m.dest$ using a reliable multicast. Next, the protocol has four consecutive steps:

Local timestamping consensus Each partition $p_x \in m.dest$ will run an intra-partition consensus to decide on a logical clock value. This logical clock value will be the vote of partition p_x

Final timestamping Each partition $p_x \in m.dest$ sends its timestamp vote to every other partitions' replica. When a node has received at least one message from every partition in $m.dest$, partitions calculate the final timestamp $m.ts$ as the maximal vote received.

Local clock update When $m.ts$ is decided, the local logical clock of every node in $m.dest$ should be updated to $p_i.ts = m.ts + 1 \mid p_i \in m.dest$.

Message delivering The final timestamp $m.ts$ is not enough to ensure total order. The protocol must ensure that no message will be decided with a smaller logical clock than any decided messages. Thus, a message is only decided when it has a final timestamp and its timestamp is smaller than any non-decided timestamp. When this message is decided, no other messages will have a smaller logical clock because the local logical clock was updated to a larger value.

[1]In [13], partitions are called replication groups. For more consistency with the other sections, the term *"partition"* is used.

TO-Multicast ensures total order in the system because: **1.** Every replica in every partition involved in a message will receive the message thanks to the reliable multicast properties. **2.** The message's final timestamp is consistent across partitions and replicas because every node involved in the message takes the maximal value of all clock votes. **3.** A message is decided only when no other messages can break the consistency.

TO-Multicast ordering protocol is genuine because it only involves nodes which are required by a message which makes it really scalable. Unfortunately, TO-Multicast requires more communications steps than other protocols. In total, it requires two intra-partitions consensus and $n * n$ unicasts where n is the number of nodes involved in the ordering.

2.4 Periodic broadcast

The Periodic Broadcast ordering protocol was introduced in the Calvin research database [41] which uses a partially replicated state machine technique to ensure ACID consistency.

Calvin sequencing layer is distributed on each node to ensure fault tolerance. Transactions are collected on each node during a small timeframe, called rounds. When this timeframe finishes, transactions are grouped in a batch on each node.

Next, the batch is replicated inside its own partitions using either asynchronous or synchronous replication. The asynchronous method is a master-slave replication where every transaction is collected on the master. When the collection timeframe is over, the master will propagate the batch to its replicas in its partition. Synchronous method is using an intra-partition consensus with the Paxos (see section 1.5) protocol. Replication provides fault tolerance in case of a replica failure.

When the replication is finished, each replica in a partition has the same set of transactions. Given a node n_{ij} of partition p_i and replica r_j, node n_{ij} sends a message to every node that has the same replica id r_j. Thus, every node in the system will receive a message from every partition as each replica sends messages to every other replicas in its replica groups. This message contains the partition id p_i of the source partition and transactions which need to be executed by the destination partition.

When every message is received, a node has all multiple partitions operations (MPOs) which need execution on that partition. Those operations plus its own operations just need to be ordered before any execution. The order is decided by selecting transactions received in a round-robin manner. As this process is completely deterministic, the total order stays consistent.

In Periodic Broadcast, a round describes the three steps explained above from the replication to the total order of transactions.

Periodic Broadcast provides an optimal latency as it only requires one communication step to dispatch MPOs but on each round, it involves every partition in the system even if no transaction needs to be sent. For a big cluster, the communication overhead can quickly become a bottleneck.

Chapter 3

CaMu ordering protocol

Before describing our new ordering protocol, the high-level goal of our protocol will be described. Next, an intuition on how the protocol works will be discussed. Finally, a detailed explanation of *CaMu* and the switching protocol will be provided. Those explanations also contain pseudo-codes of each protocol described.

3.1 Goal

Skewness in distributed datastore is a well-known problem and the topic is still an active area of research. Most of the research are focused on partition skewness which is the difference of workload between partitions in the system. Other data access pattern can also introduce a difference of workload which can reduce the performance of the system. The *CaMu* protocol aims to reduce one of this type of skewnesses introduced by partitioning, mainly multi-partitions operations access pattern skewness.

As multi-partition operations (MPOs) are involving multiple partitions, we can assume that the data access pattern between the different partitions will not be the same. A partition will certainly have more *affinity* for some partitions than others. If a partition P_a has more *affinity* with a partition P_b, the total ordering could become an overhead as P_a and P_b are communicating a lot together.

The protocol introduced here aims to reduce latency of MPOs dispatching and ordering for this kind of skewed workload.

3.2 General idea

3.2.1 Ordering protocol

The main difficulty in combining the use of TO-Multicast and Periodic Broadcast comes from the fact that each of them use different mechanisms to determine the order of transactions within each partition: TO-Multicast orders transactions by their timestamps; while Periodic Broadcast does not have the notion of timestamps, the use of synchronized rounds across all partitions allows a deterministic transactions ordering. As a result, partitions can not directly order transactions delivered by TO-Multicast and Periodic Broadcast, as illustrated in Figure 3.1. In the figure, $P1$ tries to dispatch $T5$ to $P3$ and $P4$. $P2$ also tries to dispatch $T4$ to $P3$ and $P4$.

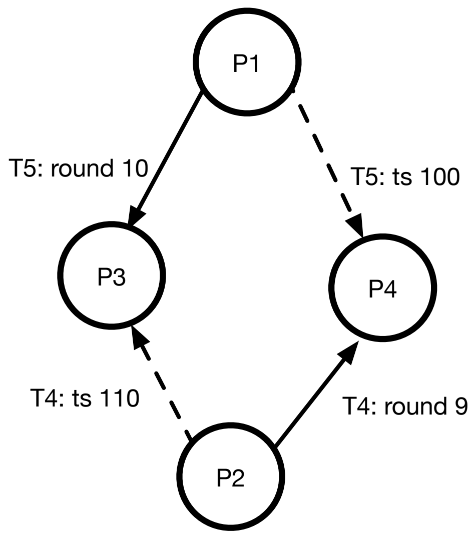

Figure 3.1: Example of partitions using different protocols to order transactions.

Note that since $P1/P2$ both communicate with $P3$ and $P4$, $T4/T5$ will be dispatched to $P3$ and $P4$ with different ordering schemes. However, if there is no mechanism to compare two transactions dispatched by TO-Multicast and Periodic Broadcast, $P3$ and $P4$ may execute $T1$ and $T2$ in different order.

To mitigate this problem, we can associate transactions dispatched by Periodic Broadcast with timestamps, just like TO-Multicast. Partitions now have a common way to order transactions but a transaction timestamp must stay consistent across partitions to ensure total order.

Consistent ordering in our protocol is achieved by making sure that the two protocols assign the same timestamp to any given transaction. Each protocol has a different scheme to assign a timestamp to a transaction.

TO-Multicast assigns a timestamp to a transaction by using an agreement between the involved partitions which means TO-Multicast knows the final timestamp of a transaction only when all votes from other partitions are received. On the other hand, Periodic Broadcast imposes a timestamp decided by the transaction receiver which means the transaction can be assigned any timestamp.

In the case where a transaction requires TO-Multicast and Periodic Broadcast for the dispatching and ordering, the transaction will be first assigned a timestamp with TO-Multicast. When the transaction has a final timestamp value assigned by TO-Multicast, Periodic Broadcast can impose the final timestamp to partitions using the Periodic Broadcast protocol.

In case where a transaction must only be dispatched with Periodic Broadcast, the transaction is assigned a special timestamp value which must follow one condition to ensure total order: the transaction timestamp must be greater than any transactions timestamp executed yet in all involved partitions. Otherwise, a Periodic Broadcast dispatched transaction could break the total order by having a smaller timestamp than an already executed transaction.

Since transactions have consistent timestamps across partitions, each partition must ensure to locally execute transactions in total order by following the ascending order of timestamps assigned to transactions. Therefore, a transaction T can be executed locally on partition P only when P is sure that, in the future, P will never receive a transaction with smaller timestamp than transaction T timestamp. Recall that in our protocol, a transaction can be dispatched to a partition from another partition, either through TO-Multicast or Periodic Broadcast. As this will become clear soon, a partition has to employ different mechanisms to handle transactions dispatched by these two protocols, to ensure the above property.

TO-Multicast knows which transactions timestamps aren't decided because partitions are involved in the clock assignation, while Periodic Broadcast imposes transactions with a given timestamp to other partitions. Therefore, for a transaction dispatched by TO-Multicast, since a partition is directly involved in the ordering of this transaction, by leveraging its local metadata it can know the minimum timestamps of transactions that may ever be dispatched to it. Thus, a partition communicating with Periodic Broadcast must inform other partitions about its state, and more precisely, the lower bound of transaction timestamp that a partition will ever dispatch to other partitions (i.e., future transaction will always have higher timestamp). The information shared is called the Maximal Executable Clock (MEC) as it restrains transactions execution of other partitions by executing only transactions with a timestamp smaller than the MEC.

As the MEC gives a higher bound on transactions timestamp, partitions ensure not to execute higher timestamps than the MEC on a given round. Thus, Periodic Broadcast only transactions can use this information to assign its special timestamp by taking the maximal MEC received. This timestamp value will follow the condition stated above because it will always be higher than the executed transactions timestamp.

3.2.2 Switching protocol

The switching protocol allows two partitions to change the communication scheme currently used between them. This switch will allow the system to react to any change in workloads. As the switching can cause overhead, the switching of protocols between two partitions will not delay the ordering process of transactions currently being ordered between these two partitions.

When two partitions are switching, the main constraint is the round number synchronization between the two partitions. If the two partitions do not switch at the same round (or agree to switch to the same round), the partition, which hasn't switch yet, will wait indefinitely a message from the other partition and thus, block the ordering.

The switching protocol differs depending on the current employed communication scheme between the two switching partitions.

When switching from Periodic Broadcast to TO-Multicast, the two switching partitions communications are scheduled by rounds. Each partition proposes a round to switch by using a separated message channel than the ordering, and then they take the maximal of the proposed round as the round to switch. Thus, the switch must occur at the same round to avoid one partition waiting infinitely for messages from the other partition. Before switching to TO-Multicast, the two switching partitions must wait until Periodic Broadcast transactions dispatching is done. Otherwise, some transactions may not be dispatched on some partitions and thus, break the total order.

When switching from TO-Multicast to Periodic Broadcast, the switching protocol should ensure that after switching, the two partitions have the same round number. Otherwise, like stated above, one of the partitions will wait for the other. When rounds are not used in one of the two partitions (i.e. the partition is only using TO-Multicast), the partition can change its round to synchronize with the other partition without any problem.

On the other hand, when both partitions are using Periodic Broadcast and the two partitions are not in the same round, both partitions will need to jump to a common round number. This jump is not straightforward as partitions using Periodic Broadcast are connected to other partitions using Periodic Broadcast, and so on. Each partition connected with Periodic Broadcast must have the same round number. Thus, every partition connected with Periodic Broadcast will need to jump round.

3.3 Ordering protocol

3.3.1 Description

CaMu uses two communication schemes with different characteristics to dispatch and order MPOs. Those protocols are called Periodic Broadcast and TO-Multicast. Periodic Broadcast is a round-based ordering protocol, provides an optimal latency but requires involvement from every node in the system with periodic communications. TO-Multicast is a genuine (involves only partitions required by the transaction), based on logical clock but requires more communication steps.

A partition communicates with another partition with either Periodic Broadcast or TO-Multicast. Periodic Broadcast is used when the two partitions talk a lot to each other. TO-Multicast is used when partitions are not communicating much together to avoid the overhead of Periodic Broadcast involvement in the ordering. A transaction can be dispatched and ordered with only Periodic Broadcast or TO-Multicast or both depending on the partitions involved.

To totally order MPOs, a common source of truth between the two protocols needs to exist. Logical clock is used as a single source of truth, even with Periodic Broadcast ordered only MPOs. We will see how during the protocol breakdown.

The protocol (Listing 1 and 2) can be decomposed in different steps:

Init phase (line 10-13) First, we get a batch containing MPOs collected in a small timeframe. Next, protocols used by partitions involved in a MPO are saved inside the transaction as a metadata. This metadata ensures a consistent dispatching even when a switching occurs during the dispatching and ordering of the transaction.

Replication phase (line 16-17) For fault tolerance, MPOs are replicated across nodes inside the same partition. An intra-partition consensus is also made on the Local Maximal Executable Clock (LMEC) by taking the minimum of all proposed LMEC inside the partition. The LMEC guarantees that no transaction with a smaller logical clock than the LMEC will be dispatched with Periodic Broadcast from this partition in future rounds.

This guarantee means that another partition can safely execute transactions with a smaller timestamp than the LMEC and still respect the ordering between those two partitions. As explained in section 3.2, the LMEC will inform the state of the partitions execution to other partitions.

Genuine dispatching (line 26-27) When a transaction in the batch has to be dispatched to some partitions using TO-Multicast, the transaction will be dispatched to those partitions with TO-Multicast. The protocol keeps track of the transactions dispatched by the genuine protocol by adding them to a special queue called *pendingQ*. The dispatching is done completely asynchronously and the main loop does not wait for the transaction to be decided by the genuine protocol.

Clock assignation phase (line 29 and 23-24) Each round, the protocol checks if TO-Multicast has transactions which are decided (dispatched and in order). Those transactions are assigned a logical clock by the TO-Multicast protocol where each logical clock is consistent across partitions.

As explained in section 2.3.2, TO-Multicast assigns logical clock in a propose-decide fashion. First, every node sends its maximal logical clock to the other nodes involved. Next, each node decides the logical clock by taking the maximal value and updates its logical clock to the maximal clock received. Finally, a transaction is decided by TO-Multicast when the transaction has the smallest logical clock compared to every transaction which are still being ordered. This ensures that decided transactions respect the total order.

When the clock is assigned by TO-Multicast, the transaction can be in two different states depending if it still needs dispatching with Periodic Broadcast or not. A transaction requires Periodic Broadcast dispatching when a partition involved in the transaction is using Periodic Broadcast.

Those states are:

READY state means that the transaction needs to be dispatched with the Periodic Broadcast protocol.

EXECUTABLE state means that the transaction is waiting to be executed.

When a transaction in the batch does not require any genuine dispatching, the transaction is assigned a special clock value *periodic_broadcast_logical_clock* which was defined during the previous round. The detail will be explained later. As the transaction still requires Periodic Broadcast dispatching, the transaction is in **READY** state.

Periodic Broadcast dispatching phase (line 37-39) As required by the Periodic Broadcast protocol, a message will be sent to every Periodic

Broadcast connected partition. This message will include two information:

- Transactions with a **READY** state in this round
- The local MEC calculated during this round

Next, the protocol waits for messages coming from the other Periodic Broadcast connected partitions. When every message is received, we have a list containing transactions and every local MEC sent. Transactions received can directly added to the execution queue.

Execution phase (line 41-48) Before the execution, the global MEC needs to be calculated by taking the minimum LMEC received (see *Replication phase* and *Periodic Broadcast dispatching phase*). The MEC provides a bounded clock execution for a round to ensure synchronization between sub-protocols (TO-Multicast and Periodic Broadcast). This limitation ensures that a partition P_a does not execute a transaction T_x with a clock value of C_{T_x} when a partition P_b has a transaction T_y which has not yet been dispatched with Periodic Broadcast with a possible future logical clock C_{T_y} such as $C_{T_x} > C_{T_y}$.

With the MEC calculated, we can execute transactions which have a logical clock lower than the MEC.

Periodic Broadcast value update (line 50-51) Before starting a new round, important variables need be updated to ensure correctness in the following rounds.

First, the *periodic_broadcast_logical_clock* is the logical clock value given to transactions which aren't assigned a logical clock by the genuine protocol (i.e. Periodic Broadcast only transactions). One condition is required to ensure total order property: *periodic_broadcast_logical_clock* value should have a greater timestamp than any executed transactions during the round. This condition is required to make sure that transactions will be executable in

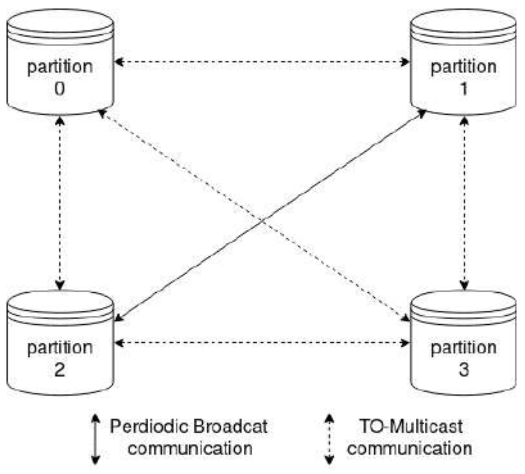

Figure 3.2: Four partitions (with one node each) using different ordering scheme between them. Dashed line are representing communication using the TO-Multicast protocol while solid line represents Periodic Broadcast communications.

the following rounds. As the MEC is bounding the execution, each Periodic Broadcast connected partition knows the maximal clock executed. For simplicity, the *periodic_broadcast_logical_clock* is the maximum of all LMEC received which is always greater than the maximal clock executed.

Second, the logical clock of the genuine protocol is updated only to make sure that our protocol advances when no genuine communication occurs. As explained in the *Clock assignation phase*, the logical clock is updated by the genuine protocol itself but when no genuine communication happens, the logical clock is not updated. As explained in the *Replication phase*, the LMEC is based on the logical clock of the genuine protocol. If this logical clock is not updated, we may no longer execute transactions as the MEC will not increase.

```
Algorithm variables
    pendingQ: a queue storing MPO waiting genuine dispatching
    executablePQ: a priority queue storing MPO waiting to be executed
    genuine: genuine sub-protocol implementation
    round: round number
    low_latency_clock: clock value assigned to exclusive low latency MPO
    genuine: interface to the TO-Multicast protocol

upon new round
    $MPOs \leftarrow$ get batched MPOs
    for $MPO \in MPOs$ do
        for $p \in MPO.partitions()$ do
            $MPO.protocols[p] \leftarrow get\_protocol(p)$
    $local\_mec \leftarrow get\_mec(genuine, decided\_by\_genuine)$

    C-PROPOSE([$MPOs$, $local\_mec$])
    wait C-DECIDE([$MPOs$, $local\_mec$])

    // Store transactions which need Periodic Broadcast sequencing.
    $readyQ \leftarrow$ new queue()
    for $MPO \in MPOs$ do
        if GENUINE $\notin MPO.protocols()$ then
            $MPO.setLogicalClock(periodic\_broadcast\_logical\_clock)$
            $readyQ.add(MPO)$
        else
            $genuine.Send(MPO)$
            $pendingQ.add(MPO)$

    $decided\_by\_genuine \leftarrow genuine.get\_decided()$
    for $MPO \in decided\_by\_genuine$ do
        if $MPO \in pendingQ$ && PERIODIC_BROADCAST $\in MPO.protocols()$ then
            $pendingQ.remove(MPO)$
            $readyQ.add(MPO)$
        else
            $executablePQ.add(MPO)$
```

Listing 1: Pseudocode describing the *CaMu* ordering protocol (Part 1).

```
36:
37:     Send the local_mec and readyQ MPOs for every Periodic Broadcast com-
    municating partition
38:     $executablePQ \leftarrow executablePQ + readyQ$
39:     $received\_MPOs, mecs \leftarrow$ **wait** messages from low latency protocol con-
    nected nodes
40:
41:     $mec \leftarrow min(mecs)$
42:
43:     **for** $MPO \in executablePQ$ **do**
44:         **if** $MPO.logical\_clock < mec$ **then**
45:             Execute $MPO$
46:             $executablePQ.remove(MPO)$
47:         **else**
48:             **break**
49:
50:     $periodic\_broadcast\_logical\_clock \leftarrow max(mecs)$
51:     $genuine.logical\_clock \leftarrow max(genuine.logical\_clock, periodic\_broadcast\_logical\_clock)$

52:
53:     $round \leftarrow round + 1$
54:
```

Listing 2: Pseudocode describing the *CaMu* ordering protocol (Part 2).

3.3.2 Ordering example

As the protocol explanation can be a bit abstract, here is an example which contains four partitions and two transactions to order. Figure 3.2 represents the protocols used to communicate between the partitions. As described in the schema, only partition 1 and 2 are talking together with the Periodic Broadcast protocol. Every other partition pair is communicating with the genuine protocol. For more clarity, we are only assuming one replica per partition.

The two transactions will be received approximately at the same time at the partition 1 and 3. The partition 1 receives the transaction **A** which also requires dispatching on partition 0 and 2. The partition 3 receives the transaction **B** which also requires dispatching on partition 2. The ordering of those two partitions is schematized in Figure 3.3. In the following

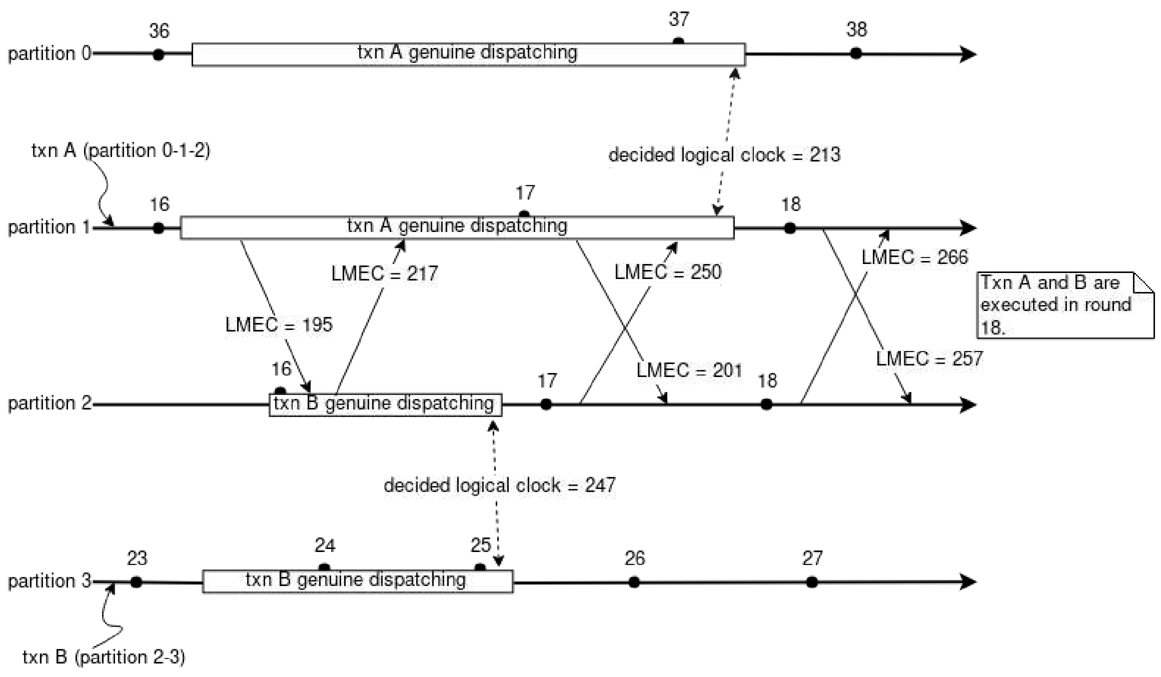
partition 0
36
37
38
txn A genuine dispatching
decided logical clock = 213
txn A (partition 0-1-2)
partition 1
16
17
18
txn A genuine dispatching
LMEC = 217
LMEC = 250
LMEC = 266
LMEC = 195
Txn A and B are executed in round 18.
partition 2
16
txn B genuine dispatching
17
LMEC = 201
18
LMEC = 257
decided logical clock = 247
partition 3
23
24
25
txn B genuine dispatching
26
27
txn B (partition 2-3)

Figure 3.3: Timing diagram representing the ordering of two transactions **A** and **B**. Some useless communications between partitions have been removed to improve readability.
Horizontal arrows represent the time and black point on them informs a round change. Dashed vertical line represents a genuine agreement on the logical clock and normal vertical line are Periodic Broadcast sequencer messages.

paragraphs, each transaction dispatching will first analyzed separately and afterwards, the ordering relation and the execution will be discussed.

Transaction A. As this transaction has a TO-Multicast and a Periodic Broadcast connected partition, the protocol first needs to dispatch the transaction with the genuine protocol to assign it a logical clock. After some time, partition 0 and 1 agree on a logical clock of 213. Next, the transaction will dispatched with the Periodic Broadcast at the round 18 as the round 17 had already begun when transaction **A** was decided. At round 18, partitions 0, 1 and 2 have the transaction **A** in their execution queue. The actual execution will be discussed in the last paragraph.

Transaction B. This transaction only requires a TO-Multicast dispatching which provides, at the end, a logical clock of 247. At round 17, the transaction **B** is in the execution queue of partition 2 and 3.

The execution of A and B. As shown in Figure 3.3, the transaction **B** is available (round 17) before **A** (round 18) even if **A** has a smaller logical clock. To ensure that transaction **B** is not executed before **A**, **B** needs to be delayed until at least transaction **A** execution to avoid breaking the total order.

The MEC is providing this delay by bounding the logical clock execution for every round. As you can see, for round 17, $MEC = min(250, 201) = 201$ and as $201 < (B_{clock} = 247)$, the transaction **B** can't be executed in round 17. In round 18, as transaction **A** has been decided by the genuine protocol and dispatched by the Periodic Broadcast, the partition $1_{LMEC} > 213$ and we have the relation $(A_{clock} = 213) < (B_{clock} = 247) < MEC$ which means that **A** and **B** can be executed in total order in this round.

3.3.3 Correctness proof

We are assuming two transactions T_a and T_b. $T_a < T_b$ relation implies T_a is ordered before T_b on every correct process involved in T_a and T_b. The "$<$" relation is the total ordering property. $T_a \prec T_b$ for a process P_a implies that T_a is ordered before T_b inside process P_a and has no other guarantee on the ordering for other processes. The "$\prec$" relation is the local ordering property.

Assume our sequencer receives two multi-partition transactions T_x and T_y on two different partitions. Assume T_x and T_y are accessing two common partitions P_a and P_b. By their timestamp values t_{T_x} and t_{T_x}, the total order relation is expected to be $T_x < T_y$. By contradiction, we assume that we have $T_x \prec T_y$ on partition P_a and $T_y \prec T_x$ on partition P_b which break the total order property. A violation of total order could happen because: **a.** T_x or T_y has a different timestamp value between partition P_a and P_b.

b. If T_x and T_y have a consistent timestamp across partitions, partition P_b executes T_y before T_x because P_b does not know that a transaction T_x exists such as $T_x < T_y$.

A transaction can be dispatched to its involved partitions either only using one of the communication patterns, or using a combination of both, which we call a *hybrid* dispatching. In the following text, we analyze whether the above cases are possible per dispatching type.

First, we examine if case **a.** is possible for each dispatching type for a transaction T_x involving multiple partitions:

TO-Multicast dispatching TO-Multicast properties[1] ensures a consistent logical clock across nodes involved in the transaction.

When TO-Multicast assigns a logical clock to a transaction, the protocol is executed in three steps:

1. Each partition makes an intra-partition consensus to agree on a vote for the final logical clock value.
2. Every node sends its logical clock vote to the other nodes involved in the transaction.
3. Each node uses the maximal vote received to be final logical clock for the given transaction.

As every node choose the maximal vote received, it's evident that every node will have the same logical clock.

Periodic Broadcast sequencing Before T_x is dispatched, the partition receiving the transaction T_x will assign a logical clock which is equals to *periodic_broadcast_logical_clock*. As the *periodic_broadcast_logical_clock* is based on the MEC which is the result of an inter-partition consensus, the *periodic_broadcast_logical_clock* will be consistent between

[1]`ftp://ftp.irisa.fr/techreports/1998/PI-1162.ps.gz`

replicas inside a transaction. The inter-partition logical clock value will also stay consistent as the logical clock is decided only by one partition.

Hybrid dispatching T_x is dispatched in two steps. First, T_x is dispatched with the TO-Multicast protocol to partitions which are communicating with the TO-Multicast protocol. When T_x is decided by the TO-Multicast protocol, it has a logical clock which is consistent as seen above across TO-Multicast partitions. Second, T_x is dispatched with Periodic Broadcast with the same logical clock. Every involved node in T_x now has received the transaction with a consistent logical clock.

Next, we examine whether case **b.** is possible for each dispatching type with two multi-partitions transactions T_x and T_y with a common partition P_a.

TO-Multicast dispatching TO-Multicast properties ensure that the transaction T_x will be decided iff there is no transaction T_y such as $T_y < T_x$ relation is possible. If there is no transaction with a smaller logical clock, no transaction will ever have a smaller logical clock and thus, no transaction will ever break the total order.

Periodic Broadcast sequencing Unlike the base Periodic Broadcast protocol, transactions are not simply deterministically ordered and then executed. Transactions are assigned a logical clock to make sure the ordering is consistent with other transactions. The logical clock assigned to a Periodic Broadcast only transaction T_x has the value of *periodic_broadcast_logical_clock* which was calculated during the previous round. This value was calculated by taking the maximum of all LMEC received such as $periodic\_broadcast\_logical\_clock_r = max(LMECs)$. As $MEC_{r-1} = min(LMECS)$ and MEC_{r-1} is the maximal logical clock executed during the previous round, the transaction T_x can be ordered safely.

Even if some transactions have the same logical clock, transactions can be deterministically ordered just like Periodic Broadcast do.

Hybrid dispatching Since an hybrid transaction T_x delays its Periodic Broadcast dispatching to get a logical clock with the genuine dispatching, the protocol must ensure that execution of T_x respects the total order. The total order is consistent if no transaction with a smaller logical clock than transaction T_x is received in the future when T_x is executed. The execution guard is given by the MEC which limits the execution to transactions which only have a smaller logical clock than the MEC value.

MEC only apply to partitions communicating with Periodic Broadcast because TO-Multicast already ensures total order by only deciding a transaction when it has the smallest logical clock. For partitions talking with Periodic Broadcast, the MEC is an agreement of the maximal executable clock of every partition and every replica (thanks to the consensus on the LMEC). As every partition sends the maximal executable clock locally and the MEC is the minimum of those values, no node can execute a transaction which breaks the total order.

Since case **a.** and **b.** have been demonstrated to be impossible, by contradiction, the ordering protocol is proven to be correct.

3.4 Switching protocol

The protocol switching is divided in different parts. First, a different protocol is used between TO-Multicast $\rightarrow$ Periodic Broadcast and Periodic Broadcast $\rightarrow$ TO-Multicast. In addition, when we are switching from TO-Multicast to Periodic Broadcast, we need to handle different cases because those cases are more tricky as we will see later.

The switching protocol is implemented as a state machine which breaks the switching into different steps. The switching logic is executed after each ordering round and this logic can take multiple ordering round to finalize the switch because each state is completely asynchronous to avoid blocking the main loop.

Parallel switchings aren't authorized for a question of correctness. For example, if two parallel switches are happening at the same time, some round synchronization process could create an inconsistent state in the system.

The adaptive algorithm, which optimize the ordering latency by changing the protocol currently used between two partitions, is completely separated from the actual switching code. The adaptive algorithm can be implemented in many ways. Right now, the adaptive algorithm is executed every ten to twenty rounds and collects data, mainly MPO access patterns, to take decision whether to switch or not with other partitions.

When the adaptive algorithm find a possible optimization, the switching command will be added to a queue which will be consumed by the switching protocol. When the adaptive algorithm is run again, the switching command queue is cleared to have the most recent switching decision.

3.4.1 Common

The first step of a switch is an agreement between the two partitions. This agreement will make sure that every node in both partitions are aware of the switch. If one of the partition is already busy with an other switch, the switch initialization is aborted and the switch will be retried later on. The agreement is also used to agree on several values that will be useful in the switch and saved in an object called *switch_metadata* for convenience:

switching_round is a round value at which both partitions can switch in a synchronized fashion

partition_type is the type of the other partition that will be used to decide which switching protocol should be used

```
1: **Algorithm variables**
2:     $state$: current state of the switching
3:     $switch\_metadata$: metadata based on the agreement of the two switching partitions
4:
5: **upon** switching agreement with partition $P_{switch}$
6:     **if** $get\_protocol(P_{switch}) = PERIODIC\_BROADCAST$ **then**
7:         $state \leftarrow INIT\_SWITCH\_TO\_TOMULTICAST$
8:     **else**
9:         **if** $switch\_metadata.partition\_type = FULL\_TOMULTICAST$ **or** $get\_this\_partition\_type() = FULL\_TOMULTICAST$ **then**
10:            $state \leftarrow INIT\_SWITCH\_TO\_PERBROADCAST\_FULL\_TOMULTICAST$
11:        **else**
12:            $state \leftarrow INIT\_SWITCH\_TO\_PERBROADCAST\_HYBRID$
13:
```

Listing 3: Pseudocode describing the common part in the switching protocol.

As explained above, the switch is divided in different cases depending on the type of protocol currently used. Line 6 to 12 (Listing 3) handles the different cases possible. Line 9 to 12 handles the different cases when switching from TO-Multicast to Periodic Broadcast depending on the partition type. A partition can have one of those three types:

Full Periodic Broadcast is a partition which communicates only with the Periodic Broadcast protocol.

Full TO-Multicast is a partition which communicates only with TO-Multicast.

Hybrid is a partition which communicates with TO-Multicast to some partitions and with Periodic Broadcast to the others.

Each case is then explained separately for more clarity as they are completely isolated from each other.

3.4.2 Periodic Broadcast → TO-Multicast

Switching from Periodic Broadcast to TO-Multicast requires 4 steps (Listing 4):

Transition phase (line 1-4) When the switching round happens (agreed upon during the *init* phase), both partitions will change from the Periodic Broadcast protocol to a transition protocol. This transition protocol is running both protocols at the same time meaning that new transactions received will be dispatched with TO-Multicast for the switching partitions while the Periodic Broadcast message will still be sent to ensure the next step in the switching will eventually terminate.

Waiting for transactions requiring Periodic Broadcast dispatching (line 5) Before switching to TO-Multicast, the switching protocol needs to make sure that transaction no longer requires Periodic Broadcast dispatching with the switching partition by checking the pending transactions queue. Otherwise, some transactions may not be dispatched to the switching partition which will break the total order property. It means that the next step in the protocol switch is delayed until this condition is fulfilled.

Switching round agreement (line 5 to 9) Like the *init* phase, an agreement is made between both switching partition and their respective replicas to find a round to switch. This agreement takes the form of a consensus with two primitives $A - PROPOSE$ and $A - DECIDE$.

Genuine switch (line 10 to 13) When the switching round is reached, the partition can switch to the genuine partition.

```
1: upon $state = INIT\_SWITCH\_TO\_TOMULTICAST$ and
   $current\_round = switch\_metadata.switching\_round$
2:     $state \leftarrow FINISH\_PERIODIC\_BROADCAST\_DISPATCHING$
3:     $set\_protocol(P_{switch}, TRANSITION)$
4:
5: upon $state = FINISH\_PERIODIC\_BROADCAST\_DISPATCHING$
   and no transaction left for Periodic Broadcast dispatching with partition
   $P_{switch}$
6:     $A - PROPOSE(P_{switch}, current\_round + delta)$
7:     $switch\_metadata.switching\_round \leftarrow wait\ A - DECIDE(P_{switch})$
8:     $state \leftarrow SWITCH\_TO\_TOMULTICAST$
9:
10: upon $state = SWITCH\_TO\_TOMULTICAST$ and
            $current\_round = switch\_metadata.switching\_round$
11:     $set\_protocol(P_{switch}, TO - MULTICAST)$
12:     $state \leftarrow$ null
13:
```

Listing 4: Pseudocode describing the switching from Periodic Broadcast to TO-Multicast.

3.4.3 TO-Multicast → Periodic Broadcast

TO-Multicast → Periodic Broadcast requires two different cases depending on the partitions types as explained in section 3.4.1.

One or two full TO-Multicast partition(s)

Before switching to Periodic Broadcast, both partitions need to have the same round number. Otherwise, the periodic message sent by the partitions won't be synchronized and the two switching partitions will block.

If at least one of the switching partition is a full TO-Multicast partition, the switching is really straightforward because round number of full TO-Multicast partitions can be changed without interfering with the protocol as no Periodic Broadcast message needs to be sent.

The protocol can be described in two steps (Listing 5):

Round change (line 1-10) As explained above, the round number needs to be synchronized between the switching partitions. The round update depends on the type of the partitions involved:

Two full TO-Multicast partitions In the *init* phase, both partitions made an agreement on some values such as *switch_metadata.switching_round.* As we are creating two new hybrid partitions, any round value is valid as long as it's synchronized. *switch_metadata.switching_round* is used because it's already available and the value is the same on every node in the switching partitions.

One hybrid and one full TO-Multicast partition An hybrid partition is talking with other partitions with the Periodic Broadcast sequencer protocol which means it's impossible to change the round number without also switching those connected partitions. Fortunately, the full TO-Multicast partition can switch to any round number. The full TO-Multicast partition will wait for the first Periodic Broadcast message from the hybrid partition to receive the current round value.

After updating round number, switching partitions switch their protocol to the transition protocol (like *Periodic Broadcast* $\rightarrow$ *TO-Multicast*).

Periodic Broadcast switch (line 11-24) Before switching to the Periodic Broadcast protocol, a round is executed with the transition protocol to synchronize LMEC between the switching partitions. LMEC are required to make sure that variables such as *periodic_broadcast_logical_clock* are updated before receiving transactions with Periodic Broadcast. After this synchronization round, the protocol can be updated to Periodic Broadcast.

Two hybrid

As explained above, Periodic Broadcast requires connected partitions to have the same round number to terminate a round. Every partition which

```
upon $state = INIT\_SWITCH\_TO\_PERBROADCAST\_FULL\_TOMULTICAST$
    if $get\_partition\_type() = FULL\_TOMULTICAST$ and
            $other\_partition\_type = FULL\_TOMULTICAST$ then
        $current\_round \leftarrow switch\_metadata.switching\_round$
    else if $get\_partition\_type() = FULL\_TOMULTICAST$ then
        $msg \leftarrow$ wait LOW LATENCY message
        $current\_round \leftarrow msg.round$
    $switching\_round = current\_round$
    $set\_protocol(P_{switch}, TRANSITION)$
    $state \leftarrow SYNCHRONIZE\_MEC$

upon $state = SYNCHRONIZE\_MEC$ and $current\_round = switching\_round + 1$
    $set\_protocol(P_{switch}, PERIODIC\_BROADCAST)$
    $state \leftarrow$ null

```

Listing 5: Pseudocode describing the switching from TO-Multicast to Periodic Broadcast with one or two full TO-Multicast partitions.

is connected by Periodic Broadcast and has the same round number at one time compared to other partitions are forming a graph, called Periodic Broadcast connected graph.

When we have two hybrid partitions, we can have two partitions which are in the same graph or in two different graphs. When we are initializing the protocol switching, it is impossible to tell if two partitions are in the same graph or not due to the network being asynchronous.

The protocol switching for two hybrid partitions requires updating the round number in one or two graphs which can described in multiple steps:

Graphs locking Before updating the round number of the graphs, the protocol switch needs to make sure that no other switch occurs in the graph. This locking is required to, first, keep track of every partition involved in the graph and, second, avoid switching with other partition inside the graph to avoid unknown partition in the mapping. If there is already a switching occurring in the graph, the switch needs to be aborted and every partition locked should be unlocked.

The graph locking can be implemented in different ways. We choose an implementation where the switching partitions are doing the heavy lifting which is easier to manage and reason about. The switching partition will run a consensus with every of its neighbours partitions to lock them. Every neighbour will share their own neighbours. Next, the switching partition will lock those partitions and receive neighbours information until every partition in the graph as been discovered.

During the graph locking, the switching protocol can also discover that there is only one graph because both partitions are in the same graph.

Switching round agreement If we use the same technique as 3.4.2 with *switching_round* and we have two distinct graphs, we may block a graph for a non-negligible amount of time because the two graphs have different round values. The lowest round value graph would have to catch up to the highest round value graph where the round difference could be big. The highest round value graph will need to wait and will block during the catch-up.

To avoid blocking one graph, the two switching partition will agree on two different values. First, *switching_round* will be the round value at which the graph will switch. This value is either the same if the protocol detects that we only had one graph or different if we have two distinct graphs. If we two different switching round value, we can reduce the blocking time but the two graphs will be at different round value. The same round value is given by $final\_round$ which jump the round value of both graphs to the same value. $final\_round$ is an agreement between the two switching partitions ensuring both graphs have the same round at the end.

Graphs round update propagation Thanks to the round locking, the two switching partitions know which partitions are in their graphs.

The round update propagation will be handled by the switching partitions. *switching_round* and *final_round* will be sent to every locked partition.

LMEC Synchronization Like in section 3.4.3, before switching to Periodic Broadcast protocol, we need to switch to the transition protocol between the two switching partitions after upgrading the round number. The transition protocol will synchronize the various values of the *CaMu* protocol.

Periodic Broadcast switching After one synchronization round, both switching partitions can switch to the Periodic Broadcast protocol safely.

For the sake of brevity, the protocol shown in Listing 6 is only describing the switching protocol for the two switching partition and not for other partitions involved in the graph.

```
Algorithm variables
    $locked\_partitions$: a set containing partition ids of locked partition

upon $state = INIT\_SWITCH\_TO\_PERBROADCAST\_HYBRID$
    $unvisited\_partitions \leftarrow$ set(get every neighbour partition using Periodic Broadcast)
    while $unvisited\_partitions.size()! = 0$ do
        $partition\_id \leftarrow unvisited\_partitions.pop()$
        $A - PROPOSE(partition\_id, PARTITION\_LOCK)$
        $unvisited\_partitions.add(wait\ A - DECIDE(partition\_id))$
        $locked\_partitions.add(partition\_id)$

upon locking is finished
    $A - PROPOSE(P_{switch}, (switching\_round, final\_round))$
    $(switching\_round, final\_round) \leftarrow wait\ A - DECIDE(P_{switch})$
    $propagate\_round\_info(switching\_round, final\_round)$

upon $current\_round = switching\_round$
    $current\_round \leftarrow final\_round$
    $set\_protocol(P_{switch}, transition)$
    $state \leftarrow SYNCHRONIZE\_MEC$

upon $state = SYNCHRONIZE\_MEC$ and $current\_round = final\_round + 1$
    $set\_protocol(P_{switch}, PERIODIC\_BROADCAST)$
    $state \leftarrow$ null

```

Listing 6: Pseudocode describing the switching from TO-Multicast to Periodic Broadcast with two hybrid partitions.

Chapter 4

Implementation

The *CaMu* ordering protocol has been implemented on top of an existing database called Calvin [44]. This database is fully-ACID compliant even with distributed transactions thanks to the partially replicated state machine properties. The database is not intended to be used in production as it was only used to validate and test PRSM techniques.

In this chapter, a high-level view of the database architecture will be presented. Next, the implementation of the TO-Multicast protocol, the new adaptive and switching protocol will be studied [1]. Finally, other small features will be discussed.

4.1 Calvin database

The Calvin database is a distributed database which means the database runs on a cluster of multiple nodes. Each database node is a multi-threaded program implemented in C++. The database is only running on Linux at the time of writing.

[1]The complete codebase can be found at `https://github.com/charlesvdv/spec_calvin`

4.1.1 Architecture

Each Calvin node is implemented in three layers to increase modularity:

Sequencing layer manages the total ordering of the commands

Scheduling layer handles locks and the scheduling of distributed transactions

Storage layer is an abstracting of the storage implementation

Each layer is only communicating with the layer above or below it. For the sequencing layer, the layer above are the clients which sends transactions. The storage layer marks the persistence and end of processing of transactions.

Clients

Clients are creating the load for the database. In a real-world context, clients are applications and services using our database to store and retrieve data. As Calvin was designed to be a research database, such clients do not exist.

In Calvin, clients are benchmarks implemented to test the database in various workloads. Calvin implements two types of clients: TPC-C [9] and a micro benchmark.

TPC-C is a benchmark specification specially designed to test performance of transactional datastores (OLTP). TPC-C simulates a wholesale supplier which must manage, sell, or distribute a product or a service.

The benchmark has five types of transactions; each having different access patterns. Some transactions are designed to be single partition operations while other transactions are designed to access another partition.

Thus, the TPC-C benchmark involves at most two partitions in a transaction.

Micro benchmark is a synthetic benchmark built to stress test the database on multiple partitions operations. In this benchmark, the user can customize the percentage of multiple partitions operations but also the number of partitions involved in a transaction. As those information stay static, the benchmark does not really match real world workloads but it is great to show edge cases and test the database on other types of workloads thanks to its customizability.

Each client is implemented in two parts. First, there is the transaction generation which will create new transactions. Second, there is the actual execution of the transaction. A client will define how transactions are executed based on the benchmark used. In TPC-C, transactions may require data from other partitions to be executed. This logic is implemented inside the client.

In theory, transactions are generated at any point in time and Calvin collects them in batch to reduce the communication overhead. In practice, the sequencing layer will ask for the current batch and the ordering layer will generate a fixed number of transactions collected in a batch. Thus, the transaction latency in Calvin doesn't account for the batching time of transactions.

Sequencing layer

The sequencing layer is using the protocol described in section 2.4. The implementation takes the form of an infinite loop running in a thread. Each iteration is a round in the protocol. At each iteration, a batch is generated by calling the client to generate transactions.

The Calvin database is using an optimization to order batch of messages. In theory, the sequencer thread should wait to receive messages from all partitions in the system as explained in section 2.4. The actual implementation directly send MPOs to the scheduler layer of other partitions. The scheduler is then responsible to order transactions deterministically based on the MPOs received.

The batch replication inside a partition is replicated using *ZooKeeper*. *ZooKeeper* [19] is a distributed key-value store used for various use-cases in large scale distributed systems.

Scheduling layer

The scheduling layer receives batches of transactions that have been ordered during a round of the sequencer layer. Different schedulers have been implemented to study PRSM transaction executions [40] but only one has been used in this thesis: the deterministic scheduler.

The deterministic scheduler is implemented like the sequencing layer using a infinite loop running in a thread. When a batch containing transactions is received, transactions contained in the batch are executed on the storage layer using the client currently in use.

Some transactions can be dependent which means they require read values from other partitions to be executed correctly. When a dependent transaction does not have the required data to be executed, the execution is delayed until enough data is received to execute the transaction properly.

Storage layer

As Calvin is a research database, the storage layer is only an in-memory object containing the tables and rows data to avoid the disk bottleneck.

Though a production ready implementation could use any storage able to store key-value pairs efficiently. A full-blown relational database like *postgres*, *mysql*, ... or an optimized key-value storage like *RocksDB* could be used to implement a storage layer.

4.1.2 Communication

Communication is an important part of the database as nodes must communicate to replicate and dispatch transactions. The communication is using *ZeroMQ* [34] which is a distributed messaging systems. *ZeroMQ* is implemented as a message queue that can run without any brokers and can be embedded in an application as a library.

In the codebase, communications are abstracted by channels which defines a local or distant destination queue. Those channels are abstracted inside a communication multiplexer object. For example, Calvin uses a channel to communicate with the scheduler.

Messages are encoded and decoded using the Protocol Buffers library [16] (also known as *protobuf*). *protobuf* is an open-source library and compiler developed by Google to handle their inter-process communications in their services. The library is language agnostic as it uses a custom Domain Specific Language (DSL) to describe the message content. When one service uses a message described in the *protobuf* DSL, the DSL must be compiled into a file specific to the language. This file defines classes or methods to encode and decode data from their binary format. In our case for C++, *protobuf* creates a header file **.h** and a source file **.cpp**.

The big advantage of Protocol Buffers library is the performance and size of the resulting message. Services are also using a common auto-generated file so each service has the same behaviour even when using different languages.

4.1.3 Deployment and configuration

The Calvin database has two executables: one is executed on every node to form the distributed database; while the other is used to deploy the configuration and run each database node. The former is called *db* while the latter is called *cluster*.

The *db* executable uses two configuration files. The first configuration is called **myconfig.conf** and defines various parameters that are useful to tune while testing and benchmarking the database. A few examples of configuration entries are given in table 4.1. The other configuration file is called **deploy-run.conf** and contains information about every node in the system. Each line consists of the partition and replica number, a node id, a resolvable host address and a port number. Those information are used by the *db* executable to learn in which partition the process is and to know the address of other nodes.

The *db* executable is executed in terminal through a command line. This command is composed of two parameters:

1. The node id of the future *db* process
2. The benchmark type: **t** for TPC-C and **m** for micro

The *cluster* executable is another executable used to deploy and run the *db* executable in a cluster in an automatic way.

One configuration file is required for the *cluster* executable and is called **dist-deploy.conf**. This file is the same file as **deploy-run.conf** because the *cluster* executable copy **dist-deploy.conf** to every node as **deploy-run.conf**. The copying is done with the *scp* program. *scp* stands for Secure Copy and uses the SSH protocol [45] to communicate. The SSH protocol stands for Secure SHell and allows remote access to distant computers.

When the *cluster* executable is called, we pass the benchmark type as an option which will be used to run *db*. Next, the **dist-deploy.conf** file is copied to every node in the cluster as explained above. Finally, the execution on remote nodes is done through *ssh*. Thus, the node running the *cluster* executable must have unrestricted *ssh* access to every node. As *ssh* is present by default in a lot of Linux distribution, the deployment can be done really easily.

When every *db* executable is running, the standard output *stdout* and error *stderr* are piped through the controller node using *ssh*. This pipe allows easy debugging in a multi-nodes setup and easily detects errors without checking every node by hand.

Variables	**Description**
max_batch_size	Defines the number of transactions in a batch
duration	The duration that the benchmark is running
multi_txn_num_parts	The number of partitions in MPO during a micro benchmark
rw_set_size	Number of keys read and written in a transaction
...	...

Table 4.1: A non-exhaustive list of tunable parameters in **myconfig.conf** file.

4.2 TO-Multicast protocol

The TO-Multicast protocol did not have any open-source implementation available so one has been implemented using the description of the protocol found in [13]. This description is using one thread to control the timestamping and one thread for each message. Each thread is implemented as a state machine.

In our case, a single threaded implementation was better because we wanted a fair comparison with the Calvin sequencer which is single threaded. Our single threaded implementation is also using a state machine approach where pending transactions are stored in a queue. Each transaction is assigned a state inside the queue. This state is updated when a transaction has made progress towards the total order. States are:

WaitingReliableMulticast The transaction was just added to the queue and requires a multicast to other nodes.

GroupTimestamp The transaction will be assigned a timestamp within its partition (also called replication group) after a intra-partition consensus.

InterPartitionVote The node sends its transaction timestamp vote to other nodes in other partitions involved in the transaction.

ReplicaSynchronization When the transaction is assigned the final logical clock, the node will run an intra-partition consensus to update the timestamp to the highest value inside the partition. The transaction is then added in new queue waiting to be decided.

The TO-Multicast implementation can be used in two different ways. First, the protocol can be used as the main ordering protocol in standalone mode or embedded inside another protocol like *CaMu* for example.

When used in standalone mode, batches are generated periodically. At each iteration, decided operations are dispatched to the scheduling layer.

When used in embedded mode, a public Application Program Interface (API) is used to submit transactions, retrieve ordered transactions and get various information about the protocol. In embedded mode, the batch generation and scheduler communications are obviously disabled. Listing 7 shows the public API exposed to *CaMu*.

4.2.1 Limitations

The implementation of TO-Multicast is not complete because the reliable multicast and the consensus was not implemented because of a lack of time.

Transactions are sent using an unicast to every partitions involved in the transaction. For the consensus, methods abstracting the consensus have been developed. Currently, those methods are only returning the same value but should be implemented to agree on a value between replicas.

```
// Dispatch a transaction with TO-Multicast.
void Send(TxnProto *message);
// Get transactions that are ordered.
vector<TxnProto*> GetDecided();

// Get the LMEC.
LogicalClockT GetMaxExecutableClock();

// Update the logical clock.
void SetLogicalClock(LogicalClockT c);
// Get the current logical clock.
LogicalClockT GetLogicalClock();

// Verify if TO-Multicast still has transaction which requires
// dispatching with the given partition and with the periodic
// communication protocol. Used for the switching protocol.
bool HasTxnRequiringDispatchingForPartition(int partition_id);
```

Listing 7: TO-Multicast public interface used by *CaMu* and the switching protocol.

In the case where the consensus are developed, the threading architecture exposed in [13] should be used. The architecture is one thread handling the logical clock and consensus while a thread pool should be handling inter-partition communications and the transaction state.

Thus, the current implementation can only run with partitions containing one replica. As we will see later, this does not have a huge impact on the evaluation of the protocol because the evaluation has been done with only one replica per partitions to maximize the number of partitions in the system.

4.3 *CaMu*

4.3.1 Ordering protocol

The implementation uses the same idea as the Calvin sequencer implementation explained in section 4.1.1.

At the start of a round, a batch is generated. At the end of a round, a batch containing transactions ordered during the round is sent to the scheduler. As transactions can be partially dispatched with the genuine protocol, transactions received in round r can be ordered in round $r + \lambda$.

The detail and pseudo-code of the *CaMu* ordering protocol can be found in section 3.3.

Limitations

Just like the TO-Multicast implementation, the consensus required to replicate transactions inside a partition has not been implemented due to a lack of time.

4.3.2 Switching protocol

As explained in section 3.4, the switching is implemented with multiple state machines where some steps are shared between the different switching protocols.

The switching protocol is running on the same thread as the ordering protocol but the switching is never blocking the ordering with a network request for example. The state of the state machine is updated when an event occurs. Those events can be network requests, pooled information (from the TO-Multicast protocol for example), etc.

Limitations

Just like the other protocols, the consensus primitive uses to make an agreement between the two switching partitions has not been implemented in a fault tolerant way. When this primitive is required, each partition sends a message and the maximum value is taken as final value. This scheme ensures consistency in the decision between the two partitions without using a consensus.

4.3.3 Adaptive algorithm

The adaptive algorithm takes decisions whether the partition should switch protocol with another partition. Right now, the adaptive protocol is really straightforward and not smart because this challenge requires a thesis on itself.

The adaptive algorithm analyzes the transactions access pattern to partitions. In each round, partitions touched by the ordering protocol are collected. After a few rounds ($X = 10$~15 rounds), if a partition has been touched a lot (more than ~75% touched during X rounds), the protocol recommended is the Periodic Broadcast. When a partition is not often touched (~25% touched during X rounds), the protocol recommended is TO-Multicast. If the current protocol is not the recommended protocol for a given partition, a switch command is added to a queue. This queue is then consumed by the switching protocol.

Values above are completely arbitrary and requires more fine-tuning to improve performance and avoid useless switching.

4.4 Other improvements

Due to some requirements, various features have been implemented to improve benchmarks and to make the database easy to work with. Those features can be used by modifying the **myconfig.conf** configuration file.

On top of the features, some bugs were also found as the database was not designed to be used with a different sequencer. Those bugs will not be discussed as they are closely tied to the implementation of the database and are not very interesting.

4.4.1 Pluggable sequencer

Since the purpose of this thesis is to create a new ordering layer, multiple ordering layers are required to validate and compare the performance. Each ordering layer should live on the same codebase to have the same behaviour/performance and the same feature sets. Thus, each ordering protocol was implemented with the same interface to make them pluggable at runtime.

Right now, the database supports TO-Multicast, Periodic Broadcast and *CaMu*.

4.4.2 Partitions access skewness

As this thesis is based on the fact that there is some skewnesses when MPOs access partitions, benchmarks should reflect this skewness to validate our protocol. Various skewness distributions were implemented to select which partitions would be chosen for a given transaction. Those skewnesses are injected when the client is created. It means skewness can be introduced when using TPC-C or micro benchmarks.

The skewness distributions implemented are discussed in section 5.1.2.

4.4.3 Closed loop clients

An open loop client sends transactions without waiting for a result; while a closed loop client will wait for a response before sending another transaction.

The base client implementation of Calvin was using the open loop client but closed loop client can show interesting information about latency sensitive workloads. This information is mostly the throughput of the system with a limited number of clients as the throughput is then a direct function of the latency.

For the closed loop implementation, open loop clients were used to generate transactions but a limitation in pending operations is enforced.

4.4.4 Independent MPOs

As this thesis is only focused on the ordering layer, the scheduler layer should be as performant as possible to avoid a possible bottleneck in our system. This means the transaction execution must be as fast as possible to have a minimum impact on the latency of the scheduling layer.

As seen in section 4.1.1 about the scheduling layer, some transactions can be dependent. Those transactions require network communications between partitions to execute properly. The independent MPOs feature assumes that transactions are never dependent so it does not wait for dependent data from other partitions. This feature decreases latency but transactions are no longer executed properly which is not required to calculate ordering layer performance.

4.4.5 Partition communication pattern

As already explained, a partition communicates with another partition with either Periodic Broadcast or TO-Multicast. To ease the development and tests, one can statically define the communication scheme for each partition in the cluster. To speed up the development, **dist-deploy.conf** and

myconfig.conf have been used to specify this information because those already have parser available.

A default protocol is defined in the configuration file **myconfig.conf**. This protocol will be used by default if no other information is given. If another protocol than the default wants to be specified for a partition, the **dist-deploy.conf** must be edited. A new entry has been added to specify for each partition a list of partitions that shouldn't use the default protocol. This way, a static communication pattern can be defined for partition inside the system.

A static switching configuration has also been added in **dist-deploy.conf** file to easily test the switching protocol in a deterministic way. The idea is to force a switch of one partition with another one after a specified running time. The new entry in **dist-deploy.conf** can be added to all partitions and consists of a list containing a partition id to switch to and the time of the switch in seconds.

Chapter 5

Evaluation

In this chapter, the benefits of *CaMu* will be measured by comparing it to two protocols: Periodic Broadcast and TO-Multicast. Those ordering protocols will be tested on various workloads defined as benchmarks.

We first explain our test setup, and then we analyze the strengths and weaknesses of our protocol.

5.1 Benchmarks

Benchmarks are an important part of the evaluation as they will expose the performance of the protocol tested. We present two benchmarks: a real-world simulation and a synthetic benchmark.

Next, we discuss the injection of skewness in the system with different statistic distributions giving various properties to the skewness.

5.1.1 Benchmark types

As seen in section 4.1.1, the Calvin database uses two benchmarks: TPC-C and a micro benchmark. In our evaluation, we use the same benchmarks as Calvin to evaluate the performance of the *CaMu* protocol.

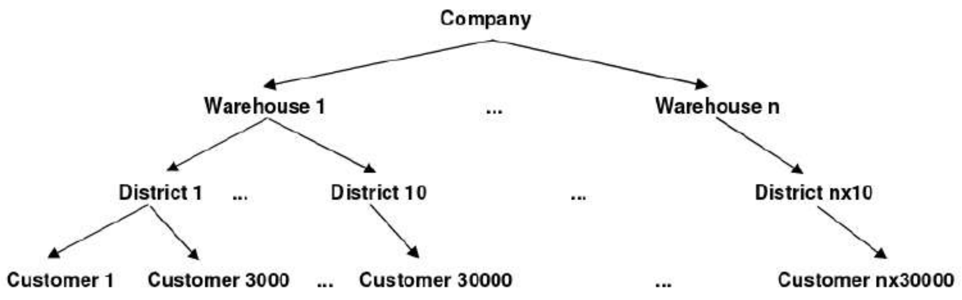

Figure 5.1: The data model TPC-C uses to represent a real-world workload. This model describes a wholesale company composed of different sales districts. Each district is composed of clients. [14]

TPC-C

TPC-C is a benchmark which is supposed to model a real-world workload by simulating a wholesale company (as shown in Figure 5.1). The company has several warehouses which contain each several sales districts. Those sales districts contain a fixed number of clients using the service offered by the company.

The benchmark is built to test OnLine Transaction Processing (OLTP) database and has five different types of transactions explained in table 5.1 [10]. Some types of transactions are generated more than others to simulate real customers behaviours.

The benchmark contains distributed transactions. *New order* and *Payment* transactions sometimes require remote data. When the number of nodes is high, the number of MPOs reaches 10% of the total transactions generated by the system.

Micro

The micro benchmark is a synthetic benchmark used in addition to TPC-C to validate the database in various distributed transaction setups. The benchmark has two tunable parameters:

- The percentage of MPOs

Type	Description	Distribution
New order	Enter a new order of a customer	45%
Payment	Update customer balance to reflect payment	43%
Delivery	Deliver orders	4%
Order-status	Retrieve recent customer purchases	4%
Stock-level	Monitor warehouse inventory	4%

Table 5.1: Transactions types emitted by the TPC-C benchmarks. Each transaction creates a different workload type such as read-only, write-heavy, etc. Transactions are not distributed equally to match a production environment. [10]

- The number of partitions in an MPO

5.1.2 Skewness sources

As already explained, MPOs partition access pattern is generally skewed in real world scenario because partitions have more *affinity* with some partitions than with others.

In real workload, the skewness is function of the partition layout and transactions submitted in the system. As there is no existing benchmark testing the skewness in MPOs partition access, some have been introduced inside the existing benchmarks. The skewness is introduced in the benchmarks by tuning the probability to communicate to a given partition.

The distribution can be used with any benchmarks presented above as it only decides the partition used for an MPO.

Uniform distribution

The uniform distribution has already been used in the Calvin database benchmarks. The distribution has the same probability to pick any partition.

Unfortunately, this distribution does not introduce any skewness in the system as a partition is communicating with all partitions with the same probability. Therefore, this distribution does not simulate real-world workload.

Even though the Uniform distribution is not realistic, we use this distribution in our evaluation because it represents the skewness of most available benchmarks.

Zipfian distribution

The Zipfian distribution [43] is a statistic law mapping very well to real-world problems such as the word occurrence frequency in a linguistic corpus, social and physical approximations [43]. The law states that *"the frequency of an item is inversely proportional to its raking by frequency"* [12].

More formally, it means that given N is the number of elements, k_n the rank of element n, s a constant characterizing the distribution, the frequency f of an item is defined as:

$$f(k_n; s, N) = \frac{1/k_n^s}{\sum_{n=1}^{N}(1/n^s)}$$

The s constant is a tuning parameter which modifies the speed at which the frequency is reduced. When $s > 1$, items will tend faster to the 0 limit while when $s < 1$, the items will tend slower to 0 limit.

Figure 5.2 represents the cumulative distribution of the Zipfian distribution. In this figure, we can easily see that a big s will increase the initial slope of the function. When $s = 1$, the first element k_1 has a probability of ~0.35. When $s = 2$, the probability of k_1 is ~0.65.

The Zipfian distribution has been chosen because we think that this distribution is suitable to describe a real-world skewness in a partitioned database. This hypothesis is supported by multiple observations. First, various papers about partition skewness are using the Zipf's law to simulate a real-world partitioning skewness [33, 38]. Next, a cloud database benchmarking tool from Yahoo called *YCSB* also uses the Zipf's law to introduce partitioning skewness in the system [7]. Finally, intuitively, we can assume that a partition *affinity* with other partitions will follow approximately the Zipfian as some partitions will be accessed a lot by MPOs because the data is highly coupled while some partitions will never be accessed because no MPOs combine those type of data together.

To pick partitions involved in a MPO, partitions are deterministically organized in a specific order. Next, the probability to choose a partition is calculated from the position of the partition in the list. The first partition in the list will have the highest probability to be chosen while the last partition will have the lowest probability.

Partitions specific order is completely deterministic because the information in **dist-deploy.conf** and more precisely, the static protocol assignation (see 4.4.5) is used to define an order. More precisely, partitions configured to use Periodic Broadcast are on the top of the list while partitions configured with TO-Multicast are at the bottom of it. This way, partitions communicating with Periodic Broadcast receive more transactions than partitions communicating with TO-Multicast.

Deterministic distribution

The deterministic case is designed to be a best scenario for the *CaMu* protocol. A partition will always send MPOs to a fixed set of partitions. This distribution is the best case because, when the switching is smart enough, every partition in the set will communicate with Periodic Broadcast which has the best latency. As TO-Multicast is genuine, not using it does not introduce any overhead.

Deterministic distribution simulates a datastore having multiple types of data which are unrelated to each other.

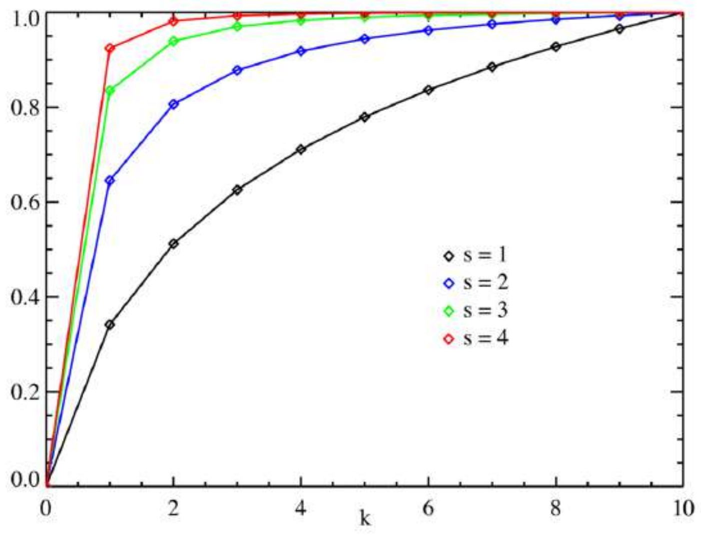

Figure 5.2: Cumulative distribution function of the Zipfian distribution with $s = 1, 2, 3, 4$ and $N = 10$. The horizontal axis is the index k. The vertical axis is the probability (from 0 to 1). [43]

The fixed set is defined in **dist-deploy.conf** just like the Zipfian distribution. In this distribution, only partitions configured to use Periodic Broadcast are added to the fixed set.

5.2 Infrastructure

As the database is distributed on multiple nodes, the evaluation of the database must use a cluster of servers. For this purpose, we used the *Grid5000* infrastructure [17]. *Grid5000* is testbed for research in various areas such as big data, distributed systems, etc. There are approximately 1000 nodes and 8000 cores split in 8 sites in France: Lille, Luxembourg, Nancy, Lyon, Grenoble, Sophia, Nantes and Rennes. The infrastructure is quite versatile by having various technologies like InfiniBand, GPU, etc.

The access to servers is done through a reservation system controlled using a command line application. When a server is reserved, we have bare-metal access to it meaning there is no overhead introduced by any virtualization technology.

A connection to a specific reserved server is done in multiple steps using mostly *ssh* to connect to the servers. First, the user must log in inside the *Grid5000* infrastructure using its *ssh* key provided in advance. Now, the user can log directly to any machine that the user reserved. If no reservation has been made, the user can still access a "site frontend" which allows access the shared storage of the site.

For the experiment, two configurations were used: a Local Area Network (LAN) and a Wide Area Network (WAN). The LAN is only using one cluster while the WAN uses multiple clusters spread across France as we can see in table 5.2.

The *graphene* cluster has been chosen to run the LAN evaluation because it is the only cluster available having at least 100 nodes. This cluster also has a 10G InfiniBand interface which provides a better latency and throughput compared to Ethernet. The average round-trip time between two nodes in the cluster is 200ns for Ethernet and 35ms for the InfiniBand interface.

The WAN cluster has been chosen to have as many different sites as possible. As we can see each cluster has a different hardware but it doesn't matter much in our application as the network will certainly be a bigger bottleneck compared to the difference in servers' performance. The connection between sites is a 10Gbit/s fiber which is dedicated to research purpose. Table 5.3 shows the latency from Lille to the other sites in the *Grid5000* infrastructure.

Conf.	Cluster information				
/	Name	Site	Count	CPU	RAM
LAN	graphene	Nancy	100	4cores@2.53GHz	16GB
WAN	genepi	Grenoble	10	8cores@2.5GHz*	8GB
	griffon	Nancy	10	8cores@2.5GHz*	16GB
	econome	Nantes	5	16cores@2.2GHz*	63GB
	parapide	Rennes	10	8cores@2.93GHz*	24GB
	suno	Sophia	10	8cores@2.26GHz*	32GB
	granduc	Luxembourg	10	8cores@2GHz*	16GB

Table 5.2: Information on servers used during the LAN and WAN experimentation. ** signifies CPU are bi-socket board.*

Site	Average round-trip time (ms)
Luxembourg	11.8
Nancy	9.57
Lyon	/
Grenoble	15.5
Sophia	19.4
Nantes	26.2
Rennes	27.5

Table 5.3: Average round-trip time at various *Grid5000* site from the Lille site. Lyon was in maintenance during the evaluation.

5.3 Setup

As seen earlier, there are various parameters that can be tuned to run the evaluation. This section discusses the configuration used to get the evaluation result discussed in the next section.

First, the configuration used for **myconfig.conf** (see section 4.1.3) is analyzed.

The batch duration defines how long transactions are batched before launching the ordering protocol. This batch duration defines the throughput of our system as the system is designed to assign a fixed number of transactions per batch. As we will see later, the latency of transactions when using *CaMu* is really sensitive to the batch duration so we took the smallest round duration possible. In the LAN, we used a 5 milliseconds batch duration for a cluster size of 100 nodes and 3 milliseconds for a cluster size of 50 nodes; while in WAN, 20 milliseconds batch duration were used. The rationale behind those numbers will be discussed in the next section.

The number of transactions in each batch (i.e. transactions processed by rounds) decides the throughput of our system with the batch duration parameter. As a small round duration was used, the number of transactions per batch is only one. Otherwise, the throughput will be too high for the system to handle. For example, in LAN, transactions per second generated by each partition is $\frac{1}{0.003} = 333.33\ txns/s$ meaning that a cluster of 100 partitions processes $333.33 * 100 = 33,333.33\ txns/s$ which is not much but as explained in the next section, the evaluation hit a bottleneck.

The number of partitions in a multiple partition operation is modified when running the micro benchmark. We used two and four partitions to evaluate each ordering protocol.

When running benchmarks, independent MPOs (see section 4.4.4) are used to reduce as much as possible the overhead introduced by the scheduler layer. This way, benchmarks are more consistent and measure more precisely the latency introduced by the ordering layer.

Only open loop clients (see section 4.4.3) have been used during the evaluation because the closed loop client was showing the same information as the open loop client in an another format.

Second, the **dist-deploy.conf** configuration file (see section 4.1.3) has been configured to add each node information based on the cluster information. As seen in *"Limitations"* sections from chapter 4, no replication scheme has been implemented inside our protocol. Thus, for 100 nodes, we have 100 partitions with one node per replicas. As we wanted to test *CaMu* on large scale clusters and no cluster has more than 100 nodes in *Grid5000*, in the end, we cannot use more than one node per partition.

As seen in section 4.4.5, **dist-deploy.conf** has been modified to allow static protocol assignment to partitions. This method has been used to define an initial state between partitions. Partitions will use a protocol by default to communicate if not stated otherwise. If the configuration states otherwise for two partitions, the two partitions will use the protocol which is not the default. TO-Multicast is used as default in our evaluation.

The switching protocol was not enabled during the evaluation because the adaptive switching algorithm is not yet performant enough to reach a latency optimum. Thus, each benchmark was run in a static setup without any switching unless stated otherwise.

For the LAN evaluation, each partition uses Periodic Broadcast with 4~6 other partitions. For the WAN evaluation, each partition is also using 4~6 Periodic Broadcast communicating partitions but those partitions are generally located at the same location. This configuration makes sense because partitions with highly related data should be located as near as possible from each other to avoid any network overhead. When running with a Uniform distribution for the skewness, all partitions are communicating with Periodic Broadcast because Periodic Broadcast provides the better latency in this case.

Each benchmark was run for 30 seconds after the database bootstrapping. Each configuration was run four times to make sure the results are consistent.

For the Zipfian distribution, the tuning parameter s is set arbitrarily to 2 to accentuate the skewness.

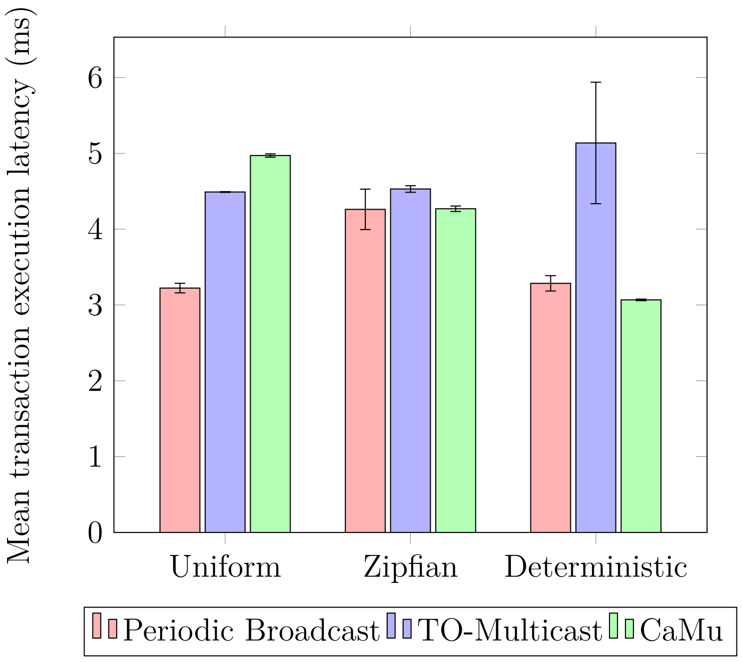

Figure 5.3: TPC-C benchmark in LAN with 50 partitions

5.4 Results

As explained in the previous section, each protocol has been evaluated in LAN and WAN. The LAN evaluation has been run on a 100 nodes clusters while the WAN was a 45 nodes clusters spanning 6 locations.

5.4.1 LAN

TPC-C benchmark

Figures 5.3 and 5.4 shows the latency of each ordering protocol with the TPC-C benchmark with 50 partitions and 100 partitions respectively. Before diving into *CaMu* performance, Periodic Broadcast and TO-Multicast latency will be analyzed.

First, we can notice that Periodic Broadcast has a better latency than TO-Multicast with 50 partitions. With 100 partitions, TO-Multicast has a better latency. The latency of Periodic Broadcast increases when the num-

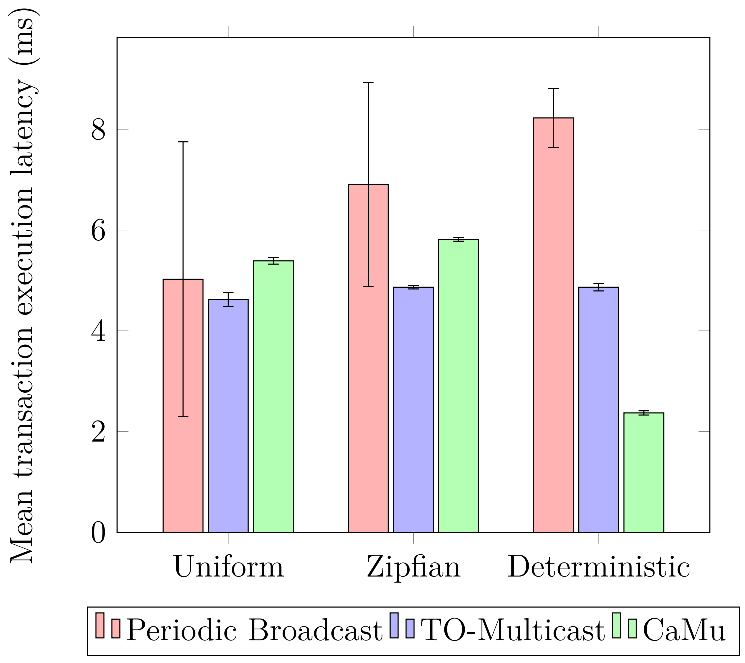

Figure 5.4: TPC-C benchmark in LAN with 100 partitions

ber of partitions increases because the protocol requires a synchronization every round of all partitions in the system. When the number of partitions is high, the synchronization becomes a bottleneck. On the other hand, TO-Multicast only involves the partition required by a transaction meaning that the number of partitions in the system does not matter for this protocol.

Next, the skewness introduced in the system with different distributions demonstrates that TO-Multicast and mostly Periodic Broadcast have worst latency with skewness than without. The increased latency of Periodic Broadcast can be explained by the fact that some partitions may be busier than others because it receives more transactions. In this case, the partitions could be slower and therefore, slowing down the synchronization process. For TO-Multicast, the protocol is less affected because the protocol is genuine and thus, a slow partition will have less impact on the ordering of other partitions.

Finally, the *CaMu* latency will be compared against TO-Multicast and Periodic Broadcast with every distribution.

A comparison of *CaMu* protocol with the two other protocols with the Uniform distribution shows that the protocol does a bit worse than the two others. As the Uniform distribution is dispatching MPOs with random partitions, the evaluation was configured to only use the Periodic Broadcast protocol because no skewness was introduced in the system. Therefore, the difference between *CaMu* and Periodic Broadcast shows the overhead introduced by the protocol.

On the other hand, when the distribution used is the Zipfian, *CaMu* is better in average than the others protocols because the protocol is either better than Periodic Broadcast or TO-Multicast depending on the number of partitions in the cluster.

The Deterministic distribution has a much better latency result because a partition will always dispatch an MPO to a fixed set of partitions. This allows a nice optimization where this fixed set is using Periodic Broadcast for low latency dispatching and other partitions are using TO-Multicast which does not impact performance. This way, we are creating small clusters of Periodic Broadcast communicating partitions with a very low latency as Periodic Broadcast does not scale well with the number of partitions. In this case, the Deterministic distribution is a best-case-scenario for *CaMu*. As each partition is communicating with a small set of partitions with Periodic Broadcast, MPOs are dispatched with a very low latency.

Notice also the difference of latency between the Zipfian distribution with 50 and 100 partitions, the former has an average latency of 4.97ms while the latter has an average latency of 5.38ms. In theory, as most partitions are using TO-Multicast to communicate, the latency should not increase as seen with the TO-Multicast evaluation above. This latency increase is coming from the fact that the batch duration has been increased from 3ms for 50 partitions to 5ms for the largest cluster. As seen in Figure

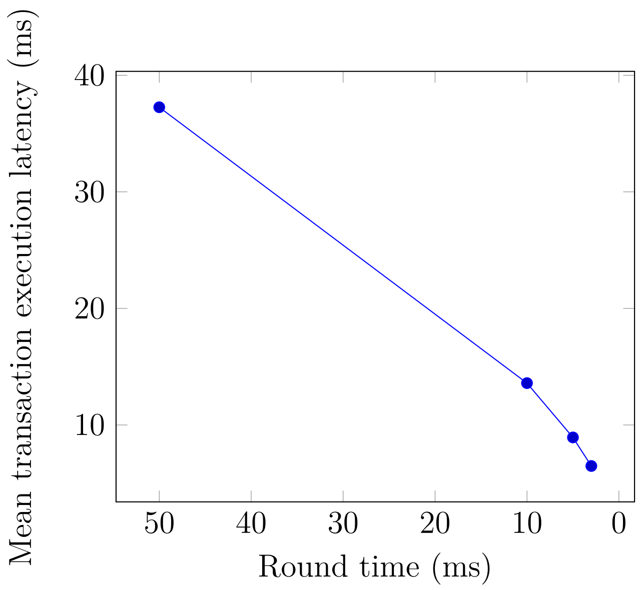

Figure 5.5: Relation between round time and mean transaction execution latency in *CaMu*

5.5, the latency of *CaMu* is a direct function of the batch duration because our protocol is highly relying on rounds to order transactions. In the implementation of *CaMu*, the protocol only sends transactions to the scheduler at the end of rounds to ensure that logical clock are properly ordered.

Unfortunately, an implementation issue with *ZeroMQ* is limiting the batch duration value. *ZeroMQ* does not support reliable message dispatching meaning that when the throughput increases, *ZeroMQ* drops messages [18]. As reducing the batch duration is increasing the throughput, the batch duration is currently limited up to a certain point. For 50 partitions, this limit is 3ms while with 100 partitions, the database is limited at 5ms. The problem is even worse when using TO-Multicast as the protocol uses a lot of small messages to assign logical clock to transactions.

Micro benchmark

Figures 5.6 and 5.7 show the results using the micro-benchmark and 100 partitions. The former is using two partitions per MPOs while the latter

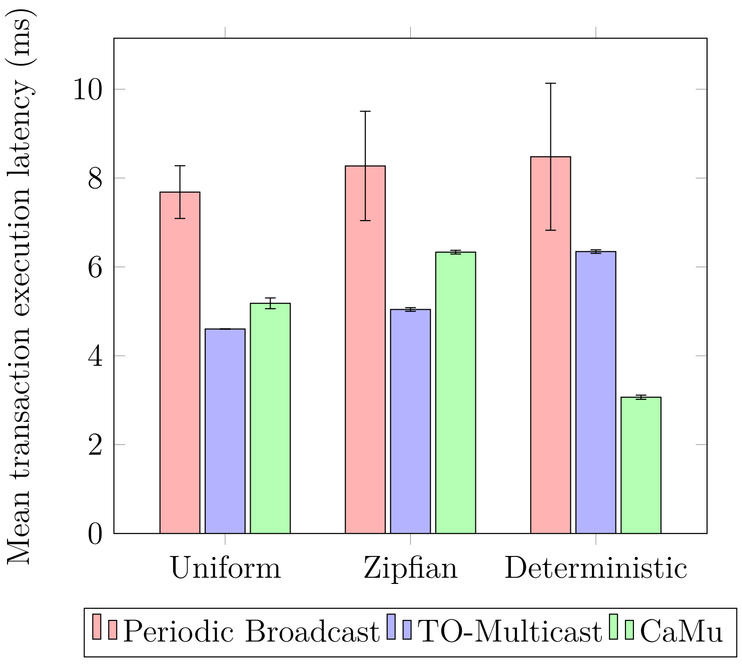

Figure 5.6: Micro-benchmark in LAN with 2 partitions accessed by MPOs and 100 partitions

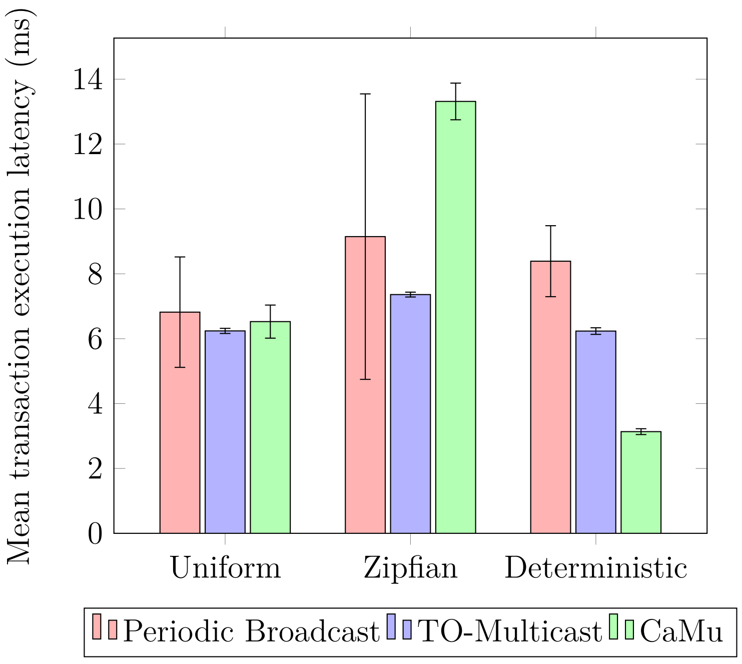

Figure 5.7: Micro-benchmark in LAN with 4 partitions accessed by MPOs and 100 partitions

is using four partitions per MPOs. The first noticeable result is the huge latency of *CaMu* on Figure 5.7 with the Zipfian distribution.

This latency bump shows one limitation of the current protocol that will be addressed in the next chapter about future works. When a MPO requires more partitions to be dispatched to, the probability to have some partitions requiring TO-Multicast ordering and Periodic Broadcast is higher than an MPO with a smaller partition set. Currently, when *CaMu* does an hybrid ordering, the MPO is first dispatched and assigned a final logical clock with TO-Multicast. In the round following the final clock assignment, *CaMu* dispatches the MPO to the rest of the partitions with Periodic Broadcast. Thus, *CaMu* is ordering the transaction sequentially as one protocol is used after the other.

This problem is only visible with the Zipfian distribution because it is the only distribution using both protocols to order partitions. The Uniform distribution is using Periodic Broadcast only while the Deterministic distribution could use TO-Multicast but if possible, a partition using Periodic Broadcast will be used first.

Notice also that the TO-Multicast protocol is handling MPOs with 4 partitions worse than Periodic Broadcast compared to MPOs with 2 partitions. TO-Multicast has a bigger latency when increasing involved partitions for a transaction because TO-Multicast needs to send more messages to order the transaction between partitions. On the other hand, Periodic Broadcast always dispatch the transaction in one round which means the latency should be constant in this case.

The Deterministic distribution with TO-Multicast shows an improved latency because the system was starting to drop messages at the default throughput and thus, the throughput has been reduced to get some results.

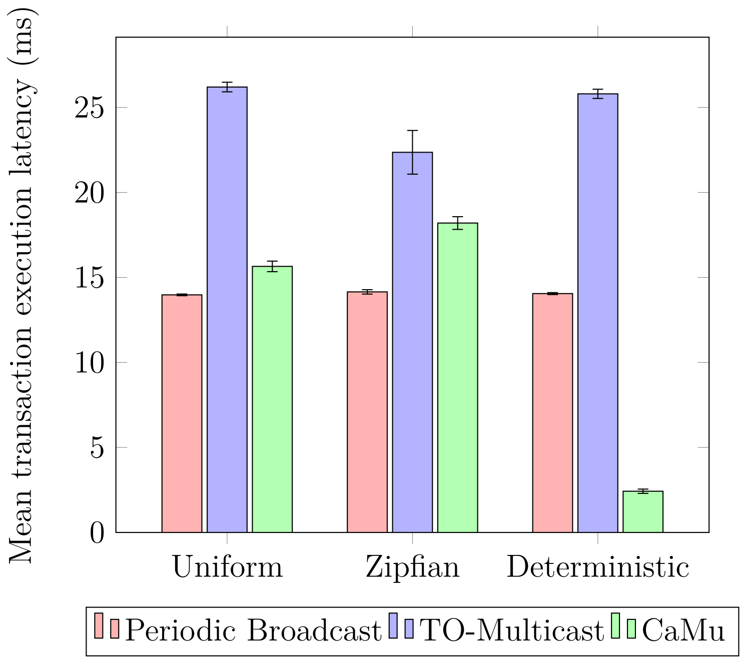

Figure 5.8: TPC-C benchmark in WAN with 45 partitions

5.4.2 WAN

TPC-C

Figure 5.8 shows the evaluation result for the TPC-C benchmark in a WAN configuration. In WAN, the TO-Multicast protocol has a much worse latency compared to Periodic Broadcast and *CaMu*. This result comes from the fact that TO-Multicast requires much more communications than Periodic Broadcast and the latency is much higher compared to the LAN. In TO-Multicast, the protocol must use at least 2 inter-partitions communications to order transactions while Periodic Broadcast requires only one. Also, the WAN cluster is relatively small and thus, as seen in section 5.4.1, Periodic Broadcast performs better in small clusters compared to TO-Multicast.

The *CaMu* latency in the Deterministic distribution is very small because, as explained in the previous section, partitions have been configured to have more *affinity* with partitions in the same cluster. Thus, the Deter-

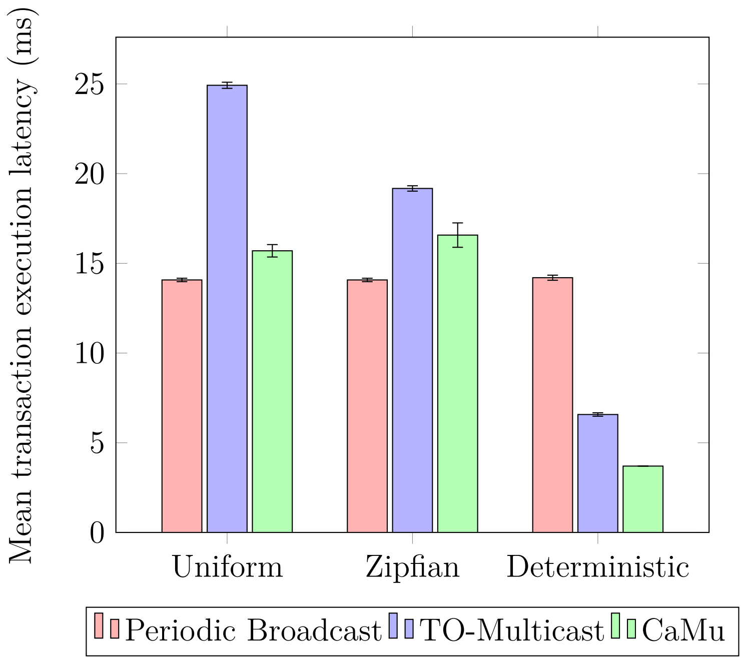

Figure 5.9: Micro-benchmark in WAN with 2 partitions accessed by MPOs and 45 partitions

ministic distribution only sends messages to partitions in the same cluster which provides a really small latency between partitions.

Micro-benchmark

Figure 5.9 and 5.10 shows the average latency for the micro benchmark in a WAN configuration with respectively two and four partitions involved in every MPO. The latency bump observed in Figure 5.7 can is also present in Figure 5.10. This latency bump is, of course, a common problem whenever using the *CaMu* protocol.

Compared to Figure 5.8, the TO-Multicast latency with a Deterministic distribution is much lower. During the TO-Multicast micro benchmark, the system was dropping with the default batch duration meaning the benchmark run a much lower throughput than other tests. This explain why the latency is so low for this benchmark.

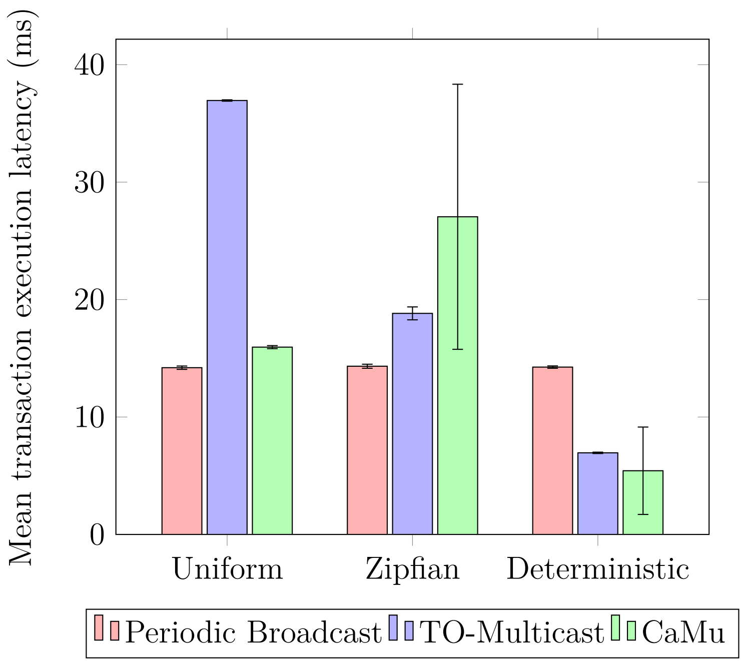

Figure 5.10: Micro-benchmark in WAN with 4 partitions accessed by MPOs and 45 partitions

5.4.3 Switching protocol

In Figure 5.11, the impact of protocol switching is shown by comparing a static and a dynamic communication configuration. The static configuration is already optimized to yield a good latency. The dynamic configuration starts with the same configuration as the static one but the adaptive algorithm can decide to switch some partitions to another protocol. The figure shows that the switching between protocols does not really impact the latency of transactions. The latency is not really impacted because the switching has been developed to avoid as much as possible blocking the ordering of transactions.

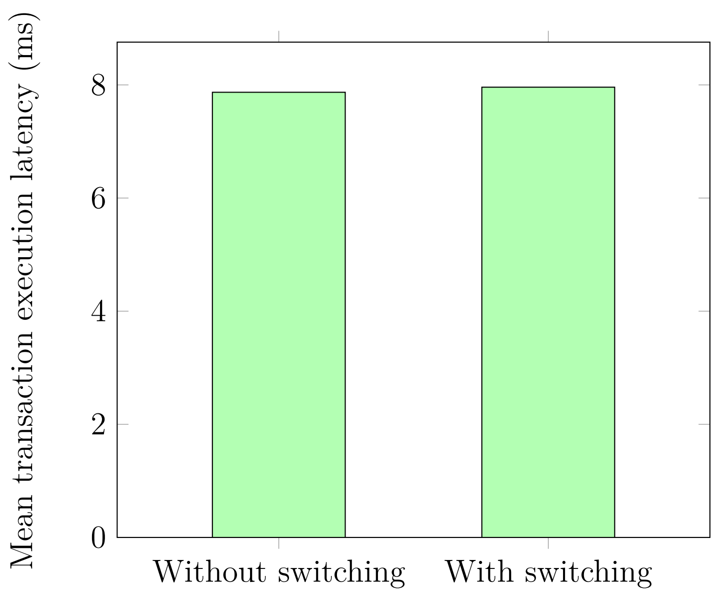

Figure 5.11: Overhead introduced by the switching in TPC-C benchmark with Zipfian distribution and 100 partitions

Chapter 6

Future work

This chapter aims to give solutions to problems highlighted in the previous section. Solutions are split into two categories: implementation and protocol. Implementation problems are issues with the underlying database implementation which can be solved while the protocol improvement directly impacts how *CaMu* works.

6.1 Implementation

As the Calvin implementation has been designed from the ground up to support only one ordering layer, some problems have been discovered while testing the database in big deployment. In this section, some clues to solve the problems encountered are discussed.

6.1.1 Messaging queue

As explained in the previous chapter, *ZeroMQ* does not support reliable delivery of messages which causes the system to drop messages when a lot of messages are received. To resolve this problem, two solutions are possible. First, a system on top of *ZeroMQ* could be re-created to re-send lost messages.

Another solution would be to use another message queue supporting reliable delivery. There are a lot of solutions available such as *ActiveMQ*, *Kafka*, etc. A non-exhaustive list can be found at [37].

Both solutions should be relatively easy to integrate into the Calvin codebase as the communication between nodes is already abstracted inside the codebase.

6.1.2 TO-Multicast batching

Right now, each transaction is handled separately by TO-Multicast but *CaMu* is handling transactions in batch. Thus, TO-Multicast could use a batching mechanism to improve its latency by reducing the number of messages required to order a set of transactions. This could also reduce the problem of lost messages as less messages will be required.

As long as there are multiple transactions which have the same partitions involved, those transactions can be batched. Instead of sending transactions with the reliable multicast, the partition receiving a set of transactions will send batches containing all partitions involved with the same partition set. Afterwards, the protocol works as designed and assigns a logical clock to the batch. As the order in a batch is defined by only one partition (i.e. the partition which received the set of transactions), each partition will execute transactions in the batch in the same order.

If no batching is possible, a transaction can still be dispatched as a batch of one transaction.

6.1.3 Adaptive switching

As explained in section 4.3.3, the adaptive algorithm used to decide whether a partition should switch between TO-Multicast and Periodic Broadcast is not really smart and could use some improvements. Right now, the adaptive algorithm is taking decisions based only on local information about the MPOs access pattern.

There are a lot of different ways to improve the adaptive algorithm. Some ideas will be discussed below.

The protocol could take into account more metrics such as average latency to order a transaction, the partitions accessed frequently, the partitions involved frequently together in MPOs, network latency between partitions, etc.

A centralized service collecting metrics and sending switching commands to partitions could be used. This service will have the advantages to have a global overview across the system which could provide better decisions. A centralized service is also easier to implement than a decentralized one.

The algorithm taking switching decisions could be based on various artificial intelligence techniques such as machine learning, decision tree, etc. Black box optimization techniques could also be used.

6.2 Protocol

During the evaluation period, some overhead caused by our protocol have been analysed. In this section, some ways to possibly improve the latency will be developed. Of course, those improvements are theoretical and there is no correctness proof nor an implementation to back up those claims.

6.2.1 Latency improvement for *hybrid* MPO dispatching

An *hybrid* MPO is an MPO which requires ordering with partitions that do not use the same ordering scheme. An *hybrid* MPO will require ordering with TO-Multicast and Periodic Broadcast.

In our evaluation, the micro benchmark with four partitions involved in one MPO shows a huge latency mostly due to hybrid MPOs. In *CaMu*, an hybrid MPO is first assigned a logical clock with the TO-Multicast protocol. When the final logical clock is assigned to the MPO, the transaction is dispatched to the rest of partitions with Periodic Broadcast. Therefore, an *hybrid* MPO requires two sequential orderings which increase the latency a lot.

To avoid the double dispatching, the transaction requiring an hybrid dispatching could only be dispatched with TO-Multicast. This way, the protocol can order those types of transactions faster in only one ordering scheme rather than a double sequential ordering.

This solution still keeps the consistency of the total ordering because *CaMu* always checks the TO-Multicast thread to make sure it will not receive a logical clock smaller than the transactions that are going to be executed.

6.2.2 Rounds removal

The evaluation has clearly shown that rounds are limiting an optimal latency in every case. The rounds have been introduced in *CaMu* to allow partitions communicating with Periodic Broadcast to synchronize themselves but rounds are clearly restraining for *CaMu*.

Periodic Broadcast uses the concept of rounds to make sure that the synchronization message arrives in order. In Periodic Broadcast, it makes sense to use rounds because no progress can be made until every synchronization message is received as the protocol needs to deterministically order transactions.

Currently, in *CaMu*, transactions can only be executed at the end of a round, even transactions only ordered with TO-Multicast which do not use rounds internally. Therefore, rounds are delaying transactions execution which is bad for latency. This delay is required to ensure that future transactions will not break the consistency. This property is enforced by the MEC.

Instead of waiting for every synchronization message to arrive before calculating the MEC, the MEC could be calculated each time a new synchronization message arrives. This way, the execution can run totally asynchronously from the ordering thread.

On top of that, rounds could be totally removed and replaced with a sequence number ensuring the order of synchronization message. The sequence number would be only assigned to a pair of partitions meaning different pair of partitions could have different sequence number values. Thus, the switching protocol can potentially be a lot easier as the rounds were the main constraint (see section 3.4).

On top of this mechanism, some optimizations could be made by modifying the frequency of synchronization messages sent depending on the progress of a partition. If a partition has a low LMEC value compared to other partitions, the partition may block the ordering on other partitions and could send synchronization message as soon as some progress has been made. If a partition has a high LMEC value, the partition can reduce the rate of messages because it means the partition is in advance compared to other partitions.

Of course, there are still open questions such as how to assign timestamps to Periodic Broadcast only transactions, the correctness of such implementations, etc.

6.2.3 Fine-grained MEC

The MEC enforces total order for partitions communicating with Periodic Broadcast. The protocol uses the maximum value of LMEC received to decide whether transactions should be executed or delayed. Currently, the LMEC is common to all partitions but the protocol could calculate a custom LMEC for every partition. This way, a partition could potentially execute transactions faster if one transaction not related to the partition is blocking the ordering.

The MEC is also used to assign a logical clock to transactions which are not dispatched with TO-Multicast (i.e. transactions which are dispatched only with Periodic Broadcast). The current protocol is taking the maximum LMEC received plus one to calculate the logical clock that will be used for transactions requiring only Periodic Broadcast dispatching. As the logical clock must be higher than the current executed clock in every involved partitions, the logical clock assigned could be the maximum of the involved partitions LMEC received in a transaction plus one. This way, on average, transactions are not delayed too much if one of the LMEC is much bigger compared to others.

Conclusion

This conclusion will be split into two sections. First, the scientific contribution of my thesis will be analyzed. The second part will describe the personal contribution of this thesis.

The main contribution of this thesis is showing the practicability of combining two distributed protocols built for the same purpose, and more precisely, two total order protocols. The *CaMu* protocol demonstrates that a clever use of TO-Multicast and Periodic Broadcast can reduce or keep unchanged the latency of transactions being ordered depending on the workload. Of course, when combining two protocols, in specific workload, one will be better than another. Thus, the workload should be biased or dynamic to use the strength of each protocol.

It has been shown that combining two protocols can combine the advantages of each protocol as the partition scalability problem of Periodic Broadcast has been solved with the genuineness of TO-Multicast. Unfortunately, the disadvantages of one protocol can impact the other like the rounds blocking the execution of transactions.

This thesis also proves that dynamic switching between two heterogeneous total order protocols is possible. On top of that, the switching protocol introduces almost no overhead on the transaction ordering.

Finally, this document gives various leads to fix performance issues found in the database or improve the latency of the *CaMu* protocol. Those improvements have been found after understanding how the protocol works thanks to our evaluation.

This thesis also brought me a lot in terms of computer science skills and knowledge but also a lot of soft skills that only great experience can teach.

First, I learned a lot about distributed system which was a field of computer science I did not know before my work on partially replicated state machine. I enjoyed learning about this field because the concept is everywhere from IoT to the Cloud passing by things like blockchain, databases, ... even a simple client-server architecture is a distributed system itself.

As the ECAM is a more practical school, I also had the chance to have a sneak peak into the academic and research world. This experience has allowed me learn new skills such as pedagogical explanations, writing proofs, improving my understanding of scientific papers, etc.

I also had the chance to do my thesis with a non French speaker which allowed me to improve my English a lot, especially my speaking and writing skills that I do not practice daily.

On top of that, I had the chance to work inside a great C++ codebase which has improved my knowledge about parallel programming, best practices, etc. The software is also using some state-of-the-art library such as Google Protocol Buffers, ZeroMQ, etc that I had to learn and use during this thesis.

List of Acronyms

ACID Atomicity, Consistency, Isolation, Durability. Properties of relational databases. 20, 48

API Application Program Interface. 55

CAP Consistency, Availability, Partition tolerance. Defines a trade-off in distributed systems.. 6, 7

CPU Central Processing Unit. 69

DSL Domain Specific Language. 52

FIFO First-In First-Out. 9

GPS Global Positioning System. 8

GPU Graphics Processing Unit. 67

LAN Local Area Network. 68, 70–72, 78

LMEC Local Maximal Executable Clock. 28, 30, 31, 37, 38, 43, 46, 86, 87

MEC Maximal Executable Clock. III, 25, 30, 31, 35, 36, 38, 86, 87

MPO Multiple Partitions Operation. II, III, 14, 54, 60, 63, 64, 66, 70, 74–77, 79, 80, 84, 85, 92

NTP Network Time Protocol. 8

List of Figures

List of Tables

Bibliography

[1] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. “Scalable state-machine replication”. In: *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE. 2014, pp. 331–342.

[2] Kenneth P Birman and Thomas A Joseph. “Reliable communication in the presence of failures”. In: *ACM Transactions on Computer Systems (TOCS)* 5.1 (1987), pp. 47–76.

[3] Eric Brewer. “CAP twelve years later: How the" rules" have changed”. In: *Computer* 45.2 (2012), pp. 23–29.

[4] Harry Brundage. *Neat Algorithms - Paxos*. `http://harry.me/blog/2014/12/27/neat-algorithms-paxos/`.

[5] Mike Burrows. “The Chubby lock service for loosely-coupled distributed systems”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 335–350.

[6] Tushar Deepak Chandra and Sam Toueg. “Unreliable failure detectors for reliable distributed systems”. In: *Journal of the ACM (JACM)* 43.2 (1996), pp. 225–267.

[7] Brian F Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[8] James C Corbett et al. “Spanner: Google’s globally distributed database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[9] TPC Council. *tpc-c benchmark, revision 5.11.* 2010.

[10] Charles Levine Dave DeWitte. *SIGMOD ’97 Industrial Session 5 - Standard Benchmark for Database Systems.* `http://www.tpc.org/information/sessions/sigmod/indexc.htm`. 1997.

[11] Xavier Défago, André Schiper, and Péter Urbán. “Total order broadcast and multicast algorithms: Taxonomy and survey”. In: *ACM Computing Surveys (CSUR)* 36.4 (2004), pp. 372–421.

[12] The free dictionnary by Farlex. *Zipf’s law.* `https://www.thefreedictionary.com/Zipfian+distribution`. 2018.

[13] Udo Fritzke et al. “Fault-tolerant total order multicast to asynchronous groups”. In: *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on.* IEEE. 1998, pp. 228–234.

[14] Fujitsu. *White paper - Benchmark overview TPC-C.* `https://sp.ts.fujitsu.com/dmsp/publications/public/benchmark_overview_tpc-c.pdf`. 2013.

[15] Google. *Cloud Spanner.* `https://cloud.google.com/spanner/`. 2018.

[16] Google. *Protocol Buffers | Google Developers.* `https://developers.google.com/protocol-buffers/`. 2018.

[17] Grid5000. *Grid5000 - Homepage.* `https://www.grid5000.fr`. 2018.

[18] Pieter Hintjens. *ZeroMQ Guide - Missing Message Problem Solver.* `http://zguide.zeromq.org/page:all#Missing-Message-Problem-Solver`. 2018.

[19] Patrick Hunt et al. “ZooKeeper: Wait-free Coordination for Internet-scale Systems.” In: *USENIX annual technical conference.* Vol. 8. 9. Boston, MA, USA. 2010.

[20] Martin Kleppmann. *Designing data intensive application - The big ideas behind reliable, scalable, maintenable systems.* O'Reilly Media, 2017.

[21] Cockroach labs. *CockroachDB.* `https://www.cockroachlabs.com/`. 2018.

[22] Leslie Lamport. *Distribution.* May 1987. URL: `https://www.microsoft.com/en-us/research/publication/distribution/`.

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

[24] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[26] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. "High performance state-machine replication". In: *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on.* IEEE. 2011, pp. 454–465.

[27] Parisa Jalili Marandi et al. "Ring Paxos: A high-throughput atomic broadcast protocol". In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on.* IEEE. 2010, pp. 527–536.

[28] David L Mills. "Internet time synchronization: the network time protocol". In: *IEEE Transactions on communications* 39.10 (1991), pp. 1482–1493.

[29] Alberto Montresor. *Distributed Algorithms - Models.* `http://disi.unitn.it/~montreso/ds/handouts/02-models.pdf`. 2016.

[30] Alberto Montresor. *Distributed Algorithms - Reliable Broadcast.* `http://disi.unitn.it/~montreso/ds/handouts/04-rb.pdf`. 2016.

[31] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[32] Andrea Omicini. *Fault Tolerance in Distributed Systems: An Introduction Distributed Systems.* `campus.unibo.it/145939/1/9-faulttolerance.pdf`. 2013/2014.

[33] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* ACM. 2012, pp. 61–72.

[34] Hintjens Pieter. "ZeroMQ". In: *`http://zeromq.org`* (2018).

[35] PingCap. *PingCap.* `https://www.pingcap.com/en/`. 2018.

[36] Fred B Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial". In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[37] Łukasz Strzałkowski. *Queues - Job queues, message queues and other queues. Almost all of them in one place.* `http://queues.io/`. 2018.

[38] Rebecca Taft et al. "E-store: Fine-grained elastic partitioning for distributed transaction processing systems". In: *Proceedings of the VLDB Endowment* 8.3 (2014), pp. 245–256.

[39] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms.* Prentice-Hall, 2007.

[40] Alexander Thomson and Daniel J Abadi. "The case for determinism in database systems". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 70–80.

[41] Alexander Thomson et al. “Calvin: fast distributed transactions for partitioned database systems”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 1–12.

[42] Wikipedia. *Distributed computing*. Mar. 2018. URL: `https://en.wikipedia.org/wiki/Distributed_computing`.

[43] Wikipedia. *Zipf’s law*. `https://en.wikipedia.org/wiki/Zipf's_law`. 2018.

[44] yaledb. *Source code of Calvin*. `https://github.com/yaledb/calvin`. 2015.

[45] Tatu Ylonen and Chris Lonvick. “The secure shell (SSH) protocol architecture”. In: (2006).