Adaptive ordering protocol for partially replicated state machine

Charles Vandevoorde

April 24, 2018

1 Goal

Skewness in distributed data store 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. This protocol aims to reduce one of this type of skewness introduced by partitioning, mainly multi-partition 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.

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

2 General idea

2.1 Ordering protocol

The main difficulty in combining the use of TO-Multicast and Calvin 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 Calvin does not have the notion of timestamps, the use of synchronized rounds across all partitions allow a deterministic transactions ordering. As a result, partitions can not directly order transactions delivered by TO-Multicast and Calvin, as is illustrated in Figure 1. In the figure, $P1$ tries to dispatch $T5$ to $P3$ and $P4$. $P2$ also tries to dispatch $T4$ to $P3$ and $P4$. 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 Calvin, $P3$ and $P4$ may execute $T1$ and $T2$ in different order.

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

TO-Multicast assigns a timestamp to a transaction by using an agreement between the involved partitions which means TO-Multicast can't deterministically know the future timestamp of a transaction. On the other hand, Calvin 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 Calvin 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, Calvin can impose the final timestamp to partitions using the Calvin protocol.

In case where a transaction must only be dispatched with Calvin, the transaction is assigned a special timestamp value which must follow one condition to ensure total order: the transaction's timestamp must be greater than any transactions' timestamp executed yet in all involved partitions. Otherwise, a Calvin 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 no transaction will have a smaller timestamp than T. As future timestamps have bigger value than already assigned timestamps thanks to logical clock properties, partition P must only ensure that T has the smallest timestamp amongs the current transactions being ordered involving P.

TO-Multicast knows which transactions' timestamps aren't decided because partitions are involved in the clock assignation, while Calvin imposes transactions with a given timestamp to other partitions. Thus, a partition communicating with Calvin must inform other partitions about its state, and more precisely, the transactions' timestamps which haven't been dispatched yet. 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.

The MEC is also used for timestamp assignation for Calvin only transactions as the MEC provides enough information about other partitions' state.

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 protocol was designed to avoid blocking the ordering protocol.

When two partitions are switching, the main constraint is the round number synchronization between the two partitions Ch ▶*The problem is for both switching because when switching from calvin to to-multicast the switch should happen at the same round on both partitions.*◀. If the two partitions don't switch at 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 Calvin to TO-Multicast, the two switching partitions communications are scheduled by rounds. 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 Calvin transactions' dispatching is done. Otherwise, some transactions may not be dispatched on some partitions and thus, breaking the total order.

When switching from TO-Multicast to Calvin, 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 Calvin 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 Calvin are connected to other partitions using Calvin, and so on. Each partition connected with Calvin must have the same round number. Thus, every partition connected with Calvin will need to jump round.

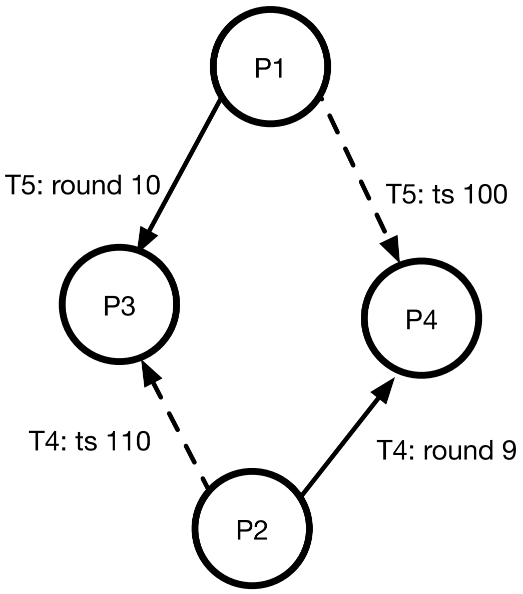


Figure 1: Example of partitions using different protocols

3 Ordering protocol

3.1 Description

The basic idea of the protocol is to use two communication schemes with different characteristics to dispatch and order MPOs. Those protocols are called Calvin and TO-MULTICAST. Calvin 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 Calvin or TO-MULTICAST. Calvin 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 Calvin involvement in the ordering. A transaction can be dispatched and ordered with only the Calvin sequencer 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 Calvin ordered only MPOs. We will see how during the protocol breakdown.

The protocol 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 needed to dispatch an 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 concensus 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 Calvin from this partition. 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 2, the LMEC will inform the state of the partition 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 genuine by adding them to a special queue called *pendingQ*. The dispatching is done completely asynchronously and the main loop doesn't wait for the transaction to be decided by the genuine protocol.

Clock assignation phase (line 29 and 23-24) Each round, the protocol check if the genuine implementation 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.

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, the transaction can be in two different states depending if it still needs dispatching with the Calvin sequencer or not:

READY state means that the transaction needs to be dispatched with the Calvin sequencer.

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

When a transaction in the batch doesn't require any genuine dispatching, the transaction is assigned a special clock value *calvin_logical_clock* which was defined during the previous round $\boxed{\text{Li}}$ ►*big question: do we need to assign these transactions timestamp? Isn't it enough to just execute them in front of all transactions in that batch?*◄ $\boxed{\text{Ch}}$ ►*No because TO-Multicast transactions executions are not bounded on a round which means that some partitions may already have executed some transactions in round X_i while other partitions will execute them in round $X_{i+\lambda}$*◄. The detail will be explained later. As the transaction still requires Calvin sequencing, the transaction is in **READY** state.

Calvin sequencer dispatching phase (line 37-39) As required by the Calvin sequencer protocol, a message will be send to every Calvin connected partition. This message will include two informations:

- 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 Calvin sequencer 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 (Maximal Executable Clock) needs to be calculated by taking the minimum LMEC received (see *Replication phase* and *Calvin sequencer dispatching phase*). The MEC provides a bounded clock execution for a round to ensure synchronization between sub-protocols (genuine and Calvin sequencer). This limitation ensures that a partition P_a doesn't 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 the Calvin sequencer 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.

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

First, the *calvin_logical_clock* is the logical clock value given to transactions which aren't assigned a logical clock by the genuine protocol (i.e. Calvin only transactions). One condition is required to ensure total order property: *calvin_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 the following rounds. As the MEC is bounding the execution, each Calvin connected partition knows the maximal clock executed. For simplicity, the *calvin_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.

3.2 Pseudocode

```
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 genuine 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 Calvin sequencing.
    $readyQ \leftarrow$ new queue()
    for $MPO \in MPOs$ do
        if GENUINE $\notin MPO.protocols()$ then
            $MPO.setLogicalClock(calvin\_logical\_clock)$
            $readyQ.add(MPO)$
        else
            $genuine.Send(MPO)$
            $pendingQ.add(MPO)$

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

    Send the local_mec and readyQ MPOs for every low latency communicating partition
    $executablePQ \leftarrow executablePQ + readyQ$
    $received\_MPOs, mecs \leftarrow$ wait messages from low latency protocol connected nodes

    $mec \leftarrow min(mecs)$

    for $MPO \in executablePQ$ do
        if $MPO.logical\_clock < mec$ then
            Execute $MPO$
```

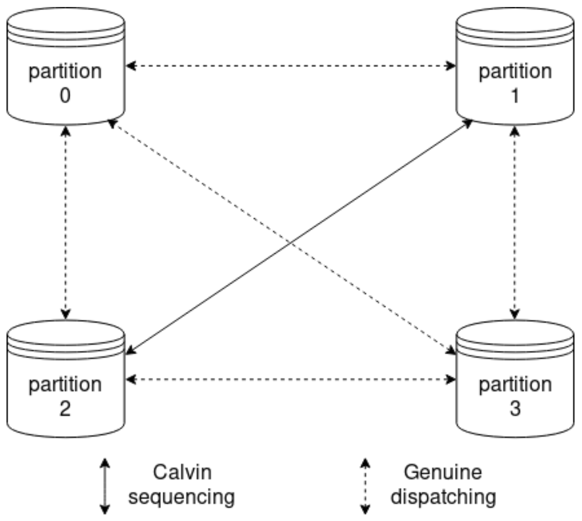


Figure 2: Example of partitions using different protocols

```
            $executablePQ.remove(MPO)$
        **else**
            **break**

    $calvin\_logical\_clock \leftarrow max(mecs)$
    $genuine.logical\_clock \leftarrow max(genuine.logical\_clock, calvin\_logical\_clock)$

    $round \leftarrow round + 1$

```

3.3 Example

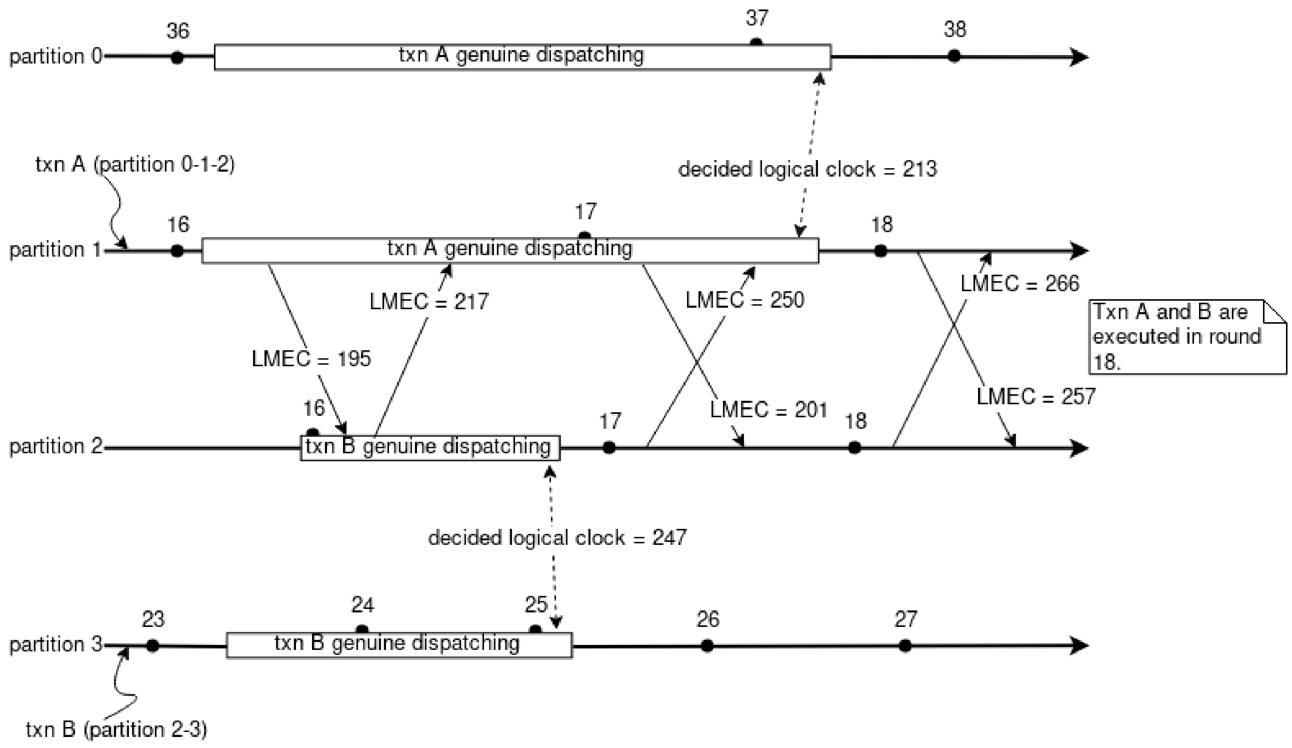
As the protocol explanation can be a bit abstract, here is an example which contains four partitions and two transactions to order. Figure 2 represents the protocols used to communicate between the partitions. As described in the schema, only partition 1 and 2 are talking together with the Calvin sequencer. Every other partition pair is communicating with the genuine protocol. For more clarity, we are only assuming one node 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. In the following paragraphs, each transaction dispatching will first analyzed separatly and afterwards, the ordering relation and the execution will be discussed.

Transaction A. As this transaction has a genuine and a Calvin 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 Calvin sequencer 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 genuine 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, 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



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
Txn A and B are executed in round 18.
LMEC = 195
LMEC = 201
LMEC = 257
partition 2
16
txn B genuine dispatching
17
18
decided logical clock = 247
partition 3
23
24
25
txn B genuine dispatching
26
27
txn B (partition 2-3)

Figure 3: Timing diagram representing the ordering of two transactions **A** and **B**.
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 Calvin sequencer messages.

executed before **A**, **B** needs to be delayed until at least transaction **A**'s 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 Calvin sequencer, 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.

3.4 Correctness proof

We are assuming two transactions T_a and T_b. $T_a \prec T_b$ relation implies T_a is ordered before T_b on every correct process involved in T_a and T_b. The "$\prec$" relation is the total ordering property. $T_a < T_b$ for a process P_a implies that T_a is ordered before T_b inside process P_a and has no other garantee on the ordering for other processes. The "$<$" 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 $T_x \prec T_y$. By contradiction, we assume that we have $T_x < T_y$ on partition P_a and $T_y < T_x$ on partition P_b which break the total order property. A violation of total order could happen because: **a.** $t_{T_{a,b}}$ 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 doesn't know that a transaction T_x exists such as $T_x \prec T_y$.

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:

Genuine 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.

Calvin sequencing Before T_x is dispatched, the partition receiving the transaction T_x will assign a logical clock which is equals to *calvin_logical_clock*. As the *calvin_logical_clock* is based on the LMEC which is the result of intra-partition consensus, the *calvin_logical_clock* will be consistent between 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 genuine protocol to partitions which are communicating with the genuine protocol. When T_x is decided by the genuine protocol, it has a logical clock which is consistent as seen above across the genuine partitions. Second, T_x is dispatched with the Calvin sequencer 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.

Genuine dispatching TO-MULTICAST properties ensure that the transaction T_x will be decided iff there is no transaction T_y such as $T_y \prec T_x$ 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.

Calvin sequencing Unlike the base Calvin sequencer, 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 assigned to a Calvin sequencer only transaction T_x has the value of *calvin_logical_clock* which was calculated during the previous round. This value was calculated by taking the maximum of all LMEC received such as $calvin\_logical\_clock_r = max(LMECs)$. 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 the Calvin sequencer do.

Hybrid dispatching Since an hybrid transaction T_x delays its Calvin sequencing 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.

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

MEC only apply to partitions communicating with the Calvin sequencer because TO-MULTICAST already ensures total order by only deciding a transaction when it has the smallest logical clock. For partitions talking with the Calvin sequencer, 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.

4 Protocol switching

The protocol switching is divided in different parts. First, a different protocol is used between TO_MULTICAST → Calvin sequencer and Calvin sequencer → TO_MULTICAST. In addition, when we are switching from TO_MULTICAST to the Calvin sequencer, 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 switchs 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.

4.1 Common

The first step of a switching 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

As explained above, the switch is divided in different cases depending on the type of protocol currently used. Line 6 to 12 handles the different cases possible. Line 9 to 12 handles the different cases when switching from TO_MULTICAST to the Calvin sequencer depending on the partition type. A partition can have one of those three types:

Full low latency is a partition which communicates only with the Calvin sequencer.

Full genuine is a partition which communicates only with TO-MULTICAST.

Hybrid is a partition which communicates with TO-MULTICAST to some partitions and with Calvin to the others.

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

4.1.1 Pseudocode

```
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 $get\_protocol(P_{switch}) = LOW\_LATENCY$ then
7:         $state \leftarrow INIT\_SWITCH\_TO\_GENUINE$
8:     else
9:         if $switch\_metadata.partition\_type = FULL\_GENUINE$ or
                   $get\_this\_partition\_type() = FULL\_GENUINE$ then
10:            $state \leftarrow INIT\_SWITCH\_TO\_LOW\_LATENCY\_FULL\_GENUINE$
11:        else
12:            $state \leftarrow INIT\_SWITCH\_TO\_LOW\_LATENCY\_HYBRID$
13:
```

4.2 Calvin sequencer → TO-MULTICAST

Switching from the Calvin sequencer to TO-MULTICAST requires 4 steps:

Transition phase (line 1-4) When the switching round happens (agreed upon during the *init* phase), both partitions will change from Calvin sequencer protocol to a transition protocol. This transition protocol is running both protocol at the same time meaning that new transactions received will be dispatched with the genuine protocol for the switching partitions while the Calvin sequencing message will still be sent Li ►*why do we need to have both protocols running together? Isn't it enough to keep delivering message using Calvin until the switch round has reached?*◄ to ensure the next step in the switching will eventually terminate.

Waiting for required Calvin sequencing (line 5) Before switching to the genuine protocol, the switching protocol needs to make sure that transaction no longer requires Calvin sequencing with the switching partition Li ►*when can we know this condition? And can't this round be known in advance in the previous phase?*◄ 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.

4.2.1 Pseudocode

Li ►*For the pseudo-code: when is switching round decided?*◄ Ch ►*The first switching round is decided during the common phase*◄

```
1: upon $state = INIT\_SWITCH\_TO\_GENUINE$ and
           $current\_round = switch\_metadata.switching\_round$
2:     $state \leftarrow FINISH\_LOW\_LATENCY\_DISPATCHING$
3:     $set\_protocol(P_{switch}, TRANSITION)$
4:
5: upon $state = FINISH\_LOW\_LATENCY\_DISPATCHING$ and
           no transaction left for Calvin 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:     $state \leftarrow SWITCH\_TO\_GENUINE$
```

```
 9:
10: upon $state = SWITCH\_TO\_GENUINE$ and
            $current\_round = switch\_metadata.switching\_round$
11:     $set\_protocol(P_{switch}, GENUINE)$
12:     $state \leftarrow$ null
13:
```

4.3 TO-MULTICAST → Calvin sequencer ■ One or two full genuine

Before switching to the Calvin sequencer, 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 genuine partition, the switching is really straightforward because round number of full genuine partitions can be changed without interfering with the protocol as no Calvin sequencing message needs to be sent.

The protocol can be described in two steps:

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 genuine 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 same on every node in the switching partitions.

One hybrid and one full genuine partitions An hybrid partition is talking with other partitions with the Calvin sequencer protocol which means it's impossible to change the round number without also switching those connected partitions. Fortunately, the full genuine partition can switch to any round number. The full genuine partition will wait for the first Calvin sequencing 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 *Calvin sequencer → TO-MULTICAST*).

Calvin sequencer switch (line 11-24) Before switching to the Calvin sequencer 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 *calvin_logical_clock* are updated before receiving transactions with the Calvin sequencer. Li ►*why? Can't we use each partition's existing LMEC?*◄ Ch ►*No because the LMEC from the switching partition is not yet received because the communication was using genuine before so no LMEC send.*◄ After this synchronization round, the protocol can be updated to the Calvin sequencer.

4.3.1 Pseudocode

```
 1: upon $state = INIT\_SWITCH\_TO\_LOW\_LATENCY\_FULL\_GENUINE$
 2:     if $get\_partition\_type() = FULL\_GENUINE$ and
                $other\_partition\_type = FULL\_GENUINE$ then
 3:         $current\_round \leftarrow switch\_metadata.switching\_round$
 4:     else if $get\_partition\_type() = FULL\_GENUINE$ then
 5:         $msg \leftarrow$ wait LOW LATENCY message
 6:         $current\_round \leftarrow msg.round$
 7:     $switching\_round = current\_round$
 8:     $set\_protocol(P_{switch}, TRANSITION)$
 9:     $state \leftarrow SYNCHRONIZE\_MEC$
10:
```

```
11: upon $state = SYNCHRONIZE\_MEC$ and $current\_round = switching\_round + 1$
12:     $set\_protocol(P_{switch}, LOW\_LATENCY)$
13:     $state \leftarrow$ null
14:
```

4.4 TO-MULTICAST → Calvin sequencer ▪ Two hybrid

As explained above, the Calvin sequencer requires connected partitions to have the same round number to terminate a round. Every partition which is connected by Calvin and has the same round number at one time compared to other partitions are forming a graph, called Calvin sequencer 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 4.2 with *switching_round* and we have two distinc graphs, we may block a graph for a non-negligeable 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 distinc 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 4.3, before switching to the Calvin sequencer protocol, we need to switch to the transition protocol between the two switching partitions Li ►*Just the two partitions, or also all connected partitions?*◄ Ch ►*Just the two switching partitions because other partitions are already sharing their LMEC because they are talking with Calvin.*◄ after upgrading the round number. The transition protocol will synchronize the various values of the *CaMu* protocol. Li ►*Overall, can't LMECs be synchronized together during switching round agreement phase?*◄ Ch ►*No because to send Calvin message we need to have synchronized round number.*◄

Calvin sequencer switching After one synchronization round, both switching partitions can switch to the Calvin sequencer protocol safely.

4.4.1 Pseudocode

For the sake of brevity, the protocol shown below is only described for the switching partition and not for partitions involved in the graph. Li ►*replace launch graph locking with a few lines of brief brief description would be more clear*◄ Ch ►*added a small pseudo-code to reflect what the algo is really doing.*◄

```
1:  Algorithm variables
2:      $locked\_partitions$: a set containing partition ids of locked partition
3:
4:  upon $state = INIT\_SWITCH\_TO\_LOW\_LATENCY\_HYBRID$
5:      $unvisited\_partitions \leftarrow$ set(get every neighbour partition using Calvin)
6:      while $unvisited\_partitions.size()! = 0$ do
7:          $partition\_id \leftarrow unvisited\_partitions.pop()$
8:          $A - PROPOSE(partition\_id, PARTITION\_LOCK)$
9:          $unvisited\_partitions.add(wait\ A - DECIDE(partition\_id))$
10:         $locked\_partitions.add(partition\_id)$
11:
12: upon locking is finished
13:     $A - PROPOSE(P_{switch}, (switching\_round, final\_round))$
14:     $(switching\_round, final\_round) \leftarrow wait\ A - DECIDE(P_{switch})$
15:     $propagate\_round\_info(switching\_round, final\_round)$
16:
17: upon $current\_round = switching\_round$
18:     $current\_round \leftarrow final\_round$
19:     $set\_protocol(P_{switch}, transition)$
20:     $state \leftarrow SYNCHRONIZE\_MEC$
21:
22: upon $state = SYNCHRONIZE\_MEC$ and $current\_round = final\_round + 1$
23:     $set\_protocol(P_{switch}, LOW\_LATENCY)$
24:     $state \leftarrow$ null
25:
```