International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. 12, No.1 (Oct-2022)

https://dx.doi.org/10.12785/ijcds/120177

A New Adaptive Causal Consistency Approach in Edge Computing Environment

Abdennacer Khelaifa[1, 2], Saber Benharzallah[3] and Laid Kahloul[4]

[1]*Biskra University, 07000 Biskra, Algeria*
[2]*LIAP Laboratory, University of El Oued, Algeria*
[3]*LAMIE Laboratory, University of Batna 2, Algeria*
[4]*LINFI Laboratory, University of Biskra, Algeria*

Received 18 Jul. 2021, Revised 20 Sep. 2022, Accepted 27 Oct. 2022, Published 31 Oct. 2022

Abstract: Edge computing is a new computing paradigm that has emerged to offload computation and storage to edge devices in order to get a shorter response time and more efficient processing. Nevertheless, adopting a consistency scheme that can conserve multiple replicas while guaranteeing a good level of consistency is an open issue. Moreover, a data store with only one consistency level is not suitable for applications that have different consistency requirements. In this paper, we propose MinidoteACE, a new adaptive consistency system which is an amelioration version of Minidote a causally consistent system for edge applications. Unlike Minidote which supports only causal consistency, our model allows applications to run also queries with stronger consistency guarantees. Experimental evaluations show that throughput decreases only by 3.5% to 10% when replacing causal operation with a strong operation. However, update latency increase significantly for strong operations up to three times for 25% update workload.

Keywords: Minidote, MinidoteACE, Causal consistency, Adaptive Consistency.

1. Introduction

Choosing a consistency model in a distributed data store is a critical decision mainly when the latter is geo-replicated. The growing requirement for short response time and wide availability has led many geo-replicated data stores to prefer weak consistency models over strong consistency models. However, using weak consistency models, such as eventual consistency [1], [2] or adaptive consistency [3], renders the programming model more complicated to application developers, who are obligated to fix explicitly data inconsistency issues introduced by these models. This has led to the emergence of the causal consistency model [4] that has been demonstrated to be the strongest consistency model in an always-available system. Causal consistency captures the potential causal relationships between events within one client and also the reads-from relation between different clients. The causal consistency model ensures that a write operation will not be applied until all the operations that precede it are applied. Causal consistency is a good choice for geo-replicated data stores, since it can reduce the anomalies issued by eventual consistency, yet it tolerates partitions and avoids long latencies associated with strong consistency. However, causal consistency is a relaxed consistency model that optimistically allows replicas to confirm updates concurrently to achieve higher availability. This sometimes leads to conflicts across replicas. A Conflict-free Replicated Data Types (CRDTs) [5] are proposed as a proven solution to handle conflicts. A CRDT is an abstract data type that ensures that replicas can be modified without coordinating with each other and if they have received the same set of updates, they reach the same state, deterministically.

In this context, AntidoteDB [6], [7] is a highly available geo-replicated key-value data store. Thanks to causal consistency and CRDTs, AntidoteDB allows developers to build solid applications without reducing performance or horizontal scalability provided by AP/NoSQL storage systems (systems that sacrifice consistency to ensure availability and Partition tolerance). Minidote [8] is a lightweight version of AntidoteDB designed for Edge Computing Environment. Like AntidoteDB, Minidote keeps providing causal consistency as the only consistency level in the system. In fact, Minidote supports two types of operations: read and write (update). Read operations are performed locally. After receiving a client read query, the corresponding node uses only the local replica to answer the query. Similarly, write operations are applied and committed locally before broadcasting their effects and dependencies to other nodes. A node that receives effects of a remote update, can apply them to the local replica after verifying the update dependencies. Minidote then uses CRDTs to

E-mail address: abdennacer-khelaifa@univ-eloued.dz, sbharz@yahoo.fr, kahloul2006@yahoo.fr

resolve inconsistencies generated by concurrent updates.

However, causally consistent systems [4], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] including Minidote face two issues: 1) They provide only a single consistency level which is not sufficient for particular kinds of applications. Taking an online auction as an example of the systems that change the recommended consistency level during the runtime. For the early bids of the auction, the system does not require strong guarantees because the data are not so important for the final deal. However, When the deadline draws near, the participants should see the latest bids to make the next bid. Therefore, the consistency requirements become higher and the data should be always modified under strong consistency. 2) They do not distinguish between system replicas and clients, in other words, each node of the cluster is considered as a replica of the service and a client of the service at the same time. The latter assumption means implicitly that every client is always associated with one of the nodes. In such a case, the client sticks to the same replica, and the system provides textitSticky availability [23] . However, clients are independent entities, which can switch from one replica to another, and which, in some situations, may lose communication to a system replica. Namely, the client needs high availability. But if a client moves from one node to an alternative one, it could mean giving up causal consistency; this can happen if the alternative node does not receive all the client's updates, and therefore it leads to breaking the read-your-writes guarantee implied by causal consistency.

To overcome these issues, we propose MinidoteACE, a new adaptive consistency strategy that enables the client to choose the level of consistency for each update he sends either causal or strong. For this purpose, we add another type of operation: “strong update”, to Minidote system to become able to support three types of operations: read, causal update (the existing update operation) and strong update. We keep the existing behaviors for read and causal update operations, however we use the implementation of Causal stability [24] to carry out strong update operations. An operation is stable if it is delivered to all the nodes of the cluster. Hence, a strong operation will not be executed until it becomes stable. Towards that end, we delay the execution of strong queries until they arrive at the other nodes. The new consistency level is weaker than the typical strong consistency as it just ensures the arrival of queries to the system nodes, not their execution. Our model provides stronger consistency guarantees to ensure high availability. If we assume that operations will be executed correctly, replicas will have all recent updates so clients will be able to regain their updates when they switch from one node to another. Hence, in addition to causal consistency, clients can also perform updates with stronger guarantees. Providing adaptive consistency allows application designers to choose the convenient consistency level for each operation according to application needs and the system's context. The application can specify which operations should be run under causal consistency and which others should have stronger consistency guarantees.

We have experimentally evaluated our model using Basho_bench [25] : a benchmarking tool that has been modified to evaluate AntidoteDB. To do that, We perform the necessary amendment on the Basho_bench benchmark to test MinidoteACE. Our evaluation proves that MinidoteACE can support a certain proportion of strong operations without affecting latency or throughput.

In this paper, we make the following contributions:

- A new adaptive consistency approach that enables clients to choose the consistency level for their queries either causal or strong according to their requirements.
- We develop MinidoteACE, an ameliorated version of Minidote that provides the new consistency strategy in edge computing environment.
- We implement our proposed contributions and perform the necessary amendment on the benchmarking tools to ensure a meaningful evaluation.

The outline of the rest of this paper is as follows. We discuss related work in Section 2. Section 3 briefly describes the principal concepts. Section 4 describes the Minidote system. Section 5 presents our proposed approach (MinidoteACE) including the protocol and the algorithms. We evaluate the performance of MinidoteACE in Section 6. Finally, Section 7 concludes the paper and makes recommendations for future work.

2. Related works

Many prior work efforts have studied data management, mainly the trade-off between consistency and availability that has been involved in building distributed data stores. Therefore, there are many systems providing different semantics to application developers. For instance, Google BigTable [26], Microsoft Azure Storage [27] and Apache HBase [28] provide strong consistency. These systems provide simple semantics, but suffer from long latencies and partition-intolerance. Other systems like Amazon Dynamo [29], Cassandra [30] and MongoDB [31] provide eventual consistency which provides excellent performance and tolerates partitions, but renders the programming model more complicated due to inconsistency handling difficulties. MinidoteACE takes an intermediate position in this trade-off by embracing causal consistency semantics.

Many previous works have recognized the convenience and applicability of causal consistency. COPS [4] was the first system that defined the concept of causal+ consistency, a causally consistent model that ensures the convergence offered by eventual consistency. Causally consistent systems need to track causal dependencies between operations in the form of a piggybacked metadata which is used to tag

operations and to capture the causal past of clients with each operation. Multiple forms have been used to represent metadata, COPS [4] for instance uses explicit causal histories that, for each operation, enumerate all the operations that are in its causal path. Logical clocks are also used to track causality in ChainReaction [10], SwiftCloud [12], and Orbe [9]. The simple logical clock has an entry for each replica that will be incremented upon a new update is applied on the replica. Other approaches like GentleRain [11], Saturn [19], Cure [14], and Legion [13] use physical clocks that tag operations with the physical local time. However, Okapi [16] and Eunomia [17] choose to make a combination between logical and physical clocks: hybrid clocks. However, tracking causality accurately requires maintaining an incremental size of metadata that may affect the system performance. Our model uses smaller size timestamps than vector clocks to encode the causality between messages, a causal graph to store the dependencies between messages, and an efficient algorithm for causal delivery and stability.

Several recent works have developed storage systems for the edge environment. DPaxos [32] and EdgeCons [33] rely on Paxos-based protocols that provide only strong consistency, and suffer from long latencies. FogStore [34] however, implements two functionalities that allow it to manage consistency between Fog nodes in an optimal manner. The first one aims to minimize the response time between the clients, the devices, and the copy of a data record by relying on a replica placement strategy. The second functionality is a context-aware mechanism that chooses the consistency level of the client's query according to the client's context. PathStore [35] also supports several consistency levels according to the client needs mainly, eventual, session, and strong consistency. Although Fogstore and Pathstore strive to optimize latency, they do not preserve causality. Gesto [36] is a hierarchical architecture that allows cloud data stores to cover edge networks while providing causal consistency. Gesto splits replicas into geo-replicated groups where each group of edge replicas has one datacenter. The native replication protocol of the cloud data store is used between data centers. However, a novel causality tracking mechanism is integrated into each group. Moreover, Gesto uses a multipart timestamp enabling scalability and fast migration. Minidote (detailed in section 4) is also a causally consistent system for the edge. Unlike our model, Gesto and Minidote provide only causal consistency which is not sufficient for all kinds of applications that require stronger consistency guarantees.

3. Background and definitions

In this section, we aim to understand the main concepts around our system. In fact, our contribution is based on the Minidote system which is a lightweight key-value storage system that is designed for Edge Computing Environment. Minidote is a causally consistent system that allows it to provide fairly high scalability and performance while keeping a good consistency level. We introduce firstly, causal consistency by giving its definition, implementation, and causal stability concept. Later, we define CRDT data types as a way of convergent conflict handling. Finally, we give an overview of the computing environment of our system: Edge computing.

A. Causal Consistency

Causal consistency captures the notion that different data store nodes should see causally-related operations in the same order. Causality [37], [38] is a happens-before relationship between two events. For two operations a and b, we say that a happens before b or, alternatively b causally depends on a, and we write $a \rightarrow b$, if and only if at least one of the taking after conditions hold:

- Thread-of-execution: a and b belong to a single thread of execution, and a precedes b.
- Reads-from: b reads the value written by a when b is a read operation and a is a write operation.
- Transitivity: There is some other operation c such that $a \rightarrow c$ and $c \rightarrow b$.

Consequently, a storage system provides causal consistency, if it does not commit any write operation before executing all its causally-related operations. For example, in a social network, Mark updates an old post on his wall. Alan sees the update and writes a comment about it. Then, Alan's comment shouldn't appear to their friend Paul before he sees Mark's update on the post as the comment of Alan causally depends on the update of Mark.

1) Causal consistency implementation

The implementation of Causal consistency can be performed using two simple steps [4]:

- Assign a set with all the preceding operations to each operation; this set we'll be called the dependencies of o and refer to it as $dep[o]$.
- Perform o once all its preceding operations (operations in $dep[o]$) have been executed.

Also, let's mention the set of executed operations $Executed$. If the process sends an operation o, then the dependencies of o will be the set of operations that have been performed at that time, i.e. $dep(o) = Executed$. If the operation p follows the operation o, then the dependencies set of p contains o :

$$o \in dep(p) \quad (1)$$

It may however occur that p does not cause o and o does not cause p. If so, o and p will be competing with each other :

$$o < dep(p) \wedge p < dep(o) \quad (2)$$

2) Causal stability

An operation o is causally stable (denoted by $stable(o)$) if it belongs to the dependencies set of each operation p

that will be executed in the system [39], [24]:

$$stable(o) \equiv \forall p : p < Executed \Rightarrow o \in dep(p) \quad (3)$$

In other words, stable(o) means that o has been executed by all the nodes of the system. Hence, every new operation will be in the future of o.

Example 3.1:Let's have a cluster with three nodes: a, b and c. o, p, q and r are operations that will be Executed in the cluster.

a performed o and then p.

b noticed o and p, and then performed q.

c noticed o and performed r.

So, the Executed sets for each node are the following, Executed

$$Executed\_a = \{o, p\}$$

$$Executed\_b = \{o, p, q\}$$

$$Executed\_c = \{o, r\}$$

Remember that: the set of operation dependencies is the set of operations performed at the moment of operation submission. We know therefore at this point that any submitted operation (at any node) will include o as a dependence. This means that o is stable. We can't say the same about any of the remaining operations.

B. Conflict-free Replicated Data Type (CRDTs)

In a causally consistent system, there is no need to order concurrent operations; two operations can be replicated in any order if they are not causally related, to avoid synchronization cost. However, If two concurrent update operations target the same object, they are in conflict and produce an inconsistency in the system. Avoiding conflicts usually requires ad-hoc handling [40], [41], [29] in the application logic by employing an automatic strategy to handle conflicts deterministically, in the same manner, at every replica. For example, the last-writer-wins rule has been adopted by most of the existing causally consistent systems [4], [42], [11], where the last update overwrites the other updates. Antidote and Minidote rely on CRDTs [5], [6] which are an abstract data type built to be replicated at multiple replicas. CRDTs have a clearly designed interface and exhibit two attractive properties: (1) there is no need for coordination between replicas associated with update operations; (2) two replicas can reach the same state after receiving the same set of updates since they guarantee state convergence by adopting mathematically sound rules. CRDTs include sets, counters, maps, LWW registers, lists, graphs, among others. As an example, a counter data type can handle the following operations: increment(C) and decrement(C). The implementation of a CRDT counter will guarantee that the state of the counter will reach the same value at different replicas whatever the order of increment and decrement operations.

C. Edge Computing

Edge computing [43] is defined as a model of distributed computing that employs technologies allowing to perform computation at the edge of the network. In contrast to the Cloud computing paradigm where data centers handle all storage and processing services, Edge computing aims to improve system scalability and data privacy, reduce latencies and ensure an effective usage of resources (mainly reducing energy consumption). In edge computing, cloud servers and edge devices work together to perform processing. Which computations are conducted on which of these components is determined by a variety of factors, including node capacity and latency requirements. We call an “edge“ any processing, storage, or networking resource located along the path between cloud data centers and edge devices. A cell phone, for example, is regarded as an edge between body things and the cloud, whereas a gateway in a smart home is considered as an edge between house things and the cloud.

4. An overview of Minidote system

Minidote [8] is a replicated key-CRDT store that provides causal consistency with atomic batch-reads and batch-updates while the data is automatically replicated on each node and concurrent updates are resolved using CRDTs. Compared to Antidote, Minidote doesn't support interactive transactions and replica sharding which makes it more lightweight and therefore allow it to run well on less powerful devices. Minidote only has to keep the latest version available, whereas Antidote must be able to serve all snapshots used by currently running transactions.

The inter-dc replication service of Antidote is replaced with a different causal broadcast service provided by Camus [44]. Camus is a CAusal MUlticast Service, that provides different back-ends to guarantee a reliable dissemination and delivery respecting causal order at all replicas using the service. The back-end used in Minidote is an implementation of tagged casual broadcast (TCB) protocol [24] which guarantees that messages/operations will be delivered respecting the end-to-end happen-before relation as seen by the application.

A. Concepts and data structures

Before we describe the different components of Minidote architecture, we will present some concepts and data structure that will be used frequently in the next paragraphs and sections.

State: Each minidote server instance has a state that describes the current status. The state contains a vector clock, a list of dots that causally precede the next update i.e, dependencies, and the dot: the tag of the current operation of this node.

CRDT object: like AntidoteDB, Minidote is a key-value data store so it identifies each object with its key, type, and bucket. So we write Object = {key, type, bucket}. When updating an object, the following parameters should be introduced to the query: the object identifier, the update operation, and the given value. For example, to increment a counter by 1, we write Update = {Obj, increment, 1}.

Vector clock: A vector clock of a system of N nodes is an array of N logical clocks used to manage a partial ordering of events in a distributed system and to keep causality relationships [45], [46]. As in Lamport timestamps, The causality relationship (called happened-before) captured is defined based on passing of information, rather than passing of time. The vector clock is used to track causal information between operations. Operations piggyback this information to allow their delivery in an order respecting the causality of events.

Dot: a pair of (node_Id, counter) which serves as a unique identifier for an operation/message of Minidote node to be broadcast.

Dpgraph: The dependency graph is used to store messages that are not ready to be delivered yet. Graphs better characterize the partial order rather than queues which are better for totally ordered events. The graph uses dots to identify its vertices where each vertex contains the operation's effects, the status of the message (missed, received, delivered, or stable), successors_set, and predecessors_set where predecessors_set/successors_set is a set of dots that precede/succeed this dot.

B. Minidote architecture

Figure 1 shows the architecture of Minidote node and its interactions with clients and other nodes. Each Minidote node contains three principal modules: Minidote_server, Camus_middleware, and CRDT storage.

1) Minidote_server

Minidote_server is the brain of Minidote node which interacts with clients and other modules. A client can perform atomic batch-reads and batch-updates using Minidote APIs: Read_objects (Keys, clock) and Update_objects (updates, clock). For read operations, the client specifies Crdt keys of requested objects and a vector clock. After execution, Minidote_server returns values of the input keys and a new vector clock. However, for update operations, the client should specify a vector clock and for each update: an object's key, an update operation, and input parameters. Minidote_server returns, after execution, a new vector clock. Since Minidote server has a state, vector clock, dot, and dependencies, for each read or update operation, it should wait if the operation's vector clock is later than the current state's vector clock. Then, it asks for locks of the object's keys, if they are not free the query should wait. Hereafter is a brief description of the main procedures of Minidote_server:

Read_objects: Minidote_server performs read operations from local storage and then sends answers to the client.

Update_objects: After acquiring locks, Minidote_server applies updates on the local data store, and then it increments the value that corresponds to the local node on the vector clock. After that and in a parallel way, it releases locks, answers the client by the new vector clock, and calls the broadcast function of the middleware which broadcasts the update effects to the other nodes.

Deliver_remote_Update: this order is received from the middleware to apply a remote update. Minidote_server sends the received update effects to its local storage, updates the vector clock of the state, and adds the received dot to its dependencies.

2) Camus_middleware

The middleware is a low-level layer that ensures causality between messages using the following functions.

Broadcasting updates (Tcbcast): When Minidote_server calls this function, the middleware creates a new vertex for it in the Dpgraph and labels it as 'received'. Then, it broadcasts a message containing the update effects, its dependencies, the dot, and the vector clock.

Receiving a remote message: Upon receiving a remote message, the middleware creates a new vertex in Dpgraph for the received dot with status 'received'. Then it checks its dependencies, if they all have been received, it calls the function Deliver. Otherwise, it creates a new vertex in Dpgraph for each dot in the dependencies that have not yet been received with status 'missed'.

Message delivery (Deliver): when this function is called, the middleware sends the corresponding message to Minidote_server and labels the corresponding vertex as 'delivered' in the Dpgraph. Then, it checks each dot in the successors_set. If all its predecessors have been delivered, it calls the function Deliver for the corresponding message. The middleware notes also that each dot in the predecessors_set has been received in the sender node. If a dot is labeled as 'received' in all the nodes, it will be labeled as 'stable' and the function Stable will be called for this dot.

Stable: when a dot is labeled as 'stable', it will be removed from each predecessors_set in Dpgraph and then, its vertex will be removed from Dpgraph.

Figure 1. A cluster of three Minidote nodes.

3) Storage

Minidote uses the CRDT libraries developed in AntidoteDB. CDRTs support high-level replicated data types such as counters, sets, maps, and sequences which are designed to work correctly in the presence of concurrent updates and partial failures.

C. Consistency Protocol

Hereafter, we will summarize the behavior of Minidote towards client requests:

Each node has two main components: Minidote_server and Camus_middleware.

For read operations, Minidote performs the query from the local instance and forwards the answer to the client.

The client (Application) performs an update u via client api at Minidote_server of a node i.

Minidote_server applies the update u locally and tags it by a new dot. Then, it calls the broadcast function of Camus that broadcasts the effects of the update u to the other nodes. At the same time, it replies to the client request by the new vector clock.

When a node j receives the effects of u through its middleware, it delivers the effects to Minidote_server to be applied if all its dependencies have been already delivered. The update u will be then inserted into the dependencies of new updates generated by the node j.

When the node i receives updates from all nodes containing u in their dependencies, it labels u as local stable.

u becomes stable in the node $j \neq i$ when it is included in the received update dependencies of all other nodes except i.

5. MinidoteACE: Proposed ameliorations and Adaptive consistency approach

Minidote provides only causal consistency and uses Crdts to handle conflicts. In MinidoteACE, we keep the same architecture as Minidote and the same behavior for read operations as well. However, MinidoteACE exposes two types of updates: causal and strong. The system keeps the same behavior as Minidote for causal consistency operations. But in the case of strong consistency operations, it follows these steps.

The client performs an update u at a node i via its Minidote_server.

Minidote_server tags the update u and broadcasts it through the middleware to the other nodes without

Figure 2. Flowchart for consistency proposal.

applying it locally.

When a node j receives u, it inserts it to the strong received updates set $SRUS_j$. The latter will be piggy-backed to all updates and commits sent by the node j.

When the node i receives updates from all other nodes containing u in their $SRUS$, it considers u as stable. Then, it delivers u locally and broadcasts a commit message to the other nodes.

When the node j receives a commit message of u, it labels u as stable and delivers it to Minidote server to be applied.

Hence, Read_objects API keeps the same behavior. However, Update_objects API needs a third parameter that specifies the consistency level of update operations. Therefore, Update_objects API takes the following form: Update_objects (updates, clock, consistency). Figure 2 summarizes the behavior of MinidoteACE for read and update operations.

A. Detailed functions and Algorithms

Read_objects: We keep the read operation behavior without modification. When the client calls the Read_objects API, Minidote_server uses the function described in Algorithm 1 to perform read operations. Minidote_server adds the current server state to the parameters introduced by the client to form the input parameters of read operation. Before performing read operations, Minidote server ensures that the keys are free and compares the operation clock with the state clock and locks the keys. After reading objects, Minidote server releases the keys and returns outputs to the client.

Algorithm 1: Read operation

```
Input: Keys, Clock, State
Output: Values, NewClock
1 wait_release_locks(Keys);
2 check_clock(Clock, State.Vc);
3 lock(Keys);
4 Values = read_crdtObjects(Keys, Clock);
5 NewClock = StateVc;
6 unlock(Keys);
```

Update_objects: We ameliorate this function by adding the possibility to handle strong consistency operations. Our ameliorations are illustrated in Algorithm 2. The client who sends the update operation should specify the vector clock, the consistency level, and the update details (object identification, operation type, and the value). Minidote server uses these elements in addition to the current state as input parameters to the update function that executes the following steps.

Firstly, Minidote_server checks the keys of the

corresponding objects, and if they are acquired by another operation, it should wait. Then, it compares the operation's vector clock with that of the current state. If the stump of the query is greater, this means that the node misses some updates so it should wait until missed updates arrive (lines 1-3). After that, it acquires the locks of targeted keys and then increments its own logical clock in the vector by one (line 4).
Secondly, Minidote_server checks the consistency parameter (lines 5-11), if the consistency is 'causal', it applies the updates locally by calling CRDT APIs. Then, it calls the Tcbcast function of the middleware to broadcast the updates to other nodes. However, if the consistency is 'strong', it broadcasts the updates directly and waits for its stability to be able to apply them locally.
Finally, Minidote_server updates its state's vector clock by the current vector clock. The latter is returned as an answer to the client query and the Keys should be released(lines 12-14).

Algorithm 2: Update operation.

```
Input: Updates, Clock, Cons, State
Output: NewClock
1  wait_release_locks(Updates.keys);
2  check_clock(Clock, State.Vc);
3  lock(Updates.Keys);
4  NewVc:= increment(StateVc);
5  if Cons= 'causal' then
6  |   apply_updates_locally(Updates);
7  |   StateDot :=
   |     tcbcast(Updates, NewVc, StateDeps, StateDot, Cons);
8  else
9  |   StateDot :=
   |     tcbcast(Updates, NewVc, StateDeps, StateDot, Cons);
10 |   wait_stability(Updates);
11 |   apply_updates_locally(Updates);
12 StateVc := NewVc;
13 NewClock:= NewVc;
14 unlock(Updates.Keys);
```

Broadcasting updates (Tcbcast): Tcbcast is a function of camus_middleware which is called by Minidote_server to broadcast updates to other nodes (Algorithm 3). When it is called, the middleware increments the dot by one to identify the message holding the new update (line 1). Then, the middleware adds a new vertex to the dependency graph for the new update (line 2). After that, it checks the consistency. If the latter is strong, the dot is added to the set of strong dots and the new vertex of the graph is labeled as 'received'. However, in the case of causal consistency, the new vertex is labeled as 'delivered' (lines 3-7). Finally, a message will be sent to the other nodes of the cluster. This message contains the updates, its identi cation (Dot), the dependencies, the consistency type, and the set of strong dots (lines 8 and 9).

Algorithm 3: Broadcasting updates(Tcbcast)

```
Input: Updates, Vc, Deps, Dot, Cons, Dpgraph
Output: NewDot, Dpgraph
1  NewDot:= increment_dot(Dot);
2  Add NewVertex to Dpgragh for NewDot;
3  if Consistency= 'strong' then
4  |   Add NewDot to StrongDots;
5  |   Mark NewVertex as 'received';
6  else
7  |   Mark NewVertex as 'delivered';
8  Msg := {Updates, NewDot, Deps, Cons, StrongDots}
9  broadcast_to_other_nodes(Msg);
```

Receiving Remote updates: This function will be called when the middleware receives a broadcasting message of remote updates from another node (Algorithm 4). Upon receiving a remote message, the receiver adds the corresponding dot to the set of strong dots if the consistency is strong. Then the function check_strong_dot is called to verify the status of strong dots (lines 2-4). After that, a new vertex in the dependency graph should be added, if it has not been added before, for the received dot (line 5). Then, the middleware creates a new vertex for each dot in the dependencies of the received dot if it has not been created before and in the latter case the vertex is labeled as 'missed' (lines 8-10). The received dot is inserted into the successorset of all the vertices of its dependencies (line 11). Finally, for a causal consistency dot, it will be delivered if all its Dependencies have been delivered by calling the function Deliver(lines 12 and 13).

Delivering messages(Deliver): Delivery means send the remote update from the middleware component to Minidote_server of the same node. This function has three steps (Algorithm 5):

label the corresponding vertex in the dependency graph as 'delivered' and send the updates to Minidote_server (lines 1 and 2).
Check the stability of the predecessorset dots by noticing their vertices in the dependency graph as arrived at the sender node. A dot becomes stable if it is noticed as arrived at all the nodes (lines 3-6).

Algorithm 4: Receiving Remote updates

Input: Msg, LSDots, Dpgraph
Output: LSDots, Dpgraph

```
1  {Updates, Dot, Deps, Cons, RSDots} := Msg;
2  if Consistency = 'strong' then
3      Add Dot to LocalStrongDots;
4  {LSDots, Dpgraph} :=
     check_strong_dot(LSDots, RSDots, Dpgraph);
5  Add NewEntry to Dpgragh for Dot, if it is not exist;
6  Mark Dot as 'received';
7  foreach Dotd in Deps do
8      if not exist a vertex of Dotd in Dpgraph then
9          Add NewVertex to Dpgragh for Dotd, if it is
             not exist;
10         Mark NewVertex as 'missed';
11     Add Dot to successors_set of Dpgraph(Dotd) ;
12 if all dots in Deps have been delivered and Cons =
     'causal' then
13     Deliver(Updates, Dot, Dpgraph);
```

Check the dots in the successors_set if they can be delivered (lines 7-9).

Algorithm 5: Deliver Msg

Input: Updates, Dot, Dpgraph
Output: Dpgraph

```
1  Mark Dpgraph(Dot) as 'delivered';
2  Send Updates to Minidote_server;
3  foreach Dotd in Predecessors_set of Dpgraph(Dot) do
4      Find Dotd in Dpgraph and notice it as arrived at
         SenderNode;
5      if Dotd is noticed as arrived at all the nodes of
         the cluster then
6          Dpgraph := Stable(Dotd, Dbgraph);
7  foreach Dotd in Successors_set of Dpgraph(Dot) do
8      if all Predecessors of Dotd have been delivered
         then
9          Deliver Dotd ;
```

Check strong dots: It is a new function that we have added to manage strong consistency operations (Algorithm 6). It is called when the middleware receives any message from the other nodes by exploring the piggybacked set of strong dots. So the local node can determine which dots among its broadcast dots have arrived at the sender node to notice them. If a dot has arrived at all the cluster nodes, the middleware labels its vertex in the Dpgraph as 'stable' and then notify Minidote_server to apply the corresponding updates locally. Finally, a commit message is broadcast to the other nodes. When a node receives the commit message, its middleware calls the functions Deliver and Stable for the corresponding dot.

Algorithm 6: Check strong dots

Input: LocalStrongDots, RemoteStrongDots, Dpgraph, SenderNode
Output: LocalStrongDots, Dpgraph

```
1  foreach Dotd in LocalStrongDots do
2      if Dotd exist in RemoteStrongDots then
3          Notice Dotd as arrived at SenderNode;
4          if Dotd has arrived at all the nodes then
5              Notify Minidote_server to Apply the
                 corresponding updates locally;
6              Dpgraph := stable(Dotd, Dbgraph);
7              Broadcast a commit message to the other
                 nodes for Dotd;
```

Stable dot: When a dot becomes 'stable', it should be removed from the predecessors_set in the overall graph and its vertex should be removed from the dependency graph as well (Algorithm 7).

Algorithm 7: Stable dot

Input: Dotd, Dpgraph
Output: Dpgraph

```
1  foreach Vertex of Dpgraph do
2      Remove Dotd from Predecessors_set if it is exist;
3  Remove Vertex of Dotd from Dpgraph;
```

B. Clarification example

Take a cluster of two nodes MinidoteACE as shown in Figure 3. We will show how MinidoteACE handles client reads and updates.

Let Ctr1 = {"g1", crdt_counter, "bucket"} be an object of MinidoteACE where his key is "g1", his type is a counter, and his bucket is "bucket".

Update_object: When Client 1 wants to update the object Ctr1, he sends the following instruction to MinidoteACE 1:

Vc1 = update_objects({Ctr1, increment, 3}, {0,0}, causal).

This instruction increments Ctr1 three times (Ctr1 = Ctr1 + 3) under causal consistency. {0,0} is a vector clock that means that this operation has no dependencies. If the instruction is executed in the first node, the resulting vector clock Vc1 will take the value {1,0}.

Read_object: To read this object, the Client 2 send the following query:

Figure 3. An example of cluster of two MinidoteACE nodes.

{Value1,Vc2} = read_objects([Ctr1],{0,0}).

He should then obtain {Value1,Vc2} = {3,{1,0}}. In fact, Client 2 can read the value that has been updated by Client 1 and replicated by the system. Moreover, the obtained vector clock indicates that Client 1 has performed one update.

C. Implementation

We have implemented MinidoteACE by working on Minidote-tcb: a branch of Minidote GitHub repository which is written in Erlang and uses Camus. We have used the branch with stability of Camus that contains the implementation of stability. The main modules that we have replaced are: Minidote API, Minidote_server, Camus and Camus_middleware. The source code of MinidoteACE is available in our GitHub repository[1]. We have made the necessary amendments in the other modules of Minidote and Camus to enable MinidoteACE to work conveniently.

6. Evaluation

We have used the benchmark Basho_bench which is developed within the project AntidoteDB[25], [47]; the latter is built on the original benchmark for Riak core. Basho_Bench[48] is a popular Erlang application that has a pluggable driver interface. Basho_Bench can be extended to serve as a benchmarking tool for data stores and generate performance graphs. The benchmarking tool is useful for repeatable and accurate performance. Basho_bench utilizes two particular indicators of performance: throughput and latency.

- Throughput: the number of operations performed over the de ned period of time, including all possible types of operations.
- Latency: the time between sending a query and the completion of the reply.

The experiments have been performed on a cluster of MinidoteACE nodes incorporated with a tra c generator node. A tra c generator runs one copy of Basho_Bench that generates and sends commands to MinidoteACE nodes which are identi ed by their IP addresses and port numbers. Figure 4 shows how trac generators and MinidoteACE nodes are organized in a cluster.

A. Experimental structure

Figure 5 illustrates our experimental setup. Hereafter, we describe the main components of the experimental architecture.

1) Tra c generator

The benchmark of AntidoteDB has two additional les, that make the original Basho_bench compatible with Antidote:

- The driver (basho_bench_driver_antidote_pb.erl): This le de nes the initialization of a benchmarking thread and how it executes transactions. We have adapted the driver le to work properly with our system. Therefore, it can generate three kinds of operations: read, causal update and strong update. Each generated operation is sent within a transaction of a single operation to the target MinidoteACE node through the protocol bu er interface.
- The con guration le (antidote_pb.con g): This le is given as a starting parameter to the Basho_bench, it contains the benchmark parameters that can be adjusted according to the test's purpose. The con guration le contains the following parameters:
  - Target: de nes the ip addresses and the ports of MinidoteACE nodes.
  - Driver: de nes the le name of the driver which makes the workloads that were con gured to be benchmarked for the speci ed targets.
  - Operations: This parameter con gures which operations the driver should run and the weight of each operation. We put the following operations in the benchmark: read, causal update, and strong update.
  - Duration: An integer that de nes the duration of the benchmark in minutes.
  - Threads: The number of parallel threads that will be run during one benchmark.

[1] https://github.com/nacer-git/MinidoteACE

Figure 4. A cluster of MinidoteACE nodes with a tra c generator.

Figure 5. Experimental structure.

Data type: de nes data types for generated data as well as operations to be applied for each de ned type in the benchmark.
Keys generators: takes the form fkey_distribution, Max_keyg which de nes the key generator distribution (Pareto or Uniform) in which the keys will be generated as well as the maximum value.
Values generators: takes the same form as keys generator.

2) Protocol Bu er Interface and data ow inside the system

We have adapted the Protocol Bu er of AntidoteDB to work conveniently with MinidoteACE. A worker process randomly selects one of the de ned operations in the con- guration le. Then, the selected operation is encapsulated in a transaction before encoding it using the protocol bu Each encoded command is sent to a target MinidoteACE node which is selected randomly too. Upon receiving a command, the Protocol Bu er API of MinidoteACE decodes the command and sends it to MinidoteACE API. The API then encodes the response of MinidoteACE and transfers it to the tra c generator. In the trac generator, there is a module that catches results and collects it in a result le.

B. Experiment Setup

The experiments were performed in an environment using a laptop with 8 GB DDR3 RAM, AMD Quad-Core Processor A8-7410 (up to 2.5 GHz), and 1 TB of Hard Disk Drive. Ubuntu 18.0.4 LTS (64-bit) was installed.

C. Performance con guration

Based on measuring performance and workload, Basho- bench is a benchmarking tool that performs reads and updates. After adapting the driver according to the input format of MinidoteACE, The operations that a driver might run

are in the format of (causal update,x,strong update,y,read,z) which means that in each generated (x+y+z) operations, Basho_bench will generate x causal update operations, y strong update operation and z read operations. When y=0, This means that the benchmark does not contain any strong operation. Therefore, MinidoteACE will act like a Minidote system which allows us to compare our consistency approach with the existing approach. To evaluate the performance of MinidoteACE system and compare it with Minidote, we launched the experiments illustrated in Table I on a cluster of 3 to 6 nodes during one minute. The grey rows contain experiments that evaluate our adaptive consistency approach (Experiments: 3, 4, 6, and 7). The white rows, however, will evaluate Minidote behavior (Experiments: 1, 2, and 5). For simplicity, we fixed the number of concurrent threads to 10, we chose the Pareto distribution for key generation and the Uniform distribution for value generation.

D. Results and discussion

In the following sections, we address the performance indicators and describe the results according to different experimental scenarios between reads, causal updates and strong updates.

1) Throughput results

The experimental results about throughput performance for each experiment are illustrated in Figure 6. As it is shown in the results, the throughput performance in the read only experiment is almost three times bigger than the other cases (2440 operations) due to the absence of update operations that have a higher latency which decreases the performance. For the other experiments, we notice that the performance decreases when we increase the proportion of update operations. Moreover, the throughput performance is inversely proportional to the number of nodes i.e, a cluster that has a small number of nodes has a higher performance. Despite strong updates having higher latency, the difference with causal updates in performance is not huge between workloads when executing them. For experiments 2 and 3, the performance will be reduced by 3.5% when replacing causal update operations by strong operations. However, it will be less than 10% when replacing two out of five causal update operations by strong operations (Experiments 5 and 7).

2) Read latency results

Figure 7 illustrates the 95th percentile read latencies that we have measured in each experiment. The results show that read operations have the smallest latency for the read only workload (about 12 milliseconds for whatever the number of nodes in the cluster). However, this latency increases when the proportion of updates or the number of nodes in the cluster increases. We note also that sticking strong updates to the workload gives a remarkable decrease of read latency (up to 30% for 10% updates between experiment 2 and 3). This happened because strong updates have higher latency so the workload should have lower throughput performance, and hence, lower latency for read operations.

Figure 6. Throughput performance.

Figure 7. 95th percentile read latency.

3) 10% update results

Figure 8 and figure 9 illustrate the latencies of causal and strong updates for experiments 2, 3 and 4 where update operations account for 10% of the workload. In these figures, the abbreviation 1C_9R_causal_upd represents the 95th percentile average latency of a causal update operation in experiment 2. Similarly, 1C_9R_strong_upd represents the 95th percentile average latency of a strong update operation in experiment 3. The same notation is used by 1C_1S_9R_causal_upd and 1C_1S_9R_strong_upd in experiment 4. The results show that replacing causal updates by strong updates increases update latency between 23% and 100% according to the number of nodes in the cluster for average latency results (experiments 2 and 3). However, for 95th latency results, latencies of the two types are almost the same. We notice also that when dividing the proportion of updates between the two types, strong update latency is greater almost four times for a cluster of three nodes and two times for a cluster of six nodes.

4) 25% update results

Figure 10 and figure 11 illustrate the latencies of causal and strong updates for experiments 5, 6 and 7 where update

TABLE I. Experiments details and abbreviations.

	Causal updates	Strong updates	Reads	Notation
Experiment 1	0	0	1	Read only
Experiment 2	1	0	9	1C_9R (10% updates)
Experiment 3	0	1	9	1S_9R (10% updates)
Experiment 4	1	1	18	1C_1S_18R (10% updates)
Experiment 5	5	0	15	5C15R (25% updates)
Experiment 6	4	1	15	4C_1S_15R (25% updates)
Experiment 7	3	2	15	3C_2S_15R (25% updates)

Figure 8. 10% updates: 95th percentile update latency.

Figure 9. 10% updates: Average update latency.

operations account for 25% of the workload. These gures use the same notation used in gures 8 and 9. Mean latency results show that when replacing one out of ve causal updates by a strong update, its latency becomes three to two times greater according to the number of nodes in the cluster. Moreover, strong update latency becomes ve to three times greater when replacing two out of ve causal updates by strong operations. However, for 95th percentile results, strong operation latency is almost double of causal operation's latency when replacing one or two out of ve causal operations for three or four nodes in the cluster. But when the number of nodes increases the gap between the two cases widens and becomes 70 milliseconds of difference for six nodes.

Figure 10. 25% updates: 95th percentile update latency.

Figure 11. 25% updates: Average update latency.

E. Limitations of the proposed approach

Although MinidoteACE allows to overcome causal consistency issues, It suers from several limitations that we

highlight in the following points:

The strong consistency level introduced in MinidoteACE enables it to execute operations under stronger consistency guarantees. However, strong consistency decreases the system performance (it increases Latency and decreases throughput). Hence. Clients should find a suitable trade-off between consistency and performance.

Evaluations show that the latency of strong updates becomes quite high when the rate of update operations exceeds 25% of overall operations.

"Strong updates" risque to be aborted due to a possibly long waiting time. The protocol used to enforce "total" requires stability information to be collected from all nodes. If this information is not collected after some timeout an "error" state is reached.

7. Conclusion and Future work

In this paper, we have presented MinidoteACE, a new adaptive consistency approach in the edge computing environment. Our model enables applications to run queries with causal or strong consistency. To achieve this aim, we have ameliorated the Minidote system by adding the ability to handle strong consistency operations. The new consistency level is stronger than causal consistency, but it does not emulate the typical strong consistency since it only checks that the update has arrived at the nodes of the cluster. We have experimentally evaluated MinidoteACE using Basho_bench: a benchmarking tool that has been modified to evaluate AntidoteDB. To do that, We performed the necessary amendment on the Basho benchmark to fit MinidoteACE. Our evaluation proves that MinidoteACE can support certain proportions of strong operations without significantly affecting latency or throughput.

To the best of our knowledge, MinidoteACE is the only causally consistent system that provides more than consistency level in edge computing environment. In future work, we aim to improve the performance of MinidoteACE by enabling it to support a higher rate of strong consistency updates. Moreover, we aim to avoid waiting for a long timeout by investigating the possibility of implementing "strong" operations while collecting only a majority of confirmations. which gives quorum consistency: another consistency level between causal and strong.

Acknowledgement

This paper highlights the work that has been carried out while the first author was visiting High Assurance Software Laboratory (HasLab). HasLab is one of the Research and development centers of INESC TEC, a leading national associate laboratory, that is headquartered at the University of Minho, Braga, Portugal. The internship has been funded by the Algerian MESRS through its PNE program. This work is also partially funded by the Algerian MESRS PRFU project No. C00L07UN390120210002.

References

[1] W. Vogels, "Eventually consistent," Communications of the ACM, vol. 52, no. 1, pp. 40–44, 2009.

[2] S. P. Kumar, "Adaptive consistency protocols for replicated data in modern storage systems with a high degree of elasticity," Ph.D. dissertation, Conservatoire national des arts et metiers-CNAM, 2016.

[3] A. Khelaifa, S. Benharzallah, L. Kahloul, R. Euler, A. Laouid, and A. Bounceur, "A comparative analysis of adaptive consistency approaches in cloud storage," Journal of Parallel and Distributed Computing, vol. 129, pp. 36–49, 2019.

[4] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 401–416.

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in Symposium on Self-Stabilizing Systems. Springer, 2011, pp. 386–400.

[6] Antidote: the highly-available geo-replicated database with strongest guarantees, https://pages.lip6.fr/syncfree/index.php/2-uncategorised/62-antidote.html, June 2021.

[7] D. D. Akkoorath and A. Bieniusa, "Antidote: the highly-available geo-replicated database with strongest guarantees," Tech. U. Kaiserslautern. https://syncfree.lip6.fr . . . , Tech. Rep., 2016.

[8] Minidote Github repository, https://github.com/LightKone/Minidote, April 2021.

[9] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in Proceedings of the 4th annual Symposium on Cloud Computing 2013, pp. 1–14.

[10] S. Almeida, J. Leitão, and L. Rodrigues, "Chainreaction: a causal+ consistent datastore based on chain replication," in Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 85–98.

[11] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in Proceedings of the ACM Symposium on Cloud Computing, 2014, pp. 1–13.

[12] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, "Write fast, read in the past: Causal consistency for client-side applications," in Proceedings of the 16th Annual Middleware Conference, 2015, pp. 75–87.

[13] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa, "Legion: Enriching internet services with peer-to-peer interactions," in Proceedings of the 26th International Conference on World Wide Web, 2017, pp. 283–292.

[14] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2016, pp. 405–414.

[15] K. Spirovska, D. Didona, and W. Zwaenepoel, "Wren: Nonblocking reads in a partitioned transactional causally consistent data store,"

in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 1–12.

[16] D. Didona, K. Spirovska, and W. Zwaenepoel, "Okapi: Causally consistent geo-replication made faster, cheaper and more available," *CoRR*, vol. abs/1702.04263, pp. 1–12, 2017.

[17] C. Gunawardhana, M. Bravo, and L. Rodrigues, "Unobtrusive deferred update stabilization for efficient geo-replication," in *2017 fUSENIXg Annual Technical Conference (fUSENIXgfATCg 17)*, 2017, pp. 83–95.

[18] K. Spirovska, D. Didona, and W. Zwaenepoel, "Optimistic causal consistency for geo-replicated key-value stores," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2626–2629.

[19] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A distributed metadata service for causal consistency," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 111–126.

[20] Z. Xiang and N. H. Vaidya, "Global stabilization for causally consistent partial replication," in *Proceedings of the 21st International Conference on Distributed Computing and Networking*, 2020, pp. 1–10.

[21] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *14th fUSENIXg Symposium on Networked Systems Design and Implementation (fNSDIg 17)*, 2017, pp. 453–468.

[22] T.-Y. Hsu, A. D. Kshemkalyani, and M. Shen, "Causal consistency algorithms for partially replicated and fully replicated systems," *Future Generation Computer Systems*, vol. 86, pp. 1118–1133, 2018.

[23] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Trade-offs in replicated systems," *IEEE Data Engineering Bulletin*, vol. 39, no. ARTICLE, pp. 14–26, 2016.

[24] C. Baquero, P. S. Almeida, and A. Shoker, "Making operation-based crdts operation-based," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2014, pp. 126–140.

[25] AntidoteDB Basho Bench, https://antidotedb.gitbook.io/documentation/benchmarking/basho_bench, June 2021.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[27] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 143–157.

[28] Apache HBase, http://hbase.apache.org/, February 2021.

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[30] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[31] MongoDB, http://www.mongodb.org/, February 2021.

[32] F. Nawab, D. Agrawal, and A. El Abbadi, "Dpaxos: Managing data closer to users for low-latency and mobile applications," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1221–1236.

[33] Z. Hao, S. Yi, and Q. Li, "Edgecons: Achieving efficient consensus in edge computing networks," in *fUSENIXg Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[34] H. Gupta and U. Ramachandran, "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM, 2018, pp. 148–159.

[35] S. H. Mortazavi, B. Balasubramanian, E. de Lara, and S. P. Narayanan, "Pathstore, a data storage layer for the edge," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018, pp. 519–519.

[36] N. Afonso, M. Bravo, and L. Rodrigues, "Combining high throughput and low migration latency for consistent data storage on the edge," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020, pp. 1–11.

[37] 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.

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

[39] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

[40] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer, "Bayou: replicated database services for world-wide applications," in *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, 1996, pp. 275–280.

[41] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. IEEE, 1994, pp. 140–149.

[42] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Presented as part of the 10th fUSENIXg Symposium on Networked Systems Design and Implementation (fNSDIg 13)*, 2013, pp. 313–328.

[43] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[44] Camus Github repository, https://github.com/lightkone/camus, April 2021.