TÉCNICO LISBOA

ENGAGE: Session Guaranties for the Edge

Miguel Leitão Belém

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. Hervé Miguel Cordeiro Paulino

November 2020

Acknowledgments

I would like to thank my parents for all the unconditional support.

I am grateful to all my friends, colleagues and Leonor for all the the time spent reading my work and giving feedback about topics that needed to be improved.

I am profoundly grateful to Taras and Nivia for all the time spent discussing details regarding this work and also for all the help given implementing and essentially evaluating this work.

Lastly, i would like to thank to my thesis advisor prof. Luis Rodrigues for giving me the opportunity of working with him and guiding me through all this course.

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.

Abstract

One of the main goals of edge computing is to support latency-constrained applications. For applications that need to access information in the cloud, 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 thesis 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.

Keywords

Edge Computing, Session Guarantees, Causal, Consistency, Low Latency

Resumo

Um dos principais objetivos da computação na periferia da rede é o de oferecer suporte a aplicações com necessidades de baixa latência. Para aplicações que precisem de aceder a informação localizada na nuvem, os baixos tempo de acesso podem ser atingidos caso os nós localizados na periferia repliquem os dados em questão. Como estes nós são limitados quanto aos recursos disponíveis, a replicação total é inviável. Deste modo, qualquer serviço de armazenamento na periferia precisa de ser capaz de oferecer suporte para replicação parcial. Além disso, na maioria dos casos, é necessário oferecer suporte a modelos fracos de coerência, evitando deste modo a latência e a sobrecarga associadas aos algoritmos que impõem a coerência forte. Neste contexto, as garantias de sessão são uma ferramenta poderosa que simplifica o design de aplicações focacionadas para a periferia. Infelizmente, os mecanismos normalmente usados para impor as garantias de sessão, como relógios vetoriais, funcionam mal quando combinados com a replicação parcial. Esta tese apresenta o sistema ENGAGE, um sistema de armazenamento que combina o uso de relógios vetoriais e serviços distribuídos de propagação de metadados para oferecer suporte eficiente para garantias de sessão num sistema construído de modo a suportar replicação parcial.

Palavras Chave

Computação na Periferia da Rede, Garantias de Sessão, Causalidade, Coerência, Baixa Latência;

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Contributions . . . 3

1.3 Results . . . 3

1.4 Research History . . . 4

1.5 Structure of the document . . . 4

2 Related Work **5**

2.1 Edge computing . . . 6

2.1.1 Model, Challenges and Limitations . . . 6

2.1.2 Edge Storage . . . 7

2.2 Consistency Models . . . 7

2.3 Event Ordering . . . 9

2.4 Implementing Causality . . . 11

2.4.1 Techniques Used to Enforce Causality . . . 11

2.4.2 Support for Partial Replication . . . 11

2.4.3 Dissemination Types . . . 12

2.4.4 Metadata Size vs False Dependencies . . . 12

2.5 High Availability Restrictions . . . 13

2.6 Supporting Different Models of Consistency . . . 14

2.6.1 Bayou [1, 2] . . . 14

2.6.2 Per-Key Session Guarantees on top of NuKV [3] . . . 15

2.6.3 SessionStore [4] . . . 16

2.6.4 Pileus [5] . . . 18

2.6.5 Tuba [6] . . . 20

2.6.6 Simba [7, 8] . . . 22

2.7 Causally Consistent Systems . . . 23

2.7.1 Lazy Replication [9] . . . 24

2.7.2 GentleRain [10] 25
2.7.3 Cure [11] 27
2.7.4 Saturn [12] 27
2.8 Comparison 29
2.9 Edge Applicability 30
3 ENGAGE **33**
3.1 Goals 34
3.2 Design 37
3.2.1 System Model 37
3.2.2 Metadata 38
3.3 Protocols 39
3.3.1 Performing Read and Write Operations 39
3.3.2 Applying Remote Updates 40
3.3.3 The Engage Extended Metadata Service 44
3.4 Example 47
3.5 Optimization 49
3.6 Correctness 49
4 Evaluation **51**
4.1 Evaluation Goals 52
4.2 Experimental Setup 52
4.3 ENGAGE vs Bayou vs Saturn 54
4.4 Benefits from Session Guarantees 55
4.5 Tolerance to Transient Partitions 57
4.6 Signaling Overhead 59
4.7 Discussion 61
5 Conclusion **63**

List of Figures

3.1 Limitations of Bayou and Saturn 35
3.2 Remote Visibility Latency and Remote Read Latency. 36
3.3 ENGAGE Architecture 38
3.4 Propagating Metadata Messages 48

4.1 Cloudlet placement (white) and broker network (blue) 53
4.2 ENGAGE vs Bayou and Saturn 54
4.3 Remote Latency with Session Guarantees 56
4.4 CDF of the Remote Visibility Latency and Remote Operation Latency. 58
4.5 Tolerance to Transient Partitions 58
4.6 Signaling Impact: ENGAGE vs Bayou 60

List of Tables

2.1 Comparison between the several systems. G,DC,P,T,TB stands for inter-datacenter, intra-datacenter, partition, tablet, table. M and MT stand for the number of data centers and number of modified tables, respectively . . . 31
2.2 Continuation of table 2.1 . . . 31

4.1 Parameters of the dynamic workload generator. . . . 52

List of Algorithms

1 Client c code . . . 39
2 Cloudlet n code . . . 41
3 Broker i . . . 45

Acronyms

VV Version Vector

VC Version Clock

FIFO First In, First Out

CDF Cumulative Distributed Function

IOT Internet of Things

WFR Write Follow Reads

MW Monotonic Writes

RYW Read Your Writes

MR Monotonic Reads

CAP Consistency, Availability and Partitions Tolerance

SLA Service Level Agreement

LST Local Stable Time

GST Global Stable Time

DT Dependency Time

NTP Network Time Protocol

PVC Partition Vector Clock

GSS Globally Stable Snapshot

MF Metadata Flush

1

Introduction

Contents

1.1 Motivation 2
1.2 Contributions 3
1.3 Results 3
1.4 Research History 4
1.5 Structure of the document 4

This thesis addresses the problem of replica consistency in the context of edge computing. It is assumed a scenario where data replicas are deployed on several cloudlets spread along a regional area, such as a city or a district. All these cloudlets might be connected to the cloud, exchanging information with it. Since cloudlets have limited resources, it is mandatory to support partial replication. It was aimed to design replica consistency mechanisms that are able to offer both low remote operation latency and low remote visibility latency for clients that access data using different types of session guarantees. To achieve this goal, it was combined the use of vector clocks and distributed metadata propagation services in a system that was named ENGAGE. This thesis describes the design, implementation, and evaluation of ENGAGE. Experimental results show that it can offer efficient support for session guarantees in a partially replicated edge storage service.

1.1 Motivation

Numerous applications of today that have clients running on the edge of the network rely on cloud structures for computation offloading and storage [13]. Unfortunately, the high network latency between clients and data centers can impair novel, latency-constrained, applications such as augmented reality [14]. 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 [15, 16]. This happens because strong consistency models, such as linearizability [17], require strong coordination among the replicas when updates are performed, which increases latency.

In order to provide clients with the best possible service, ENGAGE was conceived around the following key points:

- **Support for partial replication** - Given the fact that cloudlets are resource constrained, they are not capable of storing every data item needed to be accessed by clients of some application, as such, clients are likely to change the contacted cloudlet depending on their needs. Partial replication, besides requiring a proper policy to allocate specific data to each cloudlet (not covered in this work), has another challenge that needs to be overcame, namely the ability of a cloudlet not to be prevented from applying remote operations due to a dependency on an object not replicated locally, being able to achieve genuine partial replication.

  Systems based on vector clocks that adopt partial replication need to broadcast metadata to all other cloudlets, even if the latter does not locally replicate it, losing genuine partial replication. Instead of broadcasting metadata to cloudlets that do not replicate it, ENGAGE uses a metadata

service to propagate this information and attempts to piggyback on genuine data updates; piggybacking avoids the overwhelming cost of broadcasting every message to every cloudlet.

- **Combine low visibility latency with low remote operation latency** - Typically there are systems focused on providing clients with one of these features, ENGAGE focuses on providing both.
- **Support for client mobility** - Given the fact that this system was conceived for the edge model, it is likely that clients tend to change its location frequently, therefore it is important that ENGAGE does not force clients to remain attached to a given cloudlet. Instead, it is intended to reduce the time required for a client to move to a different cloudlet while preserving application specific consistency guarantees.
- **Support for weaker semantics** - Mobile clients demand very low response times. By supporting weaker models, ENGAGE is able to provide more desirable latency times, since weak semantics requirements are more flexible than the ones present in stronger ones.

1.2 Contributions

In this thesis three different solutions were implemented and compared using the same simulation environment. The resulting contributions from this thesis are:

- The conception of a system tailored for edge environment. It supports session guarantees [18] and successfully integrates low remote visibility latency from Saturn [12] and low remote operation latency from Bayou [1, 2]. Under certain workloads, ENGAGE is able to achieve better results than the latter systems. ENGAGE is also capable of tolerating transient partitions much better than Saturn and has a much lower signaling cost than Bayou.
- A novel mechanism of disseminating metadata messages through a tree of serializers, allowing to reduce the number of broadcast messages.

1.3 Results

After system's conception and evaluation against two other approaches, the following results were produced:

- An implementation of ENGAGE, in the PeerSim [19] simulator.
- A detailed evaluation regarding the performance, benefits from using session guarantees, capacity of tolerating transient partitions and signaling cost of ENGAGE when compared with Saturn and Bayou.

1.4 Research History

This work was performed in the context of a research project, named Cosmos, that aims at offering causal consistency on the edge network. My work builds on the work of Manuel Bravo [12], that proposed a metadata server for geo-replicated services, and of Nuno Afonso [20], that did a first attempt at adapting this service for the edge. Both systems enforce only causal-order, and have no support for session guarantees, a limitation that this thesis aims at circumventing.

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.

1.5 Structure of the document

This thesis is organized as follows. Chapter 2 starts presenting some background concepts such as the edge computing model, several semantics, how can events be ordered, restrictions to present systems with high availability; at the end of the chapter several systems are analysed and compared in order to understand which are the best features from them to implement in an edge scenario. Chapter 3 presents the system architecture and Chapter 4 reports results from an experimental evaluation that compares ENGAGE against Bayou and Saturn. Finally, Chapter 5 presents the conclusions and ideas for future work.

2

Related Work

Contents

2.1 Edge computing 6
2.2 Consistency Models 7
2.3 Event Ordering 9
2.4 Implementing Causality 11
2.5 High Availability Restrictions 13
2.6 Supporting Different Models of Consistency 14
2.7 Causally Consistent Systems 23
2.8 Comparison 29
2.9 Edge Applicability 30

This chapter starts by introducing *Edge Computing* concept 2.1. First, it is explained the model, challenges and limitations 2.1.1, followed by a brief summary about some Edge Storage Systems 2.1.1. After introducing edge, a brief analysis regarding consistency models is presented 2.2, followed by an explanation regarding the several ways of ordering events 2.3 and how some state of the art systems implement causality semantics 2.4. In 2.5 it is briefly analysed the restrictions imposed to high availability systems. In 2.6 and 2.7 relevant state of the art systems are analysed.

2.1 Edge computing

The *cloud computing* model uses large data centers that house huge amounts of servers. This model allows to provide high computing capacity at low cost, since there are fixed costs that can be shared by several users. For efficiency reasons, the number of data centers that materialize the cloud is necessarily reduced, which means that clients can observe high latencies when accessing servers, which in turn limits the type of applications that can benefit from it. In addition, the number of clients has been increasing with the proliferation of smart devices, capable of producing large amounts of information, such as televisions, appliances, smart vehicles, sensors, among others. This reality is called the *Internet of Things* (*IoT*). With this growth in the number of clients, problems regarding the use of bandwidth when accessing data centers appear, leading to the need of adapting the cloud model to the new challenges imposed by this reality.

2.1.1 Model, Challenges and Limitations

The *edge computing* concept emerged to face the challenges mentioned above. It is materialized by a network of servers, which are placed in geographically distributed points, close to the devices to which they provide services. These servers, sometimes called *cloudlets* [21] or *fog servers* [22], can provide services to devices with low latency and can collect and aggregate information generated on the Internet of Things before it is sent to the cloud. Servers may need to maintain replicas of data objects, which in turn raises the problem of choosing which guarantees of consistency should be supported when accessing this data. Unfortunately, in most cases, stronger consistency models can also induce greater latency in accessing them. For example, the linearizability [17] is a consistency model that ensures that when an operation ends, its results are visible in all replicas. This forces the replicas to coordinate themselves to execute the operations, which can introduce significant latency in the execution of the operations, and can even block them in unfavorable conditions of the network [23]. This can be an obstacle for edge environment, since one of the main motivations for this model is to offer services with low latency.

2.1.2 Edge Storage

Several works have addressed edge storage, but few have addressed the problem of latency when considering session guarantees. SessionStore [4] 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 [24], which is a hierarchical eventual-consistent object store built on CloudPath [15], a system that replicates application data on-demand. Because data is shipped on demand, clients can experience a large latency. Like us, FogStore [16] and DataFog [25] 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 requests are always served locally (furthermore, the consistency criteria supported by FogStore are not directly comparable with session guarantees). EdgeCons [26] and DPaxos [27] propose efficient consensus algorithms for the edge that are targeted at strongly consistent systems. In [28], 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 [30] 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.

2.2 Consistency Models

In a seminal work, Terry *et al.* [18] 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. Experience has shown that weak consistency makes application development difficult [11, 31]. 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.

Session guarantees can be implemented in any system that uses replication with weak semantics and that propagates data periodically, in a *lazy* way [9] in the background, and *batching* several messages in one. These guarantees can be offered without the system having to be rebuilt, that is, the idea is that the session guarantees are implemented by a new and independent component, which can easily be coupled to the base system without profound changes to it.

Session guarantees can be combined and work on demand, so there is flexibility in choosing them. A session is an abstraction for a sequence of operations (reads and writes) made by the client. Sessions do not correspond to atomic transactions, as they neither guarantee *atomicity* nor *serializability*, ensuring just a consistent view of data.

Typically, session's state is ensured by a *front-end*, which is also responsible for forwarding requests from clients to the different servers, abstracting them from this task.

The four session guarantees, which can be combined with each other, are as follows:

- **Read Your Writes** (RYW) - A client can only read from a server that has all its writes. A read does not necessarily reflect the last write made in the client's session, since it may reflect a value external to it, but it needs to be at least as recent as the last write made by the client in its session;
- **Monotonic Reads** (MR) - A read made in a session must be served by a server that has all the writes that have been seen by all reads made so far in the session by the client;
- **Write Follow Reads** (WFR) - A write made in the session is only installed in remote servers after all the writes seen by the reads made so far in the session;
- **Monotonic Writes** (MW) - A write is done on all servers only after all the writes previously made in the same session. This way, only the servers that have all the session's writes already installed are able to apply the newest one.

For a system to be able to ensure "Write Follow Reads" and "Monotonic Writes", it needs to be able to satisfy two properties:

- **Write Ordering** - When a server receives a write, it must be able to install it unambiguously after all the writes that already exist in its database;
- **Write Propagation** - The *lazy* propagation of information must be done so that the order of the write operations is maintained across all servers.

These two properties are mandatory since both guarantees influence the state of the data outside the session's scope.

When the four guarantees mentioned above are combined, *causality* [32] is reached. Causal dependencies between objects are determined by *happens-before* relations ($\leadsto$). In order to now if this relation is preserved, at least one of the following rules must hold [33]:

- **Thread of Execution** ($o_1 \rightarrow_i o_2$) - o_1 and o_2 are two operations executed inside the scope of the same thread (possibly by the same client), then $o_1 \leadsto o_2$ if 0_1 happens before 0_2.
- **Reads From** ($o_1 \mapsto o_2$) - if o_1 is a write and o_2 is a read operation that is reading the value of o_1, then $o_1 \leadsto o_2$.

- **Transitivity** - There exist an operation o' such that $o_1 \rightsquigarrow o' \rightsquigarrow o_2$.

Causality is not broken if two servers apply concurrent (not causally related) writes in a different order. To ensure that this does not happen, a stronger semantic is needed, called *causal+* [11, 34].

Some systems [5–7, 35] are able to provide *strong* consistency to its clients, such as *serializability* [36] or *linearizability* [17]. The *linearizability* requires that all writes and reads appear to have been done in the entire system instantly, between their invocation and response, forcing a read to always reflect, regardless of the server contacted, the most recent value. According to the *CAP* (Consistency, Availability and Partition tolerance) [23] theorem, it is not possible for a system to provide strong consistency while maintaining high availability in a partitioned network. As a way to circumvent this problem, many systems opt for weaker semantics like *eventual*, which only requires that the various replicas of the system eventually converge to the same state when the system stops receiving writes. This latter semantics became quite popular with the appearance of *cloud computing* paradigm, where consistency tended to be overlooked due to the scalability and availability of the system [37].

Another type of consistency model adopted in some systems [5, 6] is called *bounded staleness*. This type of semantics ensures that read values are delayed within a limit defined by the application or user. The delay is defined by a given time window *T* and the system must be able to guarantee that the read returns a value that is delayed by a maximum of *T* time units [38].

2.3 Event Ordering

One way to order different operations and ensure that they are processed in that order is to use *timestamps*. There are several ways to apply timestamps, which will all have an impact on the consistency to be satisfied and also on the performance of the system [39]. The assignment of timestamps to events is done using clocks, which can be of different types.

In [40], Leslie Lamport discusses logical and physical clocks, pointing out the advantages and disadvantages of each approach. Some systems also opt for a mixture of these two types, using a hybrid clock [41].

- **Logical Clock** - Consists in a scalar that is incremented sequentially for each operation performed. This type of clock has the particularity of capturing the *happens-before* ($\rightsquigarrow$) relation [40], however it is not able to capture the real order with which the events occur.

- **Physical Clock** - Servers need to use a clock synchronization protocol like *NTP* [42], making the skews between their clocks as small as possible, allowing all system components to be properly coordinated. This type of clock is one that we are familiar with, using it on a daily basis. This type of clocks, as long as they are properly synchronized, can order events according to the real time in

which they occurred. However, if the synchronization of the clocks is not perfect, it fails to capture the $\rightsquigarrow$ relation;

- **Hybrid Clock** - The hybrid clocks appears as the combination of the two types described above. They have a physical part to capture the events according to their real order and a logical part that allows them to capture the ($\rightsquigarrow$) relation.

Version vectors [9, 43, 44], also known as vector clocks [45], are a way to keep track of concurrent updates, this way it is possible to capture in a more fine-grained way dependencies between objects. 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. 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 violating the desired semantics.

Vector clocks are able to keep track of 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 [46] or, alternatively, a matrix clock [47]. Both approaches are extremely expensive and impractical on the edge.

Typically, a clock with greater precision tends to have a greater number of entries, allowing it to take less time to make operations that are concurrent visible, that is, they are less affected by false dependencies (section 2.4.4). False dependencies, or artifacts that result from the way the $\rightsquigarrow$ relations are captured by the system, consists in having two concurrent operations timestamped with clocks that indicate a potential dependency.

In the case of the GentleRain [10] system, in order to make a remote write visible, the server must wait for all others to send their scalar, which will have to be higher than the one associated with the operation. In the Cure [11] system, as a clock with one entry per data center is used, instead of waiting for a higher scalar from all servers, it is only needed to wait for the one from the data center where the write was initially made.

A more detailed clock, while typically allowing for greater accuracy, will consume more bandwidth when it is sent over the network compared to a scalar and also requires more space to be stored.

2.4 Implementing Causality

Systems focused on serving clients with causal semantics make some decisions regarding some aspects, namely: **1)** technique used to enforce causality; **2)** support for partial replication; **3)** way of disseminating data; **4)** presence of false dependencies.

2.4.1 Techniques Used to Enforce Causality

To satisfy causal dependencies, metadata must be kept. Depending on the technique used by each system to implement causality, the size of the metadata will vary. Systems analyzed in this thesis work are based on the following techniques:

- **Sequencer-based** - Solutions using this technique tend to rely on centralized components (*sequencers*), responsible for defining the order in which updates are installed. Lazy Replication [9] and "Per-key session guarantees" [3] systems rely on a sequencer per data center, responsible for receiving remote operations and adding them to a *log*. This *log* allows for remote operations to be processed as the local ones. Pileus, Tuba and Simba [5–7] also rely on sequencers, but instead of having one per data center they have one per portion of objects (*tablets*).

- **Background Stabilization** - This mechanism, unlike the sequencer, allows decentralization. It ensures that a remote operation is only made visible locally after being considered stable in other data centers. To implement this technique, there must be an algorithm that runs periodically in order to know the state of each center and, depending on that state, the operation may or not be able to be applied locally. This technique is used, for example, in GentleRain [10] and Cure [11];

- **Tree Dissemination** - This technique is based on a tree by which the various operations are propagated between the various data centers. The ordering of the data is done by the tree serializers in a hierarchical manner. These serializers are usually implemented on top of some of the servers in the data centers. There is no stabilization algorithm and the operation is made visible locally when it arrives. This technique is used, for example, in Saturn [12].

2.4.2 Support for Partial Replication

Besides supporting weak consistency memory models [15, 16], edge nodes must also support partial replication to unleash their full potential. Since edge nodes are resource constrained, full replication is infeasible. Therefore, any edge storage service needs to support this type of replication model. Partial replication is a way of increasing system's scalibility, since an update only affects a small amount of nodes.

Systems can take one of the following positions with respect to partial replication support:

- **No Support** - All system's replicas have a full copy of all data. Lazy Replication [9] opts for this replication model.

- **Non-genuine** - Each server only stores part of system's data base, however they need to exchange metadata regarding data not replicated locally. Gentle Rain [10] uses *non-genuine* partial replication.

- **Genuine** - Each server only operates with its subset of data and never receives metadata regarding objects not locally replicated. Saturn [12], by using its metadata dissemination service, is able to achieve *genuine* partial replication.

2.4.3 Dissemination Types

Different systems opt for different ways of disseminating information. The ones approached in our analysis are:

- **All-to-All** - The replica that receives client's request is in charge of disseminating it to the rest of interested replicas.

- **Master-Slave** - Data is divided into groups that can range from a single item to several. Each of these groups has a master replica, responsible for ensuring that all replicas that replicate the item apply the updates in the same order. All writes regarding an item that belongs to a certain group are forwarded to their master.

- **Pair-Wise** - This type of dissemination is used in systems that aim to restrict the amount of data on the network, as such, periodically two nodes start a synchronization procedure in which the sender node updates the receiver one with all updates that the last is missing.

- **Tree-Based** - This scheme relies on a group of serializers, organized as a tree. When a data center wants to propagate some update to its peers, a message containing the update is sent to the serializer that is connected to the data center. This message travels through the tree and is distributed to data centers that replicate the data item. By using this dissemination mechanism it is possible to achieve genuine partial replication and low remote visibility latency, while assuring that the network is not too congested as in an all-to-all scheme.

2.4.4 Metadata Size vs False Dependencies

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 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).

Distributed metadata services [12] 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 inform 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 while offering causal consistency.

Depending on metadata granularity, different *false dependencies* types may arise, namely: **1)** concurrent updates on the same object; **2)** concurrent writes on data items located in the same partition, shard, table or tablet; **3)** concurrent updates regarding data belonging to the same data center; **4)** concurrent updates regarding data located in different data centers.

Remote visibility latency is severely impacted by the type of false dependency, being the inter-datacenter one the most harmful one since data centers have high ping latencies, while the intra-datacenter one is not so problematic since servers inside the same data center are physically close, which is reflected in low ping latencies.

2.5 High Availability Restrictions

In [32], the requirements linked to an highly available environment are analyzed, in which *edge computing* is inserted, as well as the maximum guarantees that can be offered by this model.

A system is highly available if all users who communicate with any of its servers eventually receive a response without the server with which they communicate being blocked, waiting to receive some information, even if the network is partitioned and, consequently, some servers cannot communicate with each other.

“Monotonic Reads”, “Monotonic Writes” and “Writes Follow Reads” guarantees can be satisfied according to the high availability model, as long as a write is only made visible when its dependencies become visible on all machines. As such, users are never blocked due to their inability to find a server that is sufficiently updated to serve their needs, even in the presence of partitions. However, it is not guaranteed that the system will be able to make progress in the presence of network partitions if they last for a long time.

"Read Your Writes" guarantee can be violated even by postponing the visibility of write operations; for that, client just needs to do a read right after having performed a write (which was not soon made visible). To satisfy this guarantee it is necessary for the user to remain *sticky* (client's communicate with the same server, changing only when data is not replicated locally) to a given server, as so he will always write on a replica that is guaranteed to have all his history.

Since causality is given by the combination of the four session guarantees, it is possible to have a highly available system that provides causality, as long as the client remains *sticky* to a given server.

Stronger guarantees like *linearizability* cannot be offered, as the system may need to be blocked until it can respond, thus the causality in which the user is *sticky* to a server is considered the maximum guarantee that can be given for a system to be highly available.

2.6 Supporting Different Models of Consistency

The systems that support multiple consistency criteria will now be addressed. This systems allow clients and applications to manage which set of data items they want to access with a certain semantics. Systems such as Pileus and Tuba [5, 6] also allow clients to define an *SLA* with several semantics and time constraints, which permits systems to choose the most appropriate semantics depending on the state of the network at the time the request is received.

2.6.1 Bayou [1, 2]

Bayou is a weakly consistent storage system, conceived to operate on a mobile computing environment, in which the devices had low computational power and constantly faced network connectivity problems. Given the fact that the system was conceived for mobile computing, it was desirable to allow clients to switch between multiple replicas on demand, without the need of remaining *sticky* to one of them.

In order to reduce the inconsistencies that may result when clients access different replicas, Bayou provides session guarantees [18]. Bayou has the main objective of maximizing the system's availability, while only requiring occasional communication between the devices. In this system, each replica contains a full copy of the data base, as such, it is not capable of supporting *partial replication*.

The storage system at each replica consists in an ordered *log* of writes and data itself. Bayou's replicas are guaranteed to converge states during its *anti-entropy* [48] protocol, used to exchange information between system's replicas, which is guaranteed by its conflict detection and resolution procedures.

In order to detect possible data conflicts, namely Write-Write and Read-Write ones, Bayou uses vector clocks and timestamps. After detecting a conflict, it is used a merge procedure to solve it, which, in turn, can lead to writes reordering. To keep Bayou's replicas always available for new clients requests, conflicts do not lock or make some objects inaccessible, instead, the objects remains fully accessible

and the conflict resolution procedure is delayed. By never blocking part of the system, clients are always capable of reading old data, however this approach has the drawback that the system might be unable to make progress due to a cascade of unresolved conflicts.

Since each write might be undone and reordered due to the merge procedures, replicas need to have a way of realizing if a write operation is stable. Bayou system follows a *primary commit protocol* [49], in which it is selected a replica to act as a primary replica and coordinate the order in which writes are committed. It was chosen a primary instead of a secondary commit protocol due to the network connectivity restrictions.

Regarding structures needed by the replicas to exchange metadata, while respecting dependencies between them, each replica maintains a vector with one entry per replica. Each entry represents the last known stable update (and consequently discarded from its *log*) from each replica. When the anti-entropy protocol wants to propagate a committed write (in order to follow the *primary commit protocol*), it cannot just inspect the sender's Vector clock and checking if it contains the write, since the receiver might have the write in its *log* and as a consequence it is not reflected in its own Vector clock yet, as such, the sender needs to send all the committed writes that the receiver might be possibly missing. Given that, the sender starts by sending to the receiver all the committed writes that the latter is unaware of.

In terms of false dependencies, it would be majored by the most delayed or outdated replica contained in the *view* used by the *primary commit protocol*. The most likely type of false dependency would be the *intra-datacenter* one, however, if the view contained all replicas, the type would be *inter-datacenter*.

2.6.2 Per-Key Session Guarantees on top of NuKV [3]

This system is built on top of NuKV, an eBay key-value store focused on offering an highly available service for its clients. NuKV offers eventual consistency for replication amongst different data centers and relies in the Raft protocol [50] for replication in each partition of each data center. In order to remove inconsistencies intrinsic to the low consistency models such as the eventual, this work is focused on offering session guarantees [18] but working in a per-key way. Besides offering higher consistency semantics, authors proposed a way of circumventing the *slowdown cascades* and *delayed writes* problems.

- **Delayed Writes** - In a seminal work, Terry *et al.* [18] have introduced the notion of *session guarantees* and also proposed a way of implementing them in a real system. According to Terry approach, a client may get blocked in a write operation until the server receiving the client's operation collect all dependencies needed. In this work, the client never blocks due to a write. This is made possible through the use of *hybrid clocks* [41]. This type of clocks conjugate a logical scalar with a psychical timer, as such it is always possible for a server to mark each write operation with an unambiguous

hybrid timestamp, this way, writes are always instantly accepted and clients only need to possibly wait when performing read operations.

- **Slowdown cascades** - Partitioned systems supporting semantics stronger than eventual are impacted by this problem. Waiting for an update to be considered stable across all systems partitions might dramatically delay its visibility and, in the worst case scenario, a single delayed or disconnected partition is able to create a cascade of slowdowns. To circumvent this problem, supported session guarantees only depend on replicas contained in the data center partition that contains the object being written, avoiding cross partition communications.

The system's topology consists in a set of several data centers, each one having a full copy of all the data base. Each data center is divided into partitions, each of which ensures fault tolerance using a set of replicas that run the Raft [50] protocol between them. Each group that applies the Raft protocol elects a leader on a rotating basis, who ensures that all members apply the operations in the order indicated by him. All writes in each group are directed to their leader, acting as the group's sequencer.

Each client keeps two matrices, one dedicated to the read set and the other to the write set. Matrices have one entry for each partition in each data center. Each client also has two scalars, corresponding to the hybrid clock for the last write and read operations.

Each server maintains a vector with one entry per data center. Each entry refers to the last write made at the remote center, propagated to the server. Two scalars are also maintained, one referring to the hybrid clock of the last operation consumed on the server, and the other referring to the physical clock used to assign the hybrid clock in each operation [41].

Read operations are blocking, and the client, depending on the guarantee he wants to satisfy, sends at least one vector that corresponds to a vector (line) in one of his matrices. The vector refers to the state of the same partition in each center. The server blocks if the vector stored is not more recent then the one sent by the client.

Regarding write operations, which, as previously said, are not blocking, the client only sends the scalar referring to the last consummated read or write and the server makes sure that the timestamp created for marking the operation will always be higher than any other generated by it and also higher than the one sent by client, as explained in [41].

Since the session guarantees work by partition, it can be considered that the granularity of the false dependencies of this system is by partition.

2.6.3 SessionStore [4]

Systems designed on top of the Edge layer are focused on providing an highly available service to its clients, as such, eventual consistency is the model typically provided by them. As already described in

2.1, this semantic becomes insufficient for clients who are not *sticky* to a replica.

To circumvent this problem, SessionStore provides *Session Consistency* on top of eventually consistent replicas. *Session Consistency* is enforced by grouping related data accesses into a session and using a session-aware reconciliation algorithm to reconcile only the data required when clients switch replica.

By using this approach, SessionStore is able to reduce the amount of data that needs to be shipped before serving a client, however it incurs in an higher latency since data is shipped on demand.

SessionStore is designed for applications running on top of several edge nodes with limited resources, this way it supports partial replication and replicates data only on demand. In order to enforce Session Consistency, this system uses a *reactive* approach instead of a *proactive* one. This way *session consistency* is only ensured when clients switch replicas.

By using a *reactive* approach, data needs to be less frequently propagated across the network. Authors argue that an active approach is only beneficial in a scenario where the number of edge devices is so high that is not feasible in terms of needed resources.

The technique used by SessionStore to identify data that is needed to propagate from a replica to another consists in a SQL-like query. This approach has a low storage requirement since a simple query is able to identify many data objects.

Each client has a session; objects retrieved by the queries performed by a client are buckled in its session. When the client changes the replica that he is communicating with, new versions of the objects contained in its session are broadcast to the newest replica.

SessionStore is based on PathStore [24], which is a hierarchical eventual-consistent object store built on CloudPath [15], a system that replicates application data on-demand. The system's implementation is made on top of Cassandra [51], since it runs on top of almost every device, suiting edge requirements.

When a client performs a query against a replica, a recursive mechanism for fetching data is triggered. Nodes are organized hierarchically with cloud representing the root and weaker nodes the leafs. The client is typically near one of the leafs, as such, the request is directed to him. After receiving the client's query, that replica fetches requested data from its parents, which is a recursive process until reaching the root cloud node. Each replica caches the queries that pass through it. When new data is introduced in the system, the replica's *pull and push daemons* are responsible for making sure that the objects introduced reach replicas whose queries fall on them.

Regarding false dependencies, SessionStore clients are just dependent on data contained in its session, which in turn can be classified as *intra-datacenter* considering the edge devices that are close to each other as part of the same data center.

2.6.4 Pileus [5]

The Pileus system, unlike [3, 18], allows clients to define a *Service Level Agreement (SLA)* per request. Each *SLA* consists of a list of priorities, where each entry is called *subSLA*. The first *subSLA* is the one with the most utility, that is, it is the one that the client wants most to be fulfilled, the rest have progressively less utility. Each one has, besides the associated utility, the desired semantic and the maximum latency that the operation must take.

When *SLAs* are used, the client is able to define more than one possible semantics for the same operation, and the choice of which one is satisfied is made by the system according to the network's state when the request is issued.

The system's architecture is divided into the following components:

- **Storage Nodes** - Responsible for storing the data and are divided into *primary* and *secondary* types.
- **Replication Agents** - They are found next to the storage nodes, guaranteeing replication when passing information between them.
- **Monitors** - They are with the client and are responsible for making estimates of the state of the various storage nodes.
- **Client Library** - Each client has his own, and its objective is to maintain his state, keeping the various operations made by him. With the client's state, together with the SLA sent, this component will be responsible for calculating what is the minimum response (with the lowest timestamp) that it is acceptable for the client to receive. This timestamp corresponds to the metadata exchanged between client and server and it is a scalar. It is through this library that the client interacts with the Pileus system.

The Pileus system consists of a set of $\langle key, value \rangle$ pairs that can be grouped into tables, each of which is an isolated unit, in order to present its level of replication independently of the remaining. Each application can create one or more tables to store its data. The tables can grow infinitely and, to contemplate this case, they can be divided into *tablets*, each one will have a subset of the $\langle key, value \rangle$ pairs of the table.

Each *tablet* is replicated in several places. Some of the nodes that replicate the *tablet* coordinate to materialize an abstraction called *primary node*, which is responsible for receiving all writes that clients perform about any of the keys stored in it. This entity works as a whole, so the clients sees it as if it were a single node. *Primary Node* orders the various writes and applies them in the same order to all its nodes. This way, possible conflicts related to concurrent writes on the same key are avoided. As such, this system uses a sequencer technique by *tablet*.

The nodes that make up the *primary node* are few and distinct for each *tablet*. There are more nodes replicating each *tablet*, which are called *secondary nodes*, and which are used only to process read requests. The latter, unlike the primaries, may exist in a greater amount, so, most likely, they will be physically closer to the clients, however they will not always have the most recent data, as such, they can be used to satisfy *subSLAs* with weaker consistency semantics and with low latency requirements. The *primary nodes* propagate the data asynchronously to the *secondary nodes*.

The monitoring nodes periodically collect metrics, both related to latency (measurement of *RTTs*) and data state (the highest timestamp presented on the node reflects its state) in each of the storage nodes. With this information, monitoring nodes are able to make a prediction of each node state and consequently choose the node that will best serve the client.

The various types of supported semantics, as well as the way each monitoring node makes the choice of which node to forward the request to, are:

- **Strong** - The request is always forwarded to the primary node or to a secondary node that is known to be in the most recent state.

- **Read Your Writes and Monotonic Reads** - The state kept by the client library (namely, the minimum acceptable timestamp) is compared with the most recent read timestamp from the server. If the server has a higher or equal timestamp, it can serve the request.

- **Bounded Staleness** - It is checked if a write operation on a certain key was performed in the time window defined by the current physical time and the bounded staleness value. If there is any match, it is checked the logical timestamp marking the operation. The server is able to serve the client's request if the operation's timestamp is equal or greater than the minimum acceptable timestamp calculated and sent by client.

- **Causal** - As writes are all done synchronously and the timestamps are generated sequentially, the verification is done in the same way as for the guarantees of "Read Your Writes" and "Monotonic Reads";

- **Eventual** - The key value of any server is read without any restrictions.

After the prediction of each node's state has been made, an utility value is assigned to each of them. Then, each of the *subSLA*, that also has an utility value, is analyzed. The server's choice is made by multiplying the two utilities and checking which server has the most fulfilling one. Thus, the most desired *subSLA* for the client (first entry of the *SLA*) is not always chosen, it all depends on the state and latency linked to each node when the request is issued.

2.6.5 Tuba [6]

Tuba System tries to solve a Pileus' [5] problem of being difficult or even impossible to choose a system configuration (location of nodes and synchronization frequency between primary and secondary nodes) that serves all clients optimally. Pileus is then extended so that it is possible to self reconfigure according to users' needs. In order for this increment to be done, it is necessary to have another component, called *configuration service*, whose functions are to periodically reconfigure the system in order to maximize its usefulness and inform clients of this reconfiguration. For this to be possible, the clients must periodically inform system about its latency when communicating with each node, as well as which *subSLAs* are satisfied and which are not. The steps performed by the *configuration service* are:

1. **Constraints** - As the system always makes a *greedy* choice (it always tries to maximize its utility), it can for instance choose to always add new nodes and this is not desirable at the expense of the system growing infinitely. As such, the programmer can apply some restrictions, such as a maximum number of replication nodes.

2. **Cost model** - Each possible reconfiguration has an associated cost, which takes into account the following factors to be calculated: i) cost of saving a *tablet* on a given node, ii) cost of doing a read or write and iii) the frequency with which the secondary nodes synchronize with the primary ones.

3. **Selection** - The reconfiguration is chosen based on: i) the satisfaction of the restrictions, ii) the associated cost, iii) the latency metrics collected and iv) the satisfied entries of *SLAs* (clients send this information directly to *configuration service*).

The reconfigurations that can be made are:

- **Adjust the Synchronization Period** - This measure does not affect neither writes nor reads with strong consistency requirements, as both are directed at the primary nodes. If the periodicity with which the synchronization is done is reduced, it is possible to sometimes alleviate the secondary nodes, which means that clients physically close to it can get responses with lower latencies, although possibly with less updated data. The main strengths of this measure are that it has a low cost when compared to adding, moving or removing a replica and does not affect the client in any way, as its view over the system is preserved.

- **Add a Secondary Node** - If a peak of accesses is detected in a certain area of the globe, it may be advantageous to place a secondary replica there. *Configuration service* launches a task responsible for copying data directly from a primary node, making the new one visible only when the copy is finished. Clients are not directly affected by having an outdated view of the nodes as the old one is a subset of the new one. Since a primary node is being overloaded when making the copy, strong writes and reads may be temporarily slower.

- **Remove a Secondary Node** - When the peak of accesses reaches the end, in order to reduce costs, the *configuration service* may choose to remove the secondary replica previously placed. In this scenario, the clients' view of the system is already directly affected as he can try to communicate with a node that no longer exists.

- **Change Primary Node** - A node is created in the new location, which receives data from the primary in the background. Until the state of the two nodes does not stabilize, the new one only receives write operations from the clients, because since it is out of date it cannot serve strong reads. When the state of the two nodes converges, the old one is deleted. This reconfiguration directly affects the clients' view of the system in a similar way to removing a secondary node. To ensure convergence between both nodes, the system reaches a point that temporarily blocks write operations.

- **Add Primary Node** - Adding a primary node makes writes slower and strong reads faster. The procedure is similar to the change of primary node, however the old one is not deleted.

Since the *configuration service* can change the structure of the network, clients need to know where the primary and secondary nodes are located. To increase overall performance, clients maintain a cache of the network configuration, avoiding unnecessary communications with that component. Therefore, special care must be taken to ensure that clients do not make strong read or writes on a non-primary node. To avoid these scenarios, clients can operate in two different modes:

- **Slow Mode** - It is the default state for clients. For a strong read, client does the same normally and at the end checks if the node from which he read is still primary. For writing, clients need to acquire a non-exclusive lock on the *tablet* they want to write on and only then can perform the desired operation. The *configuration service*, in order to modify the *tablet*, needs to acquire the same lock, which can fail if any client already has it. The slow mode does not prevent the client from writing, it just forces him to acquire a lock, which makes the write to be done more slowly.

- **Fast Mode** - Clients have a *lease*, during which they are sure that the *configuration service* will not change the state of the network. It is this *lease* that allows them not to have to consult the *configuration service* in each operation and consequently to have a better performance. For the client to remain in this state, the *lease* must be renewed. When *configuration service* begins to reconfigure the system, it signals this event, preventing renewals of *lease* and forcing clients to switch to slow mode.

This system has the drawback: if the number of storage nodes is high, the *configuration service* will take considerable time to choose the reconfiguration. For this scenario, techniques to prune the search space should be applied.

2.6.6 Simba [7, 8]

This system has two components, one more focused on what happens on the user's mobile device [7] and the other contemplates the interaction between the device and the cloud [8].

The system's authors started by testing, from the user's perspective, several applications used on mobile devices, in order to learn how they deal with the various types of data consistency and granularity. Two sources of problems have been identified: i) the application programmer uses existing systems to synchronize with the cloud and problems arise because they are not tailored to suit the particular needs of the application (they are rigid in terms of supported semantics, or ignore dependencies between tabular objects and data), or ii) try to create from scratch all the synchronization mechanisms and problems arise from the complexity of this task.

The problem of ignoring dependencies between tabular data and objects sometimes cause an object to be received (for example, a photograph) and not its tabular data (for instance, the metadata of the photograph).

To overcome these problems, Simba was conceived. This system provides a simple *API*, which can be easily used by programmers when creating applications. This system therefore consists of a synchronized storage mechanism both in the cloud and on the mobile device.

Simba's granularity is by *sTable* (Simba Table), thus being able to support dependencies between objects and tabular data, one of the problems mentioned above. The data belonging to an *sTable* will all have the same degree of consistency, which means that the system cannot adapt the semantics offered by request and depending on the state of the network, contrary to what happens in [5, 6].

Each entry in *sTable* will contain data that is dependent on each other, whether they are objects and/or tabular data, as such false dependencies can be classified as being per *sTable*. Each application can have several *sTables* and each can have a different degree of consistency. The reason why there is no consistency per entry is the fact that typically mobile applications already tend to group data that they want to have the same degree in the same *SQL* table. Keeping this principle, the portability of applications to Simba is facilitated.

Each *sTable* can have the following semantics:

- **Strong**
  - **Writes** - They can only be done when the mobile device is online. When the device is offline for a long time and resumes the connection, it has to perform a procedure that allows it to be in the most updated state, and only afterwards it can accept write operations. Writes are always done first in the cloud and only locally when there is confirmation message from the cloud.
  - **Reads** - If the device is online, the data it has is always the most recent, as such reads

are always performed locally. When the device is offline, although the data is no longer guaranteed to be the most recent, local reads are still possible. This is a relaxation that allows a better overall performance of the system.

- **Causal**
  - **Writes** - Always local, even when the device is online. While strong writes are synchronous to avoid conflicts, causal ones are asynchronous, thus conflicts can occur and might be resolved automatically or with the client's assistance.
  - **Reads** - Always local, even when the device is online.
- **Eventual**
  - **Writes** - Always local, even when the device is online. Like causal writes, conflicts can arise due to their asynchronism, however they are always resolved automatically according to a *last-writter-wins* semantics.
  - **Reads** - Always local, even when the device is online.

The various accepted semantics work only *per key*. Each table sequentially generates a timestamp for each write made on each of its entries.

Data can flow in two directions:

- **Server notifies client** - This operation is called *Downstream Sync* and occurs when the server has new data regarding tables that the client is interested in. Whenever there is a strong write, the server immediately executes a *Downstream Sync* for all clients interested in the table. This will allow strong reads to later be taken locally on the client's mobile device. In the case of *Downstream Sync* referring to only one operation (in the other systems the analysis is carried out operation by operation), the client only needs to exchange with the server the highest timestamp of the table where the written data is inserted.
- **Client notifies server** - This operation is called *Upstream Sync* and occurs when a client wants to write. In order for a client to perform an *upstream sync* of an operation, he just needs to send the current timestamp of the *row* that he wants to modify and the new value he wants to write.

2.7 Causally Consistent Systems

As explained in section 2.5, causality is the strongest attainable semantic that suits *edge computing* model. As such, several systems focused on providing this semantic were studied and its main features are shown below.

2.7.1 Lazy Replication [9]

This system has a message propagation mechanism between replicas, which guarantees causality and some stronger semantics, it also considers that all nodes have a full copy of all data. It is based on the assumption that the network is not reliable, that is, it can delete, duplicate and randomly delay messages.

Messages are propagated between replicas using an epidemic protocol (*Gossip*), allowing to reduce network congestion. Clients' requests can be sent respecting the following semantics:

- **Causality** - The client maintains a list of dependencies which, together with his request, he sends to the server with which he communicates. This list will define what causally precedes the client's request.

  This list of dependencies consists of a vector or *label*, with one entry per node. For a read operation, the *label* and the key corresponding to the object to be read are sent. When the server receives the client's request or another message that has been passed through epidemic propagation from another server, it waits for the list of dependencies to be installed locally and only then returns an answer to the client. For the write operations, the value to be written is also sent by the client.

  Each server needs to maintain a *label* with one entry per server and needs to create and save, together with the data, the timestamp assigned to each operation.

  This system does not consider data centers, where data can be replicated and partitioned through multiple nodes, hence the *label* has an entry for each server. However, transposing to a multi-machine replication scenario using data centers, the *label* would have one entry per data center or partition. The most direct way to ensure causal consistency for this new scenario would be to adopt a sequencer per data center, where a node per data center would be elected as leader, thus being responsible for ensuring that all other nodes in its data center apply the operations in the same order.

- **Forced Updates** - Causal operations can be used for reads and writes, while forced operations work only for writes. These operations are stronger than the previous ones, because while in the causal ones it is just needed to ensure that the operation is installed after all the elements of the list of dependencies, here it is also needed to guarantee total order of all forced operations that occur and propagate that information to all replicas, which is achieved through the use of an extra replica that will mediate the total ordering of these operations. The replica responsible for carrying out this order is elected on a rotating basis.

  This type of operations is useful, for instance, in scenarios where it intended to prevent two competing clients from being able to register with the same username. The clients *A* and *B* both start

a forced operation, the extra replica will order the two operations and only one of them will be able to choose the name. The satisfaction of this type of operations does not interfere with causal ones occurring simultaneously, since there is a replica dedicated exclusively to serve the forced ones.

- **Immediate Updates** - As with forced operations, these are also used only for writes, however they are even stronger. Unlike the previous ones, which only guarantee total order between forced operations, here It is intended to guarantee total ordering against all external events, regardless of the chosen semantics. As such, there is no need to provide a list of dependencies as the list would consist of all the operations done so far.

  This type of operations is useful in cases where it is wanted an update to be seen immediately (before any update that can be done later) on all replicas. For instance, to ban a user from the system, it is needed to ensure that regardless of the replica he communicates with after being banned, he is unable to access the data again. By only using forced operations it would not be able to guarantee this, since causal reads are never blocked.

The latter two operations are stronger than the first, therefore, its excessive use leads to a degradation in the overall performance of the system.

Although the system provides three types of semantics, each key is locked to the semantics used in the first operation on it, so the client cannot, for the same key, vary the semantic on demand. The system is not able to adapt the chosen semantics to the state of the network at the moment.

Assuming the multi-machine scenario where each node is seen as a data center and consequently the vectors work with one entry per data center, clients are subject to *intra-datacenter* false dependencies.

2.7.2 GentleRain [10]

GentleRain is a geo-replicated system that aims to provide causality, however it tries to bring the throughput as close as possible to what is supported by eventual consistency (very high). For this to be possible, the size of the metadata used to satisfy the causal dependencies is as small as possible, constituting only a scalar.

In this system there is no partial replication since each data center has a full copy of all data. The fact that only one scalar is used to satisfy the causal dependencies means that a remote operation takes longer to be made visible, since it is necessary to be stable in all data centers before being visible.

Each data center is divided into partitions, where each is responsible for storing a portion of the data. The portion of data held by a partition is disjoint from all other partitions in the same data center.

The protocol used by the system marks all the writes with the physical clock (*timestamp*) of the server where it was made.

Each server maintains a vector and each entry corresponds to the timestamp of the last write made on the same partition but from a remote data center. Each server also has a scalar named *Local Stable Time* (*LST*) which corresponds to the lowest value of the vector and yet another scalar named *Global Stable Time* (*GST*), representing the smallest *LST* of all partitions in a given data center.

Servers in the same data center periodically exchange their *LSTs* in order to compute *GST*. Since this operation has to be done periodically and the number of servers can be quite large, making a simple propagation of messages is too expensive and it is not scalable, so a tree-based dissemination technique is used. The servers that represent the leaves of the tree send their *LSTs* to the parent node, which will see which is the smallest and give that *LST* to its parent. This procedure is done until the root of the tree is reached, which will choose the smallest *LST*, now becoming *GST*. The tree root propagates this value to its children, and the algorithm ends when the value reaches all the leaves. Unlike the sequencer technique, this method of global stabilization allows decentralization, increasing the throughput. However, the fact that a stabilization algorithm has to be performed periodically postpones the visibility of remote operations.

Each client stores the largest timestamp he has seen so far in the session, called *Dependency time* (*DT*) and the largest *GST* of which he is aware.

Write operations are marked with the value of the physical clock of the server where they were made and then are concatenated with the identifiers of the partition and the server, which allows the existence of a total order between all operations. The protocol used by this system ensures that local writes are made visible immediately, while remote ones only are when their value is less than *GST*, meaning that it is considered stable. In the background and asynchronously, writes are propagated to remote partitions.

To perform a read, the client sends his *GST* and the key he wants to read. If the server's *GST* is lower, it updates it with the client's and asks the remaining servers to send the missing operations to it. If it is higher, it will choose the highest key value that is stable, that is, that does not have a timestamp higher than *GST*. The server returns the value, its timestamp and *GST* to the client. By having this information, client will be able to update his *DT* and *GST*.

For write operations, the client sends his *DT*. The server will wait until the received *DT* is less than the value of its physical clock, then updates its vector entry with the value of its physical clock. Subsequently, the tuple $\langle value, timestamp \rangle$ is added to the value chain associated with the key. Finally, the timestamp assigned to the operation is returned to the client and he updates his *DT*.

Physical clocks only need to be loosely synchronized, and can even be replaced by logical ones. Still, GentleRain uses physical clocks to avoid a problem that may occur when logical clocks are used: if there is a large disparity in the access rate observed at different servers, logical timestamps diverge quickly and this has a negative impact on the visibility latency. Given that the *GST* always takes into account the minimum value of all *LSTs* data center, all transactions are forced to wait for the node with

the lowest access rate.

Since GentleRain only uses a single scalar, clients need to deal with *inter-datacenter* false dependencies.

2.7.3 Cure [11]

This system provides high availability and performance while ensuring causal+ consistency. Similar to GentleRain [10], it considers that each data center has a full copy of the data.

Each server has a physical clock, which is only loosely synchronized with the ones in the rest of the partition, for instance by using the NTP [42] protocol. In addition to the physical clock, it also has two vectors, each with one entry per data center. One of them has the name of *Partition Vector Clock (PVC)* where each entry concerns the same partition but from a remote data center and stores the physical timestamp for the last write coming from it. The other is *Globally Stable Snapshot (GSS)* and indicates the most recent (stable) state common to all data center partitions.

The various partitions of a data center, in a manner similar to the one used by GentleRain, exchange their *PVC* and calculate the *GSS* which will consist of the minimum *PVC* common among all.

Unlike GentleRain, whose global stabilizer (*GST*) is a scalar, requiring the operation to be expected to be stable in all data centers before making it visible, Cure has the *GSS* which, as previously said, is a vector, allowing us to just wait for the write to be stable in the data center where it was made, which in turn allows clients to face only *intra-datacenter* false dependencies unlike GentleRain that faces *inter-datacenter* ones.

2.7.4 Saturn [12]

Saturn is a metadata service that can be easily placed on top of a geo-replicated system.

The purpose of this system is to orchestrate the visibility of data between the various data centers around the globe, so that causality can be respected.

This system uses metadata of constant size, more specifically a scalar called *label*. The fact that metadata presents always a constant size allows Saturn to maximize the system throughput, unlike what happens in Cure [11], where a vector is used. Besides maximizing throughput, Saturn also aims to minimize the time needed to make remote data visible.

Saturn supports genuine partial replication, allowing data centers to never depend on data not replicated locally. By allowing partial replication it is possible to reduce the number of false dependencies that may exist, since a data center never needs to wait for data not replicated locally to make certain operations visible, as such clients face *intra-datacenter* false dependencies.

This system only manages the metadata. Data objects are transmitted between the data centers in a transversal way to the system, so that Saturn can withstand very high loads of writes.

Saturn metadata service consists of a set of *Serializers* that are organized in a tree topology, with data centers as leaves. *FIFO* is assured between tree links. *Serializers* are placed in strategic places in order to satisfy the serializability requirements of each data center as much as possible. The metadata is then propagated from the data center where the operation is performed to the centers that also replicate it. The path chosen to deliver the metadata should take approximately the same time as the data object transfer protocol, which takes place directly between the two data centers. By approaching the time when data and metadata arrive, it is possible to reduce false dependencies.

Let's assume the scenario where there are two concurrent writes (*A* and *B*) in two different data centers (*DC1* and *DC2*). These writes are both replicated by another data center, *DC3*. Saturn must order *A* and *B* and deliver to *DC3*. Let's assume that the object of *A* takes about 12ms to be transmitted and the one of *B* takes about 2ms. If the order *AB* is chosen, *B*, although it is not causally related to *A*, will only be applied afterwards and will have to wait 12ms, when in fact it would not be necessary; it was enough to choose the configuration *BA* and after 2ms *B* can be applied. A *label* delivered after the respective object also introduces false dependencies, which should be avoided. In order to try to synchronize the arrival of the object with the *label*, Saturn may introduce strategic delays in *label's* propagation.

Each data center has the functions of: i) generating a *label* for each write done by the client, ii) provide the *label* properly ordered to Saturn metadata service (Saturn assumes *linearizability* in each center) and finally iii) send the objects to the other data centers that replicate them. Saturn receives *labels* and when it delivers them to interested data centers, it delivers them respecting causality, however, depending on the center, different serializations can be applied, whichever is more advantageous for the same.

If Saturn fails, *labels* remain to be sent from one center to the other as they always go piggybacked in data objects. However, the optimization (serialization tailored for each data center) offered will be lost, thus there will be false dependencies that unnecessarily delay the visibility of remote operations.

Each client is sticky to a data center, changing to another only when the old one fails or does not replicate the data item that clients needs to access. Each client maintains a *label* that is updated whenever they write, or read a value with a more recent *label* than the one stored by him.

Saturn guarantees causality because:

- Each client must remain sticky to a data center before performing any type of operation on it. The client, through the situations mentioned above, can perform a migration to another data center. This migration involves transferring its entire history from one center to the other, ensuring that the new operations that it will carry out on the new center respect causality.

- If some *serializer* fails, the data centers will end up being notified and the tree topology will be modified. To speed up the transition process from one topology to another, the system already has some pre-computed alternative topologies.

- If the Saturn system fails completely, as the *labels* are always piggybacked in the objects, the data centers are still able to apply the operations in respect with causal order, however there is no optimization, that is, for an operation to be made visible, it will have to wait to receive a *label* greater or equal from each of the remaining data centers that replicate the object.

2.8 Comparison

In this section, the various systems analyzed so far will be compared.

On the one hand there are systems focused on serving only eventual consistency as Dynamo [52], on the other hand there are systems focused on providing strong semantics, such as Spanner [35]. These two extremes are not desired, the first because it is possible to provide stronger semantics and the system remains highly available [32], the second because semantics like strong consistency or *linearizability* cause high latency, which is not tolerable [32] for edge scenarios. As such, systems that serve exclusively one of the two extremes were set aside in our analysis.

Systems offering intermediate guarantees such as causal [9–12] and those that give the client the chance to choose the desired consistency per request [1, 3–7] will be the focus of our analysis.

SessionStore uses a session-aware reconciliation algorithm focused on reducing the amount of data shipped before serving the client, however clients might face a higher latency, since data is shipped on demand, as such, this system was put aside in our comparison.

In Pileus [5] and Tuba [6], each application can create several object tables and each one is divided into *tablets*. Writes referring to objects contained in a given *tablet* are all made in the same set of machines called the *primary node*, which process them synchronously and guarantee that all other machines that replicates the object receive the update in the same order. These two systems serialize concurrent operations happening on the same *tablet*, as such they present per *Tablet* false dependencies.

In Simba [7] the mechanism is similar: writes made on objects of a given table need to be ordered sequentially, as such, there is a need to pass through a central entity. This system, similarly to the two compared previously, present per *Table* false dependencies.

In Per-Key Session Guarantees on top of NuKV system, each partition has a Raft [50] group. In each group there is a leader who forces all other replicas in the group to apply the operations in the same order, as such this system uses a per partition sequencer. Each data center is divided into the same partitions and each replica has a vector with an entry per data center with each entry referring to

the same partition of each remote data center. This system cannot distinguish from concurrent writes happening in the same partition, as such it suffers from per *Partition* false dependencies.

In Lazy Replication [9] it is considered that each machine is not replicated. Transposing to a *multi-machine* scenario, the most direct way is to consider that each machine is now a data center, in which a leader is elected and is responsible for serializing all the operations and assuring that each replica applies the same operation in the same order, acting as a sequencer. Since each replica has a vector with an entry per data center, concurrent operations on the same data center are serialized, as such this system has *Intra-datacenter* false dependencies type.

In Bayou [1,2], for an update to be committed, it is necessary to run a *primary commit protocol*, where a replica is elected as the master and decides the order of the commits. The number of machines used in the protocol's view is variable, as such its false dependencies type also vary reaching *Inter-datacenter* type if all system's replicas are in the view.

Cure and GentleRain [10, 11], instead of using a sequencer technique, are based on global stabilization. Neither Cure nor GentleRain are able, unlike Saturn [12], to make the visibility of an operation depending exclusively on locally replicated data items. As such, false dependencies occur, since the global stabilizer does not take partial replication into account. Saturn [12], to overcome this problem, uses a tree dissemination technique that allows each replica to deal with only data that it replicates, never creating false dependencies for data not replicated locally, even so it suffers from *intra-datacenter* false dependencies. Although not genuine, Pileus, Simba and Tuba [5–7] support partial replication, while Bayou, Per-Key Session Guarantees on top of NuKV system and Ladin Lazy Replication [1–3, 9] stick with a full copy per machine/data center.

Technique used to enforce causality, false dependencies type, partial replication support, system's ability to apply a different semantic depending on the state of the network (Quality of Service), metadata exchanged between $client \leftrightarrow server$ and $server \leftrightarrow server$, type of dissemination used and supported semantics by the systems are all summarized in table 2.1. Technique used to enforce causality, metadata exchanged between $client \leftrightarrow server$ and $server \leftrightarrow server$, and false dependencies type by system are summarized in table 2.1. Partial replication support, system's ability to apply a different semantic depending on the state of the network (Quality of Service), type of dissemination used and supported semantics by system are summarized in table 2.2.

2.9 Edge Applicability

In the Edge model, the capacity of the nodes, both in terms of processing and storage, is reduced, so partial replication is recommended.

Regarding the type of dissemination scheme used, the most appropriate would be *tree-dissemination*

Table 2.1: Comparison between the several systems. G,DC,P,T,TB stands for inter-datacenter, intra-datacenter, partition, tablet, table. M and MT stand for the number of data centers and number of modified tables, respectively

System	Technique	Client ↔ Server	Server ↔ Server	False Dependencies
Bayou [1,2]	Sequencer (Primary Protocol's View)	O(M)	O(1)	G
Per-Key Session Guarantees [3]	Sequencer (Partition)	O(M)	O(1)	P
Pileus [5]	Sequencer (Tablet)	O(1)	O(1)	T
Tuba [6]	Sequencer (Tablet)	O(1)	O(1)	T
Simba [7,8]	Sequencer (Table)	O(MT)	O(MT)	TB
Lazy Replication [9]	Sequencer (Data Center)	O(M)	O(M)	DC
GentleRain [10]	Global Stabilization	O(1)	O(1)	G
Cure [11]	Global Stabilization	O(M)	O(M)	DC
Saturn [12]	Tree Dissemination	O(1)	O(1)	DC

Table 2.2: Continuation of table 2.1

System	Partial Replication	QoS	Dissemination Type	Semantics
Bayou [1,2]	✗	✗	Pair-Wise	MR,RYW,WFR,MW,Causal
Per-Key Session Guarantees [3]	✗	✗	Master-Slave	Per-Key: MR,RYW,WFR,MW,Causal
Pileus [5]	✓	✓	Master-Slave	RYW,MR,Strong,Causal,Eventual,Bounded Staleness
Tuba [6]	✓	✓	Master-Slave	RYW,MR,Strong,Causal,Eventual,Bounded Staleness
Simba [7,8]	✓	✗	Pair-Wise/Master-Slave	Strong,Causal,Eventual
Lazy Replication [9]	✗	✗	All-to-All	Causal
GentleRain [10]	✗	✗	All-to-All	Causal
Cure [11]	✗	✗	All-to-All	Causal+
Saturn [12]	✓	✗	Tree-Based	Causal

or *pair-wise*, the latter causing an increase in Remote Visibility Latency, making the first one more suitable.

An approach based on a sequencer requires the system to have a single point through which all the writes have to go, greatly reducing its throughput. As an alternative, there is the global stabilization technique, which in the case of GentleRain [10] postpones the visibility of remote operations a lot, while in the case of Cure [11] it reduces the overall throughput of the system. It is considered that the global stabilization technique is problematic in an Edge scenario, where network partitions are formed and consequently the stabilizer may not be able to progress.

Thus, it is considered that the best approach is to develop a solution that uses a tree metadata dissemination scheme similar to that of Saturn [12], as it does not use sequencer or global stabilization, allows for partial partial replication and does not increase Remote Visibility Latency.

Summary

This chapter introduced the main features of the edge model and what distinguishes it from the cloud one. Several consistency models were presented, with a distinction being made regarding which were the ones that better suits the needs and restrictions associated with the edge model. Several systems were analyzed in order to understand some of its characteristics, such as, techniques used to enforce causality, support for partial replication, data dissemination types, metadata size and false dependencies.

In the next chapter, it is proposed a novel system, ENGAGE, focused on serving edge requirements while reconciling low remote update visibility with low remote operation latency.

3

ENGAGE

Contents

3.1 Goals 34
3.2 Design 37
3.3 Protocols 39
3.4 Example 47
3.5 Optimization 49
3.6 Correctness 49

This chapter introduces ENGAGE, 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.

Section 3.1 focus on what this system intends to fulfill; In section 3.2 is described all system's components; Section 3.3 presents the protocols established between clients and servers and how the metadata service works; Section 3.4 contains an example on how does the system works; Followed by system's optimizations 3.4; Section 3.5 provides a proof of correctness for ENGAGE algorithms.

3.1 Goals

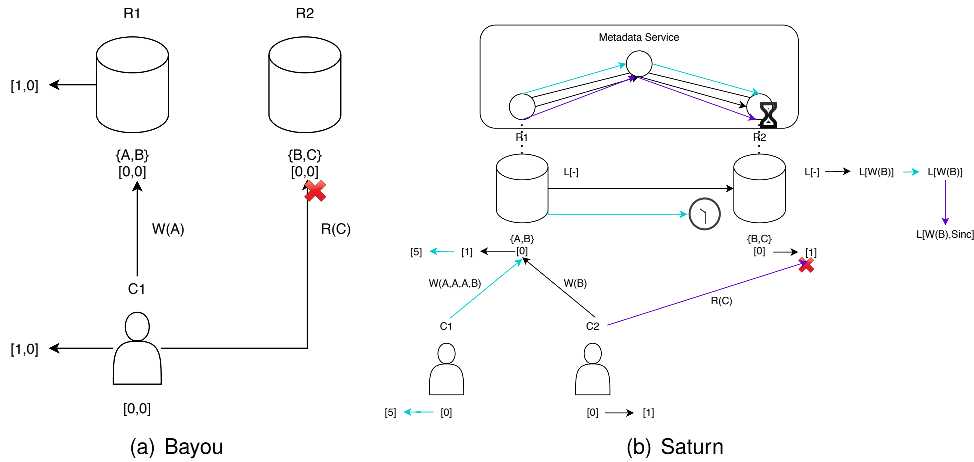
In [18], the authors have suggested a set of mechanisms to enforce the session guarantees that rely on the use of version vectors [9, 43, 44], a form of vector clocks [45]. However, these mechanisms are only efficient in settings that use full replication, i.e., all updates are propagated to all replicas. 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 this work this phenomenon is elaborated).

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 [12, 53, 54]. To address these challenges, the abstraction of a distributed metadata service has been recently introduced [12]. 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 [12] 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, as such, achieving small *remote operation latency* is also desirable.

Saturn [12] and Bayou [1, 2] complement each other. Bayou offers low remote operation latency, however derived from the partial replication incompatibility with vector clocks, it can incur high remote visibility latency. On the other hand, by using a distributed metadata service, Saturn is able to provide low visibility latency, however as it uses a single timestamp it incurs high remote operation latency.

Figure 3.1(a) illustrates why Bayou may perform porly in face of partial replication. $C1$ writes on



(a) Bayou (b) Saturn

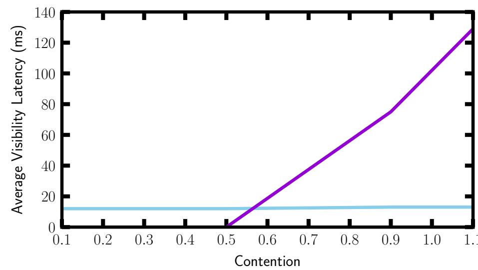
Figure 3.1: Limitations of Bayou and Saturn

object A of replica $R1$ and, as a result, both $R1$'s and $C1$'s clocks are updated. Since object A is not replicated in $R2$, the update is not propagated n the clock of $R2$ is not updated. When $C1$ issues a remote read on object C of $R2$, the request will not be served, because the state of the client is in the future of the $R2$'s state. This request will only be served when $R2$ receives from $R1$ one update with the first entry of the vector clock higher than one.

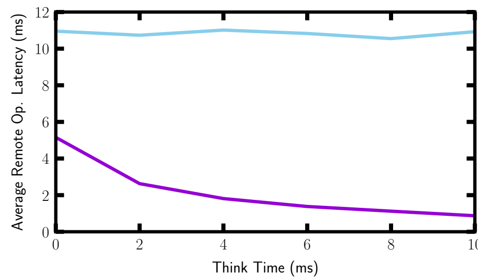
Figure 3.1(b) illustrates the limitations of using a single scalar to track causality. $C2$ writes on object B of $R1$; this update is propagated and installed on $R2$. Next, another client $C1$ makes three sequential updates to object A on replica $R1$ and, subsequently, $C1$ also updates object B on $R1$. Assume that his last update is delayed in the network when being propagated to $R2$. Consider that, at this point, $C2$ issues a remote read on object C of $R2$. Note that $C2$ depends on the first update on object B (performed by $C2$ itself) but not on the second update on object B (performed by $C1$). Unfortunately, Saturn requires the client $C2$ to propagate a *migration label* from $R1$ to $R2$, and this label to be received at $R2$, before issuing requests on $R2$. This will force $C2$ to wait for all updates performed previously at $R1$ (instead of waiting just for its own update).

The dichotomy above is illustrated in Figures 3.2(a) and 3.2(b) that show, respectively, the remote visibility latency and the remote operation latency in two separate systems, one using vector clocks and the other using a metadata service (in this case, Saturn [12]). It is postponed to Section 4 a detailed description of the experimental setup used to collect the data, and present here just the required information to understand the results. It is considered 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 it is just assumed causal consistency. This waiting time

Bayou-Causal Saturn-Causal



(a) Remote Visibility Latency



(b) Remote Operation Latency

Figure 3.2: Remote Visibility Latency and Remote Read Latency.

can contribute substantially to delay the remote operation.

Figure 3.2(a) 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 it is varied 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 3.2(b) shows the delays experienced by clients when they perform a remote operation. In this case, in the x axis it is varied 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, it is expected 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 thesis 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. It is presented 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. It is provided an extensive evaluation of ENGAGE against a system based on vector clocks and against a system based on Saturn [12], using different combinations of the session guarantees proposed in [18].

3.2 Design

This section starts by describing system components and how they are interconnected. Then, it is explained how the metadata is handled and which are the needed data structures.

3.2.1 System Model

Figure 3.3 depicts the architecture of ENGAGE. It is considered a set of edge servers, fog nodes or cloudlets, that are used to replicate data. It is assumed 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 thesis it is not address data placement: the decision of which cloudlets store each data object is orthogonal to our work; it is assumed 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.

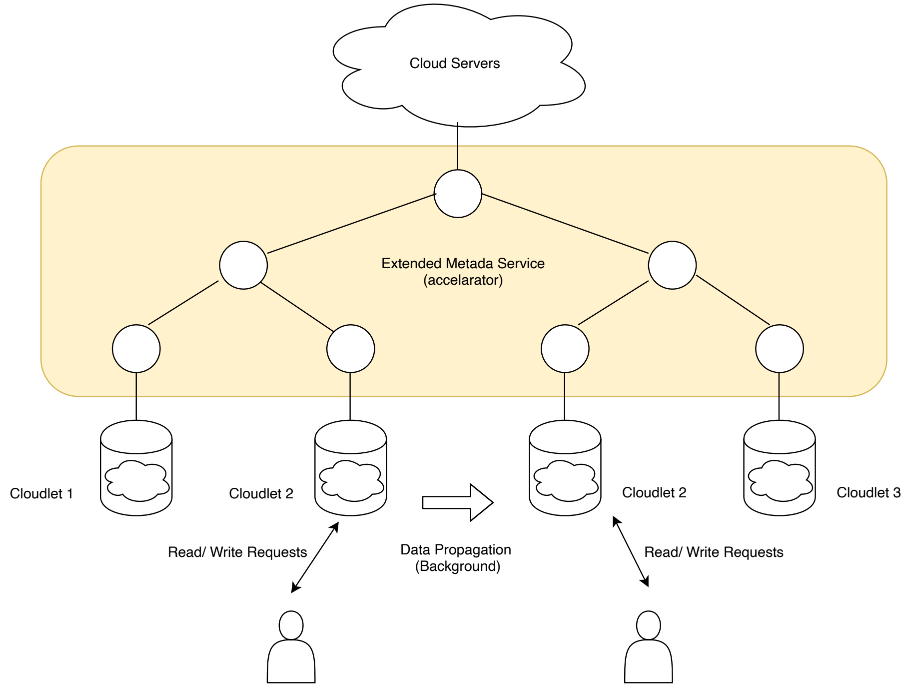


Figure 3.3: ENGAGE Architecture

3.2.2 Metadata

It is assumed that each cloudlet is linearizable [17], 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.

3.3 Protocols

This sections describes ENGAGE protocols, followed by their respective algorithms 1 to 3. The algorithms show the pseudocode for the protocols running on the client proxy, brokers and cloudlets, respectively.

3.3.1 Performing Read and Write Operations

Algorithm 1 Client c code

```
1: function READ(key_id, cloudlet_id)
2:     if MR then
3:         send ⟨ READ_REQUEST client_id, read_vector_clock, key_id ⟩ to cloudlet_id
4:         receive ⟨ READ_RESPONSE val, key_id_vector_clock ⟩ from cloudlet_id
5:     if RYW then
6:         send ⟨ READ_REQUEST client_id, write_vector_clock, key_id ⟩ to cloudlet_id
7:         receive ⟨ READ_RESPONSE val, key_id_vector_clock ⟩ from cloudlet_id
8:     if Causal then
9:         merged_vector_clock ← MERGE(read_vector_clock, write_vector_clock)
10:        send ⟨ READ_REQUEST client_id, merged_vector_clock, key_id ⟩ to cloudlet_id
11:        receive ⟨ READ_RESPONSE val, key_id_vector_clock ⟩ from cloudlet_id
12:    read_vector_clock ← key_id_vector_clock
13:
14: function WRITE(key_id, val, cloudlet_id)
15:    if WFR then
16:        send ⟨ WRITE_REQUEST client_id, read_vector_clock, key_id ⟩ to cloudlet_id
17:        receive ⟨ WRITE_RESPONSE vector_clock ⟩ from cloudlet_id
18:    if MW then
19:        send ⟨ WRITE_REQUEST client_id, write_vector_clock, key_id ⟩ to cloudlet_id
20:        receive ⟨ WRITE_RESPONSE vector_clock ⟩ from cloudlet_id
21:    if Causal then
22:        merged_vector_clock ← MERGE(read_vector_clock, write_vector_clock)
23:        send ⟨ WRITE_REQUEST client_id, merged_vector_clock, key_id ⟩ to cloudlet_id
24:        receive ⟨ WRITE_RESPONSE vector_clock ⟩ from cloudlet_id
25:    write_vector_clock ← key_id_vector_clock
26:
27: function MIGRATE(dst_id, cloudlet_id)
       ▷ This operation is not mandatory, it just allows dst_id to delete metadata regarding client c
28:    DETTACH(src_id, client_id)
       ▷ Attach is possible even if dettach fails
29:    ATTACH(dst_id, client_id)
```

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 [18], 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 [18].

On the server side it is performed 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.

Algorithm 1 presents the pseudo code that describes the client operations, namely Write, Read, and also Migration (this last one is issued when the client needs to change the cloudlet to which he is attached). This pseudo code assumes that clients are placement-unaware, as such, cloudlets are responsible for redirecting requests to the nearest replica of the target object.

3.3.2 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). It will be described

Algorithm 2 Cloudlet n code

```
    ▷ State kept by Broker i
 1: neighbours_id, self_id, cloudlet_vector_clock, objects_vector_clocks
 2: metadata_queue, data_queue, bayou_pending_updates, processed_updates, cloudlets_ids, last_applied
 3: function READ_REQUEST(client_id, key_id, client_vector_clock)
 4:     if self_id.REPLICATES(key_id) then
 5:         wait until cloudlet_vector_clock ≥ client_vector_clock
 6:         send ⟨ READ_RESPONSE val, objects_vector_clocks[key_id] ⟩ to client_id
    ▷ Method invoked on broker connected to cloudlet c
 7:     else
    ▷ Select the nearest cloudlet that replicates the key and redirect the request to the same
 8:         best_node ← NEAREST_NODE(key_id)
 9:         send ⟨ REMOTE_READ client_vector_clock, key_id ⟩ to best_node.id
10:         receive ⟨ REMOTE_READ_RESPONSE val, objects_vector_clocks[key_id], self_id ⟩ from
    best_node.id
11:         send ⟨ READ_RESPONSE val, objects_vector_clocks[key_id] ⟩ to client_id
12:
13: function REMOTE_READ(client_vector_clock, key_id, self_id)
14:     wait until cloudlet_vector_clock ≥ client_vector_clock
15:     send ⟨ REMOTE_READ_RESPONSE val, objects_vector_clocks[key_id] ⟩ to self_id
16:
17: function WRITE_REQUEST(client_id, key_id, client_vector_clock, val)
18:     if self_id.REPLICATES(key_id) then
19:         nounce ← GENERATE_NOUNCE
20:         wait until cloudlet_vector_clock ≥ client_vector_clock
21:         cloudlet_vector_clock[self_id] + +                          ▷Increment cloudlet vector
22:         objects_vector_clocks[self_id] ← cloudlet_vector_clock[self_id]     ▷Put incremented entry on
    object vector
23:         objects_vector_clocks[self_id] ← MERGE(objects_vector_clocks[self_id], client_vector)
24:         aux_vector_clock ← NEW_VECTOR_CLOCK
25:         aux_vector_clock.ADD_ENTRY(cloudlet_vector_clock[])
26:         aux_vector_clock ← MERGE(client_vector_clock, aux_vector_clock)     ▷Merge object vector
    with client one
27:         send ⟨ WRITE_RESPONSE val, aux_vector_clock ⟩ to client_id
    ▷ Method invoked on broker connected to cloudlet c
28:         last_applied[self_id] ← cloudlet_vector_clock[self_id]          ▷Needed for Bayou condition
29:         METADATA_TO_LABEL_SINK(self_id, cloudlet_vector_clock[self_id], key_id, nounce)
    ▷Propagate object vector
30:         for node_id ∈ cloudlets_id do
31:             if node_id.REPLICATES(key_id) then
32:                 node_id.RECEIVE_DATA(self_id, cloudlet_vector_clock[self_id], key_id, nounce, val)
33:     else
    ▷ Select the nearest cloudlet that replicates the key and redirect the request to the same
34:         best_node ← NEAREST_NODE(key_id)
35:         send ⟨ REMOTE_WRITE client_vector_clock, key_id, self_id, val ⟩ to best_node.id
36:         receive ⟨ REMOTE_WRITE_RESPONSE val, vector_clock ⟩ from best_node.id
37:         send ⟨ WRITE_RESPONSE val, vector_clock ⟩ to client_id
```

```
38: function REMOTE_WRITE(client_vector_clock, key_id, sender_id, val)
39:     nounce ← GENERATE_NOUNCE
40:     wait until cloudlet_vector_clock ≥ client_vector_clock
41:     cloudlet_vector_clock[self_id] + +
42:     objects_vector_clocks[self_id] ← cloudlet_vector_clock[self_id]
43:     objects_vector_clocks[self_id] ← MERGE(objects_vector_clocks[self_id], client_vector)
44:     aux_vector_clock ← NEW_VECTOR_CLOCK
45:     aux_vector_clock.ADD_ENTRY(cloudlet_vector_clock[])
46:     aux_vector_clock ← MERGE(client_vector_clock, aux_vector_clock)
47:     send ⟨ REMOTE_WRITE_RESPONSE val, aux_vector_clock ⟩ to self_id
48:     last_applied[self_id] ← cloudlet_vector_clock[self_id]            ▷Needed for Bayou condition
49:     self_id.METADATA_TO_LABEL_SINK(self_id, objects_vector_clocks[self_id], key_id, nounce)
50:     for node_id ∈ cloudlets_id do
51:         if node_id.REPLICATES(key_id) then
52:             node_id.RECEIVE_DATA(self_id, objects_vector_clocks[self_id], key_id, nounce, val)
53:
54: function METADATA_FROM_LABEL_SINK(queue)
55:     for event ∈ queue do
56:         if metadata_queue == ∅ then
57:             if data_queue.CONTAINS(event) then
58:                 data_queue.REMOVE(event)
59:                 cloudlet_vector_clock ← MERGE(cloudlet_vector_clock, event.client_vector_clock)
60:                 objects_vector_clocks[event.key_id] ← MERGE(objects_vector_clocks[event.key_id],
event.client_vector_clock)
61:             else if ¬self_id.REPLICATE(event.key_id) then                 ▷MF message
62:                 cloudlet_vector_clock ← MERGE(cloudlet_vector_clock, event.client_vector_clock)
63:             else
64:                 metadata_queue.ADD(event)
65:         else
66:             metadata_queue.ADD(event)
```

```
67: function RECEIVE_DATA(sender_id, client_vector_clock, key_id, nounce)
68:     event ← CREATE_EVENT(sender_id, client_vector_clock, key_id, nounce, val)
69:     to_continue ← true
70:     data_queue.ADD(event)
71:     pending_updates[sender_id].ADD(event)
72:     while metadata_queue ≠ ∅ & to_continue do
73:         head ← metadata_queue.PEEK
74:         if head ∈ data_queue then
75:             data_queue.REMOVE(head)
76:             cloudlet_vector_clock ← MERGE(cloudlet_vector_clock, head.client_vector_clock)
77:             objects_vector_clocks[event.key_id] ← MERGE(objects_vector_clocks[head.key_id],
    head.client_vector_clock)
78:             metadata_queue.POLL
79:         else if ¬self_id.REPLICATE(head.key_id) then                              ▷MF message
80:             cloudlet_vector_clock ← MERGE(cloudlet_vector_clock, head.client_vector_clock)
81:         else
82:             to_continue ← false
83:     state_changed ← true
    ▷ Bayou condition
84:     to_continue ← false
85:     while state_changed do
86:         for i ∈ cloudlets_ids do
87:             to_be_delivered ← pending_updates[i].GET_FIRST
88:             if to_be_delivered == null then
89:                 continue
90:             can_deliver ← true
91:             for j ∈ cloudlets_ids do
92:                 if j ≠ i then
93:                     if ((last_applied[j] ≤ to_be_delivered[j]) &
                        ((pending_updates[j].GET_FIRST == null) ||
                        (pending_updates[j].GET_FIRST[j] ≤ to_be_delivered[j]))) then
94:                         can_deliver ← false
95:                         break
96:             if can_deliver then
97:                 to_continue ← true
98:                 last_applied[i] ← to_be_delivered[i]
99:                 pending_updates[i].REMOVE(to_be_delivered)
100:                cloudlet_vector_clock ← MERGE(cloudlet_vector_clock, to_be_delivered)
101:                if self_id.REPLICATES(key_id) then
102:                    objects_vector_clocks[event.key_id] ← MERGE(objects_vector_clocks[to_be_delivered.key_id],
    to_be_delivered)
103:
```

the ENGAGE metadata service in the following section. Here it will be described 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 $i \neq \textit{orig}$, we have $V_*^{\textit{dest}}[i] \geq V_o^{\textit{orig}}[i]$.

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

Algorithm 2 captures the different sequences of operations performed by the cloudlets. Cloudlets communicate both with clients, broker nodes, and other cloudlets. Commands “READ_REQUEST” and “WRITE_REQUEST” are issued when clients want to execute Read or Write requests, respectively. These operations also handle remote operations, by redirecting the requests to the appropriate cloudlets (functions REMOTE_READ and REMOTE_WRITE). When cloudlets receive metadata messages from broker nodes the function METADATA_FROM_SINK is executed. Finally, cloudlets can also receive data messages from other cloudlets; in this case function RECEIVE_DATA is executed.

3.3.3 The Engage Extended Metadata Service

Using vector clock stability to apply remote updates is not effective under partial replication. It is recalled the example from Section 2.4.4 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 (these messages are called *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

Algorithm 3 Broker i

```
    ▷ State kept by Broker i
 1: function RECEIVE_QUEUE(queue, sender_node_id)
 2:     for event ∈ queue do
    ▷ broker i has a queue per neighbour
 3:         for k ∈ neighbours_ids do          ▷neighbours_ids represents nodes connected to i
 4:             if node[k] ≠ sender_node_id then
 5:                 node[i].queue[k] ← queue
 6:
 7: function BROADCAST_QUEUE(queue, sender_node_id)
 8:     for k ∈ neighbours_ids do
    ▷ Step 1 - Remove mf messages from queue
 9:         for event ∈ node[i].queue[k] do
10:             if event.is_mf == true then
11:                 node[i].queue[k].remove(event)
12:                 mf_queue ← event
    ▷ Step 2 - Compress mf messages into one
13:         node[i].mf[k] ← COMPRESS_MF_MESSAGES(mf_queue, k)
14:         mf_queue[k] ← ∅
15:         if node[i].queue[k] ≠ ∅ then
    ▷ Step 3 - Merge mf message with the last queue's event
16:             node[i].queue[k].merge_with_last(node[i].mf[k])
    ▷ Step 4 - Send queue for neighbour k. that can be a cloudlet or another broker
17:             if k is cloudlet then
18:                 k.METADATA_FROM_LABEL_SINK(node[i].queue[k])
19:             else
20:                 RECEIVE_QUEUE(node[i].queue[k], k)
21:             node[i].queue[k] ← ∅
22:             node[i].mf[k] ← ∅
23:         else
    ▷ Step 5 - Check Timeouts
24:             if node[i].mf[k] ≠ ∅ then
25:                 if TIMEOUT_EXPIRED(node[i].mf[k]) == true then
26:                     if k is cloudlet then
27:                         k.METADATA_FROM_LABEL_SINK(node[i].mf[k])
28:                     else
29:                         RECEIVE_QUEUE(node[i].queue[k], k)
30:                 else
31:                     node[i].queue[k] ← node[i].mf[k]
32:                 node[i].mf[k] ← ∅
33:
    ▷ Method invoked by cloudlets to introduce metadata in broker's network
34: function METADATA_TO_LABEL_SINK(sender_node_id, client_vector_clock, key_id, nounce)
35:     aux_queue ← CREATE_QUEUE
36:     event ← CREATE_EVENT(sender_node_id, client_vector_clock, key_id, nounce)
37:     aux_queue.add(event)
38:     RECEIVE_QUEUE(aux_queue, sender_node_id)
```

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 [12], can be extended to perform a dual function: it can be used to instruct cloudlets to deliver remote updates (as proposed in [12]) 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 it is proposed to connect all cloudlets by a distributed metadata service, inspired by Saturn [12]. The metadada 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 $\langle \text{UN}, \textit{src}, \textit{snb}, \textit{oid}, V_{oid}^{src} \rangle$, 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 3.3.1, 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 $\langle \text{UN}, src, snb, oid, V_{oid}^{src} \rangle$ 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 3.3.2. The update message can be safely discarded.

- Otherwise, the cloudlet waits until it has received the payload of the update directly from *src*.

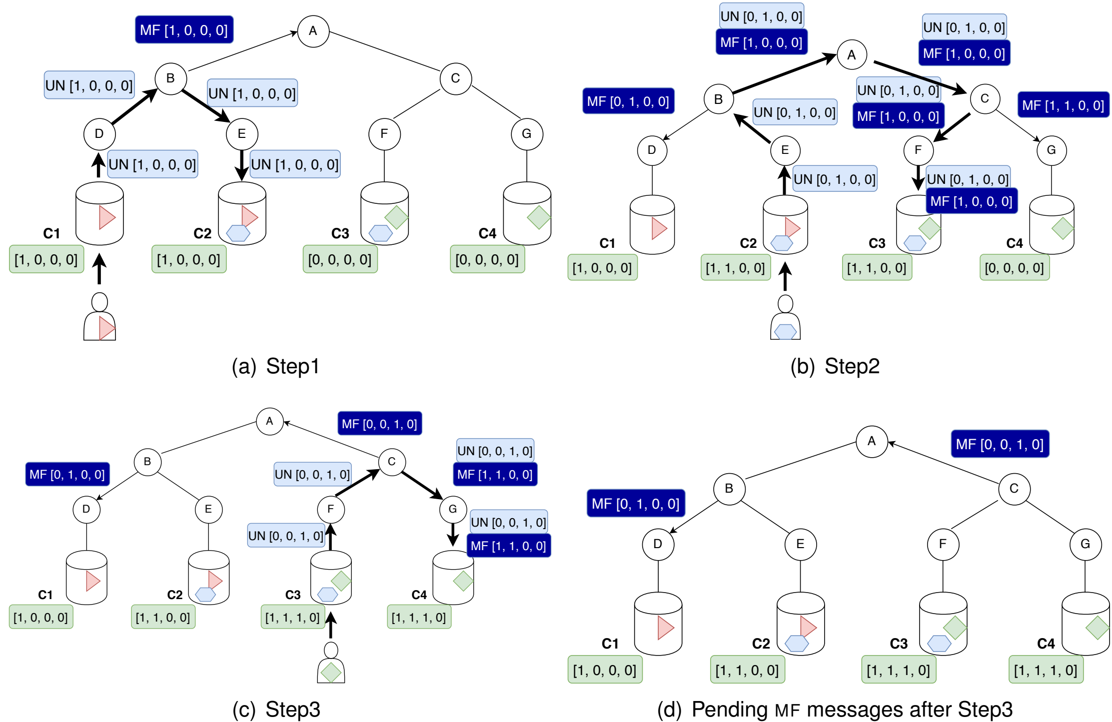
- When the cloudlet *dest* has received the update message from the metadata service *and* the payload directly from *src*, it applies the update to object *oid* and updates vector clocks V_*^{dest} and V_o^{dest} as described in Section 3.3.2.

In algorithm 3 are described all operations performed by broker nodes. Broker nodes communicate both with cloudlets and with other broker nodes. Brokers can receive metadata messages from other brokers upon receiving RECEIVE_QUEUE and can propagate metadata messages to other brokers by calling local function BROADCAST_QUEUE.

Brokers also offer a method to be called by cloudlets in order to introduce metadata messages in the label sink, namely METADATA_TO_LABEL_SINK.

3.4 Example

It is now illustrated the propagation of *update notifications* and *metadata flush* messages in the network of metadata servers with the help of Figure 3.4. 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 3.4(a)), then another client makes an update on an object replicated in c_2 and c_3 (Step 2, Figure 3.4(b)), finally, another client makes an update on an object replicated in c_3 and c_4 (Step 3, Figure 3.4(c)). 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.



(a) Step1

(b) Step2

(c) Step3

(d) Pending MF messages after Step3

Figure 3.4: Propagating Metadata Messages

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, 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.

3.5 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 [32]. Therefore, although supported 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.

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.

3.6 Correctness

The correctness of ENGAGE derives from the fact that it is build by combining techniques that have been proven to be correct and by the fact that the way these techniques are combined does not introduce any unwanted interference between their operation.

Remote updates are applied at each cloudlet according to the order of labels delivered by Saturn. This mechanism has been proved correct in [55]. Therefore, the use of Saturn ensure that cloudlets always have a causally consistent state.

When clients migrate, the consistency between the cloudlet state and the client causal past uses the techniques introduced by Bayou and described in [1,2]. These techniques are known to guarantee that a client always observes a state that is consistent with its causal past and the selected session guarantee.

Since both components run in parallel, they do not interfere with each other, as such the original proofs of correction remain valid.

Summary

This chapter has addressed all ENGAGE components and how they are all interconnected. ENGAGE presents a novel model that is able to achieve both low remote visibility latency and remote operation latency. ENGAGE also introduced a new mechanism to disseminate metadata through a tree of brokers and cloudlets while offering partial replication and support for weaker semantics. This novel mechanism besides delivering metadata messages to cloudlets is also able to propagate MF messages with minimal additional overhead.

In the next chapter is presented a comprehensive evaluation between the three implemented solutions, namely, ENGAGE, Saturn and Bayou.

4

Evaluation

Contents

4.1 Evaluation Goals 52
4.2 Experimental Setup 52
4.3 ENGAGE vs Bayou vs Saturn 54
4.4 Benefits from Session Guarantees 55
4.5 Tolerance to Transient Partitions 57
4.6 Signaling Overhead 59
4.7 Discussion 61

This chapter evaluates ENGAGE. It starts by describing the goals of the evaluation 4.1; Followed by a description of the experimental setup 4.2; Then it is presented all the evaluation scenarios 4.3, 4.4, 4.5 and 4.6; This chapter ends with a brief discussion about the obtained results 4.7.

4.1 Evaluation Goals

In the evaluation, the following research questions are addressed:

- How does ENGAGE perform in comparison with the classical vector clock approach used in Bayou [18] and with recent metadata services, such as Saturn [12]?
- 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 it was 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.

4.2 Experimental Setup

The evaluation has been performed using a version of the Peersim network simulator [19] 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 $1\ Gb/s$. The decision of using a simulator was due to the fact that it was required to use settings with a large number of nodes; it was infeasible to carry out the native implementation, the deployment of ENGAGE, and the experimental evaluation on a real testbed of those dimensions within the time available to produce this dissertation. It was implemented ENGAGE, a version of Saturn [12], and a version of Bayou [1, 2] in Peersim, in order to compare the performance of ENGAGE against the performance of these two system when used in isolation.

Table 4.1: 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

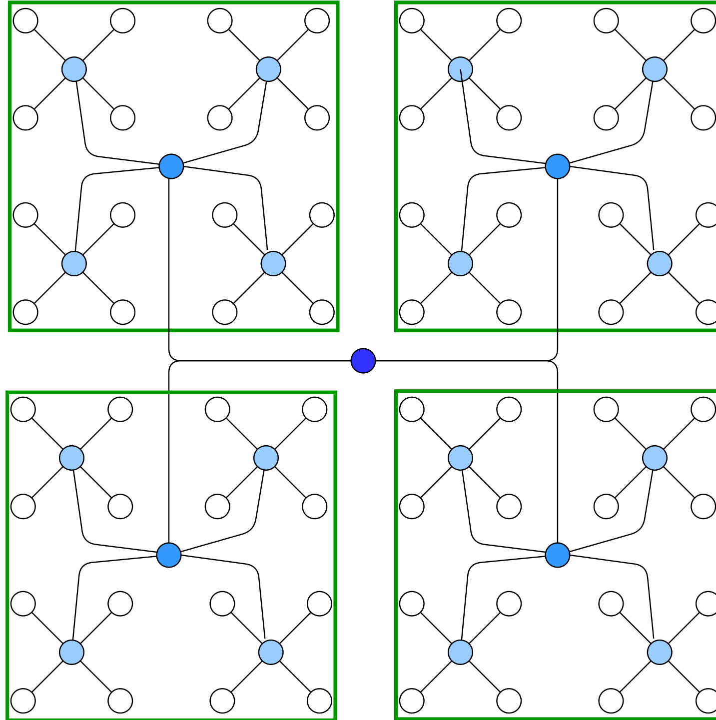
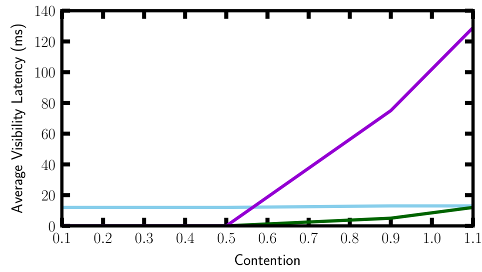


Figure 4.1: Cloudlet placement (white) and broker network (blue)

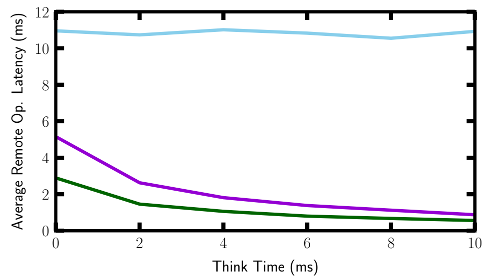
It was considered a scenario where cloudlets are deployed in a grid network, which abstracts a urban deployment. It is considered an hierarchical broker network consisting of a quadtree where the root is placed in the center of the area. Figure 4.1 illustrates the cloudlet deployment and the broker network for the case of a 8×8 grid. Cloudlets were populted 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 4.1.

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, the Zipfian distribution selects the key from a local object list, and if the operation is remote, it selects the key from a remote object list. The Zipfian Constant affects the contention of the workload: the higher the Zipfian Constant, the higher the contention. After executing an operation the client has a cooldown time before issuing another operation (*ThinkTime*): if the *ThinkTime* $= 0$, the client is considered *eager*. It is assumed 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

Engage-Causal Bayou-Causal Saturn-Causal



(a) Remote Visibility Latency



(b) Remote Operation Latency

Figure 4.2: ENGAGE vs Bayou and Saturn

message (to piggyback it with an update notification).

The workload parameters are summarized in the Table 4.1. Variations of this workload in which the value of one parameter was changed and keep the others at their default values. In the following experiments, were measured 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.

4.3 ENGAGE vs Bayou vs Saturn

In this section, it is intended to answer the first question and position ENGAGE with regard to Bayou and Saturn. In particular, it is present again the results from Figure 3.2 (used in the motivation), now including the performance of ENGAGE.

Figure 4.2(a) 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, it was varied 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, it was disabled the metadata flush of ENGAGE and Bayou (an evaluation of these mechanisms is postponed for Section 4.6). 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.

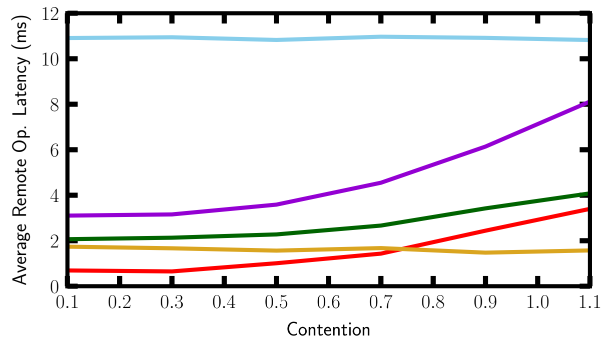
Figure 4.2(b) shows the delays experienced by clients when they perform a remote operation. In this case, in the x axis, it was varied the clients' think time, i.e., the average time between two consecutive operations. The larger the think time, the more likely it is that updates in the causal past of the client have already been propagated and applied when the client performs a remote operation. Thus, it was expected 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 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 *ThinkTime* $= 2ms$ and $20\times$ for *ThinkTime* $= 10ms$.

4.4 Benefits from Session Guarantees

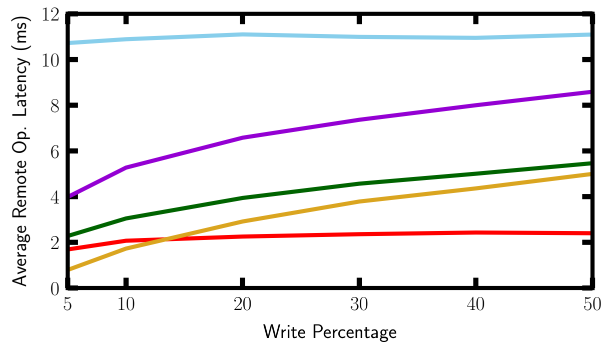
In the previous section it was shown that ENGAGE is able to match Bayou when supporting remote reads using causal consistency. It is now shown that ENGAGE can further reduce the latency of remote operations if the client uses the weaker session guaranties, instead of using full causal consistency. Figure 4.3 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 4.3(a) shows the impact of the contention level on remote latency for different systems 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

WFR MW Engage-Causal Bayou-Causal Saturn-Causal



(a) Latency vs contention



(b) Latency vs read/write ratio

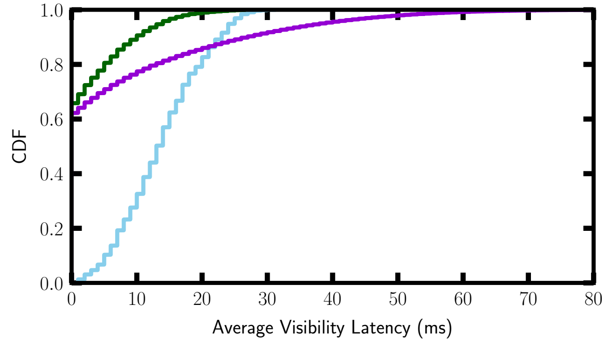
Figure 4.3: Remote Latency with Session Guarantees

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 guarantees 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 4.3(b) shows the impact of the read/write on remote latency. For most guarantees, the figure shows a similar trend. As in Figure 4.3(a), 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 that while contention has little effect on MW and a large impact on WFR, the opposite happens when the write/read ratio is increased. 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.

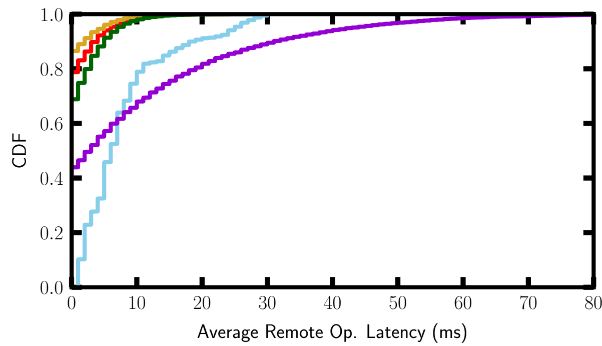
Figures 4.2(b) and 4.3(b) depict *average* latencies. It is also interesting to look in detail to the *Cumulative Distribution Function* (CDF) of the latency, both for the remote visibility latency and for the remote operation latency of the different systems. This is depicted in Figure 4.4. In this case, is was used the default parameters from Table 4.1. It is possible to 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 [56].

4.5 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 the client from being affected by a transient partition. In fact, operations that can be performed on their local cloudlet 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.



(a) Remote Visibility Latency



(b) Remote Operation Latency

Figure 4.4: CDF of the Remote Visibility Latency and Remote Operation Latency.

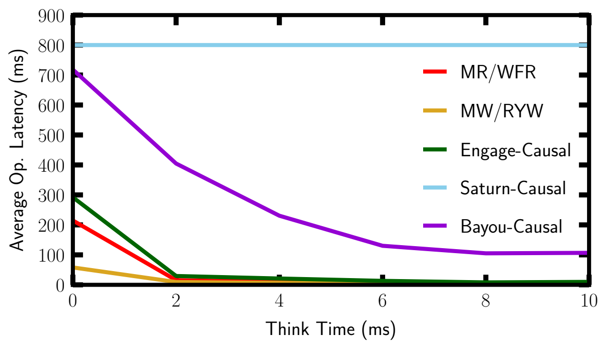


Figure 4.5: Tolerance to Transient Partitions

In Figure 4.5, it is shown 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. It was 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 the client think time is varied. 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.

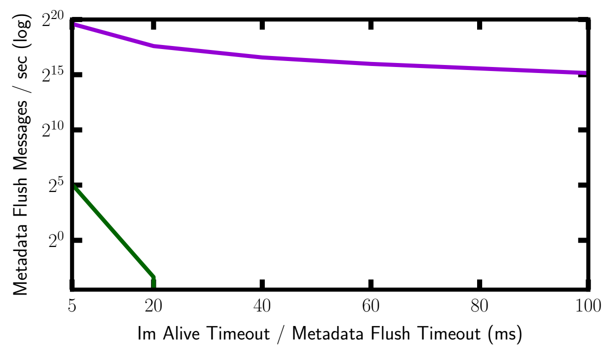
Moreover, the ENGAGE session guarantees give clients more control over what updates they need to observe to operate without violating consistency. Thus, allowing to execute operations with much lower latency than causal consistency, notably in cases where not all client's causal dependencies were propagated 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.

4.6 Signaling Overhead

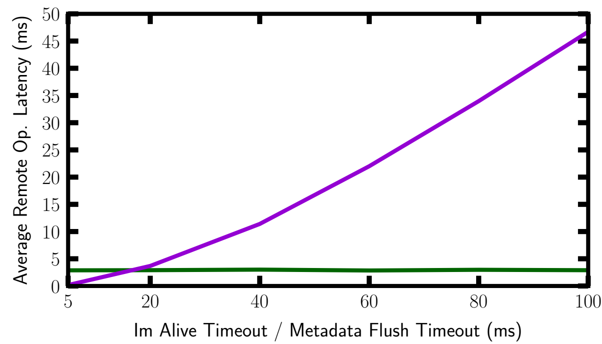
Both ENGAGE and Bayou require the exchange of control messages to update the cloudlet's vector clocks. This is of paramount importance to allow clients to be served quickly and avoid unnecessary delays due to false dependencies. In systems such as Bayou, vector clocks can be updated via the periodic exchange of "I'm Alive" messages, that carry the vector clock of the sender [54, 57]. 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 3 (for instance, see the example of Section 3.4). This often prevents ENGAGE from being required to send signaling messages just to update vector clocks. Figure 4.6(a) 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 4.6(a), in the x axis it is varied the timeout value and in the y axis it is depicted the num-

Engage-Causal Bayou-Causal



(a) Signaling Overhead



(b) Signaling Tolerance

Figure 4.6: Signaling Impact: ENGAGE vs Bayou

ber of control messages per second (note that it is used a logarithmic scale in this figure). In ENGAGE, it is only counted 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 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 4.6(b), it is possible to 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 the timeout value is incremented, 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.

4.7 Discussion

ENGAGE was conceived to combine the main characteristics of two systems without suffering from their drawbacks.

It would be interesting to conceive a system that is simultaneously able to serve an application responsible for diverting traffic by reporting accidents (low remote update visibility latency requirements) and also an application where clients need to frequently change the contacted cloudlet due to partial replication (low remote operation latency requirements). ENGAGE is able to serve both applications by combining Bayou [1,2] and Saturn [12] systems.

The fact that ENGAGE is a system to be deployed on the edge presents some limitations, so, in addition to the evaluation having to cover whether or not the ENGAGE is capable of combining the best of Bayou and Saturn, it is also necessary to measure the impact in terms of number of additional messages sent (MF messages) and whether or not it is capable of tolerating a partitioned network, a recurring situation in an edge environment. Another important aspect to be covered by the evaluation is if by providing session guarantees, clients can substantially reduce the latency associated with its requests.

Our experimental evaluation shows that the latency gains achieved with ENGAGE can be as high as $7\times$ for remote update visibility and $2.7\times$ for remote operations, compared with Bayou for high contention workloads. Moreover, ENGAGE can tolerate transient partitions much better than Saturn, reducing the

latency $2.6\times$ for eager clients and almost $27\times$ 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.

Summary

In this chapter, it was presented the evaluation of ENGAGE. The results show that ENGAGE is able to successfully combine in a synergistic way Bayou and Saturn, while generating only a minimal overhead in terms of broadcast MF messages. It was also outlined the advantages of using session guarantees and scenarios where some guarantee is more suitable than other. The next chapter ends this thesis, by reporting the most interesting conclusions and also present some ideas on how this work can be extended.

5

Conclusion

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 thesis it was 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. It was shown 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.

It would be interesting to complement ENGAGE with a Quality of Service component similarly to Pileus [5]. Allowing clients to define an SLA with its various latency and semantics requirements makes it possible for the system to provide the best response to the client, depending on the state of the network. It would also be interesting to deploy ENGAGE on top of a real topology instead of a simulation environment.

Bibliography

[1] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," *SIGOPS Oper. Syst. Rev.*, p. 172–182, Dec. 1995.

[2] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers, "Flexible update propagation for weakly consistent replication," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, ser. SOSP '97, Saint Malo, France, 1997, p. 288–301.

[3] M. Roohitavaf, J.-S. Ahn, W.-H. Kang, K. Ren, G. Zhang, S. Ben-Romdhane, and S. Kulkarni, "Session guarantees with raft and hybrid logical clocks," in *Proceedings of the 20th International Conference on Distributed Computing and Networking*, Jan. 2019, p. 100–109.

[4] S. 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.

[5] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Farmington (PA), USA, 2013.

[6] M. Ardekani and D. Terry, "A self-configurable geo-replicated cloud storage system," in *The 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield (CO), USA, Oct 2014.

[7] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. Madhyastha, and C. Ungureanu, "Simba: Tunable end-to-end data consistency for mobile apps," in *Proceedings of the Tenth European Conference on Computer Systems*, Bordeux, France, 2015.

[8] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, "Reliable, consistent, and efficient data sync for mobile apps," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara (CA), USA, Feb. 2015.

[9] 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.

[10] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, “Gentlerain: Cheap and scalable causal consistency with physical clocks,” in *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, Seattle (WA), USA, 2014.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and 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.

[19] 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.

[20] N. Afonso, “Mechanisms for providing causal consistency on edge computing,” Master’s thesis, Instituto Superior Tecnico, Universidade de Lisboa, Nov. 2018.

[21] M. Satyanarayanan, “Mobile computing: The next decade,” in *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, 2010.

[22] F. Okay and S. Ozdemir, “A fog computing based smart grid model,” in *Proceedings of the IEEE International Symposium on Networks, Computers and Communications (ISNCC)*, May 2016.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, “Don’t settle for eventual: Scalable causal consistency for wide-area storage with COPS,” in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 10 2011, pp. 401–416.

[35] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s globally-distributed database,” in *Proceedings of 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood (CA), USA, Oct. 2012.

[36] C. Papadimitriou, “The serializability of concurrent database updates,” *J. ACM*, Oct. 1979.

[37] M. Vukolic, “Eventually returning to strong consistency,” *IEEE Data Eng. Bull.*, 2016.

[38] D. Terry, “Replicated data consistency explained through baseball,” *Commun. ACM*, Dec 2013.

[39] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. Rodrigues, “On the use of clocks to enforce consistency in the cloud,” *IEEE Data Eng. Bull.*, Jan 2015.

[40] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, Jul. 1978.

[41] S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, “Logical physical clocks,” in *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*, Cortina d’Ampezzo, Italy, 2014.

[42] D. Mills, “Internet time synchronization: the network time protocol,” *IEEE Transactions on Communications*, Oct 1991.

[43] R. Guy, J. Heidemann, W. Mak, T. Page Jr, G. Popek, and D. Rothmeier, “Implementation of the Ficus replicated file system,” in *Proceedings of the USENIX Summer Technical Conference (USTC)*, Anaheim (CA), USA, Jun. 1990.

[44] 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.

[45] 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.

[46] 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.

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

[48] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Vancouver, British Columbia, Canada, Aug. 1987, p. 1–12.

[49] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed ingres," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 188–194, 1979.

[50] 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.

[51] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, p. 35–40, Apr. 2010.

[52] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOPS)*, 2007.

[53] 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.

[54] 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.

[55] M. Bravo, "Metadata management in causally consistent systems," Ph.D. dissertation, Instituto Superior Tecnico, Universidade de Lisboa and Université catholique de Louvain, May 2018.

[56] 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.

[57] 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.