Weak Transaction Support on the Edge

Taras Lykhenko
taras.lykhenko@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. This report addresses the problem of offering different consistency guarantees to clients of services provided by the edge nodes. We survey the most important weak consistency models and the main techniques that have been proposed to support weak forms of transactions in a distributed environment. We evaluate these techniques from different perspectives, namely performance, visibility latency, and fault tolerance. Using these criteria, we discuss the potential limitations and advantages of each of these techniques in the edge computing setting. Based on this analysis, we propose a new architecture to support weak transactions in the edge.

Table of Contents

1 Introduction

Cloud computing is now a widely adopted model that allows offering services in a cost-effective manner to a large number of clients. Many of the most widely used applications, from email to social networking, are based on this model. In this model, storage and processing services are provided by servers located in large centralized datacenters. The economies of scale provided by these shared infrastructures offer cost reductions and make it easier to allocate dynamically resources to match the client demand.

Despite these advantages, the current model of cloud computing also has some limitations. As the number and power of the devices that are connected to the cloud grow, new requirements emerge. Many client devices, such as modern smartphones, have now the capacity to support sophisticated applications such as augmented reality; these applications have tight latency requirements. Also, there are more and more IoT devices with the capacity of collecting data, from a myriad of sensors. This data needs to be processed but shipping all the collected information in a raw form to a central datacenter may be unfeasible. These new requirements have motivated the need to support edge computing[1], a model where the service provided by the cloud datacenter is complemented by a set of smaller servers located closer to the edge of the network (sometimes also called fog servers[2]). Edge servers have the ability to pre-process and summarize data before the results are shipped to the cloud and to offer computational services with low latency to clients in the physical vicinity.

In this report, we focus on the task of providing storage services in the edge. Such storage service should support partial replication. Data needs to be replicated for fault-tolerance and also to keep copies close to the client. However, given that the edge nodes have limited capacity, it will be in general unfeasible to keep replicas of all data items at all edge devices.

Of special concern to our work is to understand the tradeoffs involved in providing different consistency levels when clients access to data in the context of edge computing. In fact, it is well known today that it is impossible to offer simultaneously strong consistency, availability, and partition tolerance, a fact captured by the CAP theorem[3]. It is therefore relevant to search for weak consistency models that can preserve availability and still be useful for edge applications.

On one extreme one can provide a storage service where clients are allowed to interact with a replica without requiring any coordination with other replicas. In this case, reads and writes may be performed by contacting only the nearest available replica and be served immediately. Updates are subsequently propagated to other replicas in background and replicas may be observed in an inconsistent state. For instance, when reading from a replica, a client may even fail to observe the results of its own previous updates, if those updates have been performed at a different replica. Conflicting updates may also be accepted concurrently by different replicas, requiring the execution of complex conflict resolution algorithms to ensure eventual consistency of replicas.

On the other extreme, one can attempt to provide full transactional support, as typically provided by traditional databases. Transactional databases support the atomic execution of a sequence of operation with different isolation and consistency guarantees, often known as the ACID properties[4]. Unfortunately, many of the stronger consistency criteria provided by databases cannot be provided in a non-blocking way in a distributed and replicated setting and, even if when no faults occur, it can be expensive to implement in an edge scenario, particularly in the face of a large number of replicas and partial replication.

Given the limitations of the two extreme models mentioned above, an extensive amount of research has been performed in the quest for consistency models that are weaker than the serializability[4] or snapshot-isolation[4] criteria often found in centralized databases, but that still support meaningful semantics, that can simplify the development of applications. Many of these research efforts have been started in the context of geo-replicated storage services for the cloud, using models that have become generically known as NoSQL[5] but have roots in earlier works that used weaker consistency models as ways to improve the performance of database systems.

In this report, we survey the most important weak consistency models and the main techniques that have been proposed to support weak forms of transactions in a distributed environment. We will evaluate these techniques from different perspectives, namely performance, visibility latency, and fault tolerance. Using these criteria, we discuss the potential limitations and advantages of each of these techniques in the edge computing setting. Based on the analysis, we propose a new architecture to support weak transactions 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 4 we present all the background related with our work. 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 data consistency in edge computing. More precisely:

> *Goals:* We aim at identifying a small set of consistency models that can be effectively offered to clients that access services provided by edge nodes. Furthermore, we will design, implement, and evaluate a storage service supporting the selected consistency models.

The literature is very rich in systems that support a wide range of consistency models in different replicated settings, including geo-replicated cloud storage systems. However, not all of these systems can easily be deployed on edge computing scenarios, some because they maintain an amount of metadata that does not scale for a large number of server nodes, others because they require an

amount of coordination among edge servers that can impair service latency. We aim at carefully addressing related work taking these constraints into account, to select solutions that can be implemented efficiently in the edge scenario.

The project will produce the following expected results.

> *Expected results:* The work will produce i) an identification of relevant consistency criteria for the edge; ii) a specification of a set of protocols that can support these consistency models, iii) an implementation of these protocols; iv) an extensive experimental evaluation of their performance in practice.

3 Data Consistency

In this section, we discuss the guarantees that can be provided to clients accessing data that can be replicated. We distinguish two classes of consistency guarantees: those that apply individual read and write operations and those that apply a sequence of operations that are treated as a block (and that we generically denote as transactions).

3.1 Linearizability and Session Guarantees

We start by discussing *linearizability*[6], which is probably the most intuitive model of consistency. Then we describe relevant relaxations of linearizability.

In the following discussion we assume that write and read operations may take some arbitrary amount of time. In a distributed message passing system, this time is the time required for the nodes to coordinate. Each operation has a starting time (when it is invoked) and a termination time (when the operation returns). If an operation starts before another operation terminates, the two operations are said to be concurrent.

Linearizabilty [6] Under linearizability, read and write operations appear to execute instantaneously, at an arbitrary time instant between the moment the operation is invoked and the operation completes. Thus, after a write completes, all reads must observe that write (or a subsequent write). If a read is executed concurrently with a write, the read can observe the value before or after the write. However, if the read observe the new value, all subsequent reads must also observe the new value, even if they are also concurrent with the write operation.

A linearizable memory register, that supports read and writes operations is denoted to be an *atomic register*[7]. Fault-tolerant atomic registers can be implemented in message passing systems, by letting different nodes to maintain a copy of the register value and by using the appropriate coordination mechanisms to execute read and write operations. The implementation is based on quorums. A write operation only returns when receives an acknowledgment from half plus one of the servers. A read operation is more complex. First, the read must obtain values from a majority of nodes and select the most recent version from the

quorum. Then, before returning, the read operation must write back the value read. This will ensure that subsequent reads will also return that value. Only after the write back phase is concluded (i.e., a majority of acknowledgements is collected for the write phase) the read operation is terminated.

As we have just seen, implementing linearizability in a non-blocking way is expensive and requires clients to always contact a majority of replicas. It is interesting to discuss what guarantees can be provided while offering better availability (namely by allowing progress even if the client can contact only a single replica). In the next paragraphs, we discuss a number of properties that are known as "session guarantees" because they require the client to keep some state regarding its past interactions with the system. A session is assumed to be started when the client first contacts the system and get the initial session state and terminates when the client discards the session state.

In the context of our work, we say that a system is highly available if a user that can contact at least one server is guaranteed to get a response, even if a network partition prevents that server from coordinating with other servers in the system. Thus, in order to ensure high availability we need to resort to consistency levels that require little or none coordination among servers or, if such coordination exists, that it can be performed asynchronously in background. This definition of high availability allows the use of protocols that offer low latency, something that is important in a wide-area setting, were the latency among different nodes is high and network partitioning is likely to occur, making coordination among replicas a bottleneck.

Writes Follow Reads (WFR) under WFR, if a session observes an effect of an operation O_1 and subsequently executes another operation O_2, then another session can only observe effects of O_2 if it can also observe O_1's effects (or later values that supersede O_1's). Thus, the sequence of a write after a read must satisfy to Lamport's "happens-before" relation[8].

Monotonic Reads (MR) under MR, within a session, subsequent reads to a given object "never return any previous values". Reads from each item progress according to a total order.

Monotonic Writes (MW) requires that each session's writes become visible in the order they were submitted.

It is worth noting that, using these properties, the availability of a system can be increased by delaying the visibility of update operation. For instance, assume that a client makes a write operation w_2 that depends on some previous write w_1. Assume that w_2 becomes visible to other clients before w_1 has been applied at all datacenters. Another client that reads w_2 can later be blocked when attempting to read w_1 from another replica (this can happen if clients are allowed to contact different replicas). If that client is served with an older snapshot of the database, with versions that have been applied at all datacenters, that client will miss the

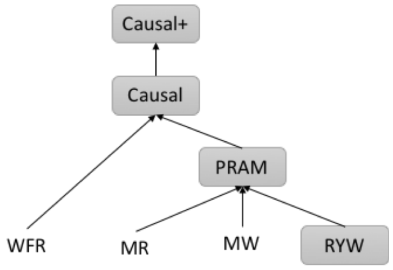

Fig. 1. Relation among the session guarantees; the session guarantees in grey are only possible in sticky availability

new update but it will also avoid being subsequently blocked when reading other objects.

Some systems are more restrictive and force clients to remain connected to a single node. The latter type of systems offers what is called *sticky availability*[9]. Sticky availability usually requires full replication, given that the server to which the client is attached must be able to serve all requests from that client. In systems that use partial replication, a server may not be able to serve requests for data that is not replicated locally without coordinating with other servers. In fact, in that case, it may be simpler to allow clients to migrate in order to access data that is only stored in other servers. With this availability model, we can ensure the following session guarantees that were previously intangible as proven by Bailis *et al.*[10].

Read your writes requires that whenever a client reads a given data item after updating it, the read returns the updated value (or a value that overwrote the previously written value).

PRAM (Pipelined Random Access Memory) lets clients observe a serialization of the operations (both reads and writes) within each session. Different clients can observe different serializations. It can be seen as a combination of monotonic reads, monotonic writes, and read your writes.

Causal consistency [11] is the combination of all of the session guarantees [12]. Note that two concurrent updates can be applied in different nodes in a different order and still respect causal consistency. In the lack of further updates, nodes would remain inconsistent indefinitely.

Causal+ consistency [13] is an extension to causal consistency that ensure that, in face of concurrent updates, one of the updates is applied last at all replicas.

Figure 1 summarizes the relation between the session guarantees mentioned above.

3.2 Transactional Properties

Transactions are sequences of operations that are grouped together and executed as one single indivisible operation. Transactions are widely used in database systems. The concurrent execution of transactions is coordinated by the database to ensure a set of proprieties, usually referred to as the ACID proprieties (ACID stands for atomicty, consistency, integrity, and durability). One of the strongest consistency criteria supported by databases is known as *serializability*. In short, serializability ensures that the concurrent execution of transactions yields the same results as a some serial executions of the same transactions. Ensuring serializability requires the database to execute some form of concurrency control. The most common approach to concurrency control consists in using locks to prevent concurrent transactions to observe inconsistent values.

Even in centralized databases some DBMS opt for a lower isolation level with weaker consistency to improve the performance of concurrent transactions. One of these weaker forms of consistency is *snapshot isolation* (SI). SI guarantees that all reads made in a transaction will see a consistent snapshot of the database, and that the transaction itself will successfully commit only if none of the updates it has made conflicts with updates performed concurrently by other transactions (i.e., transactions that have executed concurrently based on the same snapshot).

Serializability and snapshot isolation levels are often explained in terms of the *anomalies* they prevent (where an anomaly is an observed state that would never occur if transactions where executed instantaneously, one after the other). Two anomalies that are relevant in this context are the *write skew* and the *lost update*. A write skew occurs when a transaction T_1 reads an object written by concurrent transaction T_2 and T_2 also reads an object written by T_1. A lost update occurs when one transaction T_1 reads a given data item, subsequently a second concurrent transaction T_2 updates the same data item, then T_1 also modifies that data item without taking into account the value written by T_2 (and thus, the system behaves as if the update by T_2 has never occurred). Serializability prevents both the write skew and the lost update anomalies. SI only prevents the lost update anomaly.

Unfortunately, in a distributed setting, these two isolation models can only be enforced if at least a majority of nodes are able to coordinate, given that both require transactions to be totally ordered. Therefore, these models cannot be enforced without compromising the availability of the system.

3.3 Isolation Levels that Support High Availability

In order to ensure that transactions are highly available, we need to define isolation levels that can be implemented with little coordination among replicas. We use the study by Bailis *et al.*[10] on highly available transactions to enumerate a number of isolation levels that can be enforced with minimal coordination.

As with serializability and snapshot isolation, several of these weak isolation models are defined in terms of the anomalies they prevent. In this context, we identify the following additional anomalies.

Dirty Writes A Dirty Write anomaly occurs when two concurrent transactions update two or more objects and these updates are applied in different orders at different objects[14]. This is illustrated in Figure 2, where T_1 updates x before T_2 and T_2 updates y before T_1.

T1	T2
$W(x_1)$	
	$W(x_2)$
	$W(y_2)$
$W(y_1)$	

Fig. 2. Dirty Write anomaly

Dirty Reads A Dirty Read anomaly occurs when one transaction reads a value that has been written by another transaction that is still running and has not committed yet. Consider the example depicted in Figure 3. In this example Dirty Read anomaly occurs if T_3 read $x = 1$ or $x = 3$ in the case that T_2 aborted.

T1	T2	T3
$W_x(1)$	$W_x(3)$	$R_x()$
$W_x(2)$		

Fig. 3. Dirty Read anomaly

Read Skew A Read Skew anomaly occurs when in a same transaction same read yields to a different result. Consider the example depicted in Figure 4. In this example Read Skew anomaly occurs if the reads over the item x in the transaction T_2 return two different results.

T1	T2
	Rx()
Wx(1)	
	Rx()

Fig. 4. Read skew anomaly

With the help of the anomalies identified above, we can now list a number of relevant weak isolation models.

Read Uncommitted (RU) Read Uncommitted is an isolation level that only prevents the "Dirty Writes" anomaly. It does not prevent other anomalies as it makes no attempt to prevent transactions from reading uncommitted values

before a transaction has finished. Dirty writes can be avoided by defining a total order among transactions and ensuring that updates are applied to all objects according to that order.

Read Committed (RC) Read Committed is an isolation level that ensures that transactions never access uncommitted or intermediate versions of data items. RC prevents both "Dirty Writes" and "Dirty Reads" anomalies. Dirty Reads can be prevented by not allowing the client to write in the database uncommitted data. Therefore, other transactions will never read uncommitted data. This can be achieved by requiring the client to buffer his writes until commit time or by having the servers to buffer multiple uncommitted values the same data and to only apply those writes when the corresponding commit is received.

Cut Isolation (CI) under Cut isolation, if a transaction reads the same data more than once, it sees the same value each time. This isolation level prevents "Read Skew" anomaly, if this property holds on data items, it is called **Item Cut Isolation (I-CI)**, and if it holds when a transaction does a predicate-based read (e.g., SELECT... WHERE P) it is called **Predicate Cut-Isolation (P-CI)**. However this isolation level allows "Dirty Writes" and "Dirty Reads" anomalies.

CI could be achieved by the transaction caching reads locally at the client and then reading from the cache so that the values do not change in the same transaction unless the transaction itself overwrites them. Alternatively the reads could be stored in multiple versions at the server, and the transaction would only read from this versions, this could be achieved by assigning the transaction to a group of servers (transaction group) that would store this version and the following reads would only read from this group of server until the end of the transaction. The cache and multiple versions of objects are garbage collected at the end of the transaction.

Monotonic Atomic View (MAV) under MAV, once some of the effects of a transaction T_i are observed by another transaction T_j, after that, all effects of T_i are observed by T_j. That is, if a transaction T_j reads a version of an object that transaction T_i wrote, then a later read by T_j cannot return a value whose later version is installed by T_i. MAV also prevents "Dirty Writes" and "Dirty Reads" while guaranteeing all or nothing visibility of transactions.

MAV could be achieved using lightweight locks and/or concurrency control over data items [15]. This approach of achieving MAV does not satisfy high availability, because the system could stall in the presence of extended network partitions and has a significant impact on the system's throughput. There are enumerate alternatives to mitigate this problem and do an implementation of MAV without the use of locks [16, 10, 17] this systems store every data object that was ever written and replicas then gossip information about versions they have observed and construct a lower bound on the versions that can be found on every replica. At the start of every transaction the client chooses a timestamp that is lower or equal to the lower global bound, and during the transaction,

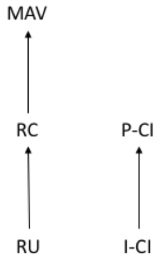

Fig. 5. Relation between the guarantees of isolation levels

the replicas return data items that have a timestamp lower than the chosen timestamp. These algorithms will be discussed in greater detail in Section 4.

There are other isolation models with higher guarantees that were proven that are not achievable with high availability [10], and for brevity are not mentioned in this report. Figure 5 summarizes all the guarantees and the relation among the different isolation levels mentioned above.

4 Related Work

In this section, we will present the surveyed solutions. For each of them, we start by explaining how the transactions are implemented, the isolation level and session guarantees that they grant. Finally, we will analyze the chosen strategy for propagating the transaction's updates.

COPS-GT COPS-GT[13] is a distributed key-value storage system that has been designed to run across a small number of datacenters. It implements a lock-free read-only transaction algorithm that provides clients with a causal+ consistent view of multiple keys in a distributed key-value store in at most two rounds of local operations.

COPS-GT assumes that each datacenter is fully replicated and linearizable. Moreover, clients only communicate with their local datacenter which makes COPS-GT sticky available.

In COPS-GT each client is responsible for maintaining its causal history. This history maintains all the client's direct and indirect (i.e., transitive) dependencies to the reads and writes operations it has performed. At the server side, each item is stored together with its list of dependencies, where each entry in the list contains the key and the version of the corresponding dependence.

Every time a client reads an item, the client appends the item's key to its history with that item's list of dependencies. When a client wants to write an item, first it computes the nearest dependency list and then sends it with the update. If the update was successful a new version is created and, it returns the

new version number that the client then adds to its history with that item's list of dependencies. After the write finishes the datastore start to replicate it asynchronously to the other datacenters. When a server receives a remote write it delays it until all the writes dependencies are visible.

To retrieve multiple values in a causal+ consistent manner, a client issues read-only transaction with the desired set of item keys. The COPS-GT as previously mentioned implements the read-only transactions algorithm in at most two rounds. In the first round, it issues n concurrent read operations to the local cluster, one for each key listed in the read-only transaction operation. Because COPS-GT clients commit writes locally and the local data store is linearized, the local data store guarantees that each of these explicitly listed keys' dependencies are already satisfied, therefore the reads on them will immediately return. The first round of reads returns the corresponding items and the items' versions and the list of these items dependencies. The causal dependencies for each item are satisfied if either the client did not request the dependent key, or if it did, the version it retrieved was greater or equal than that item's version in the dependency list.

For all the items that not satisfy this condition, the client issues a second round of concurrent read operations for the greatest version in any dependency list from the first round. This only happens when there are write operations racing the reads of the first round. These versions satisfy all the causal dependencies because they are greater or equal to the needed versions. Furthermore, the second round of reads does not introduce new dependencies because dependencies are transitive and all the new retrieved items depend on the items from the first round which enables the read-only transaction to finish in at most two rounds of reads.

Although how the client migration between datacenters are not specified in COPS-GT, it is still possible for a client to migrate from one datacenter to another. Since clients hold their entire causal history, once they arrive at the new datacenter, they can wait until all the required writes are made visible in the new datacenter before issuing new operations.

RAMP RAMP[18] is an algorithm that enforces a lock-free read-only transaction that guarantees the MAV isolation model among transactions within a datacenter. The authors presented tree variations of the algorithm, that differ between each other by the size of the metadata and the number of rounds needed to return a read-only transaction. For brevity, we will focus on the version that optimizes the number of rounds needed to finish the read-only transaction because, in our algorithms, we will also favor solutions with smaller number of communication rounds (instead of algorithms that use smaller metadata size but more rounds of communication). In this version, the read-only transactions are guaranteed to finish in at most two round of communication even in the presence of racing write and read operations.

RAMP assume that each datacenter contains all the data (i.e., they consider full replication) but the the set of items is spread over multiple servers.

RAMP implements write-only transactions that use a two-phase commit protocol that ensures that if a write of a write-only transaction is visible in one partition, all the other writes of that transaction are present in the corresponding partitions. This helps to ensure that clients do not stall due to reading an item written by a transaction that the effects are not yet present in all the corresponding partitions. After receiving a write, the server creates a new version and responds to the client with a prepared message. The new version is only made visible if the server receives a commit message from the client. After one server receives the commit message, the transaction is guaranteed to be committed in the rest of the servers and cannot be aborted.

In RAMP the client is responsible for guaranteeing that the read-only transaction returns a MAV. In the first round, the client issues n concurrent read operations over the requested item set. The read operation of the first round returns the item, the transaction number of the write transaction and a list of the keys of the other items that the write transaction modified. After all the reads have returned, if the client has read a version of an item that is included in a write transaction with a higher transaction number, the client begins the second round of reads for the specific missing version number. This scenario only occurs if the reads are concurrent with the writes or if the commit message from the client has not yet arrived at the corresponding server (but has arrived to at least another server).

A possible optimization is the following: if read request is receive for a version that is not yet visible, it is safe to make that version immediately visible. This is safe because a client only requests that version if the corresponding transaction has already committed (and, therefore, it is safe to make the version visible).

Cure Cure[19] was the first system to successfully support read and write operations within the same transaction while ensuring causal consistency and MAV isolation. Furthermore, Cure ensures that reads and writes can be executed using a single round of communication. In previous systems, such as COPS-GT and RAMP, read operations need at most two rounds of communication while guaranteeing only causally consistency or MAV isolation respectively.

Cure has been designed for a geo-replicated key-value store. Cure assumes that the full set of items is replicated across different datacenters. Moreover, each datacenter is partitioned, where each partition stores a non-overlapping subset of the key-space. The client executes all the operations of a transaction in its local datacenter (i.e., Cure implements sticky availability).

Each partition holds two vector clocks of size equal to the number of datacenters in the system. One vector clock (PVC) is responsible for tracking the remote updates received from the replicated partitions in remote datacenters. The other is responsible for maintaining the latest *globally stable snapshot (GSS)* known to that partition. The GSS is maintained by the partitions within the same datacenter exchanging their PVCs. The PVC contains the physical clock values of the commit timestamps, and it is updated in the corresponding entry when a local or a remote commit is received. Also if no local commits are received within a threshold, the replica sends a heartbeat to the other replicas.

Transactions use a two-phase commit protocol where one of the participating partitions of the transaction is assigned as the transaction coordinator which is responsible for committing the client transaction. Before starting a transaction, the client gets a transaction timestamp from the coordinator. This timestamp is the max between the entry of that datacenter in the last GSS seen by the client and the current coordinator physical clock. To ensure that the client's causal remote dependencies are satisfied, the coordinator stalls the operations until the client's last seen GSS is lower or equal to the coordinator's GSS. All the client future reads during the transaction must return version lower than the transaction timestamp and the client buffers all of its writes locally until the commit phase. During the commit phase, the client sends the buffered write set to the coordinator that will then propagate to the corresponding replicas. The replicas will prepare the new version and respond to the coordinator with their current physical time. The coordinator after receiving all the clock values chooses the maximum as the commit timestamp and sends it to the corresponding replicas. As the client's dependencies are all locally satisfied the effect of the transaction will be immediately visible to the client.

After locally committed the values are asynchronously propagated to the other replicas. The remote transaction effects become visible when the GSS advances past their commit timestamp. This ensures that all causally related transactions are already visible locally because they have a smaller commit timestamp.

In the presence of network partition between datacenters, the observed transaction from remote datacenters will be delayed until the network recovers, while local updates will continue to be made visible.

The client migration is not defined in Cure, however its possible to assume that the client could migrate to any datacenter. When the client tries to start a transaction, the coordinator will stall the client until all the causal dependencies are satisfied.

Eiger Eiger[20] is a distributed key-value storage system that extends the ideas behind COPS-GT. In addition to lock-free read-only transactions, Eiger introduces write-only transactions which ensure MAV isolation among transactions (together with a causal consistent view across multiple keys). Like COPS and Cure, clients transactions are performed in a single datacenter (to which the client is attached) and the key-value store is fully replicated across datacenters.

Eiger's read-only transaction algorithm has the same properties as COPS-GT's, however, the implementation is different, namely Eiger uses logical time instead of explicit dependencies to enforce causal consistency. Each partition in a datacenter keeps an earliest valid time (EVT) and its current logical time (LVT). EVT is the partition's logical time when it committed the last visible operation. As in COPS-GT, each client is responsible for maintaining its causal history.

The read-only transaction return in at most two rounds. The first round consists of reading from the partitions that contain the target data objects optimistically. The partitions return the current visible value, the earliest valid time (EVT) and its current logical time (LVT). Once all the first round reads

return, this metadata is used to check the consistency. All values are consistent if *the maximum EVT* $\leq$ *the minimum LVT*. If not, the transaction issues a second round of reads to the partitions that returned inconsistent values. In the second round, the client issues a read to a specific timestamp that satisfies *minimum LVT* $\geq$ *the maximum EVT*, this ensures that all the reads return and are consistent with the previously read values. The second round should be rare as it only occurs if a concurrent write operation is committed in the target partitions during the first round of reads.

Eiger's write-only transactions allow the client to write atomically across multiple data objects across multiple partitions without the use of locks. The write-only transactions are separated between local write-only transactions and replicated write-only transactions. The local version is used between the client and the local datacenter and the replicated version is used between datacenters to replicate the local write-only transactions. Both use a 2PC protocol and assign one of the target partitions of the transaction as the coordinator. In the local write-only transactions, the coordinator first prepares the writes by sending a prepare message to all the target partitions. These partitions create a new version of the data items and mark it as pending and respond with a yes vote to the coordinator. When the coordinator receives all the votes, it sends a commit message and make the new value visible and respond with an ACK to the coordinator. When the coordinator receives all the ACK, the coordinator ends the transaction and asynchronously propagates the transaction write by running a replicated write-only transactions. Upon receiving a replicated write-only transaction, the transaction coordinator must first check if all the dependencies are locally visible. When the dependency check returns, the coordinator proceeds with a local write-only transaction.

The client migration could be supported in the same way as the COPS-GT because the client keeps its entire causal history and its operations could be stalled until all the dependencies are satisfied.

It is important to note that in more than 10% write workloads Eiger has a very poor performance [19]. This is probably due to the overhead of the dependency checks of the remote updates and the read transaction probably need always two round of communication.

Clock-SI Clock-SI[21] is an algorithm that enforces Snapshot isolation (**SI**) using loosely synchronized clocks in a partitioned Data Stores. Most of the previous solutions that implemented SI in a distributed system relied on a centralized timestamp authority that managed the assignment of commit timestamps. However, this centralized authority introduced a single point of failure and could be a bottleneck in heavy workloads. Clock-SI overcame this problem by removing the timestamp authority, using instead loosely synchronized clocks to order transactions commits.

Clock-Si's read transactions return a consistent view of multiple keys across multiple partitions from a consistent snapshot. Moreover, as the Cure's read transactions, also return in only one round of communication.

However, using clocks to achieve a consistent snapshot manifests as a challenge for two major reasons. First, clock skew can cause snapshot unavailability, this occurs when a partition P_1 issues a read transaction T_1 with the snapshot timestamp of t to a partition P_2 that is in snapshot $t - \theta$, where θ is the amount the P_2 clock is behind P_1's. Therefore the Snapshot t is not yet available in P_2, and if a local write transaction in P_2 commits between $t - \theta$ and t this change must be included in T_1's snapshot. Second, a pending commit of a write transaction can cause a snapshot to be unavailable. If a write transaction T_1 with a snapshot timestamp t that updated the value of x, and started a commit operation at t' and finished the commit at t'', where $t < t' < t''$, if a read transaction T_2 starts with snapshot timestamp between t' and t'' and tries to read x, it should not return the value written by T_1 because it is not certain if the commit will succeed, however we also cannot return the earlier value, because, if T_1's commit succeeds, this older value will not be part of a consistent snapshot at t''.

Both examples are instances of a situation where the snapshot specified by the snapshot timestamp of a transaction is not yet available. Clock-SI deals with this problem by delaying the read operation until the snapshot becomes visible. In both cases, delaying a read operation does not introduce deadlocks, an operation waits only for a finite time until a commit operation completes, or a clock catches up. A possible optimization to reduce the probability and the duration of the read operation being delayed is by assigning a slightly older snapshot timestamp to the read transactions, having a cost that the read transaction return more stale data. This could the achieved by assigning to the read transaction a snapshot timestamp lower than the most recent timestamp snapshot by Δ. If we want the most recent data, the Δ needs to be set to 0. On the other hand, if we want to reduce the probability of the read operation being delayed, we could set the Δ to the maximum between the time required to commit a transaction to stable storage synchronously plus one round-trip network latency, and the maximum clock skew minus one-way network latency between two partitions.

The write transactions in Clock-SI are very similar to the Cure's [19], and the only difference is that the Clock-SI concurrent write transaction over the same items abort this is to ensure a much stronger isolation model (**SI**) than the Cure's causal consistency with MAV isolation.

SwiftCloud SwiftCloud[17] is a distributed data storage systems. A significant contribution of SwiftCloud is the ability to cache some items at the periphery network. Client nodes cache a subset of the items which reduces the latency and supports operations locally even in the presence of network partitions and failed datacenters.

SwiftCloud provides transactions with causal+ consistency and Monotonic Atomic View (MAV) as defined in section 3 supporting the merge of updates from concurrent transactions, instead of the more common last write wins policy. To achieve causal+ consistency clients maintain a session with the datacenter to ensure causality. In SwiftCloud the data objects are fully replicated across datacenters. However, the client maintains a cache that could be considered partial replication.

The client reads and writes over items in the client's local cache. Writes are then propagated to the client's local datacenter, that finally propagates the updates to the other clients and datacenters.

The client maintains a vector clock with an entry containing the most recent version number of the cached data items for each datacenter and, an entry containing the causal history of the client.

The transactions implemented in SwiftCloud are interactive, as in Cure [19] read and write sets are not predetermined and in the transaction could contain read and write operations. The read operation gets the locally available value and adds it to the client's history. A sequencer module at the datacenter in responsible for assigning the transaction identifiers. Making this centralized authority a performance bottleneck as well as a central point of failure. The update operation logs its updates when the transaction commits it writes the operation logs to stable storage locally and updates the client's vector clock with the commit timestamp of the transaction. Asynchronously these updates are propagated to the datacenter by their chronological order, after receiving it, the datacenter reassigns the timestamp with its logical clock and makes the update visible for the rest of the clients.

The datacenter has the client's current vector clock and only sends the updates that the client has not seen yet. When the client wants to add an object to its cache, it must first notify his local datacenter so that from that point on it could receive the updates for that object.

Orbe Orbe[22] is a distributed key-value store that provides read-only transactions using loosely synchronized clocks with causal consistency. Orbe requires one round of messages to execute a read-only transaction in the failure-free mode while COPS-GT requires maximum two rounds. Also, Orbe only tracks the nearest dependencies at the client side, compared to COPS-GT that the client has to track all the dependencies explicitly.

The key-value store is fully replicated across datacenters. Each datacenter is divided into N partitions, one per server and there are M different replicas. Clients performs the operations in a local datacenter.

The client keeps track of its causal history by maintaining a dependency matrix (DM_c) with N rows and M columns and a physical dependency time (PDT_c). PDT_c is the most recent update timestamp that the client depends on its session.

Each partition maintains a version vector (VV), which consists of M non-negative integer elements, that correspond to a logical clock of updates received from the replicas and the local updates. Also, every partition maintains a physical vector clock (PVV) with one entry for each replica containing the corresponding last seen physical clock value, to maintain this clock every partition sends periodically a heartbeat containing its current physical clock value.

When the client issues an update to an object, the client sends with the request its PDT_c and DM to the local server responsible for the partition. The partition upon receiving the request waits until its clock is greater than the PDT_c, this ensures that all the causal dependencies of the client are satisfied

before the update occurs. Then the partition creates a new version of the data object timestamped with the current physical clock value and the logical clock value. Attached to the new version is the DM_c, that afterward will be used on replication. The timestamp is then returned to the client that updates its PDT_c to the maximum between the received physical timestamp and its current PDT_c.

After the update to an object occurs, the server propagates the new version with the corresponding metadata. The receiving server uses the DM and the VV to verify the dependencies. It looks at the corresponding row of the same replicas and compares it to the VV. If all the values in VV are greater than or equal to the corresponding entries in the DM the server can go to the next step. If not, it means that the operation depends on other operations that the current datacenter may not store. So, the server contacts the corresponding servers, making an explicit dependency check. If they fulfill the dependencies, the remote update can be made visible and the corresponding vectors (VV and PVV) entries updated.

When the client issues a read-only transaction, it sends a read set to one partition. The partition upon receiving the request associates to the transaction a snapshot timestamp, similarly to [21] the snapshot timestamp (ST) is the current physical clock value minus Δ, where Δ is some positive number usually the time of one network round-trip. This reduces the probability of the read operation being delayed and the duration of the delay. If the partition does not contain an item from the transaction read set, the partition reads from another local partition that contains the item. However, before that partition could read the item it must first delay the read until two conditions hold: first the transaction ST must be lower than the partition current clock value, second the transaction ST must be lower than the partition's PVV. When the two conditions hold, the partition responds with the highest version of the item that is lower than ST. After all the reads finish, the client receives a set of values from the requested items and updates the PDT_c to the maximum of all retrieved items timestamp and the current PDT_c.

However, this creates a problem in the face of failing servers or network partition between datacenter, as the transaction read is delayed until $ST < PVV$, this means if a partition fails and stops sending heartbeats the transaction needs to be stalled until the partition recovers, because the replica is uncertain that all the dependencies are satisfied. To mitigate this problem when a transaction is delayed past some threshold, the transaction ST is changed to a lower value. This will return a more stale data in favor of reducing the delay of the read operation. However, if the downtime is very high, no useful progress will be made until the partition recovers, so to mitigate this problem Orbe changes to a failure mode. This mode uses a two-round read similar to Eiger [20] ensuring that more recent data is returned and the system progresses.

GentleRain GentleRain[23] is a causally consistent key-value store that implements causal consistent read-only transactions. The contribution of this system is that it uses a physical timestamp to track dependencies which reduce the com-

munication and storage overhead and eliminates dependency check messages for updates improving throughput compared to COPS-GT and Orbe.

The clients only have to store two timestamps, a physical dependency time (PDT_c) as in Orbe and the last seen global stable time (GST_c).

The server only maintains a physical vector clock (PVV) with one entry for each replica containing the corresponding last seen physical clock value and a GST. The lowest entry in the PVV is designated as local stable time LST, that is a lower bound of all the updates visible in all the replicas.

GST is the lower bound on the minimum LST of all partitions in the same datacenter. This value could be calculated by the partitions at the same datacenter periodically exchanging between each other their LST. However, merely exchanging the LST between could be a bottleneck and limits the scalability in the presence of a high number of partitions. To efficiently derive the GST GentleRain uses a tree. Child nodes send their LST to the parent node, upon receiving all the LST from the children it sends the lowest one to its parent node and this process repeats until the root node receives all the LST. The root node then calculates the GST and pushes it down the tree, saving the number of messages needed to calculate the GST.

GentleRain's update operation the client only sends its PDT_c. The server stalls the update until its clock is greater than the PDT_c received. Afterward, creates a new version assigned with the update timestamp and updates the PVV. The server returns the update timestamp to the client and propagates the update to the replicas.

The server upon receiving a remote update, the server creates a new version. However, this version is not visible to local clients until the partition GST becomes more significant than its update timestamp.

When the client issues a read-only transaction to any server in the client's local datacenter, this server will act as the transaction coordinator. The request contains the item set, the client's GST_c and PDT_c. If the coordinator's GST is lower than the GST_c, it updates the GST and starts requesting the item with a timestamp lower than the GST to the corresponding partitions. The coordinator returns the collected item values with the maximum update timestamp and the maximum GST. Upon receiving the reply, the client updates its dependency time and the GST_c.

However using the read operation it is possible that a client could read an item that is not yet in the GST, and so the read-only transactions will violate the causal consistency. If the PDT_c and GST_c are only apart by some defined threshold, the coordinator delays the read until the $PDT_c < GST_c$, in the case that the difference is greater than the threshold then the coordinator will use the protocol used in Eiger [20] for causally consistent read-only snapshots.

Yesquel Yesquel[24] is a key-value store that in the other end of the spectrum from the system mentioned above because in opposite to all the systems that offer very high availability sacrificing all the interesting features that relational databases provided such as ACID transactions, joins, between clauses, and others. The Yesquel provides all the features of a SQL relational database, however

scales as well as NoSQL in the same type of workload. To support this type of operations efficiently in a distributed fashion, Yesquel implements a distributed balanced tree (DBT) heavily inspired in a B+Tree. DBT consists of a tree where the nodes are spread across servers, the leaf nodes of the tree contain the values and the keys, and the interior nodes stores keys and pointers to other nodes.

Clients cache the inner nodes of the tree locally. This works because B+tree have a large fan-out and relatively few inner-nodes. Caching the tree allows the clients to search in the tree locally and only need to contact the leaf node, and so reducing the number of messages needed. However, the cache of the client becomes outdated as the tree is dynamic and changes over time as the nodes are split or merged due to insertion, deletion or replication. To solve this problem each node holds a fence interval, that is the lowest and the highest value key that the node contains, before fetching from the leaf node the client request a fence interval test, if the key is in the fence interval the client can proceed with the operation. If not the client goes up the tree to test the parent nodes until it finds a node that contains the wanted key, at the same time the client updates its tree locally in the cache, due to the nature of this type of trees the upper nodes rarely change and in the majority of time it can rebuild only a part of the cache without needing to fetch the whole tree again. When the client finds the node that satisfies the fence interval, it starts going down the tree and updating its local cache until reaching the leaf node.

Unlike a normal B+tree that split due to a tree node being overfull or empty, the Yesquel also slips overloaded nodes. This operation is called load splits. Yesquel estimates the workload of the node by keeping track of the number of accesses to partitions. If a node is overloaded, the node is split according to the estimated workload as each node receives approximately half of the workload. However is possible that one key is extremely popular, and no optimal split could be done, so the key is replicated and the node is split according to that key, to the lower bits of the key are attached random numbers and the client when searching for key also attaches some random generated number allowing to split the load between replicated keys.

Each client has a query processor making processing capacity increase linearly with the number of clients, which allows the system to scale well even with a high number of clients.

Yesquel's read-only transaction do not block or abort. To improve latency, the client acts as the transaction coordinator. As the state of the coordinator is irrelevant for the transaction outcome, the system can recover from fails by running periodically a function that checks the pending transaction and aborts them in cases of detecting falling servers or coordinator. Yesquel uses physical clocks to order operations similarly to the clock-Si, however, uses a much stronger 2PC with locking for write operations. Making Yesquel perform very poorly in write-heavy workloads.

ChainReaction ChainReaction[25] is a geo-distributed key-value data store that offers causal+ consistency, as the name suggests it was developed on top of chain replication[26]. ChainReaction supports causal read-only transactions.

However in contrast to other systems such as COPS-GT that assume linearizability inside the datacenter. That is all the reads are performed in the tail of the chain. In contrast, ChainReaction allows the read operation in the middle of the chain without breaking causal consistency and so improving the overall throughput.

The client tracks its causal history by maintaining a table with one entry for each item viewed during the client session. This entry contains the item's version and the chain index vector. The chain index vector contains an identifier that captures the position on the chain from where this item was last read, one entry per datacenter. All the read operation that the client does must have this metadata attached to the request.

The client update request must contain the key of the object, the new value, and a compression of all the read objects by the client since the last update. This update is forwarded to the chain head node that assigns a new version number to the update and then propagates to the following nodes. When the update is replicated in at k nodes then the update is denominated as a *k-stable* update. Only then, that the update result is returned to the client with the new version number and index of the last node that turned the update *k-stable*. When the update reaches the tail of the chain, it is denominated *DC-Write-Stable*.

After the chain head assigns a new version number to the update, this update is scheduled for replication and then propagated in batches to the other datacenters. To ensure that the update respects the causal history of the client, it is only visible after all the versions of the objects that this update depends are DC-Write-Stable in that datacenter.

When the client issues a read operation to the local datacenter and the chain length of the datacenter is equal to the chain index entry, every node in that chain can answer the request without needing to wait for a remote update. Otherwise, the request could be answered by all the nodes until the one that is specified in the chain index. However, as the updates are propagated asynchronously, it is possible that the item version that the client wants to read is not yet present in the head of the chain. So the head of the corresponding object in that datacenter needs to wait for the remote update or redirect the client to a datacenter that has that version number of the object.

The implementation of read-only transaction uses a sequencer process per datacenter. This sequencer is used to order all the put operation and reads that are part of a read-only transaction. The sequence number process maintains a different sequence number for each chain. To ensure that the client read objects respecting causal+ consistency even when updates are applied concurrently and without blocking update operations, ChainReaction keeps multiple versions of the same object. When the client issues a read transaction, it first must request the sequence number for each chain that contains the targeted objects. This sequence number is assigned by the sequencer and issues an individual read for the heads of the chain that contain the objects. The object returned has the last update sequence number, lower than the number assigned to the transaction.

Due to the asynchrony of the system is possible that the sequencer will order a put operation before the transaction and the value is not yet available in the head of the chain. This could be mitigated by stalling the read transaction until the write becomes visible or a timeout occurs leading to a transaction abort. It is also possible that the client has a dependency on an object read in a different datacenter that is not yet visible. In this case, the transaction is aborted and retried in a two-phase procedure. First, all the dependencies that have failed are verified by using a blocking read operation that blocks until the tail of the chain contains the object makes visible the version from which the client is dependent. Afterward, it reissues the transaction, and it is guaranteed to succeed, it is important to note that this case is very slow compared to the normal version.

4.1 Comparison

In this section, we will focus on comparing the advantages and the compromises of each strategy. We will subdivide this section by the type of isolation and consistency. However this comparison is not trivial due to different systems having different types of consistency and transaction isolation guarantees, so to do a reasonable comparison between the systems this section will be subdivided in comparing different techniques. Table 1 summarizes this information.

Techniques From the systems in the related work, we can point four different techniques. The techniques used are explicit checks (COPS-GT, Orbe, RAMP, Eiger), stabilization (Cure, Clock-SI, GentleRain), sequencer (ChainReaction, SwiftCloud) and explicit locks (Yesquel).

Explicit checks have the disadvantage of large size metadata and the overhead of verifying the dependencies explicitly that lower the overall throughput. System's that achieve MAV isolation (RAMP, Eiger), need at most two rounds of communication for the read-only transaction to return.

Stabilization uses loosely synchronized clocks to derive a GSS. The servers somehow need to exchange their clock values to derive a GSS. The two architectures used by the systems to exchange the clock values are a star architecture (Cure, Clock-SI) where all the servers in a datacenters exchange messages periodically between each other to derive the GSS, the other is to use a tree architecture (GentleRain) where the child nodes only exchange clock values with the parent in the tree. Using a tree reduces the number of messages needed to be exchange compared to a star architecture, on the other hand, the tree needs to be rearranged in case of falling nodes to ensure that the system can progress. This snapshot could contain some stall data. However, it improves overall performance and throughput because the transaction always returns in only one round of communication. Using loosely synchronized clocks also reduces the overall metadata, and it does not require explicit checks.

Sequencer uses a central authority through which all the updates must pass in order to give a global order. Having the advantage of reducing metadata size. However, increases the latency of all update operations by one round-trip, this centralized authority acts as a bottleneck and limits the overall throughput.

Systems	Technique	Transaction	Causality	Isolation model	Metadata	Communication rounds of Read-Only Transaction
COPS-GT	Explicit Check	**Read-Only**	✓	-	O(K)	at most 2
RAMP	Explicit Check	**Read-Only/ Write-Only**	X	MAV	O(K)	at most 2
Cure	Stabilization	**Interactive**	✓	MAV, I-CI	O(M)	1
Eiger	Explicit Check	**Read-Only/ Write-Only**	✓	MAV	O(K)	at most 2
Clock-SI	Stabilization	**Interactive**	✓	SI	O(1)	1
SwiftCloud	Sequencer	**Interactive**	✓	MAV, I-CI	O(M)	1
Orbe	Explicit Check	**Read-Only**	✓	-	O(N x M)	1
GentleRain	Stabilization	**Read-Only**	✓	-	O(1)	1
Yesquel	Explicit Locks	**Interactive**	✓	SI	O(K)	1
ChainReaction	Sequencer	**Read Only**	✓	-	O(M)	1

Table 1. Comparison of the different solutions. K is the number of all objects in the system, N is the number of partitions and M is the number of replicas.

Explicit locks do not require metadata to be exchanged. However, it significantly limits the throughput of write operations due to the need for transactions to be aborted if in the presence of a deadlock or concurrent accesses to the same data items.

All the systems mentioned above used multi-version concurrency control (MVCC) to improve concurrency of read and write operations. Also, it prevents a read transaction from aborting due to a concurrent write that overrides the required version. However, requiring more storage and having a computation overhead to garbage collect the old versions. Also, all the systems that provided causal consistency required a session between a server and a client. The only system that did not require a session was RAMP because it does not provide causal consistent reads.

4.2 Shortcomings of Current Approaches for Edge Computing

First of all, it is important to note that all the systems mentioned were designed to solve specific problems and in a specific context.

The majority of systems above assume that the data is fully replicated. We cannot provide full replication in the edge due to the nodes having much less storage compared to datacenters. The solution needs at least to implement partial replication support, and this creates many challenges due to the large size of the metadata needed to provide any consistency or isolation level between transactions.

Taking this into account, adopting explicit checks has many limitations, especially the metadata size is much greater due to the higher number of nodes, and the data being partially replicated. Even the more straightforward approach of RAMP that doesn't guarantee causality would not be feasible in the context of the edge due to the large size of metadata.

Stabilization, on the other hand, has much less metadata however requires a high amount of communication between the nodes to derive the GSS. This is more problematic in the context of the edge due to poorer network infrastructure and a higher number of nodes compared to the communication inside the datacenters. The propagation of clock values by using a start architecture would probably overload the network in contrast the tree architecture could be viable due to a much fewer number of messages needed to calculate the GSS.

Using a sequencer to order the updates has similar limitations to the other strategies. As described above the sequencer strategy has a bottleneck in the form of an ordering authority, this bottleneck is emphasized due to a higher number of nodes.

In the edge with partial replication is more likely that a client will update the same item concurrently with another client connected to another edge node making some isolation levels too strong for our context, such as SI (Clock-SI and Yesquel) for not tolerating concurrent updates on the same data items, forcing the transaction to abort.

So we need to define what the isolation level and consistency we want to support. MAV seems to be the more viable option to be implemented in the edge, however using the two rounds of communication will not work due to the geographical distance and poor network infrastructure that will increase latency. So reading from a snapshot could be a more viable option due to requiring only one round of communication. However, it will not work as is due to the higher cost to derive the GSS.

5 Architecture

5.1 Consistency Model and Base Techniques

In this section, we will discuss the modifications that need to be made to support partial replication in a geo-distributed setting. The chosen architecture for our solution is a tree, where the root of the tree is the datacenter and the partitions of the datacenter are partially replicated across the rest of the tree nodes (Figure 6). The parent node always contains all the partitions from its child nodes. This strategy facilitates the propagation of updates and the migration of the clients that we will discuss later.

Our solution will implement transactions that guarantee Monotonic Atomic View (MAV) and respect causal consistency. We opted to guarantee MAV as it does not require to compute a GSS. The GSS is too expensive to compute in the context of edge computing, the visibility latency would be too significant and the transaction read would return to stale data. Our solution will use vector

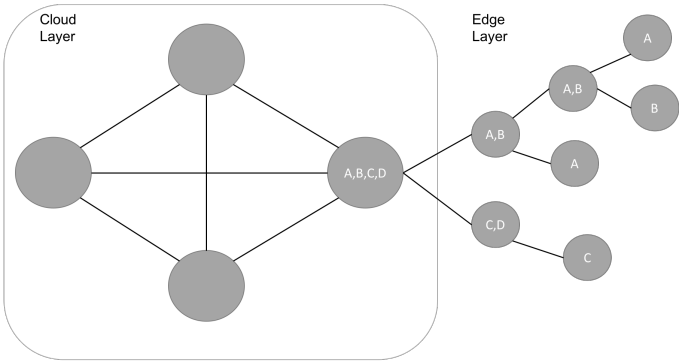

Fig. 6. Connections between all the system components.

clocks with every entry containing the physical clock value of the commit and the nearest causal dependency. We will have this vector for each of the partitions, and the size of the vector will be the number of replicas of that partition. We opted to use a vector clock instead of only a physical clock like in GentleRain, due to our need for supporting partial replication. Using only one scalar would introduce too many false dependencies and having a significant impact on the visibility latency. So we will trade-off memory space at the server and client side to give a significantly lower visibility latency. To ensure this locality the client would be attached to an edge node by a session, and when needed it will be migrated to another node. We will discuss more in-depth the client migration and the operations exported to the user next.

5.2 Client Migration

In this section, we will describe how and when the client will be migrated to another edge node and what is the criteria to choose the new node. This architecture was designed having in mind the geographic locality of the data. This means that the users from the same region usually access the same data and migrations to other nodes is not that frequent. However, in the case that a node receives an operation to update or read an item that it does not contain, the node redirects the client to another node that contains all the necessary items by going up the tree. Before the client could execute an operation, the node needs to guarantee that all client's causal dependencies are satisfied and visible. This is achieved by delaying the operation until the remote updates that the client depends on are made visible. From that point forward, the client will use this node as its local node. This approach has a few problems, namely if a set of keys turns very popular the majority of the clients would migrate to a few specific nodes that contain all the items, overloading them and leaving other nodes down the tree in an idle state or under-loaded. One approach that could be implemented is to run a distributed transaction across a set of edge

nodes that globally contain the set of items needed, and this has an added cost of coordination and more than one round communication. However it is possible that the nodes are too far apart, and the latency is too high to do this type of coordination, in this case, a more viable strategy would be if a set of keys continue to be very popular they can be replicated in the lower nodes of the tree, this avoids doing the very costly distributed transaction.

5.3 Supported Operations

In this section we introduce a set of operations that are vital to solve the proposed problem.

Read-Only Transaction The client requests a set of items from the local edge node. This node checks if it has all the needed objects and if so replies with the requested item values that have a version timestamp lower than the current clock values minus some δ, upon receiving the response the client updates its causal history. If not, it starts a client migration to a different node that contains all the items requested by the client.

Write-Only Transaction The client sends a set of items and the new values to the local edge node, this node checks if it has all the needed objects and if so creates a new version of the items and timestamps it with the current physical clock value, and returns to the client the timestamp and asynchronously starts to propagate the updates to the other replicas. If not, it starts a client migration to a different node that contains all the items affected by the transaction.

Remote update When a node receives a remote update, it delays the operation until the node receives all the causal dependencies of the update. When all the dependencies are satisfied, the update creates new version of objects. If we assign the original update timestamp of the update it is possible that due to network latency or clock skew, that a read transaction would read partially the transaction and violate MAV isolation. Therefore, our solution is to assign the current clock timestamp value rather than the original update timestamp. This approach provides MAV isolation without having to do the expensive two rounds read.

6 Evaluation

The evaluation process will focus on three topics: i) performance on the delivery of updates; ii) visibility latency; iii) fault tolerance.

6.1 Performance

We will measure the performance of the system by measuring the throughput of the system. More specifically the number of remote updates that become visible over a period of time. Also, we will measure the time that it takes to migrate a client from one node to other until it can resume a normal operation.

6.2 Visibility Latency

It is important to measure how our solution will affect the visibility latency of remote updates. More specifically, the time that it takes from receiving a remote update until it becomes visible.

6.3 Fault Tolerance

To measure the performance of our system in the presence of network partition and failing nodes. Most importantly we need to guarantee that our solution does not stops the system in this conditions. Therefore, we will compare the difference between the time it takes for remote update to become visible in different availability scenarios (i.e., normal execution, network partition, and failing nodes).

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

As the number and power of the devices that are connected to the cloud grows and produce an enormous amount of raw data, and shipping all these data to a central datacenter may be unfeasible. These new requirements have motivated the need to support edge computing[1], a model where the service provided by cloud datacenter is complemented by a set of smaller servers located closed to the edge of the network.

In this report, we surveyed the most important weak consistency models and the main techniques that have been proposed to support weak forms of transactions in a distributed environment, we elaborated a solution that supports weak transaction in the edge with little impact on performance compared to normal causal consistency without transaction, however with much greater consistency guarantees. Finally, we presented our evaluation methods and the schedule of future work.

Acknowledgments We are grateful to Manuel Bravo for the fruitful discussions and comments during the preparation of this report. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) and Feder through the projects with references PTDC/EEI-COM/29271/2017 (Cosmos) and UID/ CEC/ 50021/ 2019.

References

1. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE Internet of Things Journal **3**(5) (October 2016) 637–646
2. Okay, F.Y., Ozdemir, S.: A fog computing based smart grid model. In: Proceedings of the International Symposium on Networks, Computers and Communications, Yasmine Hammamet, Tunisia (May 2016)
3. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, Portland, OR, USA (July 2000)
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc. (1987)
5. Abramova, V., Bernardino, J.: Nosql databases: Mongodb vs cassandra. In: Proceedings of the 5th International C* Conference on Computer Science and Software Engineering, Porto, Portugal (July 2013)
6. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3) (July 1990) 463–492
7. Lamport, L.: On interprocess communication. Distributed Computing **1**(2) (June 1986) 86–101
8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (July 1978) 558–565
9. Vogels, W.: Eventually consistent. Communications of the ACM **52**(1) (January 2009) 40–44
10. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: Virtues and limitations. In: Proceedings of the 39th International Conference on Very Large Data Bases, Trento, Italy (August 2013)
11. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing **9**(1) (March 1995) 37–49
12. Brzezinski, J., Sobaniec, C., Wawrzyniak, D.: From session causality to causal consistency. In: Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, Coruña, Spain (February 2004)
13. 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 23rd ACM Symposium on Operating Systems Principles, Cascais, Portugal (October 2011)
14. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, CA, USA (May 1995)
15. Das, S., Agrawal, D., El Abbadi, A.: G-store: A scalable data store for transactional multi key access in the cloud. In: Proceedings of the 1st ACM Symposium on Cloud Computing, Indianapolis, IN, USA (June 2010)

16. Chan, A., Gray, R.: Implementing distributed read-only transactions. IEEE Transactions on Software Engineering **11**(2) (February 1985) 205–212
17. Preguica, N., Zawirski, M., Bieniusa, A., Duarte, S., Balegas, V., Baquero, C., Shapiro, M.: Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In: Proceedings of the 33rd IEEE International Symposium on Reliable Distributed Systems Workshops, Nara, Japan (October 2014)
18. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with RAMP transactions. ACM Transactions on Database Systems **41**(3) (August 2016) 1–45
19. 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: Proceedings of the 36th IEEE International Conference on Distributed Computing Systems, Nara, Japan (June 2016)
20. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, Lombard, IL, USA (April 2013)
21. Du, J., Elnikety, S., Zwaenepoel, W.: Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In: Proceedings of the 32nd IEEE International Symposium on Reliable Distributed Systems, Braga, Portugal (October 2013)
22. Du, J., Elnikety, S., Roy, A., Zwaenepoel, W.: Orbe: Scalable causal consistency using dependency matrices and physical clocks. In: Proceedings of the 4th ACM Annual Symposium on Cloud Computing, San Jose, CA, USA (October 2013)
23. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. In: Proceedings of the 5th ACM Annual Symposium on Cloud Computing, Seattle, WA, USA (November 2014)
24. Aguilera, M.K., Leners, J.B., Walfish, M.: Yesquel: Scalable sql storage for web applications. In: Proceedings of the 25th Symposium on Operating Systems Principles, Monterey, CA, USA (October 2015)
25. Almeida, S., Leitão, J.a., 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 (April 2013)
26. Van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, San Francisco, CA, USA (December 2004)