Mechanisms for Providing Causal Consistency on Edge Computing

(extended abstract of the MSc dissertation)

Nuno Cerqueira Afonso
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Eduardo Teixeira Rodrigues

***Abstract*—Today, many applications offload computation and storage to the cloud. Unfortunately, the high network latency between clients and datacenters can impair novel, latency-constrained applications, such as augmented reality. Edge computing has emerged as a potential solution to circumvent this problem. To unleash its full potential, the edge must cache data that is frequently used. However, building a storage service that is able to maintain many (partial) replicas while providing meaningful consistency guarantees is an open challenge. In this paper, we present Gesto, a data storage architecture that enables scalable causal consistency for edge networks. Gesto integrates a novel causality tracking mechanism that relies on multipart timestamps of constant size, independently on the number of edge caches. As our evaluation shows, this mechanism enables Gesto to simultaneously offer fast local operations, high throughput, fast update replication, great scalability, and, unlike previous work, quick client migrations.**

I. Introduction

Cloud computing is an established paradigm and most applications, including those that run on mobile devices, use the cloud to fetch data, upload information, or offload computations that are resource intensive and can drain the battery of the devices [1]. Image processing for face or object recognition [2], [3] is an example of a resource-eager task that can be offloaded to the cloud. This functionality is key for many augmented reality applications such as marker detection [2] or just-in-time video indexing [4]. However, a response time below 5–30 milliseconds is typically required for these applications to be usable [5]. This may be impossible to guarantee when accessing remote datacenters.

To circumvent this limitation, the use of computing resources closer to the logical extreme of the network, such that the cloud is pushed closer to clients, has emerged as a viable solution. Many approaches based on this principle are being advocated, including the use of *cloudlets* [6], *Mobile Edge Computing (MEC)* [7], and *fog computing* [8]. In addition to removing the latency bottleneck, edge computing brings other advantages, such as decreasing the load on datacenter networks [9] and making the applications more robust to datacenter outages [5].

To unleash their full potential, edge nodes should not only provide processing capacity, but also cache data that may be frequently used [10]; otherwise, the advantages of processing on the edge may be impaired by frequent remote data accesses [11]. By using cached data, end-users' requests rarely need to be served by the root datacenters.

Consequently, we claim that a key ingredient of edge assisted cloud computing is a storage service that extends the one offered by the cloud, in a way that relevant data is replicated closer to the edge. Nevertheless, designing such a storage service while providing meaningful consistency guarantees is not trivial. Specifically, we focus on the problem of supporting causal consistency on the edge, given that causal consistency was shown to be the strongest consistency criteria that can be offered without compromising availability [12]. The edge scenario enforces a set of requirements that makes the implementation of causal consistency hard:

High Scalability. An important observation is that, to ensure acceptable latency, many edge nodes will need to be deployed. For instance, to ensure the target 5–30 milliseconds latency, every heavily populated region of $95\ 114$ Km^2 (a circle of 174 Km radius) should have its own edge instance in the center. We estimate this number based on measured round-trip-times (RTTs) among Amazon EC2 regions (RTT of 22.5 ms between the Oregon and California regions, which are 783 Km apart), such that the 5 ms lower-bound is met. In Europe alone, with a total area of 10.18 million Km^2, approximately 107 replicas should be deployed.

Partial Replication. Edge replicas will only maintain partial information given that: (i) for efficiency, only information that is valuable to the clients that a replica is serving should be shipped and stored on the edge; (ii) edge nodes are resource-constrained.

Non-sticky Sessions. Clients should be able to efficiently move among replicas without having to restart their session. Clients may move, for instance, due to mobility or to read data that is not replicated in their preferred replica.

Interestingly, most systems that offer causal consistency in the cloud make exactly the opposite assumptions [13]–[21]: they have been developed for scenarios where a few datacenters exist, each datacenter usually implements full replication, and clients remain mostly connected to a single datacenter.

In this paper, we present *Gesto*, a novel hierarchical architecture that enables causally consistent cloud stores to

span over edge networks. Firstly, Gesto is able to scale with the number of edge devices by relying on constant sized multipart timestamps to track causality (the size of these timestamps does not grow with the number of edge replicas). Secondly, it supports partial replication, a replication model that is a requirement in edge networks, given that edge devices are resource-constrained. Finally, Gesto copes with the fact that users may need to attach to different replicas during their operation, and are not tied to a single one.

We have built a prototype of Gesto and we have compared its performance against other state of the art causally consistent systems, such as COPS [13], Saturn [20], and Occult [19]. Our evaluation shows that Gesto can efficiently enable causally consistent cloud stores to take advantage of edge devices.

II. Related Work

The causal dependencies of an operation are determined by happened-before relations ($\rightarrow$) [12], [22], which are defined by three rules:

Thread of Execution. If a and b are two operations executed by the same thread of execution (for instance, by the same client), then $a \rightarrow b$ if a happens before b.

Reads From. If a is an update operation and b is a read operation that reads the value set by a, then $a \rightarrow b$.

Transitivity. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

A storage system is said to be causally consistent if its clients perform v_1=*read*(k_1) and, subsequently, make v_2=*read*(k_2), where *write*(k_2,v_2) $\rightarrow$ *write*(k_1,v_1), and there is no other operation *write*(k_2,v') such that *write*(k_2,v_2) $\rightarrow$ *write*(k_2,v') $\rightarrow$ *write*(k_1,v_1).

All systems that enforce causal consistency need to maintain some form of metadata to keep track of causal dependencies. This metadata can assume multiple forms, being used for tagging operations and capturing the causal past of clients. The metadata kept by clients allows the understanding of which updates can be made visible to that client without violating causality. By comparing the metadata associated with two operations, it should always be possible to determine in which orders these operations should be applied to respect causality. Ideally, one would also like to always be able to check if two operations are concurrent. Unfortunately, the amount of information that needs to be maintained to accurately capture concurrency may be prohibitively large and affect the system's performance [19], [20]. Thus, many systems reduce the amount of metadata at the cost of losing precision, creating what has been named *false dependencies*. They are scenarios where some operations appear to be causally related, but, in fact, are not. Not surprisingly, the right amount of metadata that needs to be kept for a given scenario is one of the most studied aspects of distributed systems, both by practitioners and theoreticians [23]–[26]. A key contribution of our work is to propose a multipart timestamp that offers a good trade-off between size and precision for the network edge.

Although there has been a significant amount of work in causally consistent cloud storage services in the past years, there are no systems that join support for partial replication, efficient client migration, low update visibility, and constant metadata size (key feature to scale). Here, we briefly discuss some properties of the most relevant previous solutions.

Usually, systems use variants of vector clocks as metadata, which grows linearly with the number of replicas. Examples are PRACTI [14], SwiftCloud [15], Legion [16], EunomiaKV [17] and C^3 [21]. The size of these clocks allows them to achieve low visibility latency. Unfortunately, as the size grows with the system's scale, these systems do not scale well as the number of edge caches increases. Occult [19] introduces compression techniques that enable vectors with a reduced size. It does not introduce inconsistencies, because it relies on a master-slave replication scheme.

GentleRain [18] and Saturn [20] do an effort to keep the metadata size constant. However, they have not been designed to consider frequent client migrations, and perform poorly in this respect.

Other solutions, such as COPS [13], keep explicit track of the object versions that have been read by the client. This avoids false dependencies, potentially minimizing the update visibility latency. However, they have not been designed for partial replication. They can be extended (as we will do for our experimental evaluation), but the adopted model disallows them to efficiently trim the client's causal past. As a result, these systems need to process a very large amount of metadata, which severely limits the system's throughput.

Among all these systems, only SwiftCloud [15] and Legion [16] consider using edge caches: both allow clients to maintain a partial replica. In SwiftCloud, all the coordination between edge replicas is done through the closest datacenter. Unlike Gesto, they do not consider migration among edge replicas, forcing clients to always be attached to the same one, regardless of observing an increase on latency. In Legion, clients execute direct synchronization, bypassing the cloud. However, this is a heavy process, because clients have to exchange vector clocks for calculating the difference between their states, create structures that ensure data confluence and return them to each other. With the increase of the number of peers, clients quickly become overloaded.

III. System Model

Before presenting the design of Gesto, there are some decisions that must be highlighted. Firstly, we consider a particular type of network edge, which allows us to optimize the protocols. Secondly, Gesto separates the propagation of data and metadata. In the following paragraphs, we clarify the reasons that have led us to such choices.

Datacenters, Cloudlets and Regions. We assume a model of edge computing where the cloud is extended with smaller datacenters that are located closer to clients and maintain partial replicas of the state stored in the cloud datacenters. This model has been sometimes called the *heavy* edge

network model [27]. The smaller datacenters have been called cloudlets [6], a term that we also adopt.

We assume that each cloudlet is hierarchically attached to one (and just one) datacenter. Naturally, each datacenter can communicate with multiple cloudlets. Clients can connect to any cloudlet or directly to a datacenter. The set of system components that consists of a single datacenter, the cloudlets attached to that datacenter, and the clients attached to those cloudlets, is denoted a Gesto *region*.

Clients are not required to stay connected to the same cloudlet during their operation; they can dynamically *attach* to another cloudlet at any point. Clients may change their attachment point due to mobility, to access data that is not available in the replica they are attached to, or just to adapt to changing network conditions. Each cloudlet runs an instance of the Gesto service, that manages the client access to the data that is locally replicated.

We assume that each cloudlet maintains a partial replica of the application state (the entire state is maintained in the storage service instantiated by datacenters). A Gesto instance keeps a copy of the data that is more likely to be relevant for computations performed at its cloudlet. The algorithms used to select which data is cached at each instance are orthogonal to the contributions of this paper.

Data and Metadata. In Gesto, we separate metadata management from payload propagation. This approach has been adopted by some recent systems [14], [20], [21], [28], because it optimizes the application of remote updates, which becomes no longer constrained by the expensive dissemination of payloads in the network. When an update occurs at a given site, the system can choose any data path that is more convenient to propagate it. For instance, updates in a given cloudlet may be shipped to a cloud datacenter first, and, from there, to other cloudlets, or can be directly shipped to sibling cloudlets. On the other hand, metadata information is propagated using a specific topology, detailed below, that allows optimizations for keeping a small size.

IV. Design

In this section, we present the design of *Gesto*, a causally consistent storage service for edge networks. We start by giving an overview of Gesto's architecture and main features. Then, we analyze the metadata structure.

A. Overview

Goal. Gesto is a hierarchical architecture that aims at extending — requiring minimal changes — causally consistent cloud storage services with mechanisms to operate in the edge, bringing them closer to clients.

System Components. Gesto has the following components, which create a three-tier architecture, as depicted in Figure 1.

A *cloud storage service* is a causally consistent key-value storage system designed to run across a small number of datacenters. We assume, without loss of generality, that each datacenter replicates the whole application state. Also, we assume that each datacenter is linearizable [29].

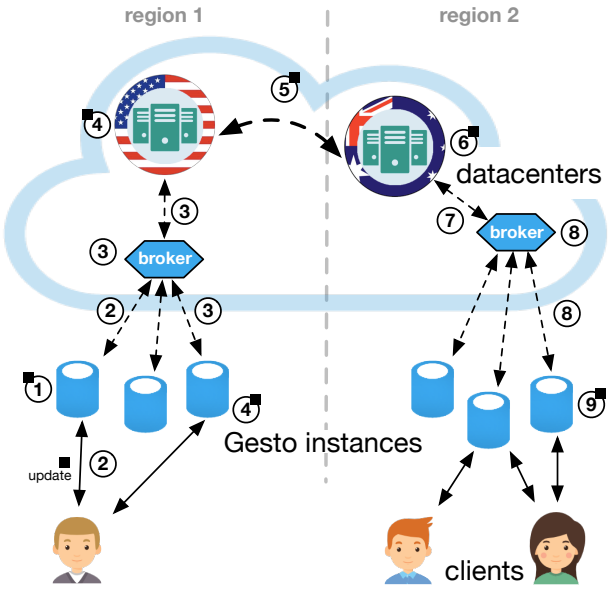

Figure 1. Gesto three-layer architecture. The numbers represent the steps that an update issued at the leftmost Gesto instance of *region 1* will follow until being propagated to the rightmost Gesto instance of *region 2*. The execution flow does not depict the payload transmission within regions (done by 5). Steps with ■ indicate that they require the update's payload to be executed.

A *regional broker* manages metadata by orchestrating its propagation within a region.

Gesto instances are partial replicas of the application state (the state maintained in the cloud storage service instantiated by datacenters) that reside in the logical extreme of the network, typically one in each cloudlet. Each Gesto instance has a local datacenter to which they are connected, namely their parent datacenter.

A *client proxy* connects clients to Gesto. It is in charge of propagating client requests to Gesto. Before propagating them, these are enriched with causal information that is used by Gesto to guarantee causality.

Execution Flow. Clients interact with the system through a client proxy by performing read and write operations. These operations can be sent to any replica (a Gesto instance or a datacenter). However, for better performance, clients should mostly interact with their closest replica. Before performing any operation, clients need to *attach* to a replica. The purpose of the attach procedure is to ensure that the target replica has a state that is consistent with the causal past of that client. Once attached, the client can issue read and write requests without further synchronization.

The process of detaching from a replica and attaching to another (for instance, to access a data item that is not cached in the origin replica) is called a *migration*. A client migration within a region is assumed to be a relatively infrequent operation (once attached, clients should perform multiple local reads/writes before migrating again), but it is not assumed to be rare. In fact, client migrations is

a requirement that derives from the assumption of partial replication, which is the only realistic assumption in face of resource-constrained cloudlets.

In the following example, we illustrate the sequence of steps that are executed from the point a client issues a write request in a given region, up to the moment the update is applied not only in other replicas of the same region, but also in replicas of remote regions. Note that the current paper is mainly concerned with the mechanisms used *inside* each region. However, Gesto is modular and can be combined with whatever mechanism the datacenter uses natively to maintain causal consistency across regions. Thus, the example below also illustrates the interplay between Gesto and the datacenter's native causal consistency mechanisms. The execution flow, depicted in Figure 1, is as follows:

Step ①: The replica (a Gesto instance) handles a local write request by assigning it a multipart timestamp (the format of this is discussed in §IV-B). Multipart timestamps (*MP_TS*) have enough information to order updates according to causality.

Step ②: The replica returns the *MP_TS* to the client, that can assume the completion of the write. Concurrently, the update is propagated to the regional replicas (including the parent datacenter) and the corresponding *MP_TS* to the regional broker.

Step ③: When the regional broker receives the *MP_TS*, it processes it and sends it to the regional replicas that store the associated object, but not to the original one.

Step ④: When a replica receives both the update payload and the corresponding *MP_TS* coming from the regional broker, it installs the update and makes it visible to local clients.

Step ⑤: When the regional datacenter has made the update visible, it generates the metadata required by the native inter-datacenter causal consistency protocol. From the perspective of this protocol, the update installed by Gesto is handled as its own local update. Then, the update is propagated to other datacenters using the native protocol. Gesto is oblivious to the internals of the inter-datacenter replication mechanism. On the receiving side, there needs to be some changes on the update, before it is applied in the Gesto instance:

Step ⑥: The native inter-datacenter protocol ensures that the update is locally applied in causal order. Just before storing it, the datacenter generates a new Gesto *MP_TS* and locally installs the update.

Step ⑦: The datacenter propagates the update payload to the regional replicas. Concurrently, it propagates the associated *MP_TS* to the regional broker.

Steps ⑧ and ⑨: They are similar to ③ (considering the datacenter as the origin replica) and ④ respectively.

As noted in §III, Gesto does not require the network path used to propagate the update payload to be the same as the path used to propagate the corresponding multipart timestamp. Gesto is agnostic to the process of propagating the update payload (different Gesto deployments can use different strategies to propagate update payloads), because remote updates are applied by the order that metadata arrives. In our exposition, we abstract this by calling a payload non-blocking multicast primitive whenever the content of updates needs to be shipped to other replicas. We make very few assumptions about this service and we do not require updates to be received in order. Differently, multipart timestamps follow a specific path. As we will explain, the path used to propagate multipart timestamps and the changes performed to these timestamps during the propagation process are instrumental to keep the size of the timestamps small.

B. Multipart Timestamps

Gesto associates with each update a multipart timestamp. A multipart timestamp is only meaningful within the region were it was generated, lacking meaning at remote regions. Clients also maintain a multipart timestamp, summarizing their causal past, that is handed to a replica when attaching. The structure is key to guarantee fast and consistent intra-region client migrations.

A multipart timestamp includes two entries, namely the *local timestamp* (*lts*) and the *regional timestamp* (*rts*). In turn, each entry is a $\langle src, clock \rangle$ tuple that includes a source field (*src*), for identifying the creator of the timestamp, and a clock field (*clock*), which captures the creation time.

The *lts* entries are created by the replica where the write is issued. Therefore, the *src* field of the *lts* indicates the origin replica of the update. It is used by clients attached to that replica to keep track of the most recent *local* update they have observed. *A local timestamp can only be compared with other local timestamps created in exactly the same replica.*

The *rts* entries are always assigned by the regional broker. Whenever an update created in an instance is propagated to other remote instances, it is assigned a new regional entry by the broker. The *rts* is used by clients to keep track of the causal dependencies for remote updates. *It is the regional timestamp that allows the ordering of updates generated in different replicas of a given region.*

The use of a single regional entry to keep track of all remote causal dependencies prevents the client from storing a different *lts* for each instance that exists in a region. Otherwise, metadata could reach very large sizes. Intuitively, Gesto keeps track of updates that happen in the local instance with more accuracy than for remote ones, given that the broker *merges* all remote updates in a single entry. The rational for this asymmetry is that we assume that clients perform much more local operations than remote ones. As we will show, experimental results confirm that this model offers a good trade-off between metadata size and accuracy.

V. Protocols

In the following sections, we detail the basic protocols of Gesto. Namely, the read and write protocols, the update propagation protocol, and the *migration* protocol to allow the movement of clients throughout the network. Due to space

constraints, the algorithms' pseudocode is only available in the full thesis.

A. Reads and Writes

Clients read and write data items to their attached replica through their client proxy. Usually, this replica corresponds to the preferred one.

Read. Gesto's read protocol executes without any synchronization, as we expect reads to be the most common operation. Thus, the client is not required to forward its own multipart timestamp. This happens because the migration procedure ensures that the local instance is up-to-date with the client's past, before allowing the client to make other operations. The replica handles read requests by returning the stored value alongside its corresponding timestamp. If the replica does not cache the target key, an error is returned. In turn, the client incorporates the update (together with the update's causal dependencies) to its causal past by merging the two multipart timestamps. It is done by picking the greatest regional entry and, if the update was generated in the contacted replica, the largest local entry.

Write. Firstly, the client generates an unique identifier for the update. Then, it forwards the update request to its preferred replica, together with its multipart timestamp. The receiving replica assigns a new multipart timestamp to the update, by preserving its regional component and getting a new local component. The new entry must have a higher clock than the last operation recorded in the client's past. This guarantees that the timestamp assigned to the update is greater than the timestamps associated to any operation observed by the client, which is essential to enable consistent client migrations. If the key is locally cached, the update is applied to the local key-value store. Then, the metadata associated with the update is sent to the regional broker and the payload is propagated to all other regional replicas, including the parent datacenter. Concurrently, the new timestamp is returned to the client. Finally, the client incorporates the new update into its causal past by simply overwriting its old multipart timestamp with the new one.

B. Intra-regional Update Replication

As mentioned, Gesto decouples the dissemination of data and metadata. In order to apply a remote update, a replica must have received both the associated payload and the metadata. The payload is directly received from the originating replica and can arrive out of order. So, they are buffered until they can be applied in causal order. Metadata is received in an order that respects causality. Therefore, updates are applied in that exact order.

Metadata goes from the origin replica to the regional broker. The broker merges the metadata produced in different replicas into a single stream, consistent with causality. Then, it propagates it to the other regional replicas. For each replica, the broker only sends the metadata associated to updates on items that are cached, filtering out the rest. This enhances performance, because it avoids processing irrelevant metadata and reduces the network traffic.

To simplify the process of creating a causally consistent stream at the broker, we require the use of FIFO channels between the broker and the replicas. We also enforce that replicas send multipart timestamps to the broker in local timestamp order. Under these constraints, the stream is causally consistent if the broker assigns regional timestamps respecting the order by which it receives multipart timestamps from replicas. In fact, an update b originating at some replica 1 can only depend on an update a originating on some other replica 2 if a was visible at 1 before b was issued, such that the client that issued b could have read a. This means that the metadata associated to a must have been received by the broker before b's metadata. Note that concurrent updates from different replicas may be serialized by the broker in any order.

There are a few subtle issues that are worth mentioning. In case of suffering a timeout, clients can submit the same write to multiple replicas, assigning it more than one multipart timestamp. Still, the update is locally applied only once, when the first multipart timestamp is received. Also, clocks are only loosely synchronized, which enables the local application of a remote update with a timestamp clock value that is greater than the local wall clock. To ensure that clock values match the causal order, the replica may have to wait for its wall clock to catch up when issuing new timestamps.

C. Client Migration

A migration is the procedure that allows a client to attach to a new replica (the *target* replica) after being previously attached to another one (the *origin* replica). A client may migrate if its current preferred replica does not replicate a data item, it became unreachable, or the client has physically moved, being closer to other replica. When faced with a cache miss, the client may directly migrate to the local datacenter or, when this information is available, to another cloudlet that also replicates the desired data item. Migration is supported by an *attach* operation, which is mandatory, and by a *snapshot* operation, which is optional and has the purpose of reducing the migration latency.

Attach. In order to perform a migration, a client issues an attach request to the target replica. The client's multipart timestamp is sent as a parameter, because it captures its causal past. In order to successfully attach to a target replica, the client must wait until that replica has a consistent state. To track which updates have already been locally applied, each replica maintains some bookkeeping information about their own region. Namely, a replica keeps track of the last update it has applied from each remote replica and the highest regional timestamp that has been locally applied. Thus, a replica can satisfy an attachment request as soon as the last received *lts* from the origin and the highest regional timestamp are, respectively, higher or equal than the local and regional entries of the client's multipart timestamp.

The second part of the attachment procedure consists on generating a new multipart timestamp that allows the client

to interact with the target replica. Since the client has not yet observed any state on the target replica, the clock of the local entry of the multipart timestamp is simply assigned with value 0. The clock on the regional entry is set to the most recent update that has been previously observed by the client. For this, the target replica checks the regional timestamp that was assigned by the broker to the last operation performed at the origin replica and that was received at the target replica. Then, it compares this to client's regional timestamp and keeps the highest value.

Snapshot. The snapshot operation is executed at the origin replica. In spite of being optional, this operation may speed-up the attachment to the target replica. The reason to implement a snapshot operation is the following. Assume that a client reads some object x on the origin replica with timestamp t_x, which makes it have t_x in the local part of its timestamp. When the client attaches to a target replica, it will need to wait until the target replica receives a timestamp greater or equal to t_x from the origin replica. Unfortunately, it may happen that the target instance does not cache object x. In fact, the target replica may cache very few items that are also cached at the origin replica. Thus, a higher timestamp from the origin replica may take an arbitrary amount of time to reach the target replica. The snapshot operation aims at ensuring that the target replica quickly receives an update.

To force an update of the metadata on the target instance, snapshot is implemented by emulating an update that is performed at the origin replica and that needs to be propagated to the target replica, thus prompting the propagation of the associated metadata. To avoid incurring the full cost of updating a data object, this operation is implemented as an update on a *faux* object that consumes no state. These updates do not have unique identifiers and do not required the exchange of payload messages; they only trigger metadata updates on the origin replica, regional broker, and target replica. There is a faux object for every pair of replicas.

Liveness. Since snapshot operations are not mandatory, we need a mechanism to ensure that migrations complete in bounded time, even when the objects that are replicated at the origin and target replicas are not updated for long periods. Thus, in absence of further updates, each replica periodically executes a spontaneous snapshot. This bounds the migration latency and guarantees that, if no failures occur, a client migration will always complete.

D. Inter-regional Update Replication

By design, Gesto is only concerned with keeping track of causality within a region and relies on any pre-existing protocol that is natively supported by the datacenter for inter-regional replication. This allows Gesto to be plugged into an existing system, in order to augment it with support for edge computing, with minimal changes to the inter-datacenter replication protocols already in place.

The inter-regional replication works as follows. When a datacenter installs a new update, either issued by a local client or originating at a regional replica, it generates whatever metadata is required by the native inter-regional protocol. Then, the datacenter ships the update, together with the inter-regional metadata, to remote datacenters. Note that Gesto is oblivious to the structure of the inter-regional metadata and to how it is generated. Similarly, the Gesto multipart timestamp does not need to be shipped to remote datacenters and can be ignored by the native protocol. This ensures the modularity of the design.

Eventually, a receiver datacenter will decide that it is safe to locally install the update. When a datacenter installs a remote update, it generates a fresh Gesto multipart timestamp for its own region. This multipart timestamp contains a local entry that is greater than any local timestamp ever assigned by the datacenter, and a regional entry that is set to 0. This is sufficient to guarantee that any client who reads this update from the datacenter can safely migrate to other regional replicas. Finally, the datacenter acts as if the update was issued by a local client: (i) installs the update, together with the multipart timestamp; (ii) sends the payload to all regional replicas; (iii) sends the timestamp to the regional broker. The broker will process the metadata as it was an update originating in its region and propagate it to the regional replicas, which in turn will make the remote update visible to local clients.

E. Correctness

We have derived proofs of correctness for Gesto's algorithms. Due to space constraints, these were not included in this paper, but can be seen in the full thesis.

VI. Reconfiguration and Faults

Like the correctness arguments, the reconfiguration strategies were not included in this paper. Furthermore, they are not evaluated, leaving them as future work.

VII. Implementation

In order to ease the implementation of Gesto and of the other systems (eventual consistency, COPS [13], Occult [19] and Saturn [20]) for making a fair performance evaluation, we have created a framework for edge storage. It considers the following components: *clients*, which are in charge of interacting with the system; *client receivers*, that get the client requests and redirect them to the right internal component; *partitions*, that store the replica state and reply to read/write requests; *timestamp senders*, that export update metadata for their replication; *timestamp receivers*, which get the remote update metadata and manage their application; *brokers*, that create a network among regional replicas. Overall, there are two possibilities for connecting the different components, due to separating (or not) the propagation of the update payload from the metadata. Solutions that make this separation, like Gesto and Saturn, use all the components. The eventually consistent system, COPS and Occult use clients, client receivers and partitions.

	Lu	Ly	Nc	Nt	R	S
Li	5.7	6.6	4.8	13.0	13.6	9.7
Lu	-	8.5	1.2	15.0	15.6	11.7
Ly	-	-	7.5	6.9	7.4	3.5
Nc	-	-	-	14.0	14.6	10.7
Nt	-	-	-	-	0.8	10.0
R	-	-	-	-	-	10.6

Table I
AVERAGE LATENCIES (HALF RTT) IN MILLISECONDS AMONG LOCATIONS: LILLE (LI), LUXEMBOURG (LU), LYON (LY), NANCY (NC), NANTES (NT), RENNES (R) AND SOPHIA (S).

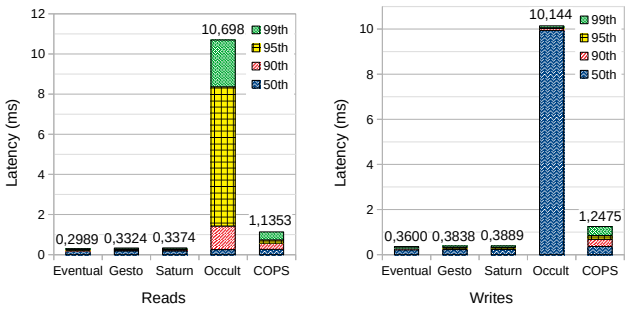

Figure 2. Each system's read and write latencies.

The framework was developed on Linux, using Erlang/OTP as the programming language. Each node executes the Erlang R16B02 virtual machine. Note that more recent releases of the virtual machine exist (at the moment of writing, the latest stable release is version 21), but this exact virtual machine was needed for running Basho Bench [30], a benchmarking tool that simulates the clients of each system.

Although the details from the implementation of each system are in the full thesis, here we just mention the most relevant decisions. Eventual consistency works as the baseline for the best achievable results, because it does not have to manage complex metadata. COPS was converted to a version that supports partial replication. The compression of client metadata is only done when the write is done to a fully replicated object. Otherwise, the returned version is added to the list of dependencies, like a read operation. Occult uses the temporal metadata compression, with a multipart timestamp with a maximum of ten entries. The datacenter is the master of all shards. At most, clients retry three times at their local replica for getting a consistent read. Afterwards, they must read from the master shard. Saturn has an internal network of brokers, which may propagate updates to replicas or to other brokers. Eventually, updates will reach all replicas that replicate them. All these systems consider that there is only one region, with a datacenter and several cloudlets.

VIII. EVALUATION

Our primary goal is to determine if Gesto can extend causally consistent cloud storage services to efficiently operate in the edge. Thus, our evaluation answers the following question: *can Gesto simultaneously offer low local operation latency, high throughput, fast update replication, scalability, and fast client migration?*

Setup. All experiments were run on the Grid'5000 [31] experimental platform using fully-dedicated servers. Each server runs Ubuntu 14.04 and has two physical CPUs ranging from four to eight cores each and from 4 to 64 GB of memory. We use a total of seven locations. Table I presents the average measured latencies among these. Our experiments focus on the intra-region operation. Thus, we place the single datacenter and the broker in Lyon, because of its centrality. The remaining locations have a cloudlet.

Clients are co-located with their preferred replica, but in separate machines. Each client eagerly sends requests to its preferred replica with zero thinking time. Each experiment runs for two minutes. The first and the last ten seconds of each experiment are ignored, to avoid experimental artifacts. The remote update visibility is computed by storing the physical time at the origin replica when the operation is issued, and subtracting it from the physical time at the target replica when the operation completes. To synchronize the clocks, each machine runs the NTP protocol before starting the test. Basho gives the other results.

Workloads. We experiment with two different synthetic workloads: **W1** and **W2**. In W1, clients only perform local operations with a distribution of 90% reads and 10% updates. In W2, clients execute a mixture of local operations and migrations, with a distribution of 70% reads, 10% updates and 20% migrations to any replica. In both workloads, we can tune the client access pattern to control the number of direct causal dependencies associated with each operation. For instance, forcing a client to read n data items and, only then, to perform the write, approximates the number of dependencies to n. W2 is just considered on §VIII-E, but W1 is used on all experiments.

A. Local Operation Latency

The first experiment compares the systems in regard to their client read and write latencies (plots on the left and right of Figure 2, respectively). The majority of the systems has faster reads than writes, with Occult being the exception. Its writes take almost the same time (there is marginal difference between the median and the 99^{th} percentile), because they are always made on the datacenter. But, in reads, there is the possibility of the local replica replying with an inconsistent version. So, in the worst case, reads take as long as writes, plus the latencies from the retries. In practice, the datacenter is visited less than ten percent of the times, because the 90^{th} percentile is below two milliseconds.

Looking at the remaining systems, eventual consistency has the combined lowest latencies, for not having to deal with the creation of metadata. Saturn and Gesto have virtually the same results, because they support fast metadata creation and the serialization of updates is not on the client path. COPS cannot match their performance, because of its explicit dependency check. When there is an update to the

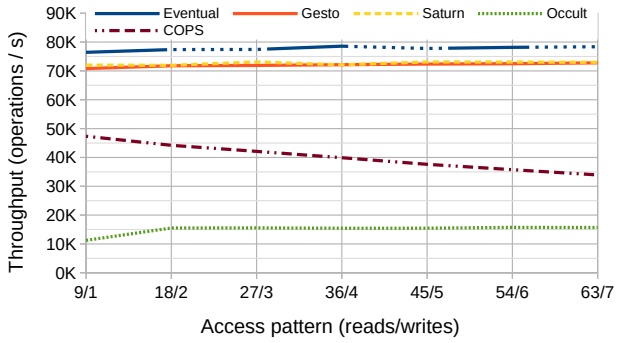

Figure 3. Each system's throughput for a given access pattern, while maintaining the same read/write ratio.

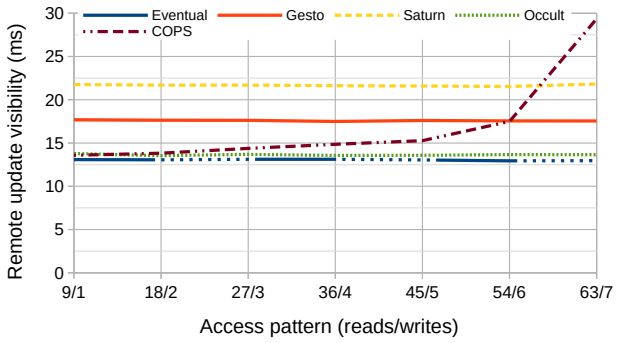

Figure 4. Each system's 90^{th} percentile of remote update visibility for a given access pattern, while maintaining the same read/write ratio.

local state, the server looks for pending dependencies that got fulfilled. The higher load delays client operations.

B. Throughput

The throughput of local operations for a given client access pattern is displayed in Figure 3. Not surprisingly, the eventually consistent system makes the most operations, mainly due to not managing any metadata. Gesto and Saturn are not far away (they are lower by less than ten percent). Between them, there is no significant difference, because the serialization of remote updates is not on the client path and their propagation is done on the background. COPS is the only system that is heavily impacted by the client access pattern. When the client's dependency list increases, the costs involved in the explicit check are large enough to incur in a system slowdown. At this experiment, Occult exhibits the lowest throughput, because of its high write latency. However, in §VIII-D, there is an increase in the number of clients, showing that it supports a fair amount without overloading. Initially, this system had even lower throughput, because the datacenter had to propagate the update to all replicas and the amount of updates that arrived at each instance was the highest. Furthermore, clients write to the datacenter and, when trying to read from their local replica, the probability of getting a non-consistent version is higher, leading to more retries.

C. Remote Update Visibility

In Figure 4, there are the remote update visibility latencies for each system's access pattern. Like in §VIII-B, only COPS shows a negative evolution, with all the other systems keeping an almost constant remote update visibility latency. Eventual is still the system with the best performance, but now is closely followed by Occult (COPS is near, when clients frequently write to the fully replicated object). Given its optimistic approach to causal consistency, Occult's slave replicas apply remote updates as soon as they receive them, without making any checks. The node organization together with the FIFO channels ensure that updates are received in the same order as they left the master, which preserves the consistency. In the worst case, Gesto shows a smaller

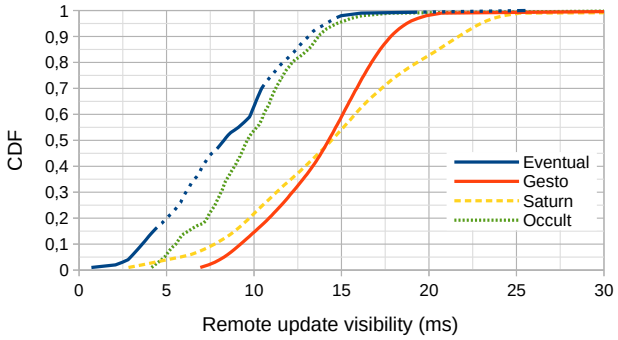

Figure 5. Cumulative distribution function for each system's remote update visibility.

remote update visibility latency than Saturn. The more direct propagation of updates and the lower number of hops in the internal nodes are what contribute to such fact.

For visualizing each system's overall remote update visibility latency, Figure 5 shows the corresponding CDF. This time, clients read a variable amount of objects between updates, leading to a more generalized behavior, while keeping workload W1. COPS was excluded from this experiment, because there is a more relaxed control of the amount of dependencies and it has high probability of bottlenecking.

The experimental results offer some interesting insights. As expected, eventual consistency is always the fastest, with the results directly matching the latency among the different physical locations. Occult gets closer to eventual consistency as the amount of remote updates increase. Given that all writes are done in the datacenter, geographically close locations incur in a higher remote update latency. This becomes less noticeable as nodes get farther apart, because the detour has a lower impact. Saturn explores the proximity of its instances by having more serializers connecting them, resulting in a fast propagation. However, it quickly slows down (it is only faster on the first five percent of all updates), because the hops introduce more delays. This is also visible when Gesto surpasses Saturn (around half of all updates).

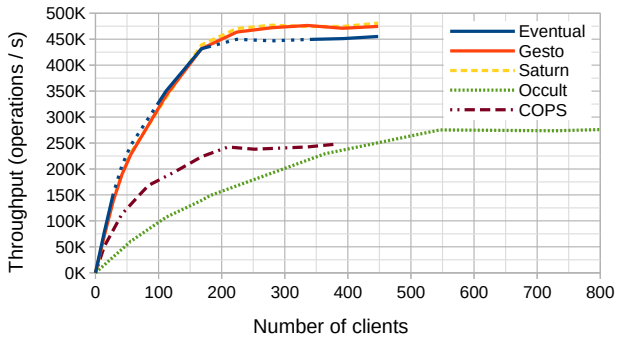

Figure 6. Each system's throughput for a given number of clients.

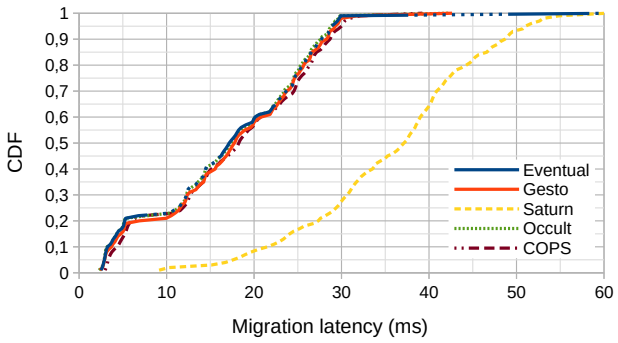

Figure 7. Cumulative distribution function for each system's migration latency.

D. Scalability

This experiment focuses on understanding how many clients are needed to reach each system's maximum throughput. Clients are equally shared among all instances. The results are depicted in Figure 6.

Eventual consistency should always have the highest throughput, but this does not happen. From 150 clients, it starts to slow down and is surpassed by both Gesto and Saturn. However, the problem is not with eventual consistency, but with the other two. After a careful analysis, the components for exporting and importing metadata are bottlenecking the propagation of updates (their queue grows faster than what they can process). This reduces the amount of delivered remote updates, allowing the response to more client requests. Optimizations to the code might solve the issue. When looking at a number of clients below 150, the performance of Gesto and Saturn is near optimal, which was also shown in §VIII-B. COPS has a promising start, but the dependency check quickly degrades performance. Its maximum throughput is roughly 45% lower than both Gesto and Saturn. Finally, Occult begins with the worst performance, due to the high client write latency. Increasing the load provides a steady evolution, getting its maximum throughput when serving 550 clients, which exceeds COPS. The simple protocols and the client side consistency check are the reasons behind such numbers.

E. Migration Latency

The last experiment is to assess the cost of performing a client migration, mainly the additional latency that is induced by this operation. There are clients with workloads W1 and W2. The CDF with each system's migration latency is displayed in Figure 7.

Again, since the eventually consistent system does not enforce any invariant when performing this operation, it is the one that offers faster migrations. Occult, Gesto and COPS exhibit near optimal migration latency. Occult relies on optimistic causal consistency to show the updates as soon as they arrive, reducing the probability of clients having to wait for consistent versions. Gesto's multipart timestamp accounts for client's read-heavy workload. Since the migration does not follow a write, the client synchronization time is reduced. Usually, the migration finishes without additional latencies. COPS exhibits a little higher latencies, because of the cost associated with the processing of its metadata. Finally, Saturn has the worst latencies, due to not optimizing client migrations. It has to generate a label with a potentially high number of false dependencies, which has an impact on latency. The sequential execution of the label creation and the attachment to the target replica are also responsible for slowing down the process.

IX. Conclusions

In this paper, we have presented Gesto, a data storage architecture that aims at providing causally consistent storage on the edge. Gesto differs from previous work on causal consistency, because it considers novel challenges that are inherent to edge computing: high number of replicas and efficient support for frequent client migrations, which results from client movement and resource constrained replicas. Our solution uses a constant amount of metadata, regardless of the number of edge replicas in each region, offers low update visibility latency, high throughput, and supports quick migrations. Furthermore, it is modular and can be plugged into a pre-existing protocol that ensures causal consistency across regions, without requiring changes to the native mechanisms.

Acknowledgments

This work was performed at INESC-ID and was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds as part of projects PTDC/EEI-COM/29271/2017 (Cosmos) and UID/CEC/50021/2013. This work has been performed in collaboration with Manuel Bravo, a previous member of the Distributed Systems Group at INESC-ID.

References

[1] M. Satyanarayanan, "A Brief History of Cloud Offload: A Personal Journey from Odyssey Through Cyber Foraging to Cloudlets," *GetMobile: Mobile Computing and Communications*, vol. 18, no. 4, pp. 19–23, 2015.

[2] C. Streiffer, A. Srivastava, V. Orlikowski, Y. Velasco, V. Martin, N. Raval, A. Machanavajjhala, and L. Cox, "ePrivateEye: To the Edge and Beyond!" in *SEC '17*. San Jose, California, USA: ACM, 2017, pp. 18:1–18:13.

[3] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-Caching for Recognition Applications," in *ICDCS '17*. Atlanta, Georgia, USA: IEEE, 2017, pp. 276–286.

[4] M. Satyanarayanan, P. Gibbons, L. Mummert, P. Pillai, P. Simoens, and R. Sukthankar, "Cloudlet-based Just-in-Time Indexing of IoT Video," in *GIoTS '17*. Geneva, Switzerland: IEEE, 2017, pp. 1–8.

[5] G. Ricart, "A City Edge Cloud with its Economic and Technical Considerations," in *PerCom '17*. Kona, Hawaii, USA: IEEE, 2017, pp. 599–604.

[6] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[7] M. Patel, Y. Hu, P. Heédeé, J. Joubert, C. Thornton, B. Naughton, J. Ramos, C. Chan, V. Young, S. Tan, D. Lynch, N. Sprecher, T. Musiol, C. Manzanares, U. Rauschenbach, S. Abeta, L. Chen, K. Shimizu, A. Neal, P. Cosimini, A. Pollard, and G. Klas, "Mobile-Edge Computing – Introductory Technical White Paper," ETSI, Tech. Rep., Sep. 2014.

[8] F. Bonomia, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *MCC '12*. Helsinki, Finland: ACM, 2012, pp. 13–16.

[9] P. Hao, Y. Bai, X. Zhang, and Y. Zhang, "EdgeCourier: An Edge-hosted Personal Service for Low-bandwidth Document Synchronization in Mobile Cloud Storage Services," in *SEC '17*. San Jose, California, USA: ACM, 2017, pp. 7:1–7:14.

[10] E. Ahmed and M. H. Rehmani, "Mobile Edge Computing: Opportunities, solutions, and challenges," *Future Generation Computer Systems*, vol. 70, pp. 59–63, 2017.

[11] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A Multi-tier Cloud Computing Framework," in *SEC '17*. San Jose, California, USA: ACM, 2017, pp. 20:1–20:13.

[12] 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.

[13] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS," in *SOSP '11*. Cascais, Portugal: ACM, 2011, pp. 401–416.

[14] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "PRACTI Replication," in *NSDI '06*. San Jose, California, USA: USENIX Association, 2006, p. 5.

[15] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, "Write Fast, Read in the Past: Causal Consistency for Client-Side Applications," in *Middleware '15*. Vancouver, British Columbia, Canada: ACM, 2015, pp. 75–87.

[16] A. v. d. Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa, "Legion: Enriching Internet Services with Peer-to-Peer Interactions," in *WWW '17*. Perth, Australia: IWWWCSC, 2017, pp. 283–292.

[17] C. Gunawardhana, M. Bravo, and L. Rodrigues, "Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication," in *USENIX ATC '17*. Santa Clara, California, USA: USENIX Association, 2017, pp. 83–95.

[18] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks," in *SoCC '14*. Seattle, Washington, USA: ACM, 2014, pp. 4:1–4:13.

[19] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades," in *NSDI '17*. Boston, Massachusetts, USA: USENIX Association, 2017, pp. 453–468.

[20] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A Distributed Metadata Service for Causal Consistency," in *EuroSys '17*. Belgrade, Serbia: ACM, 2017, pp. 111–126.

[21] P. Fouto, J. Leitão, and N. Preguiça, "Consistência Causal em Sistemas Geo-Distribuídos com Replicação Parcial," in *INForum '18*. Coimbra, Portugal: FCTUC, 2018, pp. 65–76.

[22] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[23] B. Charron-Bost, "Concerning the Size of Logical Clocks in Distributed Systems," *Information Processing Letters*, vol. 39, no. 1, pp. 11–16, 1991.

[24] M. Singhal and A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, vol. 43, no. 1, pp. 47–52, 1992.

[25] R. Schwarz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, vol. 7, no. 3, pp. 149–174, 1994.

[26] F. J. Torres-Rojas and M. Ahamad, "Plausible Clocks: Constant Size Logical Clocks for Distributed Systems," *Distributed Computing*, vol. 12, no. 4, pp. 179–195, 1999.

[27] A. Shoker, J. Leitão, P. Van Roy, and C. Meiklejohn, "LightKone: Towards general purpose computations on the edge," Tech. Rep., 2016.

[28] R. Escriva, A. Dubey, B. Wong, and E. G. Sirer, "Kronos: The Design and Implementation of an Event Ordering Service," in *EuroSys '14*. Amsterdam, The Netherlands: ACM, 2014, pp. 1–14.

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

[30] "Basho Bench," http://github.com/basho/basho_bench, Accessed: 2018-10-12.

[31] "Grid'5000," https://www.grid5000.fr, Accessed: 2018-10-12.