# ENGAGE: Session Guarantees for the Edge

Miguel Belém[1], Pedro Fouto[2], Taras Lykhenko[1], João Leitão[2], Nuno Preguiça[2], Luis Rodrigues[1]
[1]*INESC-ID*, Instituto Superior Técnico, Universidade de Lisboa
mbelem@sysmail.me, {taras.lykhenko, ler}@tecnico.ulisboa.pt
[2]*NOVA-LINCS*, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa
p.fouto@campus.fct.unl.pt, {jc.leitao, nmp}@fct.unl.pt

2022 International Conference on Computer Communications and Networks (ICCCN) | 978-1-6654-9726-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCCN54977.2022.9868846

***Abstract*—Edge computing offers support for latency-constrained applications, by replicating data in the edge. Edge storage systems need to adopt both partial replication, as only data of interest needs to be replicated, and weak consistency models, to avoid the overhead and latency induced by the coordination mechanisms of strong consistency models. In this context, session guarantees are a powerful tool that can be used to simplify the design of edge applications. This paper presents ENGAGE, a storage system that offers efficient support for session guarantees in a partially replicated edge setting. To achieve this, ENGAGE combines the use of vector clocks and distributed metadata propagation services with a payload propagation scheme tailored for the edge. We have implemented ENGAGE and evaluated its performance experimentally. The results show that, when compared with previous proposals, the combination of techniques employed by ENGAGE reduce both the number of false dependencies, that can slow down the system, and the signaling overhead, while improving the freshness of data exposed to clients.**

## I. INTRODUCTION

Nowadays, numerous applications have clients running on the edge of the network relying on cloud infrastructures for computation offloading and storage [31]. Unfortunately, the high network latency between clients and data centers can impair novel latency-constrained applications such as augmented reality [32], highly-interactive mobile applications, among others [20]. Edge computing has emerged as a potential solution to circumvent this problem. To unleash its full potential, edge nodes must replicate data that is frequently used. However, because edge nodes have limited resources, full replication is infeasible. Therefore, any edge storage service needs to support partial replication. Additionally, the bandwidth of network links at the edge can be highly variable, and is generally much lower than the bandwidth of the dedicated links that connect datacenters [33]. As such, edge applications should take these limitations into account in order to efficiently disseminate information across all replicas. Not doing so may cause the lower bandwidth of edge nodes to become the bottleneck of the system, as was previously highlighted in [7].

Furthermore, there is also evidence that edge storage should support weak consistency models [13], [25] since strong consistency models require coordination among replicas when updates are performed, which increases latency and can become impractical when the number of replicas grows.

In a seminal work, Terry *et al.* [34] have introduced the notion of *session guarantees*, a set of well defined semantics that may be used to simplify the design of distributed applications 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 not yet have been applied 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. This ensures the correctness of the results and avoids data anomalies that can be hard for application developers to predict and tackle.

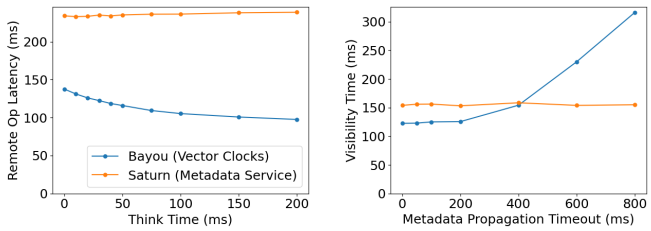
In [34], the authors have also suggested mechanisms to enforce session guarantees that rely on the use of version vectors [18], a form of vector clocks [9]. These mechanisms are only efficient in settings that use full replication, where all updates are propagated to all replicas. In systems that adopt partial replication, the execution of an operation in a replica may be delayed when it depends on an operation that will not be received in that replica. Avoiding this problem requires replicas to maintain and periodically exchange large amounts of metadata (e.g., forcing all messages to carry a vector clock for each shard in the system).

As such, with partial replication, the use of vector clocks may impair *visibility times*, i.e., the time it takes for an update to become visible in remote replicas. This limitation is of significant concern for edge applications, where low visibility times are highly desirable.

The challenges of providing low visibility times with small metadata have been recognized in the literature [4], [21]. To address these challenges, the abstraction of a distributed metadata service has been recently introduced [4]. 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. To the best of our knowledge, existing metadata services, such as Saturn [4], offer only causal consistency and have no support for (in-

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) under projects UIDB/50021/2020, NOVA LINCS (grant UID/CEC/04516/2013) and NG-STORAGE (PTDC/CCI-INF/32038/2017).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr/).




(a) Remote Operation Latency (b) Visibility Time

Fig. 1: Remote Operation Latency and Visibility Times.

dividual or combinations of) 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 operation latency and the visibility times in two separate systems. One of them uses vector clocks (Bayou [34]) and the other uses a metadata service (Saturn [4]), with both configured to provide causal consistency. We postpone a detailed description of the experimental setup to Section IV, presenting 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 their preferred replica unless they need to access a data object that is not replicated there, in which 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. This waiting time can contribute substantially to delay the remote operation. Fig. 1a shows the delays experienced by clients when performing a remote operation. In the $x$ axis we vary the average time between two consecutive operations. The larger this think time, the more likely it is that updates in the causal past of the client have already been applied when the client performs a remote operation. 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. This allows replicas to execute operations originally executed on remote replicas faster, hence providing faster replies for clients, particularly when a client has just migrated from another replica. 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.

Fig. 1b shows the operation visibility times for both classes of systems. When using vector clocks, before executing an operation, a replica needs to wait until all operation's dependencies have been applied. As a replica might not receive some of the operations, it needs to receive metadata from other replicas to find out that no operation is missing. This information can be broadcasted periodically by all replicas. The results show that the visibility time depends on the frequency of broadcast, rising sharply as the period of broadcast increases. High frequency broadcasts lead to low visibility time at the cost of an increasing number of (often redundant) exchanged messages. Metadata services circumvent this problem, offering low visibility times without the need for metadata broadcasts.

The goal of this paper is to derive a strategy that can achieve the best of both worlds, i.e., to combine low visibility times and efficient support for clients performing remote operations using different session guarantees, while also optimizing the use of the limited bandwidth of edge nodes. 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, leveraging the hierarchical nature of the metadata service to offload the cost of message dissemination to the high-performance datacenter links. We provide an extensive evaluation of ENGAGE in an emulated edge environment, comparing it with a system based on vector clocks (Bayou [34]) and a system based on metadata services (Saturn [4]), using different combinations of the session guarantees proposed in [34].

## II. Background and Related Work

### A. Background

Weakly consistent replication schemes have been introduced as a way to circumvent the performance bottlenecks associated with strong consistency, improving the availability of replicated systems [5]. In strongly consistent systems all updates need to be serialized; this requires the use of a single primary replica or the use of a consensus protocol [27]. While strongly consistent systems may block, weakly consistent systems allow updates to be performed concurrently, without coordination across replicas, offering higher availability.

**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 is missing. Experience has shown that weak consistency makes application development difficult [2]. In [34], session guarantees have been introduced as a way to simplify the application development when relying on weakly consistent replicated datastores. 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* (MW). 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. When all these guarantees are combined, the system offers *causal consistency* [1].

**Vector Clocks** Version vectors [17], [18], also known as vector clocks [9], are a popular mechanism to keep track of dependencies among update operations and identify those that are concurrent. 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. Vector clocks can also be used to enforce session guarantees [34].

**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 vector clocks for all objects in all messages [3] or, alternatively, a matrix clock [30]. 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 operations 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 yet been applied to $R_2$. Unfortunately, there is no way for $R_2$ to infer if the missing update corresponds to some object that is replicated locally or to an object that is replicated somewhere else (and will never be received).

**Metadata Services** Distributed metadata services [4] have been proposed as a solution to provide low visibility time in partial replicated systems while keeping the size of metadata very small. When an update is generated at a given replica, this information is propagated to the metadata service, which 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 low visibility times, but offer only causal consistency. Solution to extend these services to weaker consistency models, such as session guarantees, have not been explored. In this work, we present a novel protocol, which is able to combine the benefits of vector clocks and metadata services, avoiding their individual limitations and allowing efficient support for partial replication in the edge while offering different session guarantees.

### *B. Related Work*

**Session Guarantees** Session guarantees were first introduced in Terry *et al.* [34], which presented a replicated storage system, Bayou, that leverages on version vectors to support the different session guarantees. In this system, clients keep two version vectors, one to record their writes and another to record the writes that are relevant to their reads. These version vectors can then be used to check if a replica contains all the necessary operations to satisfy the client's required session guarantees. Details about the dissemination of write operations are given in a later work [28], where an anti-entropy protocol based on periodic pair-wise communications is employed. In this protocol, periodically, each replica requests another replica's version vector, and uses it to check which operations the other replica is missing, sending them in the received order. While supporting diverse network topologies (such as low-bandwidth links or unreliable networks), this protocol suffers from slow update propagation, sacrificing data visibility times. In contrast, as low latency is a key requirement for today's edge applications, our system strives to minimize visibility times, by propagating operations across replicas in a reactive way instead of leveraging on anti-entropy. Furthermore, we propose a novel mechanisms to propagate control information that allows remote replicas to execute operations faster.

**Edge Storage** Several works have addressed edge storage, but few have addressed the problem of latency when considering session guarantees. EdgeCons [15] and DPaxos [26] propose efficient consensus algorithms for the edge that target strongly consistent systems. As we discussed previously, the use of strong consistency will lead to performance penalties when the number of replicas grows, which is expected in edge datas tores. SessionStore [24] is a data store for edge applications that 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 [25], a system that replicates application data on-demand. Because data is shipped on demand, clients can experience high latency, which our solution avoids. Similarly to ENGAGE, FogStore [13] and DataFog [14] aim at offering low latency with different semantics. However, in FogStore and DataFog, the semantics drive how many replicas need to be read/written in order to execute a given operation, while ENGAGE always serves requests locally. Furthermore, the consistency criteria supported by FogStore are not directly comparable with session guarantees. In [22], the use of CRDTs [29] 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 [35] 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.

## III. THE ENGAGE SYSTEM

ENGAGE is a system that aims at combining low visibility times *and* support for session guarantees, while avoiding the costs of using matrix clocks and optimizing the use of available bandwidth in edge nodes. 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 and decrease bandwidth usage for edge nodes.

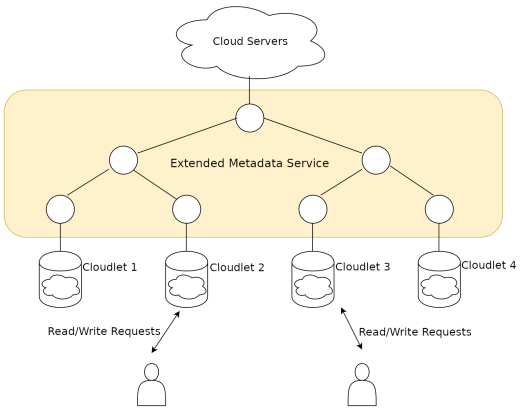



Fig. 2: ENGAGE Architecture

### A. System Model

Fig. 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 vary from 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 objects to cloudlets is known by all cloudlets. Typically, data replicated in the cloudlets will also be stored in on or more cloud datacenters (which are placed as higher nodes in the tree), but this is not necessary for the operation of ENGAGE.

ENGAGE clients have a preferred cloudlet, to which they forward all requests. If the preferred cloudlet does not replicate the target data object of an operation, the client must send the request to the nearest replica that does (client migration). 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. Extended Metadata Service (Overview)

Cloudlets are attached to an extended metadata service that supports the propagation of updates and ensures they become visible at other replicas, in an order that does not violate causality. Contrary to systems such as Saturn, where the metadata service is used exclusively to propagate metadata, in ENGAGE we extend this service to additionally support the propagation of the update payloads in a more efficient manner, that takes into consideration the topological properties (e.g., link capacity) of the edge network.

The extended metadata service is implemented by a set of cooperative servers, or *extended metadata brokers*, organized in an acyclic graph (or tree). The leaf servers of the tree are deployed at the edge cloudlets, i.e, every edge cloudlet runs a local extended metadata broker that is used to convey both metadata and data to, and from, remote cloudlets. The extended metadata brokers that run as inner nodes of the tree are placed strategically to optimize aggregation and data dissemination. They can be deployed in cloud datacenters, cloudlets with more resources, or even in some edge cloudlets that are well placed in the network (in terms of connectivity to other cloudlets).

### C. Metadata

We assume that each cloudlet is linearizable [16], which means that when one update becomes visible for a client, it becomes visible for all clients of that cloudlet. Thus, each cloudlet keeps a 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 data 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 data object $o_2$.

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

- For each object $o$ that is replicated in a cloudlet $i$, a vector clock $V_o^i$ is stored with the object. The vector clock captures all updates in the causal past of that object.

- Each cloudlet $i$ also keeps a *cloudlet vector clock* $V_*^i$ that captures the state of the local database. This clock encodes the highest sequence number of update operations executed from each remote cloudlet.

- Finally, each client $c$ 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.

### D. 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 [34], namely: *Read Your Writes* (RYW), *Monotonic Reads* (MR), *Writes Follow Reads* (WFR), and *Monotonic Writes* (MW). From the point of view of the client operation, ENGAGE offers no novel contribution.

On the server side, we employ a number of mechanisms that effectively alter the original algorithm, to support multiple objects that keep different clock values. When performing a read or write operation on data object $o$ using cloudlet $i$, the client $c$ provides its own $V_c^R$, $V_c^W$, and the desired session guarantees for the operation. The cloudlet 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 $V_*^i \geq V_c^W$.

In the case of a read operation, 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. For a write operation, the cloudlet

assigns a sequence number *snb* to the update, by incrementing the local counter that serializes all updates. Then, it creates a temporary *update vector clock* $V^{up}$ that has all entries set to 0 except its own entry $i$, which is set to *snb*. Next, it updates several clocks as follows:

- It sets $V^i_* = \text{MAX}(V^i_*, V^{up})$.
- It sets $V^i_o = \text{MAX}(V^i_o, V^{up}, V^R_c, V^W_c)$.
- It sets $V^W_c = \text{MAX}(V^W_c, V^{up})$.

After these updates, it returns the new value of $V^W_c$ to the client. In parallel, it schedules the update, tagged with $V^i_o$, to be sent through the extended metadata service to the other cloudlets that replicate $o$.

Note that, in practise, the client only needs to provide its vector clocks when executing the first operation in a new cloudlet after migrating. After the first operation, session guarantees are assured without the client sending its vector clock: when using RYW, the client executes operations on the cloudlet, so the clients' previous write operations are always reflected on the cloudlet since the servers are linearizable; when using MR, updates being applied in causal order combined with the client being sticky, results in clients always reading a version of the key that is greater or equal than the previous version that the client has read.

### E. Applying Remote Updates

In every replicated storage system, for an update executed at a given replica to become visible globally, it needs to be propagated to all other replicas that replicate the target object.

Here, we will describe how these updates can be applied in causal order based on *vector clock stability*, which is a common technique employed in causal replication systems [2], [11], [12], [34], where each replica keeps a vector clock, with one position per replica. This vector clock is used to verify if all dependencies of a remote operation have been executed locally, in order to make that operation visible and/or to check if all operations in the causal history of a client have been applied locally.

In ENGAGE, this technique would work as follows. When an update performed at a replica *orig*, tagged with vector clock $V^{orig}_o$, is received at some other replica *dest*, it is put in a list of pending updates and it remains there until both of the following conditions are met:

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

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

- It sets $V^{dest}_* = \text{MAX}(V^{dest}_*, V^{orig}_o)$.
- It sets $V^{dest}_o = \text{MAX}(V^{dest}_o, V^{orig}_o)$.

However, using vector clock stability to apply remote updates is not effective under partial replication. Imagine that a cloudlet $i$ receives a remote operation $u$ which depends on another operation $u'$. If $u'$ is an operation over an object not replicated in $i$, it will never be received, and the cloudlet has no way to know if it is suposed to receive it or not.

To solve this problem, nodes can, periodically, send to each other the values of their cloudlet vector clocks. In this paper, we call these *metadata flush* (MF) messages. These messages generate additional traffic and make the remote update latency a function of the period used to exchange them. Note that, under partial replication, MF messages are necessary not only to apply remote updates but also to serve remote operations (i.e., supporting client migration).

### F. The ENGAGE Extended Metadata Service

A key insight behind the design of ENGAGE is that a metadata service, such as the one proposed in [4], can be extended to perform multiple functions: it can be used to instruct cloudlets to deliver remote updates (as proposed in [4]), it can be used to disseminate updates themselves 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 through a distributed extended metadata service, inspired by Saturn [4]. The medatada service is implemented by a set of servers, denoted *extended metadata brokers* (or simply: *brokers*), that are distributed in different locations of the network that interconnects the replicas, and can either be co-located with cloudlets and data centers, or execute independently (as seen in Fig. 2). The brokers are organized as an acyclic graph and each cloudlet is connected to one of these brokers. Unlike Saturn, that only propagates update *labels* (i.e., a scalar that uniquely identifies an update), ENGAGE propagates two types of messages: *update notifications*, that carry update operations with their associated vector clocks, and *metadata flush* messages, which only carry the vector clocks associated with update operations.

Update notifications are tuples associated with a concrete update. They include the following fields $\langle$UN, *src*, *snb*, *oid*, *payload*, $V^{src}_{oid}\rangle$, 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, *payload* is the update operation itself and, finally, $V^{src}_{oid}$ 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 $V_{\text{MF}}$ is a vector clock that will be used to update the cloudlet vector clocks.

When a cloudlet processes a write request from a client and a new update $u$ is created, as explained in Section III-D, the cloudlet also creates an update notification message that it delivers to the local broker. When a broker receives an update notification, it performs the following sequence of actions for all tree 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, it is piggybacked with the update notification, cancelling any timeout associated with it.

- 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 into a single MF message, with a vector clock that has is the maximum of both clocks. If there was no other MF message pending on edge $e$, a timer is started to propagate the MF message.
- When the timer associated with an edge expires, the broker 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 $V_{\text{MF}}$ to update $V_*^{dest} = \text{MAX}(V_*^{dest}, V_{\text{MF}})$. Finally, when an update message $\langle \text{UN}, \textit{src}, \textit{snb}, \textit{oid}, \textit{payload}, V_{oid}^{src} \rangle$ is received by a cloudlet, the acyclic layout of the extended metadata service guarantees that every operation that could be a dependency has already been delivered locally (either as a MF message or an update message) and, as such, the update can be applied immediately (or after its dependencies finish executing).

### G. Payload Propagation

The design of ENGAGE allows it to support two different strategies to propagate the update payload:

**Broker-Assisted Payload Propagation:** In this mode, which we use as the default one, the payload of an update is propagated in the metadata broker network as part of the update notification message, as explained previously. Using the metadata service to propagate update operations (and not only their metadata) allows ENGAGE to optimize bandwidth usage in an edge scenario, as discussed further in Section IV-E.

**Direct Propagation:** ENGAGE can also be applied to a more traditional geo-replicated scenario, where replicas are datacenters connected by a dedicated high-performance network. In this scenario, propagating the payload of update messages directly between datacenters can decrease visibility times, while the extended metadata service ensures all replicas can progress even with partial replication.

## IV. EVALUATION

Our evaluation addresses the following research questions:

- What is the signaling cost of ENGAGE, in comparison with Bayou [34]?
- How does ENGAGE perform in comparison with classical vector clock approaches, such as Bayou, and with recent metadata services, such as Saturn [4]?
- Can ENGAGE bring advantages to clients that exploit session guarantees, decreasing their observed latency?
- Can ENGAGE be efficiently used in an edge scenario with heterogeneous network links?

### A. Experimental Setup

For the experimental evaluation, we implemented ENGAGE, Bayou, and Saturn over the Cassandra [19] distributed database, which only offers eventual consistency, and using a framework for building distributed protocols, Babel [10]. Since the target of this work are edge scenarios, our implementation of ENGAGE uses the Broker-Assisted Payload Propagation presented in Section III-G. In Saturn, update labels are propagated using the metadata service, while update payloads are disseminated directly between replicas, as explained in [4]. Our implementation of Bayou, while following the original article [34], has two modifications: $i$) Update operations are propagated eagerly to remote replicas, instead of using periodic anti-entropy synchronizations. This change does not alter the protocol guarantees in any way, it simply speeds up the propagation of operations, allowing us to make fairer comparisons with the other solutions; $ii$) To allow progress under partial replication, as explained in Section III-E, replicas periodically send each other metadata messages containing their vector clock values.

Experiments were run on the Grid5000[1] computing platform, with each replica running in an individual machine with an Intel Xeon Gold 5220, and 96GiB of memory. Clients were executed on similar machines, but with multiple client threads per machine, using the YCSB [6] benchmark tool. In order to control the latency and bandwidth of the networks links, we used the Linux Traffic Control[2] tool.

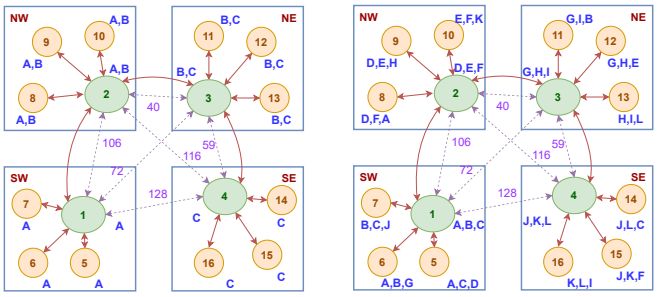
Fig. 3 illustrates the settings used in the experimental evaluation. We considered an edge scenario, where replicas are distributed in four geographic locations (or regions), with a main datacenter (numbered 1 to 4) and three edge replicas (5 to 16) per location. The bandwidth of edge replicas is limited to 1Gbps of download and 500Mbps of upload. The dotted lines represent the latency between replicas, with the latency from the edge nodes to their local datacenter being $10ms$. The connections used by the ENGAGE and Saturn metadata services are represented by the solid lines. To model the use of partial replication, we partition the data across replicas, with each data partition being represented by a letter.

While data placement is not the focus of our work, it has an impact on the performance of partially replicated systems. As such, we consider two distinct data placement schemes:

- a *non-uniform locality-driven data placement* (Fig. 3a), where each data partition is replicated in every replica of 2 regions (e.g., partition A is placed at replicas 1, 5, 6, 7 of the Southwest region and on replicas 2, 8, 9, 10 of the Northwest region), and not all regions replicate the same number of data partitions. ENGAGE and Saturn's metadata services are setup in a way that minimizes the number of hops between regions that replicate the same data partitions. In this setting, we only use three data partitions, allowing for a detailed analysis on how visibility times of operations are affected by each system.
- an *uniform data placement* (Fig. 3b), in which we assign a region to each partition, replicating it in the datacenter and two edge nodes of that region, plus in an edge node of

[1]https://www.grid5000.fr

[2]https://man7.org/linux/man-pages/man8/tc.8.html




(a) Locality-Driven Placement (b) Uniform Placement

Fig. 3: Experimental Settings

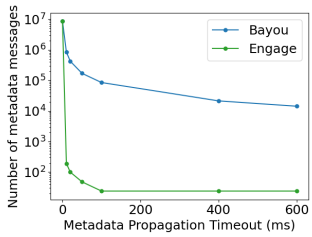
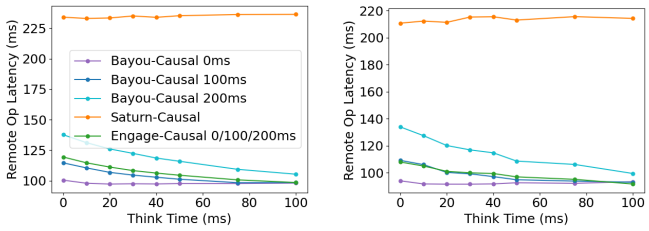



Fig. 4: Signaling Impact: ENGAGE vs Bayou




(a) Locality-Driven Setting (b) Uniform Setting

Fig. 5: Remote Op. Latency: ENGAGE vs Bayou and Saturn

another region (e.g., partition A is replicated in replicas 1, 5, 6 of the Southwest region, and in replica 8 of a different region). In this distribution, the load imposed on the system is uniform, i.e. every update operation is executed in exactly 3 nodes in the partition's region, and 1 node in another region, leading to every edge replica executing the same number of operations per second.

Unless stated otherwise, clients behave as follows: each client is assigned a local replica, against which it executes operations in a closed loop; each operation has a 50/50 chance of being either a read or a write operation (using YCSB workload A). Depending on the metric we want to assess, requests can exclusively target partitions replicated on the local replica (selected at random), or can also target partitions that are only replicated at remote replicas. Clients are co-located with their local replica (i.e. their latency to the cloudlet is negligible) and the latency between clients and cloudlets in remote regions is the same as the latency between the client's and the remote cloudlet's regions.

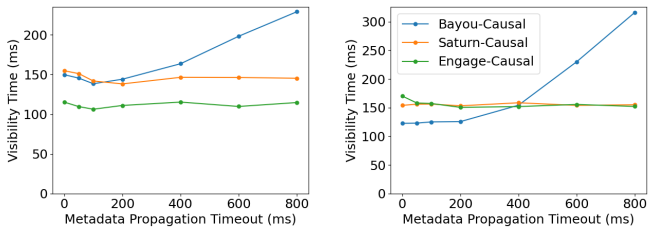
A timeout value is used to control the frequency of control information: for Bayou the timeout controls how often each node broadcasts a metadata message and, in ENGAGE the timeout is used to control for how long a broker holds a MF message. Except on experiments where we vary it, this timeout value is set to 100 ms. Furthermore, and except when stated otherwise, we use causal consistency as the target level of consistency for all operations in the experiments.

In the experiments that we report next, we consider variations of a subset of these parameters, keeping the others on their default values, in order to measure different aspects of the studied solutions. All results reported in this paper were obtained by running multiple independent instances of each experiment. While we do not report confidence intervals, we have observed negligible variations on the results across these independent runs.

### B. Signaling Overhead

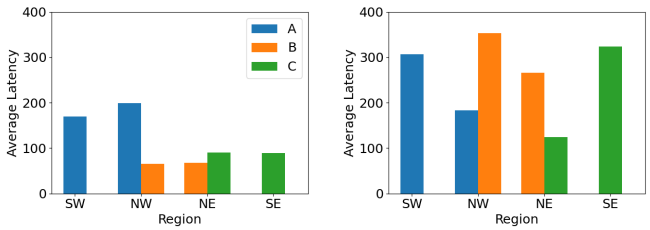
We start by comparing the signaling overhead of ENGAGE and Bayou. Both systems require the exchange of control messages to keep the vector clocks of each cloudlet up-to-date. This is of paramount importance to allow clients to be served quickly and avoid unnecessary delays due to false dependencies. In systems such as Bayou, vector clocks can be updated via the periodic exchange of metadata messages, that carry the vector clock position of the sender [8]. ENGAGE uses metadata flush (MF) messages for the same purpose. Unlike Bayou, in ENGAGE a MF message from a cloudlet can be piggybacked on the update messages sent from other cloudlets, as explained in Section III. This often prevents ENGAGE from sending signaling messages just to update vector clocks.

Fig. 4 shows the number of control messages exchanged, both by Bayou and by ENGAGE, as a function of the timeout value. In the $x$ axis we vary the timeout value and in the $y$ axis we depict the number of control messages per second (in logarithmic scale). We omit Saturn, since it does not rely on vector clocks. The reported results are based on the locality-driven setting (Fig. 3a), however, using the uniform setting we obtained similar results. In ENGAGE, we only count the MF messages that were not 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 the signaling overhead is, given that control messages are only sent when the timeout expires. While in Bayou, the number of control messages is simply a function of the timeout values used, the piggyback mechanism of ENGAGE allows it to benefit much more from larger timeouts. In fact, it can be observed that, just for a timeout of $5ms$, the number of control messages propagated by Bayou is already multiple orders of magnitude higher than ENGAGE, and after a timeout of $100ms$, all the ENGAGE MF messages can be piggybacked in some update message with high probability. This shows that the piggyback mechanism of ENGAGE is highly effective.




(a) Locality-Driven Setting (b) Uniform Setting

Fig. 6: Visibility Time: ENGAGE vs Bayou and Saturn




(a) Engage (b) Bayou

Fig. 7: Visibility Time per Partition

### C. ENGAGE *vs Bayou and Saturn*

In this section, we compare the performance of ENGAGE against Bayou and Saturn, in terms of the client delay and the visibility latency incurred by these systems.

Fig. 5 shows the latency experienced by clients when they perform a remote operation. In this experiment, clients executed 10 operations on their local replica, followed by an operation in a random remote replica. We present the results for both data placement schemes depicted in Fig. 3, while varying the timeout value used to control the frequency of metadata messages in Bayou, and the timeout of MF messages in ENGAGE. In the $x$ axis, we vary the clients' think time, *i.e.*, the time between a client receiving an operation response and sending its next operation. The larger the think time, the more likely it is that updates in the causal past of the client have already been applied when the it performs a remote operation.

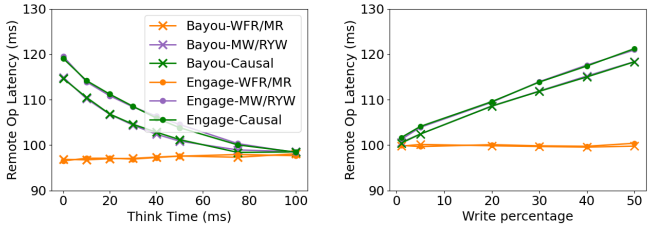
ENGAGE and Bayou are systems based on vector clocks that have fine-grained information about which updates the client has observed and that need to be locally applied before executing the client operation. Using this information, they can reply faster. Saturn does not keep detailed information regarding the past of each client. Thus, when a client executes a remote operation, it needs to propagate a migration label to the remote cloudlet, making the client always dependent on the last operation executed or received by the local cloudlet. As such, Saturn is unable to leverage on the think time of clients, exhibiting a (constant) high remote operation latency.

The latency values of ENGAGE do not depend on the metadata propagation timeout since, as shown in Section IV-B, as long as there are write operations being propagated, MF messages can be piggybacked before their timeout expires. In contrast, the metadata timeout is a key parameter for the remote operation latency in Bayou. We can see that with a timeout value of $0ms$, remote operations have low latency, however this incurs in a high number of transmitted control message. With a timeout value of $100ms$, the latency of both Bayou and ENGAGE become similar and, with an higher timeout value of $200ms$, Bayou starts showing high latency values for small think times, while still requiring an high number of control messages.

Fig. 6 shows the visibility times of operations, i.e., the time it takes for a local update to be applied on all remote replicas of the updated data object, using both data placement settings. The visibility time of operations is an important metric in edge systems with partial replication as it influences both the migration time of clients (unavoidable when using partial replication), and data freshness, i.e., if clients are accessing up-to-date or old data. In the $x$ axis, we vary the timeout value used to control the frequency metadata messages in Bayou, and the timeout of MF messages in ENGAGE.

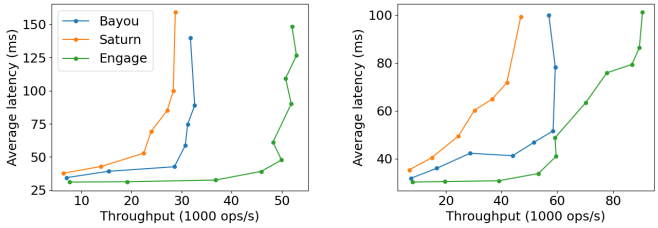
For lower timeout values, in the uniform setting (Fig. 6b), ENGAGE and Saturn require update operations or operation metadata, respectively, to go through multiple hops in the metadata service to reach all regions, resulting in higher visibility times than Bayou, where all data is propagated directly between replicas. However, in the locality-driven setting (Fig. 6a), since regions with shared data partitions are only one hop away, by funnelling all inter-region data through the high-bandwidth datacenter connections, ENGAGE achieves lower visibility times, while in Bayou and Saturn each edge node needs to propagate the payload of operations to multiple nodes, decreasing their efficiency. Since Bayou needs to receive updates from remote cloudlets before it can apply a remote update, the visibility times of operations rise sharply when the timeout value increases. In contrast, ENGAGE and Saturn rely on a metadata service to apply updates. Therefore, they do not depend on metadata from operations on objects they do not replicate, resulting in constantly low visibility time. 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 unaffected, allowing it to achieve low remote operation latency and visibility times regardless of the used timeout value.

Fig. 7 shows this behaviour in more detail. This figure shows the average visibility times per data partition per region in the experiments denoted in the last data point of Fig. 6a. In Fig. 7b we observe that in Bayou, operation visibility in a region is delayed when an operation originates from a region replicating data partitions not replicated locally. For instance, operations over partition $B$ that are received by replicas from Region NW have higher visibility times than operations over partition $A$. This happens because operations on partition $B$ may have dependencies of operations over partition $C$, that will only be fulfilled once replica 2 receives a metadata message from either replica 3 or 4. The result is




(a) Latency vs Think time (b) Latency vs read/write ratio

Fig. 8: Remote Operation Latency with Session Guarantees




(a) Locality-Driven Setting (b) Uniform Setting

Fig. 9: Throughput vs Latency

that visibility times in Bayou are a function of the period used to exchange metadata messages. On the other hand, since ENGAGE employs a metadata service, once a replica receives an update notification, it can immediately execute the operation. This happens because any pending MF messages upon which an update operation depends are flushed by that same operation and delivered to all replicas.

### D. Benefits from Session Guarantees

In this section, we show that we can further reduce the latency of remote operations when the user/application can operate using weaker session guaranties, and does not require full causal consistency. Fig. 8 shows the average latency of remote operations when different session guarantees are used The results in the figure capture how the latency for the different session guarantees is affected by parameters such as the previously mentioned think time and the read/write ratio. We omit Saturn from the plots as it does not offer support for configurable session guarantees. As previously described in Section III-D, in ENGAGE, two clocks are kept on the client-side, and used to check if an operation is safe to execute. As such, we employ three combinations of guarantees: WFR+MW, where the write clock is used for read and write operations; MW+RYW, where the read clock is used; and causal consistency, where both vector clocks are used.

Fig. 8a shows the impact of the think time of clients on remote operation latency for different systems and session guarantees. When using *WFR* and *MR*, as these are the weakest sessions guarantees leading to clients rarely ever blocking, the think time of clients has very little impact, and latency remains stable. This happens since it is very unlikely for clients to read the effects of an update operation in a cloudlet and then execute an operation in a remote cloudlet that did not receive that operation. When using *MW* and *RYW*, however, the think time of clients starts to have a visible effect on the latency of remote operations. When a client executes an update operation and then immediately executes a remote operation in a different replica, it is likely that its previous operation is not yet visible in the remote replica, thus leading the remote replica to block the client until that operation is made visible. The higher the think time of clients, the higher the chance that their previous operations are already visible in the remote replica to which the client migrates to next, thus decreasing remote operation latency times. Causal consistency is achieved by combining all sessions guarantees, (by using both vector clocks), however, since the read clock almost never blocks client operations, the results for causal consistency end up being similar to the ones of *MW* and *RYW*.

Fig. 8b shows the impact of the read/write ratio on remote operation latency. For most guarantees, the figure shows a similar trend, with the *WFR* and *MR* guarantees being unaffected by the variation of the percentage of write operations. For the other session guarantees, the latency tends to grow with the number of write operations, which is not surprising, given that additional write operations will make it the more likely that a remote operation from a client depends on one or more recent updates that are still in transit to the remote replica.

In both figures, Bayou was configured to use a metadata message timeout of $100ms$, as it was the value that made it closer to ENGAGE in the previous results (Fig 6). While this timeout leads to slightly lower latencies, it also means that the number of metadata messages is around 5 orders of magnitude higher than the number of MF messages of ENGAGE.

### E. Leveraging the metadata service for the edge

In this section, we measure the advantages of leveraging the metadata service to disseminate update operations in an edge scenario, as opposed to having cloudlet send them in a point-to-point manner, as explained in Section III-G. In this experiment, clients execute 95 operations on their local replica, following by migrating to a random remote replica and executing 5 remote operations, and then returning to their local replica. Fig. 9 shows the maximum throughput of each system in the two previously presented settings (Fig. 3). In these experiments, the timeout for both Bayou metadata messages and ENGAGE MF messages is set to 100ms, which, according to the previous experiments, is a good balance between visibility times and number of metadata messages. Additionally, we set the size of data objects to 2048 bytes, in order to better understand how the communication patterns of the considered protocols are affected when available bandwidth is limited.

The first relevant observation is that, in both cases, Saturn is unable to reach the throughput of the other systems. This happens due to two main reasons: i) the higher visibility times of Saturn (studied in the previous sections) force clients to spend more time waiting, when migrating between replicas;

ii) the lack of fine-grained dependency tracking in Saturn prevents replicas from applying remote operations concurrently; instead, updates must be applied in the order by which labels are received from the metadata service, further contributing to the increase of visibility times.

Efficiently routing the payload of update operations becomes an important aspect when comparing ENGAGE with Bayou in an edge setting, where edge replicas have access to much lower bandwidth than in traditional datacenter deployments, and the data partitions are complexly distributed. By setting up the metadata service of ENGAGE in an hierarchical way, such that internal nodes in the tree are the ones with the highest available bandwidth, we can leverage on it to propagate the payload of update operations. The benefits of this approach can be seen particularly in Fig. 9a, where the raw throughput of ENGAGE surpasses that of Bayou. In this figure, we can see that, while the other systems are limited by the bandwidth of edge replicas (that need to propagate each operation to multiple other replicas), ENGAGE is able to smartly use available bandwidth to increase its performance. Additionally, in a real world deployment, where the number of replicas could reach the hundreds or thousands, the benefits of this propagation scheme would become even more noticeable.

## V. CONCLUSIONS

Given that latency constrained applications are one of the main drivers for edge computing, offering 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 the use of vector clocks and metadata services to achieve *both* low visibility times and low remote operation latency, while leveraging on the hierarchical nature of edge deployments to optimize bandwidth usage, maximizing throughput. We show that, by using session guarantees, ENGAGE allows the programmer to fully exploit the application semantics, improving its performance, and outperforming systems based on full causal consistency. At the same time, ENGAGE avoids stalling remote updates due to false dependencies, offering low operation visibility times. Finally, ENGAGE is able to achieve these goals while reducing the amount of control messages that are required to operate to a negligible fraction of the total system traffic.

## REFERENCES


[1] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[2] D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *ICDCS*, Nara, Japan, June 2016.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM TOCS*, 9(3):272–314, 1991.

[4] M. Bravo, L. Rodrigues, and P. van Roy. Saturn: A distributed metadata service for causal consistency. In *EuroSys*, Belrade, Serbia, April 2017.

[5] E. Brewer. Towards robust distributed systems. In *PODC*, Portland (OR), USA, July 2000.

[6] B. Cooper, A. Silberstein, E. Tam, Ra. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

[7] P. Costa, P. Fouto, and J. Leitão. Overlay networks for edge management. In *NCA*, pages 1–10. IEEE, 2020.

[8] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *SOCC*, Santa Clara (CA), USA, October 2013.

[9] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *ACSC*, Queensland, Australia, February 1988.

[10] P. Fouto, P. Costa, N. Preguiça, and J. Leitao. Babel: A framework for developing performant and dependable distributed protocols. *arXiv preprint arXiv:2205.02106*, 2022.

[11] P. Fouto, J. Leitão, and N. Preguiça. Practical and fast causal consistent partial geo-replication. In *NCA*, pages 1–10. IEEE, 2018.

[12] C. Gunawardhana, M. Bravo, and L.s Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. In *ATC*, 2017.

[13] H. Gupta and U. Ramachandran. Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In *DEBS*, Hamilton, New Zealand, June 2018.

[14] H. Gupta, Z. Xu, and U. Ramachandran. DataFog: Towards a Holistic Data Management Platform for the IoT Age at the Network Edge. In *HotEdge*, Boston (MA), USA, July 2018.

[15] Z. Hao, S. Yi, and Q. Li. EdgeCons: achieving efficient consensus in edge computing networks. In *HotEdge*, Boston (MA), USA, July 2018.

[16] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, July 1990.

[17] J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *SOSP*, Pacific Grove (CA), USA, October 1991.

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4):360–391, 1992.

[19] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[20] J. Leitão, P. Costa, M. Gomes, and N. Preguiça. Towards enabling novel edge-enabled applications. *CoRR*, abs/1805.06989, 2018.

[21] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, Lombard (IL), USA, April 2013.

[22] C. Meiklejohn, H. Miller, and Z. Lakhani. Towards a solution to the red wedding problem. In *HotEdge*, Boston (MA), USA, July 2018.

[23] S. Mortazavi, B. Balasubramanian, E. de Lara, and S. Narayanan. Toward session consistency for the edge. In *HotEdge*, Boston (MA), USA, July 2018.

[24] S. Mortazavi, M. Salehe, B. Balasubramanian, E. de Lara, and S. PuzhavakathNarayanan. SessionStore: a session-aware datastore for the edge. In *ICFEC*, Melbourne, Australia, May 2020.

[25] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara. Cloudpath: A multi-tier cloud computing framework. In *SEC*, San Jose (CA), USA, October 2017.

[26] F. Nawab, D. Agrawal, and A. El Abbadi. DPaxos: managing data closer to users for low-latency and mobile applications. In *SIGMOD*, Houston (TX), USA, June 2018.

[27] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, Philadelphia (PA), USA, June 2014.

[28] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, pages 288–301, 1997.

[29] N. Preguiça, C. Baquero, and M. Shapiro. Conflict-free replicated data types (CRDTs). In *SSS*, Grenoble, France, October 2011.

[30] F. Ruget. Cheaper matrix clocks. In *WDAG*, Terschelling, The Netherlands, September 1994.

[31] K. Saito, A. Mikami, K. Ariga, H. Yasutake, S. Kimura, and H.e Hane. Case studies of edge computing solutions. *NEC Technical Journal*, 12(1), 2017.

[32] M. Schneider, J. Rambach, and D. Stricker. Augmented Reality Based on Edge Computing Using the Example of Remote Live Support. In *ICIT*, Toronto, Canada, March 2017.

[33] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, Oct 2016.

[34] D. Terry, A. Demers, K. Petersen, M. Spreitzer, Ma. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, Austin (TX), USA, Oct 1994.

[35] S. Zhang, W. Zeng, I-L. Yen, and F. Bastani. Semantically enhanced time series databases in IoT-Edge-Cloud infrastructure. In *HASE*, Hangzhou, China, January 2019.