Causality Tracking Trade-offs for Distributed Storage

Hugo Guerreiro
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
Lisbon, Portugal
hugo.guerreiro@tecnico.ulisboa.pt

Luís Rodrigues
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
Lisbon, Portugal
ler@tecnico.ulisboa.pt

Nuno Preguiça
NOVA LINCS
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Lisbon, Portugal
nuno.preguica@fct.unl.pt

Nivia Quental
INESC-ID
Lisbon, Portugal
nivia.quental@tecnico.ulisboa.pt

***Abstract*—After the seminal paper by L. Lamport, which introduced (scalar) logical clocks, several other data structures for keeping track of causality in distributed systems have been proposed, including vector and matrix clocks. These are able to capture causal dependencies with more detail but, unfortunately, also consume a substantially larger amount of network bandwidth and storage space than Lamport clocks. This raises the question of whether the benefits of these more complex structures are worth their cost. We address this question in the context of partially replicated systems. We show that for some workloads the use of more expensive clocks does bring significant benefits and that for other workloads no visible benefits can be observed. The paper provides a characterization of the scenarios where each type of clock is more beneficial, helping designers to develop more efficient distributed storage systems.**

***Index Terms*—Distributed Computing methodologies, Distributed Storage, Network algorithm, causal Consistency**

I. Introduction

The notion of causality is a fundamental concept in the context of distributed systems, and the ability to capture cause-effect relations accurately is a requirement of many distributed tasks, such as taking global snapshots [1], implementing distributed mutual exclusion [2], or maintaining the consistency of data [3]. After the seminal paper by L. Lamport [4], that introduced (scalar) logical clocks, several other data structures for keeping track of causality in distributed systems have been proposed, including vector [5], [6] and matrix clocks [7], among others. Each of these mechanisms offers a different trade-off between the space complexity and the ability to track causal dependencies accurately. Note that Lamport clocks consist of a single scalar (the space complexity is $\mathcal{O}(1)$), vector clocks typically require an entry per node in the system (the space complexity is $\mathcal{O}(N)$), and matrix clocks, as their name suggests, have a space complexity of $\mathcal{O}(N^2)$. Thus, despite their potential advantages, vector and matrix clocks consume a substantially larger amount of network bandwidth and storage space. This raises the question of whether, in practice, the benefits of these more complex structures are worth their cost.

From the abstract point of view, it is clear that larger clocks can capture causality more accurately and reduce the remote visibility latency. Assume that all updates are timestamped with some form of logical clocks to keep track of causality. Let $t(u)$ be the timestamp assigned to update u. It is well known that, with Lamport clocks $u_x \rightarrow u_y \implies t(u_y) > t(u_x)$ but that $t(u_y) > t(u_x) \not\Longrightarrow u_x \rightarrow u_y$. This means that, when using Lamport clocks, a node that has received u_y and not u_x, cannot immediately assess if it is safe to deliver u_y. We call this artifact of Lamport clocks a *false dependency*. False dependencies have a negative impact on the remote visibility latency, by forcing updates to wait for other updates that may not be in their causal past. Vector clocks suffer less from this problem. However, to completely reduce the number of false dependencies with vector clocks, updates need to be timestamped with not only one vector clock, but with a set of vector clocks, one vector clock for each data item. When the number of items is larger than the number of nodes in the system (arguably, the most common case) accurate clocks degenerate in a matrix clock. For any reasonably sized system, this has a significant overhead, both in terms of storage space and in terms of bandwidth utilization. Thus, in practice, the costs of maintaining accurate clocks can outweigh the potential advantages they may bring in terms of reduced visibility latency.

We study the trade-off between the space complexity of the vector clocks and the remote visibility latency in partially replicated storage systems under different workloads and different replica deployments. We show that for some workloads/deployments the use of more expensive clocks does bring significant benefits and that for others no visible benefits can be observed.

Unfortunately, there are many factors that affect the per-

This work was partially supported by the FCT via project NG-STORAGE, with reference PTDC/CCI-INF/32038/2017, and by projects INESC-ID (ref. UIDB/ 50021/ 2020) and NOVA LINCS (ref. UIDB/ 04516/ 2020)

978-1-7281-8326-8/20/$31.00 ©2020 IEEE

formance of a certain type of clocks, including: how many replicas of each object are kept and where these replicas are located, the read/write ratio of the workload, the access frequency to each object, the time between consecutive operations executed by clients, among others. This makes the task of identifying the best clock for a given scenario very hard.

To solve this puzzle, we introduce two novel features that capture relevant patterns for the performance of causality tracking mechanisms: the *Update Generation Rate Asymmetry* (GRA), defined as the ratio between the fastest and slowest average update frequency for all nodes in the system, and the *Object Ownership to Objects in Causal Past Ratio* (OPR), defined has the ratio between the number of objects replicated at a node and the objects with updates in the causal frontier of an update. With these new features, the paper provides a characterization of the scenarios where each type of clock is more beneficial, helping system designers to develop more efficient distributed storage systems.

The rest of the paper is organized as follows. Section II introduces the different logical clocks considered in the paper and Section III details how they can be used to enforce causal consistency. Section IV presents an analysis of the algorithms from a theoretical point of view. Section V presents an empirical study of said algorithms. In Section VI we present a set of guidelines to help decide between the different algorithms followed by some related work in Section VII. Finally, Section VIII concludes the paper.

II. Causal Order in Message Passing Systems

A. Causal Order

In a distributed system it is possible to define a partial order of events using their (potential) cause-effect relation. Lamport formalized this concept through the *happened-before* relation ($\rightsquigarrow$) between events [4]. An event e_1 is said to have *happened-before* an event e_2 ($e_1 \rightsquigarrow e_2$) iff:

(i) If e_1 and e_2 are events that happened in the same process and e_1 occurred before e_2 then $e_1 \rightsquigarrow e_2$;

(ii) If e_1 is the event that corresponds to a message being sent by a process and e_2 is an event that corresponds to that message being received by some other process, then $e_1 \rightsquigarrow e_2$;

(iii) If $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$ then $e_1 \rightsquigarrow e_3$.

The design of distributed applications can be simplified if there is a communication layer that delivers messages to processes in an order that respects the *happened-before* relation [8], [9]. Such communication primitive is said to enforce *causal order* and mechanisms to enforce this order have been widely studied in the literature [9]. In the following section, we present some of the more relevant mechanisms that have been used to keep track of causality and to enforce causal order in distributed systems.

B. Causal Multicast

We start by presenting a generic algorithm to implement causal multicast. The algorithm, depicted in Alg. 1 can be adapted to use different types of clocks. The algorithm assumes that multiple multicast groups can exist in the system but that causality is maintained across groups, similarly to what was provided by the Isis system [10]. Groups can be mapped to application-level abstractions, such as distributed objects. Later in the text, we show how groups can be used in the context of distributed storage systems.

The algorithm assumes that, for each group in the system, there is a set of reliable FIFO channels connecting every pair of processes. These FIFO channels are used exclusively to send messages for that group. FIFO channels can be trivially achieved in practice, for instance, by using TCP/IP connections to support the message exchange. Each process p_i maintains a logical clock denoted $local\_clock_i$. The format of this clock is implementation-dependent: it can be one Lamport clock, one vector clock, one matrix clock, or even a set of clocks (one for each group). The initialization of this clock is encapsulated by the primitive INIT_CLOCK.

For sending a message m, the process starts by updating its local clock in function PREPARE. The message m carries a logical clock, denoted $clock_m$, with the value of the local clock. The message is sent, using the FIFO channels, to the set of destination peers, typically the set of all processes in the group. Additionally, the message is locally delivered.

When receiving a message m for group g from process j, the process adds the message to the list of pending messages for g and j, $pending_i[g][j]$. A message that reaches the top of the pending list for some process p_j and group g (i.e. for which prior messages from p_j to g have already been delivered) can be delivered if its dependencies are satisfied, i.e., if all dependent messages have already been delivered – this is checked in function MESSAGE_SAFE. For helping in this check, each process p_i keeps a record, $delivered\_upto_i[g][j]$, with the largest clock for which a message from p_j to g has already been delivered in p_i or for which it is known that such message does not exist. This information is updated both when a new message is delivered (function UPDATE_LOCAL_CLOCK), and when a new message reaches the top of the list of pending messages (function MESSAGE_READY). This latter case is used to register that messages with smaller clocks from p_j to g do not exist – this follows from using FIFO channels: if an undelivered message with a smaller clock existed, it would be the message on the top of pending messages.

C. Causal Order with Lamport Clocks

In Alg. 2 we present the functions to instantiate the generic algorithm for the case where Lamport clocks are used to keep track of causality. In this case, for each group g, each process keeps a Lamport clock that is initiated to 0 (function INIT_CLOCK).

When sending a message, the local Lamport clock associated with the group is incremented (function PREPARE). We note that the message is timestamped with the full local clock, which includes the set of clocks for all groups to trace dependencies across groups.

Algorithm 1 Generic Causal Order Implementation

```
1: Let 𝒩 be a set of processes
2: Let 𝒢 be a set of multicast groups
3: procedure INIT_NODE(i)
4:     INIT_CLOCK (local_clock_i);
5:     delivered_upto_i[g][j] ← 0; ∀g ∈ 𝒢, ∀j ∈ 𝒩
6:     pending_i[g][j] ← ∅; ∀g ∈ 𝒢, ∀j ∈ 𝒩
7:     my_groups_i ← groups to which process i belongs
8: procedure SEND(m, g, destination_set)
9:     PREPARE(g, destination_set)
10:    clock_m ← local_clock_i;
11:    for k ∈ destination_set do
12:        FIFO_SEND (m, clock_m, g, i, k);
13:    DELIVER(m, clock_m, g, i);
14: procedure FIFO_RECEIVE(m, clock_m, g, j)
15:    ENQUEUE(pending_i[g][j], ⟨m, clock_m⟩);
16: when ∃m : FIRST(m, pending_i[g][j]) do
17:    MESSAGE_READY (m, clock_m, g, j);
18: done
19: when ∃m : FIRST(m, pending_i[g][j]) ∧
    MESSAGE_SAFE(m, clock_m, g, i, j) do
20:    UPDATE_LOCAL_CLOCK(m, clock_m, g, j);
21:    DELIVER(m, clock_m, g, j);
22: done
```

The MESSAGE_READY function, called when a message m reaches the top of the pending list from process p_j for group g, updates $delivered\_upto$ record to register that all messages with smaller Lamport clocks from p_j to g either have been delivered or do not exist (this follows from the use of FIFO channels, as explained before).

The MESSAGE_SAFE function verifies that a message for group g is safe to be delivered by checking that all of its dependencies have already been delivered. This can be assessed by checking that: i) all messages for g with smaller clocks from all other processes have already been delivered ($condition\_1$); ii) the dependencies for other groups the local process belongs to have already been delivered ($condition\_2$).

The UPDATE_LOCAL_CLOCK function, called when a message is delivered, updates $delivered\_upto_i[g]$ to reflect the delivered message, and the local clock by taking the maximum of the local clock and the message's clock for each group. This guarantees that, when sending a new message, the message clock for the group will be larger than the clocks of all dependencies, which makes the use of the conditions defined in MESSAGE_SAFE correct.

D. Causal Order with Vector Clocks

A well known extension of Lamport clocks are vector clocks [6], [11]. A vector clock keeps multiple logical clocks, one for each process in the system, precisely recording the last message from each process - e.g., if the system has N processes, the vector clock will contain N logical clocks. Alg. 3 presents the functions to instantiate the generic causal multicast algorithm for vector clocks.

The implementation of these functions is conceptually similar to the corresponding implementation for Lamport clocks, described in the previous section. There are two major differences. First, for each group, we keep a vector clock instead of a Lamport clock, with each entry in the vector clock initiated with 0; when a message is sent on a group g, only the ith entry in g's vector clock is incremented. Second, as a message records precisely its causal past, the function that assesses if it is safe to deliver a message can verify that, for each process, the dependencies have already been delivered.

Algorithm 2 Causal Order with Lamport Clocks

```
1: procedure INIT_CLOCK(clock)
2:     clock[g] ← 0, ∀g ∈ 𝒢;
3: procedure MERGE_TS(ts_1, ts_2)
4:     result_ts[g] ← MAX(ts_1[g], ts_2[g]), ∀g ∈ 𝒢
5:     return result_ts
6: procedure PREPARE(g, destination_set)
7:     local_clock_i[g] ← local_clock_i[g] + 1;
8: procedure MESSAGE_READY(m, clock_m, g, j)
9:     delivered_upto_i[g][j] ← clock_m[g] − 1
10: function MESSAGE_SAFE(m, clock_m, g, i, j)
11:    condition_1 ← clock_m[g] − 1 ≤ delivered_upto_i[g][k], ∀k ∈ 𝒩 :
       k ≠ j
12:    condition_2 ← clock_m[g'] ≤ delivered_upto_i[g'][k], ∀g' ∈
       my_groups_i : g' ≠ g, ∀k ∈ 𝒩
13:    return condition_1 ∧ condition_2
14: procedure UPDATE_LOCAL_CLOCK(m, clock_m, g, j)
15:    delivered_upto_i[g][j] ← clock_m[g]
16:    local_clock_i ← MERGE_TS(local_clock_i, clock_m)
```

Algorithm 3 Causal Order with Vector Clocks

```
1: procedure INIT_CLOCK(clock)
2:     clock[g][k] ← 0, ∀g ∈ 𝒢, ∀k ∈ 𝒩;
3: procedure MERGE_TS(ts_1, ts_2)
4:     result_ts[g][k] ← MAX(ts_1[g][k], ts_2[g][k]), ∀g ∈ 𝒢, ∀k ∈ 𝒩
5:     return result_ts
6: procedure PREPARE(g, destination_set)
7:     local_clock_i[g][i] ← local_clock_i[g][i] + 1;
8: procedure MESSAGE_READY(m, clock_m, g, j)
9:     delivered_upto_i[g][j] ← clock_m[g][j] − 1.
10: procedure MESSAGE_SAFE(m, clock_m, g, i, j)
11:    condition_1 ← clock_m[g][k] ≤ delivered_upto_i[g][k], ∀k ∈ 𝒩 :
       k ≠ j
12:    condition_2 ← clock_m[g'][k] ≤ delivered_upto_i[g'][k], ∀g' ∈
       my_groups : g' ≠ g, ∀k ∈ 𝒩
13:    return condition_1 ∧ condition_2
14: procedure UPDATE_LOCAL_CLOCK(m, clock_m, g, j)
15:    delivered_upto_i[g][j] ← clock_m[g][j].
16:    local_clock_i ← MERGE_TS(local_clock_i, clock_m);
```

E. Causal Order with Matrix Clocks

Matrix clocks [7] expand vector clocks in an extra dimension. Instead of maintaining one logical clock for each process in the system, matrix clocks maintain one logical clock for each link in the system, allowing a process to know not only the messages received from each of the other processes, as with vector clocks, but also what messages each of the other process has received, allowing to track precise information about indirect dependencies.

When processes are logically organized in a clique, and every process has a unidirectional link to every process in the system (including to itself), the number of links is quadratic. In Alg. 4 we present the functions to support causal ordering using matrix clocks. The structure of the code is, again, very similar to the two implementations presented before, for Lamport and vector clocks, with the exception that the clocks

Algorithm 4 Causal Order with Matrix Clocks

procedure INIT_CLOCK(*local_clock*$_i$)
  local_clock$_i[g][j][k] \leftarrow 0, \forall g \in \mathcal{G}, \forall j, k \in \mathcal{N}$;
procedure MERGE_TS(*ts*$_1$, *ts*$_2$)
  result_ts$[g][j][k] \leftarrow \text{MAX}(ts_1[g][j][k], ts_2[g][j][k]), \forall g \in \mathcal{G}, \forall j, k \in \mathcal{N}$
  return *result_ts*
procedure PREPARE(*g, destination_set*)
  local_clock$_i[g][i][j] \leftarrow$ *local_clock*$_i[g][i][j] + 1, \forall j \in$ *destination_set*;
procedure MESSAGE_READY(m, *clock*$_m$, g, j)
  delivered_upto$_i[g][j] \leftarrow l(m)[g][j][i] - 1$.
procedure MESSAGE_SAFE(m, *clock*$_m$, g, i, j)
  $condition_1 \leftarrow$ *clock*$_m[g][k][i] \leq$ *delivered_upto*$_i[g][k], \forall k \in \mathcal{N} : k \neq j$
  $condition_2 \leftarrow$ *clock*$_m[g'][k][i] \leq$ *delivered_upto*$_i[g'][k], \forall g' \in$ *my_groups* $: g' \neq g, \forall k \in \mathcal{N}$
  return $condition_1 \wedge condition_2$
procedure UPDATE_LOCAL_CLOCK(m, *clock*$_m$, g, j)
  delivered_upto$_i[g][j] \leftarrow$ *clock*$_m[g][j][i]$.
  local_clock$_i \leftarrow$ MERGE_TS(*local_clock*$_i$, *clock*$_m$);

carry much more detailed information, and sending a message on group g, updates the ith vector clock by incrementing all kth entries of said vector, where k are all processes that will receive the message.

III. Causal Consistency in Shared Memory

A. Causal Consistency

The *happened-before* relation was originally defined in the context of message-passing systems, but it also applies to shared memory systems where processes interact by reading and writing on objects. In this case, we can define the cause-effect relations as follows:

(i) **Thread of Execution**. If a and b are two operations executed by the same thread of execution, then $a \rightsquigarrow b$ if a is executed before b.
(ii) **Reads From**. If a is an update operation and b is a read operation that reads the value written by a, then $a \rightsquigarrow b$.
(iii) **Transitivity**. If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

Let $w(o_a)$ and $w(o_b)$ be two write operations on, o_a and o_b. Let $r(o_a)$ and $r(o_b)$ be two read operations by the same client, where $r(o_a)$ is executed before $r(o_b)$ and where $r(o_a)$ returns the value written by $w(o_a)$ and $r(o_b)$ returns the value written by $w(o_b)$. This execution satisfies *Causal Consistency* (CC) [3] if there is no write $w'(o_b)$ such that $w(o_b) \rightsquigarrow w'(o_b) \rightsquigarrow w(o_a)$.

Causal consistency is extremely relevant in the context of highly-available distributed storage as it has been proven to be the strongest consistency model that can provide availability in face of transient network partitions [12], [13]. Stronger consistency models, such as serializability [14] or linearizability [15], require processes to reach consensus and to establish a total order among concurrent operations, therefore, being prone to blocking [16]. One of the most common strategies to achieve causal consistency in distributed and replicated storage systems is to ensure that updates performed at remote nodes are applied to any replica in causal order.

Algorithm 5 Generic Causally Consistent Storage Implementation

Let $\mathcal{N}$ be a set of processes
Let $\mathcal{G}$ be a set of multicast groups
Let $\mathcal{K}$ be a set of object keys
procedure INIT_NODE(i)
  INIT_CLOCK (*local_clock*$_i$);
  delivered_upto$_i[g][j] \leftarrow 0; \forall g \in \mathcal{G}, \forall j \in \mathcal{N}$
  pending$_i[g][j] \leftarrow \emptyset; \forall g \in \mathcal{G}, \forall j \in \mathcal{N}$
  my_groups$_i \leftarrow$ *groups to which process i belongs*
procedure INIT_CLIENT(*client_clock*)
  INIT_CLOCK(*client_clock*)
procedure READ(*client_clock*, k)
  $\langle value, ts \rangle \leftarrow$ STORAGE_READ(k)
  client_clock $\leftarrow$ MERGE_TS(*client_clock*, *ts*)
  return(*value*)
procedure WRITE(*client_clock*, k, *value*)
  $g \leftarrow$ MAP_KEY_TO_GROUP(k)
  PREPARE(g, REPLICAS(k))
  update_clock $\leftarrow$ MERGE_TS(*client_clock*, *local_clock*)
  for $j \in$ *replicas*(k) **do**
    FIFO_SEND($\langle k,$ *value*$\rangle$, *update_clock*), g, i, j)
procedure FIFO_RECEIVE(m, *clock*$_m$, g, j)
  ENQUEUE(*pending*$_i[g][j]$, $\langle m,$ *clock*$_m\rangle$);
when $\exists m$: FIRST(m, *pending*$_i[g][j]$) **do**
  MESSAGE_READY (m, *clock*$_m$, g, j);
done
when $\exists m$: FIRST(m, *pending*$_i[g][j]$) $\wedge$ MESSAGE_SAFE(m, *clock*$_m$, g, i, j) **do**
  DELIVER(m, *clock*$_m$, g, j);
  UPDATE_LOCAL_CLOCK(m, *clock*$_m$, g, j);
done
procedure DELIVER($\langle k,$ *new_value*$\rangle$, *new_ts*, g, j)
  $\langle$*old_value*, *old_ts*$\rangle \leftarrow$ STORAGE_READ(k)
  merged_value $\leftarrow$ MERGE_UPDATE(*old_value*, *new_value*)
  merged_ts $\leftarrow$ MERGE_TS(*old_ts*, *new_ts*)
  STORAGE_WRITE(k, $\langle$*merged_value*, *merged_ts*$\rangle$)

B. Causal Storage

As we have done for causal multicast, it is possible to derive a generic algorithm to ensure causal consistency in a distributed storage system. The algorithm is presented in Alg. 5 and it builds on the generic causal multicast algorithm introduced in the previous section.

The extensions of Alg. 5 with regard to Alg. 1 are mostly focused on tracking and handling each client's read and write dependencies (variable *client_clock*). When the client performs a write, a new timestamp is associated with the update. This timestamp is computed by combining the value of the *client_clock* with the clock of the storage node to which the client is attached. Updates are afterward multicasted to all nodes that replicate that object. These updates are delivered in causal order and are kept pending at remote nodes until is safe to apply them. When an update is delivered, the value written by the client is merged with the locally stored object value and the corresponding timestamps are also merged. Note that, similarly to Alg. 1, Alg. 5 can also be instantiated to use Lamport, vector, or matrix clocks: this can be achieved by selecting the appropriate implementations of the INIT_CLOCK, MERGE_TS, PREPARE, MESSAGE_READY, MESSAGE_SAFE, and UPDATE_LOCAL_CLOCK procedures from Alg. 2, Alg. 3, or Alg. 4, respectively.

TABLE I: Metadata Configurations

		One single group		One group per object	
		1		K	
Lamport clock	L	1L	$O(1)$	kL	$O(K)$
Vector clock	V	1V	$O(N)$	kV	$O(KN)$
Matrix clock	M	1M	$O(N^2)$	-	-
N < K		1L < 1V < 1M < kL < kV			

The algorithm also assumes the existence of a MAP_KEY_TO_GROUP function that can map objects to the groups used by the multicast algorithm. The next section discusses how object keys can be mapped to groups.

IV. Metadata Costs

The implementation of causally consistent storage proposed in the previous section, captured by Alg. 5, can be configured to use different amounts of metadata. There are two main mechanisms that affect the amount of metadata required by the algorithm. The first is the type of clocks that are used to keep track of causality, i.e., if the algorithm is instantiated to use Lamport clocks, vector clocks, or matrix clocks. We use the letters "L", "V", and "M" to identify each of these alternatives. The other is how to map object keys to multicast groups. In this paper, we consider only two scenarios, namely: updates for all objects are propagated using a single multicast group (we identify this option by the prefix "1") or each object uses a different group, of its own, to propagate updates (this option is identified by the prefix "k").

The two mechanisms can be combined in different ways, as depicted in Table I. For example, it is possible to use Lamport clocks to track causality, and keep a different clock for each object (configuration named "kL") or to use a vector clock to track of causality but maintain a single vector clock for the entire system (configuration "1V"). These mechanisms address orthogonal aspects of the system's operation. The choice between Lamport, vector, or matrix clocks allows to capture different levels of detail about the *processes* that produce updates and the *processes* that are going to receive the update. The choice of the mapping function allows to capture different amounts of detail about which objects have been targeted by each update. We illustrate this fact with the following examples:

- Consider a system with 4 process, p_1, ..., p_4, using kL and a process p_3 that receives an update u from p_2 with timestamp 5 for object o without having received previously an update for o with timestamp 4 from p_1. On one hand, because each object uses its own clock, the process p_3 knows that the missing update is for object o and not for some other object. On the other hand, because Lamport clocks are being used, p_3 cannot know which other process(es) did generate the update(s) in the past of update u. Due to this lack of detail, p_3 needs to wait for an update from both p_1 and p_4 before delivering u.
- Consider a system that uses 1V and a process p_3 that receives an update u from p_2 with timestamp $[4, 1, 0, 0]$ for object o without having received previously an update from p_1 with timestamp $[4, 0, 0, 0]$. On one hand, because vector clocks are used, it knows that the missing update must be received from p_1, and not from p_4. However, because a single vector is used for all objects, p_3 cannot guess which object generated the missing update. Due to this lack of detail, p_3 may be forced to wait for an update from p_1, even if update with timestamp $[4, 0, 0, 0]$ was performed on an object not replicated by p_3.

To avoid both types of false dependencies, illustrated by the examples above, one may use one vector clock for each different object (configuration kV). Unfortunately, as Table I shows, this is the most expensive configuration (note that, in most practical systems, the number of objects is much larger than the number of nodes in the system). Conversely, the configuration 1L is the configuration that uses less metadata, as a single Lamport clock is used to track causality for all objects. The table also shows how the other configurations compare to each other regarding the amount of metadata they require. Because updates are always propagated to all replicas of a given object, the configuration MO does not bring any advantages over kV, and therefore we do not consider it in our study.

V. Empirical Study

This section presents a study to characterize the scenarios where using more expensive metadata brings advantages that justify the additional bandwidth and storage space.

A. Experimental Setting

All experiments were performed using the PeerSim simulator [17], extended with a networking module that provides point-to-point First In, First Out (FIFO) channels, and allows network latency to be configured.

We consider a setting of $N = |\mathcal{N}|$ nodes that collectively store $K = |\mathcal{K}|$ objects. Storage nodes represent data centers placed in different geographical locations. The latency among nodes is configurable. Every object $k \in \mathcal{K}$ is partially replicated and stored in a subset of all nodes *replicas*$(k) \subseteq \mathcal{N}$, i.e., we considered a setting with partial replication. Full replication is a particular case of this general setup, where every node replicates all objects.

Each node has a fixed number of clients that perform read and write operations on objects replicated on that node. Clients execute a loop where they perform a sequence of 0, 1, or more reads followed by a single write. The number of reads that precede each write defines the read/write ratio of the workload. Clients select objects to read or write randomly according to some distribution. There is a configurable think time between any two consecutive client operations, for simulating processing time, user interaction, and client-server latency.

The set of parameters that control our experiments, listed in Table II, is the following:

TABLE II: Parameterization

Configuration	Variable	Distribution	Value
Number of nodes	N	-	16
Network latency	$\delta_{i,j}$	gaussian	$[30ms, 400ms]$ taken from [18]
Number of objects	K	-	1600
Replication degree	$R(k)$	constant	$[2, N]$
Objects per node	$S(n_i)$	uniform	$R \cdot K/N = R \cdot 100$
Clients per node	C_i	constant	10
Client join rate	J_i	gaussian	50ms
Client think time	T_i	constant exponential	$[10ms, 1000ms]$ $\lambda \in [10ms, 100ms]$
Read/Write ratio	rwratio	constant	$\frac{writes}{reads} = 0.1$
Object access pattern	$P_i(K)$	uniform zipfian	$1/S(n_i)$ $\alpha = 0.9$ (from [19])

- The total number of nodes is denoted N; in all experiments we have $N = 16$. Each node (server) has 10 clients.
- The network latency, among any pair of nodes (i, j), follows a Gaussian distribution with average $\overline{\delta}_{i,j}$. The averages are taken from the real latencies observed in AWS datacenters [18], with 6 nodes in Europe, 4 in North America, and 6 in Asia.
- The number of objects K is constant in all experiments and set to 1600.
- All objects have the same number of replicas, and we vary the number of replicas from 2 to N, with the latter corresponding to full replication.
- The number of objects stored at each node is uniform (i.e., all servers store $R \cdot K/N$ objects).
- We consider that each server has 10 clients; We also consider that clients join the system and start performing updates at different moments in time. This is dictated by the J_i variable that states the rate at which new clients join the system.
- We consider two scenarios for the think time used by clients. In the first scenario, the think time of all clients is constant, with all clients producing updates at the same pace. In the second scenario, clients of different nodes have different think times, according to the average (λ) of an exponential. This means that each individual client will produce updates at different rates.
- All clients use the same write-read ratio, namely 0.1.
- Finally, each object has a pre-determined probability $P_i(k)$ of being accessed by any client. This probability can follow either a uniform distribution where all objects in a certain server i have the same probability of being accessed, or the access pattern follows a Zipfian distribution where some objects are accessed more frequently than others.

B. Characterizing the Workloads

The large number of features that defines a scenario, make the analysis of distributed storage systems particularly hard. In fact, it is infeasible to experiment with all possible combinations of the features enumerated in Table II. Still, in our experimental work, it was possible to observe that some mechanisms exhibited the same performance in a multitude of apparently distinct scenarios. This observation has motivated us to find a new set of features that can capture the properties of the scenarios that are relevant for the performance of the different causality tracking mechanisms.

Therefore, we propose two novel features that help characterize the workload for the purpose of understanding the impact of different metadata techniques. We recall that vector clocks capture information about which nodes have sent updates in the causal past and that the mapping function allows us to capture information about which objects were updated in the causal past. The following parameters take these facts into consideration:

- **Update Generation Rate Asymmetry (*GRA*):** We define the update generation rate asymmetry as the ratio between the fastest and slowest average update frequency for all nodes in the system.
- **Object Ownership to Objects in Causal Past Ratio (*OPR*):** Given the set of updates received in a node, U, the causal frontier contains all updates $u \in U$ for which there is no other updates that happened after u, i.e., $\nexists u' \in U : u \rightsquigarrow u'$. The causal frontier defines the direct dependencies of an update, i.e., the updates that are relevant to track causality. We define OPR as the ratio between the number of objects replicated at a node and the objects with updates in the causal frontier.

C. Scenarios

- **Variable *OPR* scenario**: In this scenario we vary the OPR and change the asymmetry of the system by considering multiple fixed GRA values.
- **Variable *GRA* scenario**: In this scenario we vary the GRA and increase the asymmetry of the system by considering different OPR values as well as the T_i and J_i distribution.
- **Highly uniform scenario**: In this scenario we consider an highly uniform system and vary the underlying network latency.

D. Costs and Benefits

We now discuss the costs and benefits of the different choices that we consider in this analysis.

The size of the metadata used by the different techniques was presented in Table I. As noted before, metadata consumes network bandwidth and storage, because the metadata needs to be sent with each update and needs to be stored to be later retrieved. The larger the metadata, the more processing time and memory is needed to compare and merge clocks. In our experiments, we abstract from storage and CPU/memory costs and only take into account the bandwidth costs.

Naturally, if the deployment is constrained in terms of storage, processing, or bandwidth, the forms of causality tracking that consume more metadata may saturate the system. This is illustrated in Figure 1 that shows the point where different configurations saturate the links among the nodes. However, a more interesting problem is to understand if the techniques that use more metadata bring benefits in an unconstrained system. Therefore, even if the system is not saturated, there is no reason to waste additional resources unless some benefits

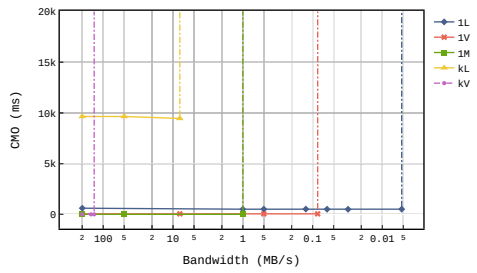


Fig. 1: Bandwidth saturation: *CMO* 99th percentile; varying Bandwidth. $N = 16$; $K = 1600$; uniform think time $T(i) = 15$; uniform $R(k) = 5$; uniform access pattern; *GRA* $= 0.7$ *OPR* $= 0.2$

can be extracted. As such, in the remaining of our experiments, we always consider that the available bandwidth is enough to avoid link congestion when the target causality track is used. This allows us to assess if a mechanism can bring benefits when enough resources are available.

In this paper we consider the *consistency maintenance overhead* (*CMO*) on remote update visibility latency as the benefit function. We measure *CMO* as the time it takes for an update u to be applied at node i *after* u has been received by node i. The smaller the value of *CMO* the better. As described, an update u received at a given node i can only be applied after the node knows that all updates in its causal past have been applied. As discussed, in this process, u may need to wait for a real dependency that is missing (because the system is asynchronous, messages can be delayed in the network) or may be delayed due to a false dependency, that results from the lack of detail of the metadata scheme. As schemes that use larger metadata are likely to exhibit less false dependencies, we expect schemes that use larger metadata to introduce fewer delays in applying remote updates, contributing to a smaller latency in the visibility of remote updates.

E. Experimental Analysis

We begin our experimental analysis by observing how varying the *OPR* affects the *CMO* metric. All these experiments are presented in Figure 2. In this scenario, we start with a relatively uniform system and increase its asymmetry by considering different values of GRA; additionally, for each GRA value, we vary the OPR. By comparing Figures 2a, 2b, 2c and 2d we can mainly extract two interesting results: First, as we consider increasing GRA values, we can notice that the performance of 1V and 1L starts deteriorating. Second, as the OPR increases, independently of the GRA value, the CMO of both mentioned algorithms also decreases; In fact, the point from which this phenomenon appears to happen is when *OPR* > 0.35. This happens because as objects get more replicated the average *OPR* also increases, which in turn results in nodes receiving more updates with objects in their causal past that they also replicate. Regarding algorithms with more detail, the kL algorithm presents an unexpectedly poor performance as we can see in Figure 2e, in fact as the OPR increases, the CMO also increases. We postpone the intuition on why the kL algorithm performs poorly to Section VI and do not include kL in the rest of our experiments due to its general poor performance. As seen in Figure 2f both 1M and kV improve the performance as the OPR increases. Notice, that kV and 1M have similar behaviors when we change the OPR, but kV has slightly better performance; this is due to the fact that kV can use one FIFO connection per object, increasing the concurrency of each individual server. Keep in mind that this is only true considering we are not reflecting the effect of bandwidth in our experiments.

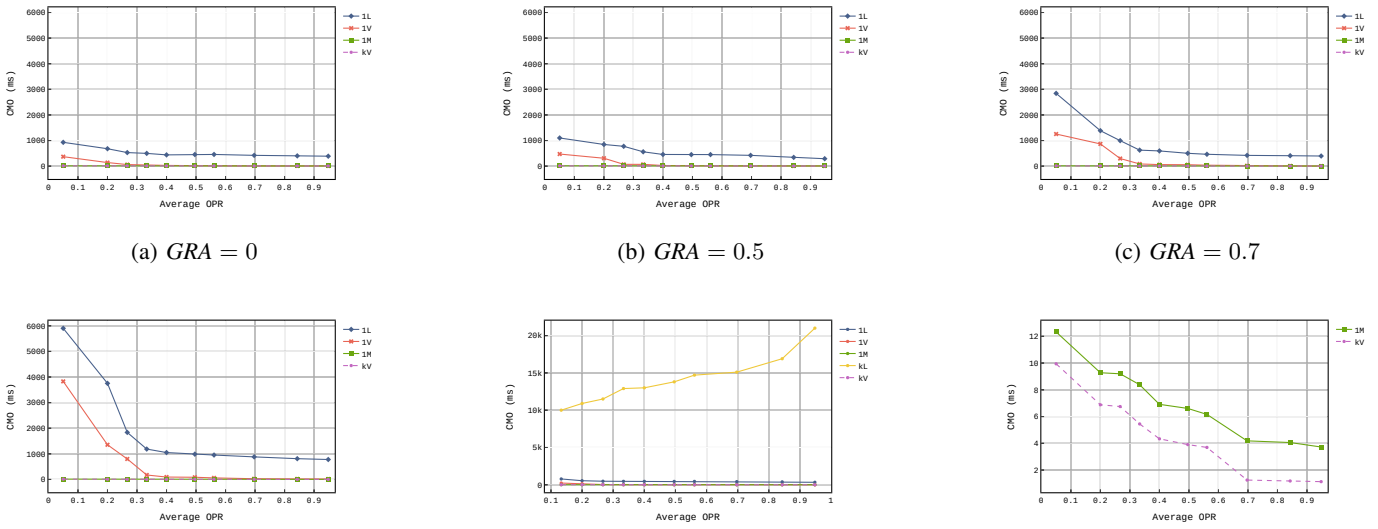
We now address the scenarios where we vary the GRA (Figure 3). As one may have already noticed, increasing the asymmetry of the system induces higher CMO values for 1V and 1L. We can increase the asymmetry in various ways: different object access patterns; object placement; the number of replicas, etc. In Figure 3a we can observe that 1V performs especially well in full replication, and despite the GRA varying, its performance is similar to the 1M and kV algorithm. As we further increase the asymmetry (Figures 3b and 3c) the 1V algorithm starts performing fairly bad. In fact, for GRA values larger than 0.7 the algorithm shows a shift in performance. An interesting phenomenon can be observed in Figure 2f; the 1M and 1V algorithms seem to perform better as the GRA increases. This happens because updates are more diverse, meaning that most likely a certain FIFO connection will be less saturated with constant updates, resulting in a lower load for each link, and, consequently inducing lower CMO values.

It is clear that, if clients access the system at the same moment and have the same uniform think time, updates will be generated at the exact same rate. Intuitively, in such a scenario, one would expect the 1L algorithm to output *CMO* values closer to the 1V algorithm. However, this is not the case and the 1L algorithm still experiences a high *CMO*. This is an artifact of the asymmetry introduced by the latencies of the underlying network. In Figure 4 we can observe such relationship. Each server can only apply messages as fast as the slowest node generates them; since, in this case, every server generates new updates at the same rate, the asymmetry happens when messages take different times to arrive at the various servers. The 1L algorithm would only perform well if every server generates messages at the exact same rate and all messages arrive at every server at the same time, which is a highly unlikely scenario in real systems.

VI. Results and Takeaways

Our experiments highlight some interesting properties of the different metadata choices. In the following section, we enumerate the main takeaways from our study and derive a decision chart that aims at helping system designers to pick the right technique for their applications.

1L only performs well in symmetric scenarios: Lamport clocks cannot capture concurrency and do not allow us to identify the source of updates in the causal past of another



(a) *GRA* $= 0$ (b) *GRA* $= 0.5$ (c) *GRA* $= 0.7$

(d) *GRA* $= 1$ (e) *GRA* $= 0.7$; kL included (f) *GRA* $= 0.7$; 1M and kV experiments only

Fig. 2: Variable *OPR* scenarios: CMO 95th percentile for varying *OPR* values;$N = 16$; $K = 1600$; $J_i = 45ms$; multiple *GRAs* considered; variable uniform R(k); zipfian P(K)

update. This forces a node to wait for updates from every other node in the system in order to make a single update visible. When nodes produce updates at different paces, the entire system is affected by the rate of the slowest node in the system.

Even in symmetric scenarios, 1L is affected by the network latency: Assume that nodes produce updates at exactly the same pace. In order to deliver an update u with logical clock x, node i needs to receive an update with timestamp $\geq x$ from every other node. Even if these updates have been produced exactly at the same real-time instant as update u, there is a delay of approximately the average network latency. This is clearly visible in Figure 4. This suggests that Lamport clocks are not advisable in geo-replicated scenarios, where inter-node latencies are large and diverse.

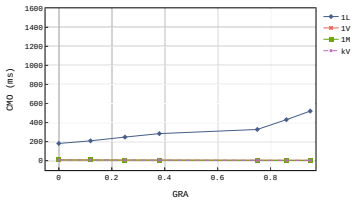
kL brings no advantages w.r.t. 1L, expect in cases of extreme symmetry or extreme partial replication: Although kL keeps additional detail over 1L, which is substantially more expensive, it brings little or no advantages to most scenarios. In fact, to use kL can even be detrimental to the performance of the system. This happens because the *CMO* of an update depends on both the GRA for that object and the *OPR*. Objects that are seldom accessed will have an extremely high *CMO*. Whenever they are updated, they will delay any updates that read that object. There are, however, two exceptions to this. The first exception is when we have a uniform $R(x) = 2$, which, in that case, a certain node doesn't need to wait for all other nodes to perform an update on the said object; this results in the kL algorithm to perform similarly to kV. The other corner case is when all objects are updated at the same rate by all nodes; in such a scenario, kL would perform similarly to 1L.

1M and kV have similar performance: Both 1M and kV have similar performance in our experiments. Thus, the choice between 1M and kV only depends on the size of the metadata. For scenarios where the number of objects is larger than the number of nodes, the 1M is more advisable. For scenarios where the number of objects is smaller than the number of nodes, kV is better. Note that, in many practical systems, it is possible to group objects in a few numbers of *partitions*, where objects in the same partition use the same replication strategy. In this case, the use of kV can be a sensible choice.

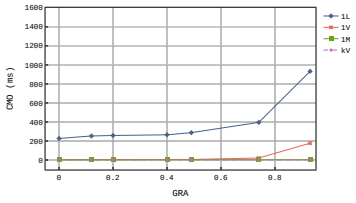
Despite 1M and kV having similar performances, the kV algorithm can perform slightly better: The kV algorithm has one FIFO connection per object. When the system has enough network capacity, this results in a higher level of concurrency when compared with having only one FIFO connection such as in 1M. Despite having this benefit, the differences in the *CMO* are actually very low, which, in some cases, doesn't seem to pay up the much metadata cost.

1M and kV perform better in more asymmetric systems: The 1M and kV experience fairly lower false dependencies than other algorithms. This fact, allied with asymmetry to update generation and object placement, results in links between servers to be less saturated which itself results in updates arriving faster to nodes (because there are fewer messages to be FIFO ordered) and lower *CMOs*. This phenomenon can be observed in Figure 3d

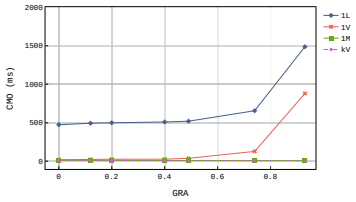
1M/kV significantly outperform 1V in systems where *CMO*



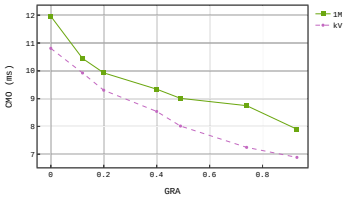
(a) *OPR* $= 1$; $J_i = 50ms$; $R(k) = N$, uniform $P(k)$



(b) *OPR* $= 0.26$; $J_i = 0ms$; zipfian $P(k)$



(c) *OPR* $= 0.26$; $J_i = 50ms$; zipfian $P(k)$



(d) *OPR* $= 0.26$; $J_i = 50ms$; 1M and kV included; zipfian P(k)

Fig. 3: Variable GRA scenarios: CMO 95th percentile for varying *GRA* values;variable exponential T_i; $N = 16$; $K = 1600$; uniform R(K); multiple *OPRs* considered;

is low: In partially replicated systems, nodes do not store a replica of every object. Therefore, it is likely that in the causal past of updates that need to be applied at node i there are updates to objects that are not replicated by i. Keeping different vector clocks for each object, or a matrix clock offers the necessary detail to prevent nodes to wait for updates they will never receive. This is clearly visible in Fig. 2, where the 1V algorithm gains performance as the *OPR* increases.

Based on these insights, it is possible to construct a decision tree to help system designers select the most suitable metadata scheme for their storage system, as a function of the deployment and workload characteristics. The decision tree is depicted in Fig. 5. First, if the system is fully replicated, there is no need to use additional metadata and the choice is between the 1L and the 1V algorithms. In this scenario, if the system is not highly uniform (we consider a highly uniform system to be $OPR = 1$; $GRA = 0$; $J_i = 0$; *uniform* T_i; *uniform* $P(K)$; $sd(\delta_{ij}) = 0$, where sd is the standard deviation), the wise choice is to use a vector clock, otherwise, if the system designer supports having higher *CMOs* (e.g. bandwidth/processing power may be scarce which justifies the use of less metadata at the expense of a higher *CMO*) then it is better to use a Lamport clock. If we are instead on a partially replicated scenario, we first check if our *OPR* is greater than 0.35. As we have seen, the higher the *OPR*, the better the 1V algorithm will perform. If the *OPR* is lower than 0.35, it means that most likely we also have few replicas for each object; In such a case, we must assess whether the system is uniform. From our previous analysis, we can correlate GRA with the system asymmetry. For *GRA* > 0.7 the 1V algorithm starts showing performance problems, therefore we should use a vector only when the *GRA* is lower than 0.7. We use the kV algorithm if the scenario where the number of objects is smaller than the number of nodes; otherwise, it is generally better to use a Matrix clock for the entire system. There is still a special case where objects are replicated in only two servers and the number of objects is smaller than N which in that case you should use kL.

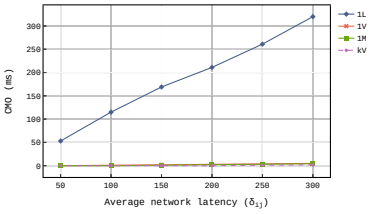


Fig. 4: All uniform scenario: *CMO* $99th$ percentile; varying average δ_{ij}; $N = 16$; $K = 1600$; $J_i = 0$; uniform $T_i = 15$; $R(k) = N$; *OPR* $= 0$; *GRA* $= 0$; uniform $P(k)$. In this scenario, the worst case δ_{ij} is equal to the average δ_{ij}

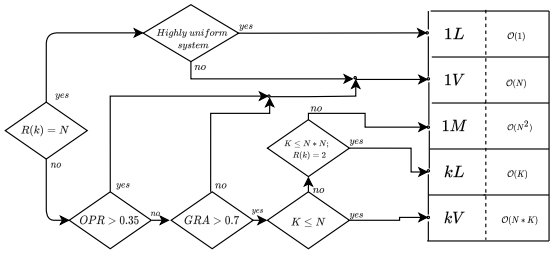


Fig. 5: Decision chart for the various metadata schemes.

VII. Related Work

There is a considerable amount of work that addresses causal consistency, however, to the best of our knowledge, only a small part of this work provides an analysis of the trade-offs imposed by the different causality tracking mechanisms.

Bravo [9] makes a thorough analysis of the numerous systems that support causal consistency and shows that there is a direct correlation between the metadata size and the number of false dependencies, even for systems with optimized mechanisms. Bravo also raises awareness to the fact that efficient causal consistency in partially replicated scenarios is difficult to achieve, and it's a trade-off between minimizing the amount of metadata being handled and the loss of concurrency by minimizing said metadata.

Cheriton and Skeen [20] perform a very detailed exposition of the limitations of causally ordered communication and correlate visibility latency with the amount of buffering of updates due to missing dependencies. Still from a more theoretical point of view, Bailis et. al. [21] extend this analysis in the context of causal consistency. Their work also identifies throughput (i.e. the rate at which clients generate new updates) and visibility latency as competing goals. Additionally, they raise awareness to the poor scalability of the mechanisms used to ensure causal consistency.

While our work focuses on studying the trade-offs of logical clocks, Bravo et. al. [22] make an interesting analysis between the advantages of using different types of clocks to enforce causal consistency by analyzing different existing systems. Their observations address the impact of the client's access patterns on the performance of systems that use logical clocks and present physical clocks as an alternative to avoid some of the limitations of logical clocks. However, unlike our work, their study does not identify which workload patterns favor a certain type of metadata.

The analysis, in all these studies, is based on simplified models that fail to capture the interactions among the different parameters that characterize the workload of causally consistent storage systems. To the best of our knowledge, our work is the first to identify a set of features that help to understand the performance of different mechanisms. Our study is also the first to make an experimental assessment of how different metadata choices affect the remote visibility latency in distributed storage systems.

VIII. Conclusions

Ensuring causal consistency is at odds with the mechanisms used to keep track of causal dependencies. On the one hand, one can opt to use more detail to better track causality. On the other hand, this results in consuming a substantially larger amount of network bandwidth and storage space. In this paper, we addressed the problem of whether the benefits of using more complex structures are worth their cost in the context of partially replicated systems. We have shown that for some workloads the use of more expensive clocks does bring significant benefits and that for other workloads no visible benefits can be observed. To help system designers to pick the right mechanisms for their applications, the paper introduces two novel features, GRA and OPR, that capture relevant properties that affect the performance of different causality tracking mechanisms. Based on these features and an extensive experimental evaluation, we derived a decision chart that characterizes different scenarios where each type of clock is more beneficial.

References

[1] M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM TOCS*, vol. 3, no. 1, p. 63–75, Feb. 1985.

[2] G. Ricart and A. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Commun. ACM*, vol. 24, no. 1, p. 9–17, Jan. 1981.

[3] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, Mar 1995.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[5] D. Parker, G. Popek, and et. al., "Detection of mutual inconsistency in distributed systems," *IEEE Transactions on Software Engineering*, vol. 3, pp. 240–247, 1983.

[6] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[7] S. Sarin and N. Lynch, "Discarding obsolete information in a replicated database system," *IEEE Transactions on Software Engineering*, vol. 1, pp. 39–47, 1987.

[8] A. Acharya and B. Badrinath, "Recording distributed snopshots based on causal order of message delivery," *Information Processing Letters*, vol. 44, no. 6, pp. 317–321, 1992.

[9] M. Bravo, "Metadata management in causally consistent systems," Ph.D. dissertation, Université Catholique de Louvain, 2018.

[10] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 3, p. 272–314, Aug. 1991.

[11] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, 1991.

[12] P. Mahajan, L. Alvisi, M. Dahlin *et al.*, "Consistency, availability, and convergence," *University of Texas at Austin Tech Report*, vol. 11, p. 158, 2011.

[13] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 141–155, 2016.

[14] C. Papadimitriou, "Serializability of concurrent database updates," Massachusetts inst of tech cambridge lab for computer science, Tech. Rep., 1979.

[15] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[16] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[17] A. Montresor and M. Jelasity, "Peersim: A scalable P2P simulator," in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE, 2009, pp. 99–100.

[18] "Home: Cloudping - aws latency monitoring." [Online]. Available: https://www.cloudping.co/grid

[19] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *IEEE INFOCOM*, vol. 1, 1999, pp. 126–134.

[20] D. Cheriton and D. Skeen, "Understanding the limitations of causally and totally ordered communication," in *Proc. SOSP*, 1993, pp. 44–57.

[21] P. Bailis, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proc. SOCC*, Oct. 2012, pp. 1–7.

[22] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. Rodrigues, "On the use of clocks to enforce consistency in the cloud." *IEEE Data Eng. Bull.*, vol. 38, no. 1, pp. 18–31, 2015.