Mechanisms for Providing Causal Consistency on Edge Computing

Nuno Cerqueira Afonso
nuno.c.afonso@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. *Edge computing* is a new paradigm that aims at leveraging the increasing processing power and network bandwidth that is available to the devices that operate on the edge of the network. To this purpose, one aims at finding new techniques that allow to execute in a decentralized manner complex tasks, by combining processing on edge devices and processing in large datacenters (typically in the premises of large cloud providers). In this work we are interested in studying techniques that can ensure *causal consistency* to edge computing applications. Causal consistency is a key consistency criteria for many distributed applications and has been widely studied in other contexts. Unfortunately, most known techniques to enforce causal consistency may perform poorly in face of the large number of participants and dynamic topologies that characterize edge computing. We survey related work, focusing on performance, adaptability and fault tolerance, and discuss how these techniques can be extended to be applied on the edge.

Table of Contents

1 Introduction

The cloud computing paradigm is widely used today to provide services to end users. Cloud computing implements a model of computation where data is sent from the devices of end-users to a central location for processing. Cloud computing has many advantages. For instance, it may be more effective to run a large datacenter than many small datacenters. However, cloud computing also has several limitations. Given that more and more devices, such as TVs, media servers and consumer appliances, have significant processing capacities (what is called the Internet of Things or simply IoT), there is an exponential growth of sources of information. As the number of data sources increases, the amount of information that needs to be shipped to a central datacenter may raise scalability problems. Also, processing data in remote locations increases the latency experienced by end users.

To circumvent the limitations above, there is a growing interest in finding techniques that allow to distribute computations among datacenters and the edge devices themselves, a paradigm that has been called *edge computing*. As with any other distributed computing paradigm, the consistency of data is a central concern in edge computing. Among the several consistency criteria that have been proposed in the literature, *causal consistency* emerges as extremely relevant for edge computing. In fact, it has been shown that causal consistency is the strongest consistency criteria that can be enforced without risking blocking the system when failures or partitions occur [1] and thus ensuring availability in the well known Consistency, Availability, Partition-Tolerance (CAP) tradeoff formulation [2].

Given the relevance of causal consistency, techniques to implement this consistency model in cloud computing settings have been widely studied during the last decade [3–17]. Unfortunately, it is unclear which of these techniques, if any, can be applied to edge computing. In fact, edge computing is characterized by large number of participants connected by a highly dynamic topology, a setting that is substantially different from that of cloud computing, where the number of datacenters is relatively small and well connected.

In this document we survey the most relevant techniques that have been proposed in the literature to enforce causal consistency. We evaluate these techniques from different perspectives, namely performance, adaptability, and fault tolerance. Using these parameters, we discuss the potential limitations and advantages of each of these techniques for an edge computing setting. Based on the analysis, we propose a new architecture to support causal consistency in the edge.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3, we present the edge computing paradigm. Section 4 gives an overview of causal consistency. Section 5 describes the proposed architecture to be implemented and Section 6 describes how we plan to evaluate our results. Finally, Section 7 presents the schedule of future work and Section 8 concludes the report.

2 Goals

This work addresses the problem of ensuring causal consistency on the edge computing paradigm. To achieve this goal, we do a comprehensive survey of existing techniques to support causal consistency in different scenarios, in order to derive a solution that can offer the best performance for the large number of nodes and dynamic topologies that characterize edge computing.

> *Goals:* We aim at building a new update propagation system that supports causally consistent updates on edge computing scenarios.

In this setting, nodes may communicate among themselves and also with a cloud service. The network must be able to support fast reconfiguration for nodes that join or become unreachable. The heterogeneity of devices requires our system to support genuine partial replication, given that will be impossible for edge devices to maintain full replication of the data involved in the global computation. The updates must be replicated with low latency, so that the interested nodes quickly receive them even with a dense network. The system should be fault-tolerant and updates that become visible should not be subsequently lost due to failures. For scalability, the size of the metadata required to enforce causal consistency must not depend on the number of participants neither on the number of objects.

For a reason that will become clear later in this report, our system will be inspired by the work in Saturn[17]. We will study how to adapt Saturn to operate on different network topologies and design new reconfiguration strategies that can perform well in face of dynamic networks. We will evaluated our solution considering different aspects, namely: tolerance to failures, network reconfiguration latency on the addition and removal of nodes, update propagation latency, and amount of missing updates. Our work should produce the following expected results:

> *Expected results:* The work will produce i) a specification of a new causal system; ii) an implementation for edge computing; iii) an extensive experimental evaluation of the performance and adaptability to changes.

3 Edge Computing

The term *edge computing*[18] captures a model of distributed computing that aims at leveraging the processing capacity that exists in the devices that operate on the edge of the network. Edge computing augments the cloud computing paradigm, where all processing is performed in datacenters, in order to achieve better scalability, lower latencies, better data privacy, and a more efficient usage of resources (including a more effective energy consumption). In edge computing, processing is performed cooperatively by edge devices and cloud servers. Which computations are performed on which of these components depend on a number of factors, including both the capacity of the nodes and the latency requirements.

In the following sections, we first provide two illustrations of edge computing applications and then identify the main challenges that emerge when developing middleware to support edge computing.

3.1 Use Cases

We will present two different use cases that illustrate how edge computing can improve the user experience and reduce the computation time. The first consists in the cooperative creation of indexes that support faster searches. The second consists in processing events on the edge and aggregating results from multiple events before these are propagated to the cloud.

Cooperative Index Creation Nowadays, media companies use different cloud providers for different types of content. Usually, unreleased content is stored in a private cloud and already released content is stored in one or more public clouds.

For a better user experience, companies' websites have a search bar. Given the large number of possibilities, there must be a precalculated index that hints where the content may be, reducing the waiting time.

If some node is searching for a keyword that is not indexed, it will be in charge of the index creation. It must search in the company's public clouds and gather the results. Because of the high traffic load, it is probable that different nodes create indices for the same things. For instance, if it is the release day of a popular show, then there will be a high amount of concurrent similar searches.

On the other hand, unpopular or old content can be removed from the cloud. In the client perspective, we do not want to receive information about something that cannot be displayed. So, the removal of content must result in the removal of the associated indices.

For both situations, the management of indices must be carefully handled. When different nodes create equal indices, the results must be merged. Other issue surfaces from concurrent computations indexing non-existing objects. Therefore, causal consistency is the solution. Its benefits are explained in Section 4.

Distributed Event-driven Computation Event-driven computation is an application domain where events are detected by edge nodes and then processed by the infrastructure. Examples of existing concurrent solutions are complex model creation for decision support systems and MapReduce[19] jobs. On a traditional cloud based implementation, all events are shipped to a central datacenter where they are processed. However, in many applications, there is no need to maintain information about individual events but only about aggregated results from different events. For instance, the final model used for classifying if a patient will show up for his appointment. Each event was a previous record from a different patient, but, in the end, we will only desire to have the classifier without the detailed data.

In this case, the edge computing paradigm allows events to be processed by edge nodes, that can aggregate results from different events before shipping them to the cloud. Even if the processing required by each event cannot be performed by a single edge device, it is possible to model the processing job as a sequence of tasks that can be performed by different edge nodes.

As a way of increasing the probability of handling all events, there may exist redundant sequences of computations. With the higher level of concurrency, a node may receive older or repeated versions of an event and related events may arrive in a random order. If a programmer desires to have an *exactly-once* processing semantics and track partial order in a scalable manner, causal consistency is the way to go, using a conflict-resolution strategy.

Considering the example of a model creation, we chose decision trees. Each subtree is delegated to a different edge device, being the starter device responsible for merging the partial results. In order to reduce the amount of repeated computations, each participant stores its state in a regular manner. If there is a failure, the parent device must redelegate the intermediate work to other device. However, there may just have been a communication failure, leading to repeated results in two different participants. When merging the outputs, each device must be able to detect which are the newer versions of the subtrees and discard duplicate updates.

3.2 Challenges and Requirements

To materialize the potential of edge computing, a number of challenges need to be overcome [18]. Here, we identify the most relevant ones.

The most obvious challenge is the heterogeneity of the nodes involved in the computation, which include a mixture of powerful and reliable nodes that run on datacenters and small, resource-constrained, unreliable nodes that run on the edge. Even among the edge nodes there is a large heterogeneity, since laptops and mobile phones are much more powerful than other IoT devices embedded in home appliances. An immediate consequence of this heterogeneity is that different devices have different capacities, and it is unrealistic to expect full replication of the data that an application manages.

Another challenge is that the number of devices is extremely large and, therefore, the size of control information required to enforce system guaranties, such as causal consistency, cannot be a function of the number of nodes. For instance, straightforward techniques such as relying on full vector clocks to keep track of causality are impractical.

Network conditions are also very heterogeneous and dynamic. Latencies among edge nodes vary widely and some parts of the topology can be very dynamic while others are mainly stable. Algorithms that operate on the edge must take this diversity into account.

Finally, the edge network is subject to a phenomena known as *churn*[20], which consists of a frequent change in the operational status of edge devices. They can become connected and disconnected very frequently, or even fail permanently.

4 Enforcing Causal Consistency

In this section, we start by introducing causal consistency. Then, we identify the main approaches that have been used to capture causal dependencies, in order to ensure that updates are applied in a order that respects causality. Afterwards, we survey the main systems that support causal consistency in geo-replicated settings. Finally, we discuss the potential limitations of these approaches when applied to edge computing.

4.1 Causal Dependencies

It is today well known that, in a distributed system subject to faults and network partitions, it is impossible to achieve simultaneously strong consistency, availability, and partition tolerance. This fact has been captured in the CAP theorem [2]. Therefore, we are faced with the need to drop one of these desirable properties. A reasonable tradeoff is to weaken the consistency of the system, while still providing precise properties that can simplify the application design. Causal consistency has the interesting property that has been shown to be the strongest level of achievable consistency without blocking the system [1].

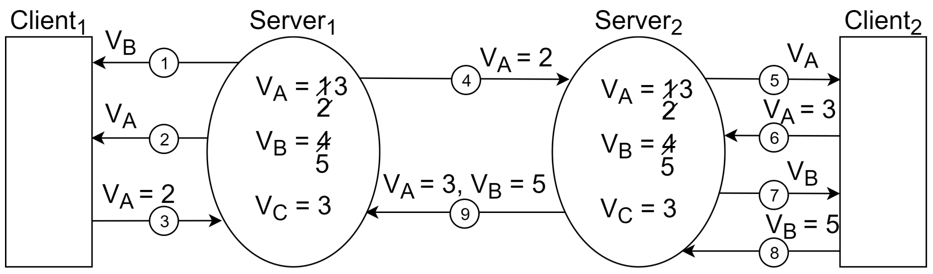

Fig. 1. Demonstration of the causal consistency rules.

Causal consistency ensures that updates are observed in a order that is consistent with causality. Knowing exactly which updates are causally related requires knowledge about the application semantics. Thus, most systems preserve the ordering among any updates that can *potentially* be causally related. The conditions for (potential) causal dependencies have been precisely identified by Lamport in [21]. Here, we illustrate these conditions with the help of Figure 1.

First, all operation that are executed in serial order by the same thread are assumed to be causally dependent. In the figure, this is illustrated in the sequence of operation executed by Client$_1$: Operation Op_3 is executed after Op_2, which in turn is executed after Op_1. This creates a dependency among these operations, denoted: $Op_1 \rightarrow Op_2 \rightarrow Op_3$.

On Client$_2$, we have a similar scenario and thus: $Op_5 \rightarrow Op_6 \rightarrow Op_7 \rightarrow Op_8$.

Furthermore, in a shared memory system, causal dependencies can also be created when a thread reads an update that has been created by another thread. In our example, given that Client$_2$ reads the version of A written by Client$_1$, operation Op_5 depends on Op_3.

Finally, causality is transitive. Since $Op_2 \rightarrow Op_3$, $Op_3 \rightarrow Op_5$, and $Op_5 \rightarrow Op_6$, we also have $Op_2 \rightarrow Op_6$.

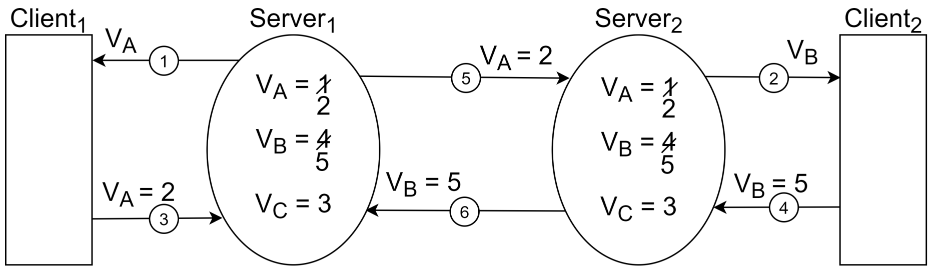

Fig. 2. Run with concurrent operations.

Sometimes, it is not possible to derive a causal dependency between two operations. In Figure 2, $Op_1 \not\rightarrow Op_2$ and $Op_2 \not\rightarrow Op_1$. Similarly, $Op_3 \not\rightarrow Op_4$ and $Op_4 \not\rightarrow Op_3$. Both pairs of operations are *concurrent* and they can be applied in any order.

To capture that fact that two operation are causally related, the system is required to maintain some metadata. Typically, this is achieved by timestamping each operation with some form of (physical or logical) clock. In the following section, we describe the main techniques that have been used in the literature to perform such timestamping.

Note that causal consistency, when used in settings where data is replicated, does not ensure that replicas become consistent, even if all updates are delivered to all replicas. Inconsistency is possible because concurrent updates can be applied in different orders to different replicas. Inconsistency can be avoided by using data types that ensure convergence, regardless of the order by which concurrent updates are performed (known as Concurrent Replicated Datatypes, or CRDTs). Systems that rely on CRDTs are SwiftCloud[9], Legion[10] and Cure[11]. Another alternative is to use some criteria to totally order concurrent updates, such as the *last-writer-wins rule*. In this case, it is possible to ensure that replicas eventually converge to the same state, a property that has been named *causal+ consistency*. Examples of systems that offer this type of guarantees are Cluster of Order-Preserving Servers (COPS)[4], Orbe[5], ChainReaction[8], Okapi[12] and Physical clock Optimistic Causal Consistency (POCC)[15].

4.2 Metadata Management

In order to ensure that updates can be applied in an order that respects causality, each update needs to be tagged with some amount of metadata. We call such metadata a *timestamp*. The timestamp encodes information regarding the causal past of the tagged operation. Since causal order is a partial order, each operation can have more than one predecessor in the causal graph. Therefore, to accurately capture the fact that the causal past of an operation can consist of multiple concurrent past operations, the timestamp may need to have multiple components, what is called a *multipart timestamp*.

Existing solutions differ on the techniques they use to encode each part of the timestamp and on the size of the multipart timestamp (i.e, how many "parts" are stored and exchanged). Furthermore, they also differ on the entity that is in charge of assigned timestamps to each update.

Timestamp Part Encoding In a multipart timestamp, all entries have the same encoding. According to the state of the art, we can have three different approaches: *logical clock*, *physical clock* and *hybrid clock*.

Logical Clock A logical clock is comparable to a counter and is incremented after each update (COPS[4], Orbe[5], PRACTI[6], Lazy Replication[7], Chain-Reaction[8], SwiftCloud[9] and Occult[16]).

Physical Clock A physical clock is like a watch, because it is monotonically increasing (Legion[10], Cure[11], GentleRain[14], POCC[15] and Saturn[17]).

Hybrid Clock A hybrid clock combines the previous two (Okapi[12] and Eunomia[13]). When the physical clock part is delayed, the new value is obtained by incrementing the logical clock. This is a way of avoiding unnecessary waiting time to reach a consistent value.

Timestamp Size As previously mentioned, different systems adopt timestamps with a different number of "parts". The smaller they are, the smaller the network bandwidth needed for exchanging them. Moreover, the storage space occupied with metadata is also smaller. However, it introduces a penalization in the visibility latency of concurrent operations. In the following sections, we will expose this issue and it will be discussed with the edge computing paradigm in mind.

In the survey of the existing solutions, we identified four different sizes: *one part per object*, *one part per server*, *one part per datacenter* and *one part for the entire system*. In the remainder of this section, we will analyze them alongside an example.

One Part per Object Looking at Figure 1, we can see an example of it. Different objects have independent logical clocks, but their dependencies must be related to prevent an incorrect update order.

The client's causal history is updated after reading or updating objects. If the read object version is newer than the previously seen, then the client updates the corresponding part on its causal history timestamp. Otherwise, it stays the same. After updates, the client can clean its causal history, storing only the part related with the creation timestamp of the operation. Taking the causal consistency transitivity rule into account, we conclude that this compression is safe, because all the future operations will potentially depend on the current update with all of its dependencies. These dependencies are obtained by the client's causal history at the time of the operation.

The update creation timestamp is calculated in the receiving server, after it has equal or newer versions than the ones in the dependencies. It gets the the highest timestamp for any local version of that specific object and increments it by one. The result will be tagged with the new object value.

Returning to Figure 1, Op_3 depends on Op_1 and Op_2. So, this will be expressed by keeping the version 1 for object A and 4 for object B in the Client$_1$'s causal history. Given that Server$_1$ stores the version 1 for object A, then the update timestamp will be 2. Afterwards, Client$_1$'s causal history only stores the version 2 for A.

New remote updates are processed in a similar fashion, also waiting for their dependencies to be fulfilled. On our example, both servers can immediately apply the remote updates, because they store the object versions in the updates' dependencies.

One system that adopts a similar kind of dependency tracking is COPS[4].

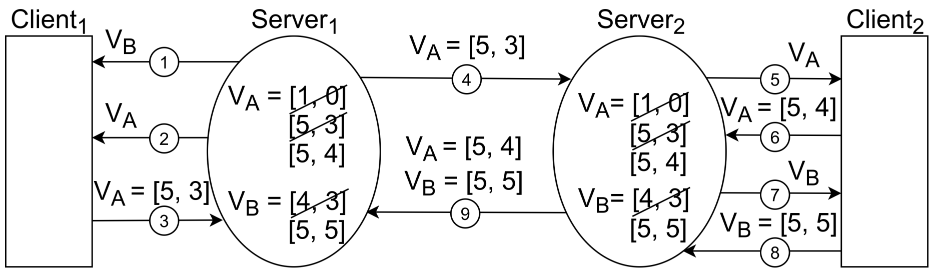

Fig. 3. Causal consistency with timestamps with one part per server.

One Part per Server In Figure 3, both servers have a full replica of all objects in the system. For simplicity, we assume that there are only two objects: A and B.

As we can see in the multipart timestamp of each version, there are two entries. The first belongs to Server$_1$ and the second to Server$_2$. Clients track their causal history by storing the highest partwise timestamp ever seen. It can come from reads or after creating updates. When a server receives an update from a client, it creates the new timestamp by incrementing its clock and replacing its part on the client dependencies timestamp.

For instance, after Op_1, Client$_1$'s causal history is tagged with the creation timestamp of the available version of object B ([4, 3]). Op_2 does not affect the dependencies, because the returned version of A has partwise smaller values than the version of B. Op_3 updates the version of A on Server$_1$. The associated dependencies are tagged with the client dependencies timestamp. The new version timestamp keeps the Server$_2$'s part, but changes the Server$_1$'s entry. Given that the current clock is at 4, its entry will be set to 5, resulting in [5, 3].

After receiving the remote update from Server$_1$, Server$_2$ can immediately make it visible, because it has received and applied all the updates until the highest predecessor of the Server$_1$'s timestamp part. Otherwise, it would have to wait for this condition to be fulfilled.

A system that follows this strategy is Lazy Replication[7]. PRACTI[6] and Legion[10] are similar, but allow partial replication of the objects. Orbe[5] has an alternative representation, displaying the different parts as a matrix. It assumes full replicas over datacenters and each datacenter has equal partitions to distribute the objects. So, the matrix has N rows (one per partition) and M columns (one per replica).

One Part per Datacenter Following the Orbe[5] perspective, we can reduce even more the size of the timestamp. If we compress all the operations of all servers inside a datacenter to a single part of the timestamp, it will have as many parts as the number of datacenters.

As in the previous category, the client's causal history timestamp is updated during both operations, storing the partwise maximum of the returned timestamps. However, the update creation timestamp may not come from incrementing the local server clock. In spite of being one possibility, we must guarantee that the values are monotonically increasing. So, the server may change its current clock value if its datacenter part in the client's causal history has a higher value. Then, it increments it by one, swapping the corresponding part.

If we consider that both Figure 3 servers belong to the same partition but in different datacenters, the displayed timestamps would be the same. In case of a client contacting a different partition, the result could be different, because it could have a newer part value for the current datacenter. When performing an update, the receiving server would eventually advance its local clock value.

Remote updates follow a similar approach as in the previous category. Usually, there is either an intra-datacenter protocol for calculating the latest updates received from foreign datacenters or First In, First Out (FIFO) channels for communication. They are strategies for ensuring that there is no older missing update.

Examples of systems that use timestamps with this size are: ChainReaction[8], SwiftCloud[9], Cure[11], Okapi[12], Eunomia[13] and POCC[15].

A small variation to this technique is to have one part per partition. One implication of such choice is the change of the internal structure of the datacenter, which will be explained in the following sections. The system that proposes this alternative is Occult[16].

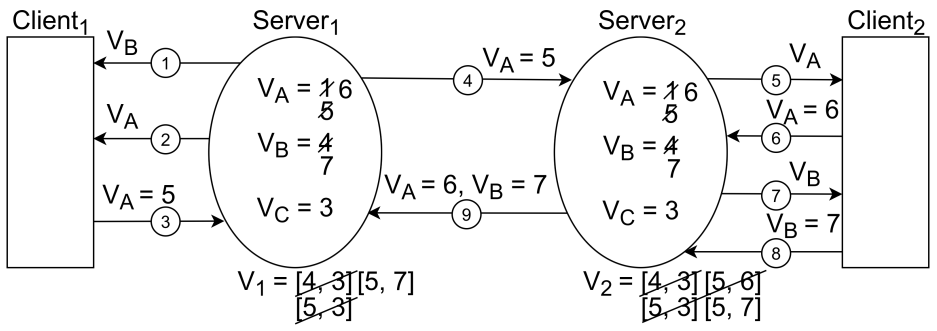

Fig. 4. Causal consistency with timestamps with one part for the entire system.

One Part for the Entire System The different versions of an object are identified by a timestamp with only one part. It has a fixed size, because it does not depend on the number of objects, servers, replicas nor partitions.

The client's causal history is represented by another timestamp. It is the maximum timestamp ever seen by the client. On reads, it is only updated if the returned object version has a higher timestamp. For writes, it is always changed, because the returned server timestamp is guaranteed to have a higher value.

Following the example in Figure 4, Client$_1$'s causal history is represented by 4 after Op_1. Op_2 does not affect it, given that the returned timestamp is smaller. The output timestamp of Op_3 must be larger than the one of the Client$_1$'s causal history and unique in Server$_1$. So, Server$_1$ assigns the timestamp 5 for the operation.

Remote updates are only safe to apply after being sure that all datacenters will apply them. To track it, servers still need to manage a vector-like structure with one entry per datacenter. Each entry stores the latest update timestamp received from the corresponding datacenter. Over time, the intra-datacenter servers exchange their vectors and compute the global minimum. The calculated value can be seen as a marker, allowing the application of remote updates with equal or smaller timestamps. In order to simplify the example, figure 4 does not show this situation.

One example of this kind of dependency tracking is GentleRain[14].

Saturn[17] introduces a novel design, considering that each datacenter has a specialized component for creating the timestamps. In the system, the timestamps are called *labels* and each server update is tagged with one.

Local updates are immediately visible, but remote updates have to wait until the corresponding label and data are received. Causal consistency is ensured by a predefined topology for propagating the labels between different datacenters.

Timestamp Assignment In distributed systems, the simplest approach for creating a solution is to *centralize* it. For the assignment of timestamps, it is one possibility. However, there are two more: *centralized server in the datacenter* and *server contacted by the client.*

Centralized Service Kronos[3] is a centralized service that is able to manage causal relations between any abstractions. Clients are responsible for creating events and giving them an order. The service can be contacted to get a consistent sequence of operations.

Centralized Server in the Datacenter The servers that store the objects contact a different entity inside the datacenter in order to tag updates. By doing this, there is no need for an intra-datacenter synchronization protocol. Saturn[17] is a system that uses this strategy.

Server Contacted by the Client On the majority of the surveyed solutions [4–16], the server contacted by the client is able to directly timestamp the update. Some of them need *stabilization protocols* before applying remote updates.

4.3 Existing Systems

In this section, we will present the surveyed solutions. For each of them, we start by explaining how a client can read objects. Then, we go through the client updates. Finally, we will analyze the chosen strategy for replicating the updates.

Kronos[3] Kronos[3] is a centralized service that keeps a *dependency graph.* Nodes represent *events* and edges define the causal relations between events.

When a client wants to obtain a *reference* to an event, it must contact Kronos[3]. A reference is an identifier of an event and it is managed by the service. After gathering a set of references, clients must know a possible order for applying the associated events. So, they must contact Kronos[3], providing the set as input. The output is a sequence with all the set items that does not violate causality. Different clients may get different results, because concurrent operations can be in a different order.

The addition of events to the graph also takes two phases. In the first, a client creates the event, receiving its reference. Then, it is able to assign an order, using the new event reference together with the other existing ones. There are two ways of ordering the events: *must* and *prefer.* Must specifies a strict ordering. Either all must conditions inside a call are fulfilled or the operation is aborted. The prefer ordering is used for assigning relations between concurrent operations. Its validity is only checked after Kronos[3] analyzes the must conditions. If they introduce inconsistencies, the operation does not abort, but the ordering is discarded.

Given that this is not a geo-replicated solution, there is no remote server that sends new operations. This contributes for a simpler design, but creates a performance bottleneck.

Kronos[3] is an isolated system that only deals with the causality of events. This introduces a novel abstraction, because different events may have much different granularities.

The biggest weakness is related with the way clients get a correct ordering of events. If a client has not received a reference that depends on other that was sent to the server, then the ordering will create an inconsistency at the client state. So, either the client must have all the references to get a global ordering or there must be an application dependent protocol to prevent this situation.

COPS[4] COPS has a timestamp with one part per object. The operations follow the same steps as described in Section 4.2. Each part has a logical clock.

Clients are in charge of managing their causal history. In COPS, this history is called *context*. Every time a client reads a new object, it adds the version to the context. Datacenter servers always give the latest consistent object version, without looking at the client context.

When a client wants to update an object, it starts by using the context to calculate the *nearest dependencies*. The context can be seen as a dependency graph and the transitivity rule is used for finding them. A nearest dependency is obtained when a version of an object does not happen before any other version. Afterwards, the client sends the object's key, new value and the nearest dependencies to the local server. This server has all the dependencies, because the client keeps a session with it. So, it can immediately apply the update, tag it with the dependencies and with the new timestamp. This timestamp is returned to the client and it replaces all the previous context.

The remote update protocol follows similar steps to the one in the local client update. When a server receives an object from other replica, it must check if the dependencies are fulfilled. Unlike the previous situation, there may exist other updates that belong to the dependencies and are not present. So, the receiver must delay the update until it can be safely applied.

An interesting feature is that a client can have different contexts for the same server connection. Given that dependencies are tracked per client context, it allows independent operations to have a lower number of common dependencies, reducing the amount of time remote updates wait to be visible (*visibility latency*).

The timestamp format introduces some compression of metadata, visible by existing only a scalar per object and not a vector with an entry per replica. This results in a similar amount of metadata as in Kronos[3], but with a higher visibility latency.

Orbe[5] This system explores the datacenter layout for choosing the number of timestamp parts. Each datacenter is divided in N partitions and there are M different replicas. Considering that each partition is assigned to a different server, it is possible to address it through a matrix-like structure with N rows and M columns. Clients track their causal history with a copy of this matrix. It is filled with the logical clock of the corresponding server.

A client read only sends the key of the object. The server replies with the latest available version's value, creation clock and the replica index. If the entry pointed by the index has a smaller value than the returned clock, then the client overwrites that position with the newer value.

To update an object, a client sends the key, the new value and its matrix to the local server responsible for the partition. The matrix corresponds to the update dependencies and is used on replication. Before tagging the creation time, the server has to increment its local clock, guaranteeing that the assigned value is unique. After storing both the object's value and metadata, the server replies with the update's clock and its index. Similarly to COPS[4], the client resets its causal history to default values, only saving the returned clock at the given index.

For replicating updates, each partition talks with its peers from different datacenters. The updates are sent by their creation order with all the associated data (key, value, creation clock, dependency matrix and replica index). To track the applied remote updates, there is a vector with one entry per replica and it has the clocks from the latest updates from each replica. The receiving server uses the matrix and the vector to verify the dependencies. For its partition, it looks at the corresponding row of the same replicas and compares it to the vector. If the vector is larger than or equal to the row, the server can go to the next step. For other partitions, every time there is a value different from the default one in the partition column, it means that the operation depends on other elements that the current datacenter may not store. So, the server contacts the servers that are in this situation, making an *explicit* dependency check. If they fulfill the dependencies, the remote update can be made visible and the corresponding vector's entry updated.

As an attempt to reduce the amount of metadata, the authors only store the matrix entries that are different from the default ones. This strategy allows clients to only store the contacted servers' clocks as a dependency for future operations. However, once a client reads a newer version from a server, the matrix must be updated in the corresponding entry. This can lead to a matrix that has all the entries filled, resulting in no gain for the worst case.

Given the coarser grain in which the dependencies are tracked, the impact of false dependencies will be greater. For instance, updates on one server will depend on all the previous updates applied in the same server. So, even when there is not a direct version dependency between objects, it will be treated as so. The increased waiting time before being able to apply the updates will have an instant impact in a higher visibility latency.

PRACTI[6] If we deconstruct the name of this system, we get its main features: partial replication (PR), arbitrary consistency (AC) and topology independence (TI). The granularity of partial replication is the lowest, because each replica chooses the individual objects it wants to store. Like Orbe[5], the timestamp has one part per server and it uses logical clocks.

When a node wants to participate in the network, it chooses three sets of other nodes: one for receiving the updates, other for prefetching objects for its *interest set* and another for requesting newer consistent versions of the locally stored objects. The interest set is any group of objects the node stores and it may have one or more interest sets.

Each node stores a global timestamp and as many timestamps as interest sets. They are updated at different times and express different things. We will analyze them when talking about the replication of updates.

Nodes read their local copy of the object when it is *valid.* The validity is checked on the timestamps and will be explained on the replication of updates.

Like reads, updates are locally done. Each version is tagged with the value of the node's logical clock. Afterwards, it is sent to the nodes that subscribed to the object updates.

The replication of updates is done by two types of messages: *invalidations* and *bodies.* The first kind contains only metadata and can be either *precise* (one message for object) or *imprecise* (one message for a group of objects and/or multiple updates), but must follow a causal ordering of delivery. Imprecise messages have a *target set*, *start* and *end* clocks. Bodies have precise metadata and the actual data and they do not need a specific order of delivery.

Upon receiving invalidation messages, the node may update the object's interest set timestamp as well as the one from the local node. The first is only updated with precise invalidations, but the second joins both. When the second has a newer version, the local state is *imprecise.* If the body for a precise invalidation has not yet arrived, the interest set's state is *invalid.* Invalid objects cannot be read, because it prevents inconsistencies for stronger models. However, for causal consistency, the older values can be returned, meaning that there are stale reads.

Supporting different levels of consistency has an impact in the amount of metadata and on the time taken to apply each update. Given that all the other systems are only defined for causal consistency, this will not be referred as a negative characteristic.

Although nodes can choose to keep only a subset of all existing objects, they must receive all invalidation messages. This is needed because of the unstructured topology, which could otherwise lead to inconsistent information on the nodes. To reduce the impact of the communication overhead, imprecise invalidations work as a summary of updates whose data is not locally stored.

The topology independence is particularly interesting when we have the edge computing paradigm in mind, because there is no rigid structure that forces a joining protocol for a new node.

Lazy Replication[7] Each node has full replication of the data, meaning that it must store all the objects in the system. Like PRACTI[6], there is not a predefined network topology and the timestamps have one part per node.

In Lazy Replication[7], there are two types of timestamps: *Unique Identifiers (UIDs)* and *labels.* The first tags updates and the second compresses multiple

UIDs. It creates the client causal history and the dependencies for all operations. Servers also keep timestamps for tracking their current version.

When reading an object value, the client sends the key and its label. The server waits until its version includes all the client history. The response has the latest object value and the corresponding timestamp. Before delivering the value, the client updates its label by including the partwise maximum between its previous label and the received timestamp.

When a client tries to write into a replica, it is constantly contacting the known entities by sending the object key, new value and its label. Possibly, it may send it to different ones and create duplicates. Given that we want an *at-most-once* message delivery, each client has its Client Identifier (CID). If we include it with the other update metadata, different replicas are able to recognize the duplicate updates. On the server side, if it is a new update, it increments its clock, updating its local label. The update UID is obtained by replacing the local server's part of the client label by the new clock value. As a response, the server sends the update UID, that is able to replace the client label while keeping the causal history. Finally, the client acknowledges the reception, marking the completion of the operation.

Updates are replicated through *gossiping*, ensuring that eventually every node sees them. The adopted strategy is a *log replication*. Each replica has its own log with all the operations that changed the local state. After some period of time, it sends it to its direct neighbors, which will filter duplicate entries and apply the missing operations. Filtering is done by looking at the update UID with the CID.

As a garbage collection strategy, messages that take more than a well defined time interval to arrive are discarded. The acknowledges of the client writes are included in the log, because they work as a marker to prune the older log entries. If future acknowledges are discarded by the time condition, the size of the log is reduced.

ChainReaction[8] This system uses timestamps that have one part per datacenter. Each part has a logical clock that belongs to a server of the corresponding datacenter.

As the name hints, ChainReaction[8] is built over a chain replication architecture. In the following paragraphs, we will explain how it affects the different operations.

Clients track their causal history with a table with one entry per viewed object. Its version timestamp and the *chain index vector* are the metadata associated with the object's key. The chain index vector has as many entries as the number of datacenters and it stores the latest position in the chain that the object has been replicated to.

If the chain index entry for the local datacenter is equal to the predefined chain length, the client may send a read request to any node of the chain. All of them will be able to answer the request without waiting for a remote update. Otherwise, the request can be made to any server until the one that is specified in the entry. When the client changes datacenter, the previous object version may not be available. So, the head of the corresponding chain must wait for it or make an explicit request to a datacenter that has it. The server returns the value with the version timestamp and the local datacenter chain index. If it is a new version, both entities are changed. If not, the chain index entry for the local datacenter keeps the highest value.

Every update is sent to the head of the chain and propagated to the following nodes. In the message, the client sends the object's key, new value and a compression of all accessed objects' metadata since the last update. The head of the chain starts by assigning a new object version by incrementing its replica part of the timestamp. Afterwards, the update is sent to the other nodes until it is *k-stable*. A *k*-stable update is replicated in, at least, *k* nodes of the chain. Only then, the head is able to return to the client the object version and the index of the latest receiving node.

When an update reaches the tail of the chain, it is *DC-Write-Stable* for that replica. Updates are delayed until every object that it depends on is in this condition, preventing clients from reading inconsistent versions. DC-Write-Stable objects do not need to be kept in the client's accessed objects table. If all chain replicas are in this condition, the update is *globally* stable.

Remote updates are scheduled immediately after a chain head receives a client update. For this replication, there is only needed the update timestamp. The datacenter has a specialized entity responsible for exchanging these updates and it is called a *remote-proxy*. It also has a timestamp in order to track the previously applied updates. When a remote update arrives, its timestamp is compared with the one of the receiver's proxy. If the latter's timestamp has at least the same entry values for other datacenters and one less for the sending datacenter, the corresponding dependencies are stable on the current datacenter. So, this update can be applied. Otherwise, it must wait until these conditions are verified.

It is important to note that a node may have different positions for different objects: for one it may be the head, for other the tail and for another be in the middle. Different types of nodes need different fault tolerance strategies. The head is substituted by the second node, the tail is not replaced and the middle nodes are hopped over. The previously chosen *k* for a *k*-stable update works as the minimum number of nodes for ChainReaction[8] to work properly.

SwiftCloud[9] The timestamp has one part per datacenter and one more for the client. The client part is necessary because each client keeps a local object cache and may send the updates for different datacenters before they are acknowledged. This is a similar strategy to the Lazy Replication's[7] CID. Each part stores the corresponding entity logical clock.

When a client wants to access SwiftCloud[9], it must maintain a session with the firstly contacted node. Clients request objects they want to cache and they are included in their *interest set*. All operations are done in the cached objects and may update the client's causal history timestamp.

A read gets the locally available value and merges the update timestamp with the client's history. Updates are tagged with dependencies (a copy of the client's timestamp) and with an initial timestamp of the client's clock. In the background, the new updates are sent to the datacenter node by their chronological order. After receiving it, the node reassigns a timestamp with its logical clock and makes the update visible to other clients.

For delivering remote updates, the node has the client's current version vector and only sends it the changes that the client has not seen. If a client wishes to change the number of elements in its set, it must notify the associated node. When there are new objects to replicate, the node delivers the most recent consistent versions like remote updates. In both situations, the client's causal history timestamp is updated.

Regarding inter-datacenter replication, the receiving datacenter must wait until the local version is consistent with the update dependencies. When these updates become k-stable, they can be sent to the clients that want to replicate them. To detect which updates are k-stable, each datacenter stores a version vector that is causally updated when it receives $k - 1$ acknowledge messages from other datacenters.

To reduce the size of the timestamp, the dependencies may not take into account all the existing datacenters. They can only mention the ones that were read/written before the update.

Legion[10] The main motivation of this system is to change the interactions in collaborative applications. In the past, there were datacenters with replicas of the objects and the clients had to use them as a middle man. This had an impact in high delays to receive the updates and blocking the dissemination of changes when the datacenter was unavailable. As a result, Legion[10] allows the communication of clients directly through themselves.

A client device does not have the available storage to be a full replica of a datacenter. To circumvent this problem, each client replicates one or more *containers*. A container has all the objects that belong to some collaborative work. Although clients may only use a subset of the container, they still have to replicate all of it. This can be seen as a *non-genuine* partial replication method, like in PRACTI[6]. Different versions of a container are detected with a timestamp with one part per replicator node. Unlike the previously presented systems, Legion[10] has entries with physical clocks.

When a client wants to join a container, it must connect to a number of *near* and *distant* nodes. They are chosen by their *timing* distance to well known servers, which is calculated by the round trip times of ping messages. The difference between the two kinds of nodes allows a lower visibility latency for updates as well as a higher tolerance to network partitions.

Reads and updates are applied to the locally available object versions. An update creates a new container version, which timestamp is changed in the local node entry to the current clock value.

The updates of the objects are kept in a *version chain*, allowing the transmission of only the differences among the old and new container states. This is done in the background and the differences are encoded with CRDTs. Between each client pair, there is a FIFO channel, which preserves the causal ordering on the replication of updates.

In order to cope with legacy applications, there is a special client that is in charge of the communication with the datacenter. Older applications may not have the changes for the direct client communication, leading to a client sending local updates only to the datacenter. The special node maintains a global consistency by sending the changes resulting in the inter-client communication to the datacenter and disseminating newer datacenter versions among its peers.

Cure[11] In spite of not being one of the goals of this thesis, this was the first system to successfully allow causally consistent transactions with both object reads and updates. Other solutions allowed transactions with either one or the other.

The causal dependencies are expressed by a timestamp with one part per datacenter and physical clocks. However, the timestamp creation process is more complex than the ones of previous systems. It is explained when we present the transactions that include object updates.

Before a client can access objects, it must contact a *coordinator* to obtain a transaction identifier. A coordinator is any server of the local datacenter that is responsible for committing the client transaction. The identifier works as a boundary for the updates that a client can access. During a transaction, the client can see objects whose version has a lower timestamp than the one from the identifier.

Clients can read one or more objects at the same time. However, they keep the transaction identifier as metadata without being changed. Regarding updates, a client may also make more than one at the same time, but they are not final. They will only be stored after a successful commit.

When a client commits a transaction that has object updates, all of them will have the same timestamp so that the atomicity property is guaranteed. The timestamp calculation involves the partitions whose objects were updated. Each partition proposes the current value of its clock to the coordinator. After gathering all the proposals, the coordinator chooses the highest value and notifies the participating partitions of the result. It will replace the content of the local datacenter entry in the dependencies timestamp to create the update timestamp.

The replication of updates is a pairwise process between the same partitions on different datacenters. After a specific time interval, pending committed updates are transmitted to other datacenters. Replicas have two timestamps for controlling the visibility of this kind of updates. The first tracks all the updates (local and remote clients) and the second is used for showing the remote updates. The receiving server advances the sender's timestamp entry to the update timestamp on the all operations structure. To know which remote updates can be made visible, partitions in the same datacenter periodically exchange their all operations timestamps to compute the second timestamp. Each part will be the minimum value of the received timestamps for that specific entry. Remote updates whose dependencies have lower or equal timestamp than the global timestamp are the ones that are visible.

Okapi[12] Clients track their causal history using timestamps with one part per datacenter. Servers have three timestamps of the same size: one for all the received updates, other for globally stable updates (similar to Cure[11]) and another for *universally* stable updates. An universal stable update is an update that is globally stable in all datacenters and its importance is explained when we present the replication of updates. Okapi[12] uses hybrid clocks as entry values.

In total, clients have two timestamps and a *dependency time*. One is a consistent version of a local server's universal stable timestamp and the other is a merge of the previous one with the dependency time in the local datacenter's entry. A dependency time is returned for objects that are not yet universally stable but were locally created. It is the clock value of the server in which the update was created.

When a client wants to read an object, it sends the key and the local copy of the universal stable timestamp. If the server is behind, it assumes the received timestamp for its local copy. At maximum, the returned version has a timestamp as large as the client's timestamp. With the object value, the server also returns its universal stable timestamp and, as said in the previous paragraph, possibly the dependency time which are stored at the client side.

Writes are appended with the new object value and the merged client timestamp. The update creation timestamp is assigned at the server, by using the latest clock value in its corresponding entry of the merged client timestamp. The server timestamp for all updates is also updated in its entry. As a confirmation, the client receives the new dependency time associated with the update.

The replication of updates is similar to the one in Cure[11]. However, the primary replica only sends its part of the timestamp. The receiver knows which entry to update, because there is a pairwise communication. The global timestamp is calculated as in Cure[11]. The universal timestamp uses the global ones, which are now exchanged between same partitions of different datacenters. As previously, each entry is the minimum value of all the exchanged timestamps.

The universal timestamp guarantees that clients observe the same updates when changing datacenters. This did not happen in Cure[11], being one of the authors' motivations.

Eunomia[13] At the time of the proposal of this system, the two most common types of mechanisms to enforce causal consistency were: *sequencers* and *global stabilization procedures*. The first had the problem of being in the client's critical path, introducing a delay for the completion of the operations. The second uses more complex dependency checking, which needs more communication between partitions of different datacenters.

In order to combine the best of both approaches, Eunomia[13] uses a *site stabilization procedure*. There is a new component in the datacenter, which is responsible for exchanging remote updates with other datacenters. When the current datacenter has the primary replica, the update is transmitted to this new component in the background. The process is explained when we address the propagation of remote updates.

Each client keeps a timestamp with one part per datacenter, storing the highest hybrid clock for each datacenter. The Eunomia[13] service has a timestamp for controlling the latest applied remote updates and a vector with one entry per partition.

When a client wants to read an object, it sends the key to the partition server. The server returns the object's value and creation timestamp. The client keeps the partwise maximum between the received timestamp and its own local timestamp.

For an update, the client sends the object key, new value and its local timestamp. This timestamp is used as the update timestamp basis. It is only changed in the local datacenter entry by giving it a higher value. The server forwards both data and metadata to the Eunomia[13] service and the update timestamp to the client. The client can replace its local timestamp, because the new one is guaranteed to be larger.

The replication of updates only happens when Eunomia[13] is sure that it will not receive any update with a lower timestamp from a local partition. It keeps track of the datacenter clocks of previously received updates on the partition vector. Every update that has a lower value on the local datacenter entry than the minimum value of all vector entries is safe to be replicated to other datacenters. For a datacenter that receives a replicated object, it puts it in the sender's pending updates queue. After a certain period of time, there is a dependency check to make those updates visible. If, for every entry that does not belong to neither the sender nor the receiver, the local timestamp has a higher or equal value than the update timestamp, then the update is applied. The local timestamp is accordingly changed to allow future updates. It is important to note that queues are sorted with the earlier updates on the head and the newer on the tail.

GentleRain[14] Unlike the previous systems, GentleRain[14] tags updates with a timestamp with one part for the entire system. This means that it always has the same size, independently of the number of objects or participants. Regarding the clock type, it uses a physical one.

Clients store two timestamps: a *global stable time* and a *dependency time*. The first is calculated by the servers and acts as a marker for making remote updates visible. The second is used in some interactions with the local datacenter. Servers maintain a vector with one entry per datacenter that allows the computation of the global stable time. There is also a *local stable time* that works as an intermediate result. Their differences are explained when we talk about the replication of updates.

When a client wants to read an object, the request includes the key and the client's global stable time. If the server's value is smaller than the one provided by the client, it replaces its local copy of the global stable time. The response has the newest object version with a timestamp lower than the global stable time, its value and the server's global stable time. The object's update timestamp is stored in the client dependency time if it is larger.

For updates, the client attaches its dependency time to the object's key and new value. The server responsible for the object must guarantee that the update has a larger timestamp than the client dependency time. This may result in blocking the operation until the internal server clock respects the condition. Afterwards, the clock replaces the current datacenter entry in the server's vector and is assigned as the update timestamp. The server returns the update timestamp to the client, which will replace its dependency time. At the same time, the server enables the background replication of the update.

Each remote datacenter entry on the local version clock is updated when an update coming from that specific datacenter arrives. It will be changed to the update's timestamp value. However, there is a possibility of not making the update visible, because of the lower value of the global timestamp. The calculation of the global time has two intra-datacenter stages. The first is to disseminate on every time interval the minimum entry value of each partition's local version vector (local stable time). After gathering all values, each partition can get the global stable time, which is also the minimum of all the exchanged values.

POCC[15] This system has an optimistic approach on causal consistency. Every time servers receive an update (client or remote), they directly include it in the version chain. To ensure causal consistency, clients are in charge of managing the returned value to the application in their upper layer. The authors argue that there is no need for servers to keep heavy dependency check mechanisms, because *usually* clients ask for updates that are already replicated.

Clients have two timestamps with one part per datacenter. One tracks the dependencies of all operations and the other is updated on reads. Servers have a timestamp that controls the stored objects. Each entry stores a physical clock value.

For joining the system, a client must create a session to a particular server in the local datacenter. Either the server has the objects the client wants to access or it forwards the operations to a responsible one.

When a client tries to read an object, it sends the object's key and its read timestamp. If the server's timestamp is larger or equal to the client's read timestamp, it can answer the request with a consistent object version. Otherwise, it must wait until it receives a new version from other replica. The server replies with the object's value and all the associated metadata. The most important ones are the object dependencies (it will update both the read and all operations timestamps of the client) and creation clock (it only updates the all operations timestamp).

It is possible to identify a situation in which a read never terminates. For instance, if a client depends on an update from a remote datacenter and there is a network partition that separates both datacenters, then the client hangs for an indeterminate amount of time. To solve this problem, the client has a time interval to complete the operation. Otherwise, it starts a new session to the current datacenter, but in a pessimistic approach (similar to the one in Cure[11]).

To update an object, the client sends the object's key, new value and the all operations timestamp. The server waits until its current clock is higher than the maximum value of the received timestamp, ensuring a higher clock than any of its dependencies. The clock value will replace the local datacenter entry in the server's timestamp and tag the object creation clock. The object dependencies are expressed by the all operations client timestamp. The server responds to the client with the update creation clock, which replaces the local datacenter entry in the all operations timestamp.

Updates are asynchronously replicated by their creation clock order. The receiver server adds the object version to the version chain and updates the local timestamp on the sender's entry to the update creation clock.

Occult[16] This system follows an optimistic approach. The timestamps have one part per partition, instead of per datacenter. It is the only system to adopt this structure, which leads to a master-slave replication technique. Each entry stores the logical clock of the corresponding partition.

Clients have one timestamp for tracking their causal history. Servers only keep their logical clock. However, updates are stored with their dependency timestamp.

In order to distribute the load of read operations, a client can contact any partition replica by sending the object's key. The contacted server immediately answers with the latest object value, its dependency timestamp and the current logical clock. Afterwards, the client must check if the version is consistent with its causal history by comparing the logical clock with the corresponding partition entry value. The inconsistency may occur when the update was made on the master and it has not yet reached the current replica. So, a client can try more times at the same replica and, if it is still unsuccessful, try the master instead. The last attempt is guaranteed to be successful, because the master has the most updated version of the partition data. Finally, the client updates its own timestamp to reflect the new dependencies.

Updates are always sent to the master server of the partition. The request includes the object's key, new value and the client timestamp. The master uses the client timestamp to derive the update timestamp, by assigning the latest logical clock to the corresponding partition entry. This clock value is also returned to the client, which will be used to update its timestamp.

Updates start being replicated before a master server returns from a client object update. They are sent to the slaves with the associated timestamp and the master clock. The second will be set as the slave's current clock value. All updates are replicated by their creation order.

Given that the number of partitions tend to be very large, there are three optimizations to the timestamp size: *structural compression*, *temporal compression* and *isolating datacenters*. As a tradeoff, the compression of metadata leads to a higher amount of false dependencies, affecting the visibility latency of updates in a negative way.

The structural compression works as a hash function. We need to define the size of the timestamp and it should be much smaller than the number of partitions. Afterwards, we assign each partition's position by applying its number modulo the size of the timestamp. In spite of different partitions being mapped to the same entry, it only stores the maximum clock value. This avoids a client from reading an inconsistent version of an object, but may result in a client blockage. If a partition has less updates than other that is also mapped in the same entry, the reads will always fail, because the client depends on a *newer* version. So, the clocks of all partitions must be synchronized.

The temporal compression also uses a predefined number of entries, but keeps detailed information for the most recently used partitions. If a timestamp has N entries, then the first $N-1$ entries will store the partitions with higher clocks and the remaining entry will compress all the other partitions by storing their maximum value. This is a dynamic structure that can be updated on every read and write. On reads, the client must have two sorted vectors by decreasing clock value and choose the maximum entry from both, creating a new vector with a possibly third different order. On writes, it is easier, because there is only the returned partition clock to compare with the vector entries. Starting from the last position, when there is an entry with a larger value than the received one, then it should be on the previous position. The compression of the values of the remainder partitions must also be considered.

The final optimization tries to reduce the visibility latency of updates when the slave and the master are on different datacenters. Instead of only having an entry with a clock, it is replaced by a vector with an entry per as many different datacenters with the same partition. In the worst case, it can be seen as an approximation of Orbe[5] by having a partition replica in each datacenter. On reads, all the datacenter entries must be updated to the latest values. On writes, the partitions that have entries on the same datacenter as the master are updated with the received clock.

Saturn[17] This is the second system to use a timestamp with a single part for the entire system. In this case, the metadata is a *label*, which works as a tag for updates. It is created in a *gear*, which exists in every server. The *label sink* is responsible sending updates to other datacenters and they are received by a *remote proxy*. The clocks used in the labels are physical ones.

Clients track their causal history by keeping an updated label that is used in the operations. The Saturn's[17] new components are responsible for handling all metadata, but the labels are stored alongside the object data.

If a client is reconnecting to a datacenter, it must firstly *attach* to it. It sends the previously seen label and it will ensure that the client's causal history is consistent with the datacenter's state. Sometimes, the client must wait for remote updates to arrive at the new datacenter.

For a read, the client only needs to send the object's key. The gear intercepts the request for recovering the label associated with the stored value. The client receives both and replaces its label if the returned one is newer.

Client updates have the object's key, new value and the client's label. The server's gear creates a new label and stores it with the new value on the server's persistent storage. Then, the update data and metadata is sent to the label sink for being replicated. Finally, the new label is returned to the client, which replaces the old value.

The object replication separates the propagation of the object's data and label. The transmission of labels follows a well defined tree topology through *serializers*, but data can be delivery in any order. If a serializer does not have lower nodes that replicate the object associated with the arrived label, then it does not propagated it to them. This small detail gives Saturn[17] the genuine partial replication capability.

Remote updates are only visible when the label arrives. The false dependencies are closely related with the arrival of the different labels, meaning that concurrent operations are executed in that order. If a label arrives before its corresponding data and the time to transfer all of it is high, other remote updates will be blocked. So, the label dissemination tree is built in order to minimize the difference in the times of arrival of the metadata and the data. Sometimes, it is necessary to introduce some delays in the transmission of labels, because they tend to be smaller and to arrive earlier.

When some serializer fails, datacenters will eventually become aware of it, changing to a different tree topology. The transition is made without stopping the system, but updates received from the new configuration are only applied after being sure that all other datacenters share the new tree. This is accomplished by using a special type of label, that signals the change of the tree for each datacenter.

To ease the *migration* of clients to other datacenters, there is another special type of label that is replicated to the new client's datacenter. It will ensure that all the causal history of the client is available at the destination.

Finally, if we look at the edge computing desires, this system fulfills most of them. It supports *genuine* partial replication, the size of the metadata does not depend on the number of participants nor the number of objects and it improves on the amount of the false dependencies.

4.4 Comparison

In order to summarize the most important system characteristics, Table 1 has a classification for the analyzed solutions according to the chosen technique, the metadata size, the amount of false dependencies and the support for partial replication with its genuineness. By looking at Table 1, we can compare the systems by the different columns, but there is too much detail. A more natural distribution of the solutions is on Figure 5, which is a plot comparing the different metadata sizes with the amount of false dependencies. The Δ at the end of some systems indicates the support for partial replication.

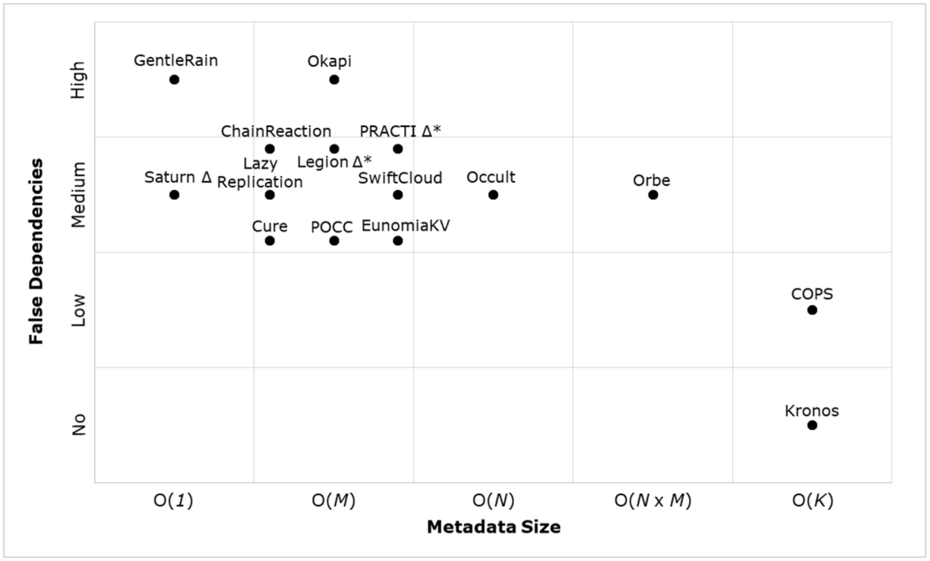

Fig. 5. Graphic distribution of the systems according to their false dependencies and metadata size. Δ stands for genuine partial replication and Δ^* is for non-genuine. Low false dependencies stand for only one level on the table. Medium is either two or three. High is all of them.

On Figure 5, we can see that a higher metadata size usually means a lower amount of false dependencies. Given that Okapi[12] keeps an inter-datacenter synchronization, it has a more complex stabilization protocol and takes longer for remote updates to become visible. Kronos[3] and COPS[4] have the same amount of metadata, but the second compresses the object's clock to only a scalar while the first considers that operations are made at only one datacenter. Saturn[17] is able to improve on GentleRain's[14] amount of false dependencies, also keeping a constant metadata size.

Regarding the different techniques on Table 1, there are five categories: explicit check, sequencer, stabilization, optimistic and tree. Explicit check exists when a server knows the objects that are missing to satisfy the dependencies (Kronos[3] and COPS[4]) or asks other specific entities about their current state (Orbe[5]). Solutions that use a sequencer technique either have a log exchange replication (PRACTI[6], Lazy Replication[7], SwiftCloud[9] and Legion[10]) or a replication strategy with a well defined sequence (ChainReaction[8]). Stabilization methods can be global, universal or site. Global stabilization uses a protocol involving the local datacenter partitions (Cure[11] and GentleRain[14]). Universal stabilization does the same as global, but it is followed by an inter-

Systems	**Technique**	**Metadata**	**False Dependencies**	**Partial Rep.**
Kronos	Explicit check	Dep. graph O(K)	$\times$	$\times$
COPS	Explicit check	Keys' vector O(K)	E	$\times$
Orbe	Explicit check	Servers' vector O($N \times M$)	$E + S$	$\times$
PRACTI	Sequencer	Nodes' vector O(M)	$E + S$	Nodes$^{\times}$
Lazy Replication	Sequencer	Replicas' vector O(M)	$E + S$	$\times$
ChainReaction	Sequencer	DCs' vector O(M)	$E + S + ID$	$\times$
SwiftCloud	Sequencer	DCs' vector O(M)	$E + S + ID$	Client*
Legion	Sequencer	Peers' vector O(R)	$E + S$	Client$^{\times}$
Cure	Stabilization	DCs' vector O(M)	$E + S + ID$	$\times$
Okapi	Stabilization	DCs' vector O(M)	$E + S + ID + ED$	$\times$
Eunomia	Stabilization	DCs' vector O(M)	$E + S + ID$	$\times$
GentleRain	Stabilization	Scalar O(1)	$E + S + ID + ED$	$\times$
POCC	Optimistic	DCs' vector O(M)	$E + S + ID$	$\times$
Occult (no optimizations)	Optimistic	Partitions' vector O(N)	$E + S$	$\times$
Saturn	Tree	Scalar O(1)	$E + S + ID$	Server$^{\checkmark}$

Table 1. Comparison of the different solutions. K is the number of all objects in the system. N is the number of partitions. M is the number of replicas and R is the same, but per container. E, S, ID and ED represent inter-datacenter element, intra-server, intra-datacenter and inter-datacenter dependencies, respectively. $^{\checkmark}$ and $^{\times}$ stand for genuine and non-genuine partial replication, respectively. Regarding SwiftCloud's partial replication, there is a client cache that only replicates its interest set, but the datacenters have a full copy.

datacenter protocol (Okapi[12]). Site stabilization is done by a single entity of the datacenter, which is only responsible for the replication of updates (Eunomia[13]). Optimistic mechanisms make clients responsible for guaranteeing the causality rules (POCC[15] and Occult[16]). The tree technique was developed for Saturn[17] and is related to the dissemination of metadata.

Although we included SwiftCloud[9] on partial replication (Table 1), it does not allow direct client communication and servers have a full copy of the data.

4.5 Shortcomings of Current Approaches for Edge Computing

Firstly, it is important to understand that all the systems work in the conditions they were implemented. The change to edge computing has much different characteristics, leading to all of them not working *as is*.

Looking at the different techniques to enforce causal consistency, we can identify some strategies that are not feasible. For instance, if we adopted an explicit check, this would mean that the amount of metadata was dependent on the number of objects (Kronos[3] and COPS[4]) or that nodes needed to exchange messages for inquiring about objects they would not replicate (Orbe[5]). Both options do not scale to the size of the edge. Sequencer techniques are based on log exchange with repeated operations (PRACTI[6], Lazy Replication[7], SwiftCloud[9] and Legion[10]) or follow very strict replication strategies (ChainReaction[8]). Stabilization creates an inter-node dependency for making remote updates visible (Cure[11], Okapi[12], Eunomia[13] and GentleRain[14]), which would be problematic with the higher degree of disconnections on the edge.

The only two reasonable techniques are: optimistic and tree-based. The former is controlled by the client and may resort to waiting (POCC[15]), while the second delivers the updates after getting the corresponding label. So, the node can decide what to do without consulting any third party.

For the timestamp size analysis, it must be a scalar. It is the only approach that complies with the unknowingly large edge size. If we consider it together with the amount of false dependencies, Saturn[17] is the best option.

The majority of the solutions assumes full replication in the datacenter. As seen in Section 3, edge computing nodes do not have the same memory capacity. Regarding the partial replication techniques, the genuine approach is the most desirable, because nodes would only receive messages for objects of their interest. Furthermore, it would reduce the number of messages on the network. Once again, Saturn[17] is the system that can offer it.

In spite of being the system that fills most of the requirements, Saturn's[17] optimization algorithm for building the dissemination tree is not the best strategy for the new paradigm. It does not cope with the much more dynamic environment, needing a much higher number of reconfigurations.

If we consider all of these issues, we can see that there is space for improvement on the edge computing paradigm. The following section explains some optimizations to tackle it.

5 Architecture

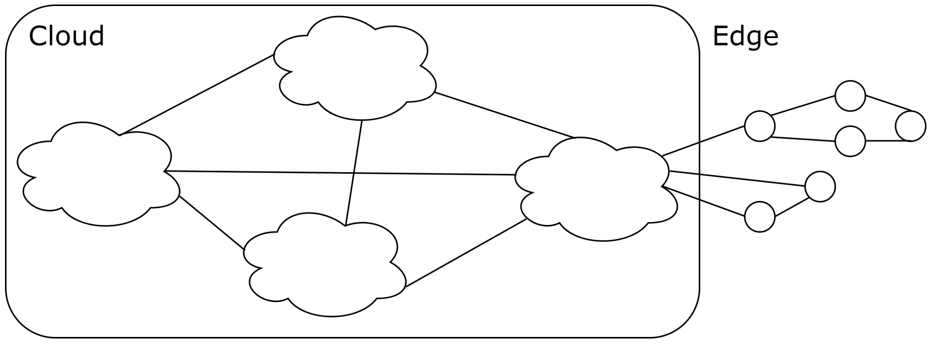

Fig. 6. Connections between all the system components.

As concluded at the end of Section 4, Saturn[17] is the best starting point for the new system. In the remainder of this section, we will describe the operations, the optimizations and the additional fault tolerance strategies for running Saturn[17] on the edge. We consider that there are geo-replicated datacenters and clients connect to them and/or to other clients (Figure 6).

The object replication between datacenters will not be addressed, because the surveyed solutions of Section 4.3 already solve this issue. To reduce the amount of adaptations, Saturn[17] is used for this situation.

5.1 Operations

Like all the presented solutions, we consider that clients are able to read, update and replicate objects. We will describe them, taking into account the edge computing paradigm heterogeneousness property. So, clients replicate a subset of all existing objects, but all the partial views must be globally consistent. Another situation that must be addressed is the change of the number of replicated objects for each node. It may include new objects or remove the ones that are not relevant.

The timestamps follow the same structure as Saturn's[17] labels with physical clocks. Each node tracks its causal history with a label and is able to create new ones with a local gear. Label sinks and remote proxies are still present, but it is in a per node basis.

Reading Objects Like in Saturn[17], a node only needs the object key to get its value. However, it always reads the local copy of the object. The gear fetches the associated version label, which is used for updating the local node's causal history timestamp if it has a higher value.

Updating Objects An update needs the object's key, new value and the causal history. The gear creates the new label, tags the update and replaces the local version of the object. Afterwards, both data and label are sent to the label sink for being replicated. Finally, the client's causal history label is replaced by the new one.

Replicating Objects Once again, following Saturn's[17] architecture, data and labels are separately replicated. This is one of the components that enables the genuine partial replication. The other one is the propagation of labels through the tree. Instead of relying in external serializers, each node will take part in the serialization network. Nodes in a higher depth may not replicate the object and they will not receive the updates. Unlike Saturn[17], remote updates may not be visible when the label and corresponding data is available. The reason behind this decision is related to fault tolerance and it is addressed in Section 5.3. An update that has a lower label than the stored one is discarded.

Changing the Number of Objects During the normal execution of the system, nodes may decide to increase or reduce the amount of replicated objects. Maybe they started working on a different subject and need more data or the local storage is full and need to free up some space. We assume that these situations are not frequent and can be dealt like new nodes joining the system. It will result in a repositioning in the tree, which allows them to receive updates for the new set of objects. This processed is presented in the following section.

5.2 Optimizations

In the new paradigm, the main issue with Saturn[17] is the heavy tree optimization problem when the number of nodes is high. It takes too long to reach a possible solution. With the high churn, the amount of times the algorithm needs to run also increases. To deal with it, we consider additional measures on the building of the initial tree, the tree reconfiguration and one alternative topology.

Initial Tree To reduce the size of the problem, we consider the concept of *regions*. A region can be seen as a group of nodes that is connected to other groups or a datacenter. The region granularity can be as small as wanted: continent, country, city or even building. For each one, the optimization problem builds a tree and, with the subtrees, it is rerun for getting a global tree that connects all regions. The smaller trees can be built concurrently, which reduces the total amount of time for defining the network.

Tree Reconfiguration The existence of regions allows a more dynamic and modular tree, offering better scalability to the edge paradigm. When the number of participants inside a region changes, only the local tree is recalculated. This originates a parallel reconfiguration process and regions without changes are not affected by outside nodes.

Alternative Topology An alternative is to use a spanning tree instead of the Saturn's[17] tree. They are known for their scalability and fast creation. Each time period, if a certain amount of participants changed, there would exist a recalculation of the spanning tree with all the current nodes.

5.3 Fault Tolerance

The amount of failures in the edge computing paradigm is much higher than in the traditional datacenter. However, clients must be able to use the service without noticing the problems. We identified two strategies for making the issues transparent. We will adopt a redundant network and stable updates.

Redundant Network To cope with the initial impact of a client disconnection, there needs to be a redundant network. Instead of relying on a single tree, a node also belongs to a second one. It is calculated in the background and is used when the first has some problems. The change between them follows Saturn's[17] online reconfiguration. Each node produces a *change epoch* label and propagates it through the first tree. Afterwards, all new labels use the backup structure. However, the new updates are only applied when the change epoch label is received.

Update Stability We must ensure that nodes do not see updates that will eventually be lost. For example, before disconnecting, a node replicates a locally written object to a single node. The second participant cannot make this change visible, because its state would be inconsistent with all the others. To avoid this situation, we consider the concept of k-stable updates introduced by ChainReaction[8] and SwiftCloud[9].

Although we need to evaluate the different alternatives as explained in Section 6, we believe that the *divide-and-conquer* technique applied to the already existing Saturn[17] tree configuration will produce the best results. Between it and the spanning tree, the former is already prepared for genuine partial replication.

6 Evaluation

The evaluation process will focus on three topics: i) performance on the delivery of updates; ii) adaptability to network changes; iii) fault tolerance. Given that there are two different network topologies, all the processes are repeated for both.

6.1 Performance

As usual in causal systems, the two most important performance metrics are: visibility latency and throughput of remote updates. The first can be measured by the difference between the update creation clock and the clock of when the same update becomes visible in a remote node. The throughput is the amount of remote updates that become visible over a period of time.

We expect worse results than the original Saturn[17] version. The higher number of nodes and branches on the tree will directly impact the time it takes for two distant nodes to communicate.

6.2 Adaptability

On the edge computing paradigm, the number of participating nodes changes in a much higher rate than in the typical datacenter scenario. So, the network must be able to provide fast reconfiguration to the addition and removal of participants.

For the addition of nodes, we will measure the time it takes for a node to be able to start replicating local updates. For the opposite case, we will measure the time between the removal of the node and the creation of the new stable topology. To find the impact of the region granularity, the previous time measurements will be obtained for regions with different number of nodes.

6.3 Fault Tolerance

In order to check the system's behavior in the presence of failures, we must look at the redundant network and the updates generated by failed nodes. On both, there will be tests in which either only a single node or a subset of them is removed.

On the redundant network, we must check if updates are still being replicated after failures. We expect a temporary loss of performance without stopping the propagation. When the structure returns to a stable state, the performance should return to the previous levels.

Updates generated by a disconnected node are sent to other single node. The second node must not make the data visible, avoiding inconsistencies with the remainder of participants. The check consists on requesting the problematic version of the object at the node.

7 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4 - May 23: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15: Deliver the MSc dissertation.

8 Conclusions

Edge computing is a new paradigm that mixes cloud computing with less powerful devices that can also make useful computations. It avoids sending the huge amount of data to the cloud provider, because there is not enough network bandwidth to allow it. The advancements on processing power of smaller devices supported this decision.

On concurrent work, causal consistency is needed to avoid the existence of inconsistent states between different nodes. It is the preferred consistency type, because systems can still be always available and tolerant to failures.

The intrinsic characteristics of edge computing do not allow an easy portability of the existing causal solutions to this domain. Saturn[17] is the closest one, needing some changes in its tree topology. We defined some different architectural options that must be tested in respect to three aspects: performance, adaptability and fault tolerance.

According to the presented schedule, a more detailed design, implementation and evaluation are left for future work. Although we think that the presented strategies provide a good starting point, we may find that they do not work as expected, leading us to even other alternatives.

Acknowledgments We are grateful to Manuel Bravo for the fruitful discussions and comments during the preparation of this report. We would also like to thank João Marçal for his tips and corrections of initial drafts of this report.

References

1. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing **9**(1) (1995) 37–49
2. Brewer, E.A.: Towards Robust Distributed Systems (Abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, Portland, Oregon, USA, ACM (2000) 7
3. Escriva, R., Dubey, A., Wong, B., Sirer, E.G.: Kronos: The Design and Implementation of an Event Ordering Service. In: Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14, Amsterdam, The Netherlands, ACM (2014) 1–14
4. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais, Portugal, ACM (2011) 401–416
5. Du, J., Elnikety, S., Roy, A., Zwaenepoel, W.: Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In: Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13, Santa Clara, CA, USA, ACM (2013) 1–14
6. Belaramani, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., Zheng, J.: PRACTI Replication. In: Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3, San Jose, CA, USA, USENIX Association (2006) 5

7. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing High Availability Using Lazy Replication. ACM Transactions on Computer Systems **10**(4) (1992) 360–391
8. Almeida, S., Leitão, J., Rodrigues, L.: ChainReaction: a Causal+ Consistent Datastore based on Chain Replication. In: Proceedings of the 8th ACM European Conference on Computer Systems, Prague, Czech Republic, ACM (2013) 85–98
9. Zawirski, M., Preguiça, N., Duarte, S., Bieniusa, A., Balegas, V., Shapiro, M.: Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In: Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, ACM (2015) 75–87
10. Linde, A.v.d., Fouto, P., Leitão, J., Preguiça, N., Castiñeira, S., Bieniusa, A.: Legion: Enriching Internet Services with Peer-to-Peer Interactions. In: Proceedings of the 26th International Conference on World Wide Web, Perth, Australia, International World Wide Web Conferences Steering Committee (2017) 283–292
11. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Nara, Japan, IEEE (2016) 405–414
12. Didona, D., Spirovska, K., Zwaenepoel, W.: Okapi: Causally Consistent Geo-Replication Made Faster, Cheaper and More Available. CoRR **abs/1702.04263**(February) (2017) 1–12
13. Gunawardhana, C., Bravo, M., Rodrigues, L.: Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA, USA, USENIX Association (2017) 83–95
14. Du, J., Iorgulescu, C.C., Roy, A., Zwaenepoel, W.: GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In: SOCC '14 Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, ACM (2014) 1–13
15. Spirovska, K., Didona, D., Zwaenepoel, W.: Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, IEEE (2017) 2626–2629
16. Mehdi, S.A., Littley, C., Crooks, N., Alvisi, L., Bronson, N., Lloyd, W.: I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, USENIX Association (2017) 453–468
17. Bravo, M., Rodrigues, L., Roy, P.V.: Saturn: a Distributed Metadata Service for Causal Consistency. In: Proceedings of the Twelfth European Conference on Computer Systems, Belgrade, Serbia, ACM (2017) 111–126
18. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge Computing: Vision and Challenges. IEEE Internet of Things Journal **3**(5) (2016) 637–646
19. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, USA, USENIX Association (2004) 137–149
20. Stutzbach, D., Rejaie, R.: Understanding Churn in Peer-to-peer Networks. In: Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement, Rio de Janeiro, Brazil, ACM (2006) 189–202
21. Lamport, L.: Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM **21**(7) (1978) 558–565