ENGAGE: Session Guaranties for the Edge

(extended abstract of the MSc dissertation)

Miguel Leitão Belém
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—One of the main goals of edge computing is to support latency-constrained applications. For applications that need to access information stored in the edge, this can be achieved by storing data replicas on edge nodes. Because edge nodes are resource constrained, full replication is infeasible. Therefore, any edge storage service needs to support partial replication. Also, in most cases, edge storage needs to support weak consistency, to avoid the latency and overhead associated with algorithms that enforce strong consistency. In this context, session guarantees are a powerful tool that simplify the design of edge applications. Unfortunately, the mechanisms typically used to enforce session guarantees, such as vector clocks, perform poorly under partial replication. This paper presents ENGAGE, a storage system that combines the use of vector clocks and distributed metadata propagation services to offer efficient support for session guarantees in a partially replicated edge storage.**

***Index Terms*—edge, session guarantees, causal, consistency, low latency**

I. INTRODUCTION

Numerous applications of today that have clients running on the edge of the network rely on cloud structures for computation offloading and storage [1]. Unfortunately, the high network latency between clients and data centers can impair novel, latency-constrained, applications such as augmented reality [2]. Edge computing has emerged as a potential solution to circumvent this problem. To unleash its full potential, the edge nodes must replicate data that is frequently used. However, because edge nodes are resource constrained, full replication is infeasible. Therefore, any edge storage service needs to support partial replication. Furthermore, there is also evidence that edge storage should support weakly consistent memory models [3, 4]. This happens because strong consistency models, such as linearizability [5], require strong coordination among the replicas when updates are performed, which increases latency.

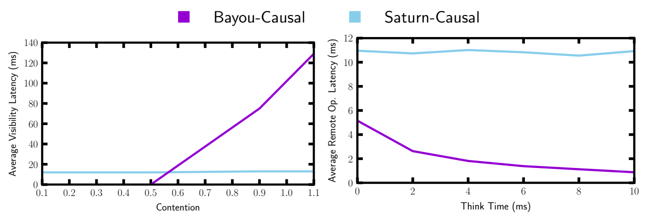
In a seminal work, Terry *et al.* [6] have introduced the notion of *session guarantees*, a set of well defined semantics that may be used to simplify the design of distributed application using weakly consistent stores. Session guarantees are relevant in scenarios where a client may access different replicas of a weakly replicated system. In particular, if a client, after performing a number of read and/or write operations on a given (origin) replica, needs to access another (destination) replica, it may observe a state that is inconsistent with its causal past: updates that the client has performed or observed on the origin replica may have not been applied yet at the destination replica. Depending on the semantics of the application, the client may be forced to wait for some (or all) of these operations to be applied at the destination replica before being served, to ensure correctness of the results.

In [6], the authors have also suggested a set of mechanisms to enforce the session guarantees that rely on the use of version vectors [7]–[9], a form of vector clocks [10]. However, these mechanisms are only efficient in settings that use full replication, i.e., all updates are propagated to all replicas. In systems that implement partial replication, one may be required to maintain and exchange large amounts of metadata (for instance, by forcing all messages to carry many vectors clocks, one vector clock for each shard in the system) or may cause update propagation to stall (later in the paper we elaborate on this phenomenon).

This impairs the *remote visibility latency*, i.e., the time it takes for an update performed in one replica to become visible in remote replicas. There are many edge applications where small remote update visibility is highly desirable (for instance, in vehicular applications, events such as accidents should be propagated to other roadside units, to divert traffic from the hazard). Thus, this limitation of vector clocks is of significant concern for edge applications.

The challenges of providing small remote visibility latency with small metadata have been recognized in the literature [11]–[13]. To address these challenges, the abstraction of a distributed metadata service has been recently introduced [11]. A metadata service is a helper service that instructs replicas regarding the order by which they should apply remote updates without violating a given consistency criteria. However, to the best of our knowledge, existing metadata services such as Saturn [11] only offer causal consistency and have no support for session guarantees. Therefore, they may force clients that have weaker requirements to suffer unnecessary delays when performing remote reads.

The dichotomy above is illustrated in Figures 1a and 1b that show, respectively, the remote visibility latency and the remote read latency in two separate systems, one using vector clocks and the other using a metadata service (in this case,



(a) Remote Visibility Latency (b) Remote Operation Latency

Fig. 1: Remote Visibility Latency and Remote Read Latency.

Saturn [11]). We postpone to Section IV a detailed description of the experimental setup used to collect the data, and present here just the required information to understand the results. We consider a partially replicated system and clients that have a preferential replica (typically the nearest one). By default, clients perform reads and writes on its preferred replica unless they need to access an object that is not replicated there; in this case they perform a remote operation. When a client is forced to contact another replica, it must wait until the replica is consistent with its past; in this figure we just assume causal consistency. This waiting time can contribute substantially to delay the remote operation.

Figure 1a shows the remote visibility latency, i.e., the time it takes for a local update to be applied remotely, for both classes of systems. Updates are applied in causal order, such that clients are required to keep a single vector clock, and do not need to maintain a separate vector clock for each object. In this figure, in the x axis we vary the diversity in access frequency to different objects. For small values all objects are accessed at the same pace, for larger ones, some objects are accessed much more frequently than others. Systems based on vector clocks need to receive updates from all nodes before applying remote updates (to determine the correct order), and the update visibility latency increases sharply when the access frequency is skewed (because some updates are much less frequent than others). Metadata services were invented to circumvent this problem and, in fact, as it can be seen, these offer small visibility latency regardless of access skews.

Figure 1b shows the delays experienced by clients when they perform a remote operation. In this case, in the x axis we vary the think time of the clients, i.e., the average time between two consecutive operations. The larger the think time, the more likely it is that updates in the causal past of the client have already been propagated and applied when the client performs a remote operation. Thus, we expect the remote operation latency to decrease as the think time increases. Systems based on vector clocks have fine grained information about which updates the client has observed and that need to be locally applied in order to avoid violating the client consistency requirements. By using this information, they can reply faster. Unfortunately, systems based solely on a metadata service do not keep detailed information regarding the causal past of each client; they have to conservatively wait for all updates that *may have been* observed by the client to be applied. Thus, these systems are unable to leverage the think time of the client and are penalized by depicting a (constant) high remote operation latency (the horizontal blue line at the top of the figure).

The goal of this paper is to derive a strategy that can achieve the best of both worlds, i.e., to combine small remote visibility latency and efficient support for clients performing remote operations using different session guarantees. We present ENGAGE, a storage system that achieves this goal by combining, in a synergistic manner, the use of vector clocks and distributed metadata propagation services to offer efficient support for session guarantees in partially replicated edge storage. We provide an extensive evaluation of ENGAGE against a system based on vector clocks and against a system based on Saturn [11], using different combinations of the session guarantees proposed in [6].

II. Background and Related Work

Weakly consistent replication schemes have been introduced as a way to circumvent the performance bottlenecks associated with strong consistency and to augment the system availability [14]. In strongly consistent systems all updates need to be serialized [15]; this requires the use of a single primary replica or the use of a consensus protocol [16]. Weakly consistent systems allow updates to be performed concurrently and without coordination at different replicas. Also, while strongly consistent systems may block, weak consistency offers higher availability as requests can be served locally by any replica.

Session Guarantees Unfortunately, without any additional support, weakly consistent systems allow applications to observe inconsistent states. For instance, a client may perform an update at a given replica and later, be forced to contact another replica and observe a state where its update its missing, or to observe states that do not respect causality. Experience has shown that weak consistency makes application development difficult [17, 18]. Session guarantees have been introduced in [6] as way to simplify the application development in weakly consistent replicated systems. This seminal paper identifies four relevant properties for a client accessing a weakly consistent datastore, namely, *Read Your Writes* (RYW), *Monotonic Reads* (MR), *Writes Follow Reads* (WFR), and *Monotonic Writes* (MR). These properties can be matched to the application semantics and define a framework where the programmer can specify which properties should be ensured for each individual operation, such that the system maximizes the availability while still preserving high-level consistency. It is interesting to note that, when all these guarantees are combined, the system offers *causal consistency* [19].

Vector Clocks Version vectors [7]–[9], also known as vector clocks [10], are a way to keep track of concurrent updates. Each replica keeps a sequence number that it uses to identify updates performed locally. The vector clock keeps one entry for each replica, with the value of the last update that was received from that replica. For instance, consider a system with three replicas. Consider an object stored in some replica with vector clock $[0, 2, 1]$: this vector clock captures the fact that the state of the object includes 2 updates performed at replica R_1 and 1 update performed at replica R_2. Vector clocks can be used to enforce session guarantees [6]. For that purpose each c client keeps a vector clock with the most recent version of the object it has observed (V_c^R), and a vector clock that captures all write operations it has performed (V_c^W). The values of these clocks can then be checked against the version stored by a given replica, to check if it is safe to serve the request without violation the desired semantics.

Limitations of Vector Clocks Vector clocks are able to keep track of this partial order accurately, for a *single* object. To keep track of all causal dependencies accurately, it would be necessary to store and exchange the vectors clock for all objects in all messages [20] or, alternatively, a matrix clock [21]. Both approaches are extremely expensive and impractical on the edge. A common strategy to limit the size of metadata is to use a single vector clock for the *entire* object store. This creates what is known as *false dependencies*, i.e., scenarios when the operation of a client may be stalled because of independent operation performed by other clients on unrelated objects. The use of a single vector clock for the entire data store also performs poorly with partial replication. Assume that the read set of a client is captured by clock $V^R = [1, 0, 0]$ and that this client attempts to read some object from replica R_2 whose clock is still $[0, 0, 0]$. The clocks indicate that the client has observed some update that has not been applied to R_2 yet. Unfortunately, there is no way for R_2 to infer if the missing update corresponds to some object that is replicated locally (and should be received) or to some object that is replicated somewhere else (and will never be received).

Metadata Services Distributed metadata services [11] emerged as a solution to provide small visibility latency in partial replicated system while keeping the size of metadata very small. When an update is generated at a given replica, this information is propagated to the metadata server. The metadata server will later tell the relevant replicas when it is safe to apply the update. Metadata services have proven to be an interesting mechanism to provide short visibility latency but enforce only causal consistency. It is unclear if and how these services can be extended to support weaker consistency models such as session guarantees.

Edge Storage Several works have addressed edge storage, but few have addressed the problem of latency when considering session guarantees. SessionStore [22] is a data store for edge applications that also supports session guarantees. However, instead of optimizing for latency, SessionStore uses the semantics to reduce the amount of data that needs to be shipped before serving a client. SessionStore is based on PathStore [23], which is a hierarchical eventual-consistent object store built on CloudPath [3], a system that replicates application data on-demand. Because data is shipped on demand, clients can experience a large latency. Like us, FogStore [4] and DataFog [24] also aim at offering low latency with different semantics. However, in FogStore and DataFog, the semantics drive how many replicas need to be read/written for executing a given operation, while we always serve requests locally (furthermore, the consistency criteria supported by FogStore are not directly comparable with session guarantees). EdgeCons [25] and DPaxos [26] propose efficient consensus algorithms for the edge that are targeted at strongly consistent systems. In [27], the use of CRDTs [28] is suggested to avoid the cost of strong consistency; however, the paper does not address the problem of offering consistency to the clients, when they access different edge servers. Timeseries DBs [29] focuses on establishing semantic specifications to handle fault detection and providing diagnosis in IoT-based monitoring systems for critical systems. Differently from our proposal, Timeseries DBs is not focused on latency optimization.

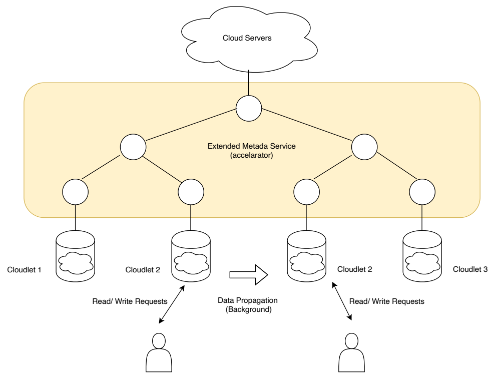


Fig. 2: ENGAGE Architecture

III. THE ENGAGE SYSTEM

ENGAGE is a system that aims at combining low visibility latency *and* support for session guarantees while avoiding the costs of using matrix clock. It does so by combining, in a synergistic manner, the use of vector clocks to keep track of the read set and write set of clients, and the use of metadata services to speed up the propagation of updates.

A. System Model

Figure 2 depicts the architecture of ENGAGE. We consider a set of edge servers, fog nodes or cloudlets, that are used to replicate data. We assume the number of cloudlets to be in the order of a few dozens to one hundred. The system uses partial replication, i.e., not every cloudlet replicates every object. In this paper we do not address data placement: the decision of which cloudlets store each data object is orthogonal to our work; we just assume that some data placement policy is in place and that the assignment of data object to cloudlets is known, at least by all cloudlets. Typically, data replicated in the cloudlets will also be stored in a cloud datacenter, but this is not necessary for the operation of ENGAGE. Cloudlets are connected by a backbone network that is used to propagate updates among replicas.

ENGAGE supports two types of clients, namely *placement-aware* and *placement-unaware* clients. Placement-aware clients know the location of the objects, and send requests directly to the nearest replica when performing an operation. Placement-unaware clients do not know the object locations. Therefore, these clients have a preferred cloudlet, to which they forward all the requests. If the preferred cloudlet does not replicate the target of an operation, in turn, the cloudlet forwards the request to the nearest replica that does. Typically, the preferred cloudlet is selected based on the network latency from the client to the cloudlet. If clients are mobile, they may change their preferred cloudlet on-the-fly.

B. Metadata

We assume that each cloudlet is linearizable [5], which means that all updates performed at a cloudlet can be serialized and when one update becomes visible for a client, it becomes visible for all clients of that cloudlet. Thus, each cloudlet keeps a unique sequence number that is used to uniquely identify updates that are performed locally on behalf of clients; this sequence number is shared by all objects. For instance, if client c_1 makes an update on object o_1 and this update is assigned sequence number x, the next update on that cloudlet will be assigned sequence number $x + 1$, even if it is performed by some other client c_2 on some other object o_2.

ENGAGE uses vector clocks to keep track of the updates that are observed by clients. Vector clocks have one entry per cloudlet. Multiple vector clocks are maintained by ENGAGE as follows:

- A vector clock V_o^i is stored with each replica o^i of each object o. The vector clock captures the causal past of all updates that have been applied to replica o^i.

- Each cloudlet i also keeps a *cloudlet vector clock* V_*^i that captures the state of the local database. This clock is computed by taking the maximum value of the clock values of all objects replicated in cloudlet i, i.e., $V_*^i = max(V_o^i)\ \forall o \in i$.

- Finally, each c client keeps two vector clocks: V_c^R, that captures the past of all objects the client has read, and V_c^W, that captures all write operations it has performed.

C. Performing Read and Write Operations

When a client performs a read or a write operation it can specify one or more session guarantees to be ensured. ENGAGE supports the session guarantees of the original Bayou paper [6], namely: *Read Your Writes* (RYW), *Monotonic Reads* (MR), *Writes Follow Reads* (WFR), and *Monotonic Writes* (MR). From the point of view of the client operation, ENGAGE offers no novel contribution. Instead, we are faithful to the original implementation proposed in [6].

On the server side we perform a number of adaptations to the original algorithm, in order to support multiple objects that keep different clock values. When performing a read or write operation on object o using cloudlet C, the client c provides its own V_c^R, V_c^W, and the desired session guarantees. The cloudlet i holds the request until it is *safe* to execute. In order to check if the cloudlet is in a state that is consistent with the guarantees specified by the client, the cloudlet compares the value of its own vector clock V_*^i with the values of V_c^R and V_c^W as follows:

- If the client requests WFR or MR, it is safe to execute the operation if $V_*^i \geq V_c^R$.
- If the client requests MW or RYW, it is safe to execute the operation if $V_*^i \geq V_c^W$.

If the operation is a read, the cloudlet sets $V_c^R = \text{MAX}(V_c^R, V_o^i)$, and returns the state of the object and the new value of V_c^R to the client. If the operation is a write, the cloudlet assigns a unique sequence number *snb* to the update, by incrementing the local counter that serializes all updates. It then creates a temporary *update vector clock* V^{up} that has all entries to 0 except the entry i associated with cloudlet that is set to *snb*. Then it updates several clocks as follows:

- It sets $V_*^i = \text{MAX}(V_*^i, V^{up})$.
- It sets $V_o^i = \text{MAX}(V_o^i, V^{up}, V_c^R, V_c^W)$.
- It sets $V_c^W = \text{MAX}(V_c^W, V^{up})$.

After these updates, it returns the new value of V_c^W to the client. In parallel, it schedules the update to be sent, tagged with V_o^i, to the other cloudlets that replicate o. The update can be shipped immediately, or in background using epidemic dissemination.

D. Applying Remote Updates

When an update performed at cloudlet *orig*, tagged with vector clock V_o^{orig}, is received at some other cloudlet *dest*, it is applied in causal order with respect to all other remote updates. There are two complementary mechanisms that can be used to decide when an update can be applied, namely, using *vector clock stability* or using the ENGAGE *extended metadata service*. The update is applied as soon as one of these mechanisms indicates that the update is safe (whichever triggers first). We will describe the ENGAGE metadata service in the following sections. Here we will describe how updates can be applied based on vector clock stability.

The remote update is put in a list of pending updates and it remains there until the following conditions are met:

- From all updates received from *orig*, the update has the lowest sequence number, *and*
- For all other entries $i \neq orig$, we have $V_*^{dest}[i] \geq V_o^{orig}[i]$.

When these conditions are met, the update is applied to the object and cloudlet *dest* performs the following updates to its own metadata:

- It sets $V_*^{dest} = \text{MAX}(V_*^{dest}, V_o^{orig})$.
- It sets $V_o^{dest} = \text{MAX}(V_o^{dest}, V_o^{orig})$.

E. The ENGAGE Extended Metadata Service

Using vector clock stability to apply remote updates is not effective under partial replication. We recall the example from Section II to illustrate the problem. Assume that cloudlet 2 receives a remote update u for object o from cloudlet 1 with vector clock $V_o^u = [1, 1, 0]$. Assume that cloudlet 2 is still in the initial state, and its cloudlet vector clock has value $V_*^2 = [0, 0, 0]$. According to the rules stated in the previous section, the update cannot be applied safely on cloudlet 2 because $V_o^u[0] > V_*^2[0]$. In fact, the update has in its causal past some previous update u' generated by cloudlet 0 with sequence number 1. Under full replication, cloudlet 2 would eventually deliver u' which, in turn, would allow to deliver u. Unfortunately, under partial replication, cloudlet 2 may never receive u'. Furthermore, cloudlet 2 has no way to know if it is supposed to receive u' or not.

To solve the problem above, nodes can periodically send to each other the values of their cloudlet vector clocks (in this paper we call these messages *metadata flush* (MF) messages). This generates additional traffic and makes the remote update latency a function of the period used to exchange MF messages. Note that, under partial replication, MF messages are necessary not only to apply remote updates but also to serve remote reads. This happens because, to enforce session guarantees, cloudlets need to compare the client read/write with their own cloudlet vector clock; therefore they need to keep the values of V_* up-to-date by receiving MF messages.

A key insight behind the design of ENGAGE is that a metadata service, such as the one proposed in [11], can be extended to perform a dual function: it can be used to instruct cloudlets to deliver remote updates (as proposed in [11]) and, *with minimal additional overhead,* it can also be used to propagate MF messages, such that cloudlets can keep vector clocks up-to-date, regardless of the objects they replicate.

Thus we propose to connect all cloudlets by a distributed metadata service, inspired by Saturn [11]. The medatada service is implemented by a set of servers that are distributed in different locations of the backbone network that interconnects the cloudlets. The servers are organized as an acyclic graph and each cloudlet is connected to one of these servers. Unlike Saturn, that only propagates update *labels* (a label is a scalar that uniquely identifies an update), ENGAGE's metadata service propagates two types of control messages that carry a vector clock: *update notifications* and *metadata flush* messages.

Update notifications are tuples associated with a concrete update. They include the following fields $\langle \text{UN}, \mathit{src}, \mathit{snb}, \mathit{oid}, V_{oid}^{src} \rangle$, where *src* is the identifier of the cloudlet where the update was originated, *snb* is the sequence number assigned by *src* to the update, *oid* is the identifier of the object that has been updated and, finally, V_{oid}^{src} is the vector clock assigned to the update by the *src* cloudlet. Metadata flush messages are tuples that include the following fields $\langle \text{MF}, V_{\text{MF}} \rangle$ where V_{MF} is a vector clock that will be used to update the cloudlet vector clocks.

When a cloudlet processes a write request, and a new update u is created as explained in Section III-C, the cloudlet also creates an update notification message that it delivers to the local metadata server. When a metadata server receives an update notification, it performs the following sequence of actions for all edges e (except for the incoming edge):

- If the edge e is in the path from *src* cloudlet to another cloudlet that replicates *oid*, it forwards the update notification eagerly on that edge. If there is a MF message pending on that edge, the MF message is also forwarded piggybacked with the update notification and any timeout associated with the MF message is cancelled.
- Otherwise, it transforms the update notifications into a MF message, by preserving the associated vector clock. The resulting MF message is then scheduled to be propagated asynchronously on that edge. If there is already another MF message scheduled for transmission on the same edge, both MF messages are merged on a single MF message, with a vector clock that has the max of both clocks. If there was no other MF message already pending on edge e, the metadata server starts a timeout timer to propagate the MF message later.
- When the timeout associated with an edge expires, the metadata server forwards the pending MF message.

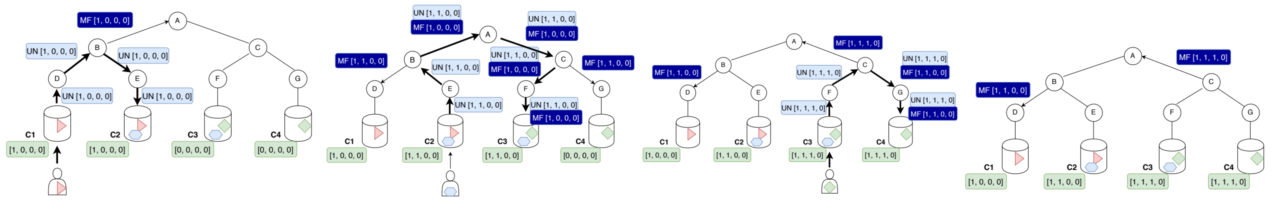
When a MF message $\langle \text{MF}, V_{\text{MF}} \rangle$ is received by a cloudlet *dest*, either isolated or piggybacked with some update notification message, the cloudlet *dest* uses V_{MF} to update $V_*^{dest} = \text{MAX}(V_*^{dest}, V_{\text{MF}})$. Finally, when an update message $\langle \text{UN}, \mathit{src}, \mathit{snb}, \mathit{oid}, V_{oid}^{src} \rangle$ is received by a cloudlet, it performs the following checks:

- If $V_*^{dest} \geq V_{oid}^{src}$, then the update has already been received and delivered via the vector clock stability described in Section III-D. The update message can be safely discarded.
- Otherwise, the cloudlet waits until it has received the payload of the update directly from *src*.
- When the cloudlet *dest* has received the update message from the metadata service *and* the payload directly from *src*, it applies the update to object *oid* and updates vector clocks V_*^{dest} and V_o^{dest} as described in Section III-D.

F. Example

We now illustrate the propagation of *update notifications* and *metadata flush* messages in the network of metadata servers with the help of Figure 3. The figure shows a network with 4 cloudlets. Geometric figures in the cloudlets represent partially replicated data objects: for instance, the pink triangle is replicated in cloudlet c_1 and c_2 only. The ENGAGE extended metadata service is implemented by a network of 7 servers $(A, B, \ldots, G)$ organized in a tree rooted at server A. The figure illustrates a sequence of events where a client first makes an update on an object replicated in c_1 and c_2 (Step 1, Figure 3a), then another client makes an update on an object replicated in c_2 and c_3 (Step 2, Figure 3b), finally, another client makes an update on an object replicated in c_3 and c_4 (Step 3, Figure 3c). In the figure, the green boxes at the bottom represent the values of the *cloudlet vector clock* and the blue boxes represent metadata messages; update notifications are represented in light blue, tagged as UN, and metadata flush messages are represented in dark blue, tagged as MF.

In Step 1, the client makes an update on c_1; this creates an update notification that is propagated in the network of metadata servers via the path $c_1 \rightarrow D \rightarrow B \rightarrow E \rightarrow c_2$. Since c_3 and c_4 do not replicate the object that has been updated, the notification is not propagated on the link $B \rightarrow A$. Instead,



(a) Step1 (b) Step2 (c) Step3 (d) Pending MF messages after Step3

Fig. 3: Propagating metadata messages.

the update is transformed in a metadata flush message that is scheduled for future transmission.

In Step 2, the client makes an update on c_2; this creates an update notification that is propagated in the network of metadata servers via the path from c_2 to c_3. Because the object written is not replicated in cloudlet c_1, the notification is not propagated on the link from B to D; instead it is transformed in a metadata flush message that is scheduled for future transmission. When the notification is propagated over the link from A to C, it "piggybacks" the MF message that was pending from Step 1: that MF message is sent together with the notification, in a single message, on the path to c_3. Because $C4$ does not replicate the object, the notification is not propagated on the link $C \rightarrow G$; instead it is transformed in a metadata flush message that is merged with the MF message from Step 1.

In Step 3, another client creates an update that generates a notification that is propagated in path $c_3 \rightarrow F \rightarrow C \rightarrow G \rightarrow c_4$. This notification flushes the pending MF message waiting on link $C \rightarrow G$ from Step 2. At the end of this step there are still two MF messages pending on the metadata broker network: the MF message generated on Step 2 on the link $B \rightarrow D$ and the MF message generated on Step 3 on the link $C \rightarrow A$. These will be piggybacked on future update notification messages or propagated alone, after some timeout.

G. *Optimization*

A client is said to be *sticky* if it performs all the operations on the same set of servers (typically, the nearest to the cloudlet they are attached to). It has been shown that availability under consistency criteria such as causal consistency or RYW cannot be guaranteed unless clients are sticky [30]. Therefore, although we support client mobility, in ENGAGE clients remain sticky while stationary. The knowledge of clients being sticky combined with remote updates applied by causal order, allows for implementing RYW and MR guarantees without blocking operations, which reduces the overall latency of the system.

When using RYW, the client executes operations on the same set of servers, so the clients' previous write operations are always reflected on the cloudlet as the servers are linearizable. When using MR, updates are applied by causal order combined with the client being sticky. The client will always read a version of the key that is greater or equal than the previous version that the client has read. To achieve non-blocking operations, the client zeros its vector clock before sending the RYW or MR operation. Thus, the client vector clock will be lower or equal than the cloudlets vector clock.

TABLE I: Parameters of the dynamic workload generator.

Parameter	Default	Range
Write %	10%	5%-50%
Access Locality	10%	-
Zipfian Constant / Contention	0.8	0.1-1.1
Think Time (ms)	0	0 - 10
I'm Alive Timeout (ms)	25	5 - 100

Note that supporting sticky clients does not interfere with supporting mobile clients. If the client changes its set of servers due to change of location, for the first RYW or MR operation, it needs to send its complete vector clock. For the following operations, the client zeros its vector clock before issuing RYW or MR operation.

IV. EVALUATION

In the evaluation, we address the following research questions:

- How does ENGAGE perform in comparison with the classical vector clock approach used in Bayou [6] and with recent metadata services, such as Saturn [11]?
- Can ENGAGE bring advantages to clients that exploit session guarantees to reduce the latency experienced by clients when accessing the data?
- Can ENGAGE help in tolerating transient network partitions?
- What is the signaling cost of ENGAGE?

For this purpose we have run a performance evaluation of ENGAGE against Bayou, a system based solely on vector clocks, and against Saturn, a system based solely on a metadata service.

A. *Experimental Setup*

The evaluation has been performed using a version of the Peersim network simulator [31] running in the event-based mode to capture the asynchrony of the interactions, configured with extensions that simulate network latency and finite bandwidth. The channels between two points ensure FIFO order with a bandwidth limit of 1 Gb/s.

We have considered a scenario where cloudlets are deployed in a grid network, which abstracts a urban deployment.

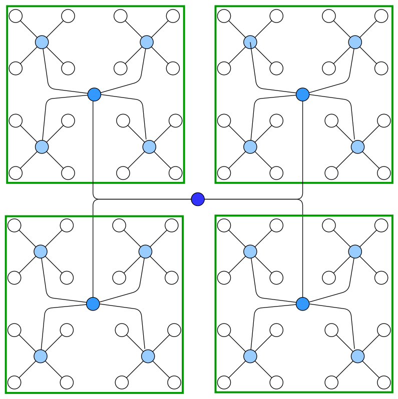


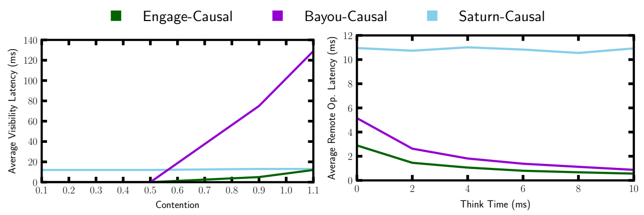
Fig. 4: Cloudlet placement (white) and broker network (blue)

We consider an hierarchical broker network consisting of a quadtree where the root is placed in the center of the area. Figure 4 illustrates the cloudlet deployment and the broker network for the case of a 8×8 grid.

We populate cloudlets with a set of objects that are partially replicated. Every object is replicated in exactly one node per bucket and each bucket has size of $k = 16$. The buckets are represented as green squares in Figure 4.

When a client issues an operation, it chooses with a certain probability whether it corresponds to a read or write operation (Write %) and if the operation is remote or local (Access Locality). The keys accessed by each request are selected using a Zipfian distribution. If the operation is local, using the Zipfian distribution selects it from a local object list, and if the operation is remote, it selects from a remote object list. The Zipfian Constant affects the contention of the workload, higher the Zipfian Constant higher the contention. After executing an operation the client has a cooldown time before issuing another operation (*ThinkTime*), if the *ThinkTime* $= 0$ we say that the client is *eager*. We also assume that, unless there is a transient network partition, as soon as an update is performed in a given replica, it is propagated immediately to the remaining replicas. A timeout value is used to control the frequency of control information: for Bayou the timeout controls how often each node broadcasts an "I'm alive message" and in ENGAGE the timeout is used to control for how long a broker holds a MF message (to piggyback it with an update notification).

The workload parameters are summarized in the Table I. We consider variations of this workload in which we change the value of one parameter and keep the others at their default values. In the following experiments, we measure two metrics, namely the *remote visibility latency* and the *remote operation latency*. The remote visibility latency is the time from which the replica received the update until it can apply it using the correct semantics. The remote operation latency is the time from which the remote cloudlet received the client's request until it can respond using the correct semantics.



(a) Remote Visibility Latency (b) Remote Operation Latency

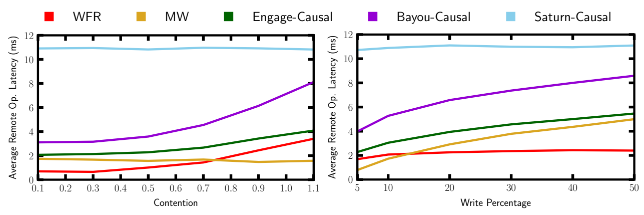
Fig. 5: ENGAGE vs Bayou and Saturn

B. ENGAGE *vs Bayou and Saturn*

In this section, we try to answer the first question and position ENGAGE with regard to Bayou and Saturn. In particular, we present again the results from Figure 1 (that we have used in the motivation) now including the performance of ENGAGE.

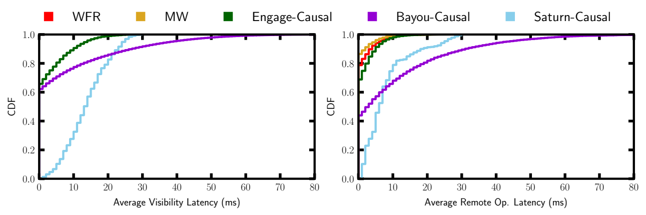
Figure 5a shows the remote visibility latency, i.e., the time it takes for a local update to be applied remotely. In this case, in the x axis, we vary the diversity in access frequency to different objects. For small values, all objects are accessed at the same pace, and for large values, some objects are accessed much more frequently than others. In this experiment, we disabled the metadata flush of ENGAGE and Bayou (an evaluation of these mechanisms is postponed for Section IV-E). Instead, there is a set of keys that are replicated in every cloudlet; updates on these keys keep vector clocks up to date. For a high skewed workload, cloudlets communicate with each other at different rates. Bayou needs to receive updates from all the cloudlets before it can apply a remote update. Thus, the update visibility latency increases sharply when the access frequency is skewed. In opposition, ENGAGE and Saturn use a metadata service to apply updates. Therefore, they do not depend on metadata from operations on objects they do not replicate. As a result, ENGAGE and Saturn exhibit a visibility latency that is $10\times$ lower than Bayou for high contention workloads.

Figure 5b shows the delays experienced by clients when they perform a remote operation. In this case, in the x axis, we vary the clients' think time, i.e., the average time between two consecutive operations. The larger the think time, the more likely it is that updates in the causal past of the client have already been propagated and applied when the client performs a remote operation. Thus, we expect the remote operation latency to decrease as the think time increases. ENGAGE, and Bayou are systems based on vector clocks that have fine-grained information about which updates the client has observed and that need to be locally applied to avoid violating the client consistency requirements. Using this information, they can reply faster. Saturn does not keep detailed information regarding the past of each client. Thus, when a client executes a remote operation, Saturn needs to propagate a migration label to the remote cloudlet through the metadata service, making the client always dependent on the last operation executed or received by the local cloudlet. As such, Saturn is unable to leverage the think time of the client to lower the access



(a) Latency vs contention (b) Latency vs read/write ratio

Fig. 6: Remote Latency with Session Guarantees



(a) Remote Visibility Latency (b) Remote Operation Latency

Fig. 7: CDF of the Remove Visibility Latency and Remote Operation Latency.

latency and exhibits a (constant) high remote operation latency (the horizontal blue line at the top of the figure), resulting in a $7\times$ higher remote operation latency than ENGAGE for $\textit{ThinkTime} = 2ms$ and $20\times$ for $\textit{ThinkTime} = 10ms$.

C. *Benefits from Session Guarantees*

In the previous section we have shown that ENGAGE is able to match Bayou when supporting remote reads using causal consistency. We now show that ENGAGE can further reduce the latency of remote operations if the user uses the weaker session guaranties, instead of using full causal consistency. Figure 6 shows the remote latency associated when different session guarantees are used (for RYW or MR it shows the latency of remote reads and for MW and WFR it shows the latency of remote writes). The figure shows how the latency for the different session guarantees is affected by parameters such as the contention level and the read/write ratio.

Figure 6a shows the impact of the contention level on remote latency for different system and session guarantees. In Saturn, a remote operation always requires the exchange of a migration label from the origin cloudlet to the remote one. This makes the remote operations in Saturn depend on the network latency, regardless of the workload pattern. Not surprisingly, causal consistency, being stronger than any of the session guarantees in isolation, is the criteria that leads the client to experience larger latency. This is more noticeable in Bayou than in ENGAGE, as our system is able to update remote clocks faster, by leveraging on normal data flow to flush MF messages (instead of depending exclusively, on "I'm Alive messages", as Bayou). When weaker session guaratees are chosen, ENGAGE offers even lower latency. Since Monotonic Reads and Read Your Writes never need to block the client, the latency is always zero (in the figure, these lines overlap with the x-axis). Monotonic Writes tend to remain constant, as the worst-case consists of writing consecutively on objects located in distant cloudlets, a scenario that is not greatly affected by varying contention. Write Follow Reads growth follows causality, as the worst-case is reading a freshly written object before the remote write operation, this case is boosted with higher contention because it is easier to read an object that was freshly written. Figure 6b shows the impact of the read/write on remote latency. For most guarantees, the figure shows a similar trend. As in Figure 6a, Saturn tends to remain constantly high. Also, as expected, the latency tends to grow slightly with the write ratio. However, it is interesting to notice

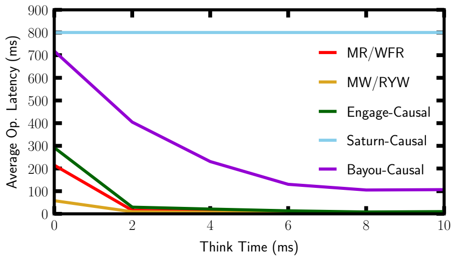


Fig. 8: Tolerance to Transient Partitions

that while contention has little effect on MW and a large impact on WFR, the opposite happens when we increase the write/read ratio. This is not surprising, given that the larger fraction of writes, the more likely it becomes that a client depends on some recent update that is still in transit.

Figures 5b and 6b depict *average* latencies. It is also interesting to look in detail to the *Cumulative Distribution Function* (CDF) of the latency, both for the remote visibility latency and for the remote operation latency of the different systems. This is depicted in Figure 7. In this case, we used the default parameters from Table I. We can observe that Bayou has high tail latency, in the 90th percentile, Bayou has almost $2.7\times$ the remote operation and $6\times$ the visibility latency when compared with ENGAGE. This shows that ENGAGE is much more suitable than Bayou for latency-critical applications and must achieve small and predictable tail latencies (e.g., 95th or 99th percentile) to work properly [32].

D. *Tolerance to Transient Partitions*

All networks can be subject to transient partitions, where a node or a set of nodes becomes temporarily disconnected from the rest of the network. During a transient partition, the propagation of updates that are performed at a given cloudlet may be delayed or postponed. A prominent feature of session guarantees is that they have the potential for shielding client from being affected by a transient partition. In fact, operations that can be performed on their local client can be executed without coordination, and are not affected by the partition. Remote operations may depend on updates affected by a partition, but with session guarantees clients have more control of what updates they need to observe to operate without violating consistency.

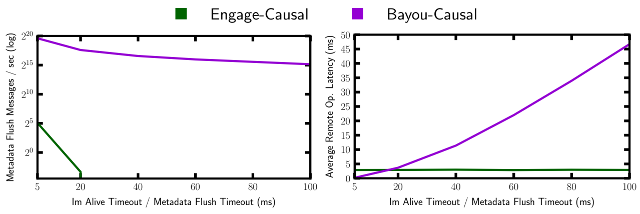
In Figure 8, we show the average time to execute an operation using different session guarantees of the clients that migrated to the nearest cloudlet due to a transient fault. We set the transient partition time to $800ms$ (i.e., during $800ms$ no client or cloudlet could send or receive messages from the partitioned cloudlet) and observe how the different systems behave as we vary the client think time. As Saturn does not keep detailed information regarding each client's past and requires to propagate a remote label through the metadata service, the clients either break causality or need to wait until the transient fault is healed. In contrast, ENGAGE and Bayou can perform fine-grained dependency checking, using the information stored in the client vector clock. This allows some clients to execute remote operations without violating causality, even if the remote cloudlet is not completely up-to-date, as long as the missing information is not in the client's causal past. Interestingly, ENGAGE is able to outperform Bayou. This happens because Bayou needs to receive messages from all other cloudlets to apply any remote updates (therefore, all updates are affected by the network partition) while in ENGAGE, only the updates that have origin in the partitioned cloudlet are delayed.

Moreover, the ENGAGE session guarantees give clients more control over what updates they need to observe to operate without violating consistency. Thus, allowing to execute operations with much lower latency than causal consistency, notably in cases where not all client's causal dependencies were propagated before the transient fault, as the client would need to wait for the transient fault to be resolved. MR/WFR achieves 36% lower latency, and MW/RYW achieves almost $5\times$ lower latency than causal consistency.

E. Signaling Overhead

Both ENGAGE and Bayou require the exchange of control messages to update the cloudlet's vector clocks. This is of paramount importance to allow clients to be served quickly and avoid unnecessary delays due to false dependencies. In systems such as Bayou, vector clocks can be updated via the periodic exchange of "I'm Alive" messages, that carry the vector clock of the sender [13, 33]. ENGAGE uses metadata flush (MF) messages for the same purpose. However, unlike Bayou, in ENGAGE MF from a cloudlet can be piggybacked on the updates messages sent from other cloudlets, as explained in Section III (for instance, see the example of Section III-F). This often prevents ENGAGE from being required to send signaling messages just to update vector clocks. Figure 9a shows the number of control messages exchanged both by Bayou and by ENGAGE as a function of the timeout value (the timeout value indicates when a control message needs to be explicitly sent in absence of a suitable update). Note that Saturn is not depicted, as it does not rely on vector clocks (with the latency penalty shown in previous sections).

In the Figures 9a, in the x axis we vary the timeout value and in the y axis we depict the number of control messages per second (note that we use a logarithmic scale in this figure). In ENGAGE, we only count the MF messages that were not



(a) Signaling Overhead (b) Signaling Tolerance

Fig. 9: Signaling Impact: ENGAGE vs Bayou

piggybacked with update messages, this captures the extra messages incurred by ENGAGE over the Saturn's metadata service. Obviously, the larger the timeout the smaller is the signaling overhead, given that control messages are only sent when the timeout expires. However, it is interesting to see that ENGAGE benefits much more from a larger timeout than Bayou. In fact, it can be observed that, for a timeout of $5ms$, Bayou sends approximately $32000\times$ more control messages than ENGAGE, and after a timeout of $20ms$, all the ENGAGE MF messages can piggybacked in some update message with high probability. This shows that the piggyback mechanisms of ENGAGE become more and more effective as the timeout increases.

In Figure 9b, we can observe how the timeout value affects the remote operation latency in both systems. For a timeout of $20ms$, both systems present almost the same remote operation latency. However, as we increment the timeout value, the Bayou latency sharply increases. From these experiments, it is clear that the timeout value has a large impact on the performance of Bayou, while the performance of ENGAGE stays mostly unchanged. This allows for ENGAGE to achieve low remote operation latency with high timeout values, avoiding the need to send unnecessary MF messages.

V. CONCLUSIONS

Given that latency driven applications are one of the main drivers for edge computing, to offer low latency when accessing data on the edge is of paramount importance. In this paper we have presented ENGAGE, a novel architecture for supporting session guarantees for partially replicated edge storage systems. ENGAGE combines, in a synergistic way, the use of vector clocks and metadata services to achieve *both* low visibility latency and low remote operation latency. We show that ENGAGE allows the programmer to fully exploit the application semantics to improve the performance of operations: by using session guarantees the application avoids the latency imposed by strong consistency, and can outperform systems based on full causal consistency. At the same time, ENGAGE avoids stalling remote updates due to false dependencies, offering small remote visibility latency.

Our experimental evaluation shows that the latency gains achieved with ENGAGE can be as high as $7\times$ for remote update visibility $2.7\times$ for remote operations, compared with Bayou for high contention workloads. Moreover, ENGAGE can tolerate transient partitions much better than Saturn, reducing the latency 2.6× for eager clients and almost 27× for non eager clients, while offering alternative session guarantees to causality that can further reduce the latency. Finally, ENGAGE has a much lower signaling cost than Bayou.

ACKNOWLEDGMENTS

This work was partially supported by the FCT via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271) and project UIDB/ 50021/ 2020. Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Nivia Quental and Taras Lykhenko.

References

[1] K. Saito, A. Mikami, K. Ariga, H. Yasutake, S. Kimura, and H. Hane, "Case studies of edge computing solutions," *NEC Technical Journal*, vol. 12, no. 1, 2017.

[2] M. Schneider, J. Rambach, and D. Stricker, "Augmented Reality Based on Edge Computing Using the Example of Remote Live Support," in *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*, Toronto, Canada, Mar. 2017.

[3] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A multi-tier cloud computing framework," in *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing (SEC)*, San Jose (CA), USA, Oct. 2017.

[4] 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 (DEBS)*, Hamilton, New Zealand, Jun. 2018.

[5] 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, Jul. 1990.

[6] 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 the 3rd International Conference on on Parallel and Distributed Information Systems (PDIS)*, Austin (TX), USA, Oct 1994.

[7] R. G. Guy, J. S. Heidemann, W.-K. Mak, T. W. Page Jr, G. J. Popek, D. Rothmeier *et al.*, "Implementation of the Ficus replicated file system," in *Proceedings of the USENIX Summer Technical Conference (USTC)*, Anaheim (CA), USA, Jun. 1990.

[8] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 360–391, Nov. 1992.

[9] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," in *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, Pacific Grove (CA), USA, Oct. 1991.

[10] C. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proceedings of the 11th Australian Computer Science Conference (ACSC)*, Queensland, Australia, Feb. 1988.

[11] M. Bravo, L. Rodrigues, and P. van Roy, "Saturn: A distributed metadata service for causal consistency," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belrade, Serbia, Apr. 2017.

[12] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Lombard (IL), USA, Apr. 2013.

[13] S. 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 *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston (MA), USA, Mar. 2017.

[14] E. Brewer, "Towards robust distributed systems," in *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Portland (OR), USA, Jul. 2000.

[15] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally-distributed database," in *Proceedings of 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood (CA), USA, Oct. 2012.

[16] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Philadelphia (PA), USA, Jun. 2014.

[17] M. Schroeder, A. Birrell, and R. Needham, "Experience with grapevine: The growth of a distributed system," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, p. 3–23, Feb. 1984.

[18] 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 *Proceedings of 36th International Conference on Distributed Computing Systems (ICDCS)*, Nara, Japan, Jun. 2016.

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

[20] 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.

[21] F. Ruget, "Cheaper matrix clocks," in *Proceedings of the 8th Workshop on Distributed Algorithms (WDAG)*, Terschelling, The Netherlands, Sep. 1994.

[22] S. H. Mortazavi, M. Salehe, B. Balasubramanian, E. de Lara, and S. PuzhavakathNarayanan, "SessionStore: a session-aware datastore for the edge," in *Proceedings of the 4th IEEE International Conference on Fog and Edge Computing (ICFEC)*, Melbourne, Australia, May 2020.

[23] S. Mortazavi, B. Balasubramanian, E. de Lara, and S. Narayanan, "Toward session consistency for the edge," in *Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, Boston (MA), USA, Jul. 2018.

[24] H. Gupta, Z. Xu, and U. Ramachandran, "DataFog: Towards a Holistic Data Management Platform for the IoT Age at the Network Edge," in *Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, Boston (MA), USA, Jul. 2018.

[25] Z. Hao, S. Yi, and Q. Li, "EdgeCons: achieving efficient consensus in edge computing networks," in *Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, Boston (MA), USA, Jul. 2018.

[26] F. Nawab, D. Agrawal, and A. El Abbadi, "DPaxos: managing data closer to users for low-latency and mobile applications," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, Houston (TX), USA, Jun. 2018.

[27] C. Meiklejohn, H. Miller, and Z. Lakhani, "Towards a solution to the red wedding problem," in *Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, Boston (MA), USA, Jul. 2018.

[28] N. Preguiça, C. Baquero, and M. Shapiro, "Conflict-free replicated data types (CRDTs)," in *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Grenoble, France, Oct. 2011.

[29] S. Zhang, W. Zeng, I.-L. Yen, and F. B. Bastani, "Semantically enhanced time series databases in IoT-Edge-Cloud infrastructure," in *Proceeedings of the IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, Hangzhou, China, Jan. 2019.

[30] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," in *Proc. of the 39th International Conference on Very Large Data Bases (VLDB)*, Trento, Italy, Aug. 2013.

[31] A. Montresor and M. Jelasity, "Peersim: A scalable P2P simulator," in *Proceedings of the IEEE 9th International Conference on Peer-to-Peer Computing (P2P)*, Seattle (WA), USA, Sep. 2009.

[32] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Providence (RI), USA, Sep. 2016.

[33] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th ACM Annual Symposium on Cloud Computing (SOCC)*, Santa Clara (CA), USA, Oct. 2013.