An improved causal consistency mechanism for geo-replicated data stores

Călin Iorgulescu
LABOS, I&C, EPFL

***Abstract*—Reliable distributed data stores are an important pillar of today's large-scale online systems. In order to accommodate the strict requirements imposed, careful trade-offs between *consistency*, *availability* and *partition tolerance* are made. Often data stores that employ a weaker consistency model, but have higher availability guarantees, are used to provide an "always-on" experience. However, this can expose the user to potential inconsistencies, the mitigation of which requires more complex application logic and tweaking.**

This paper presents a new causal consistency mechanism for key-value data stores that achieves performance and availability comparable to ones using eventual consistency. The idea is to preserve the virtues of strongly consistent storage services [1], while supporting incremental scalability and being highly decentralized and heterogeneous [2]. The work builds upon the design of Lloyd et al. [3], achieving near linear scalability with respect to the number of servers used per datacenter.

***Index Terms*—causal consistency, key-value data stores, geo-replication, high-availability**

I. Introduction

MODERN Internet services greatly rely on data storage services in order to provide a seamless user experience. These data stores must be resilient to failures, capable of handling large amounts of data, and must scale in order to accommodate an ever increasing number of users. In addition to this, they must be flexible enough to provide differentiated services, based on SLAs[1] conforming to application requirements. For example, a social networking site might tolerate displaying some posts with several seconds delay, but will need to ensure that the displayed posts are in the right order (i.e., the context is not lost).

Due to the wide spectrum of applications existing today, choosing a sufficiently flexible interface for the storage service allows for better integration and more efficient resource utilization. This work further focuses on services that provide a *key-value* access vector. That is, data is stored as objects which can be *queried* and *updated*. Other solutions, such as filesystems or databases, can either have insufficiently explicit semantics, or they can be overly-specialized, respectively. In [2], Amazon reports that many of its applications only require primary-key access, and using a relational database would be inefficient and would decrease scalability.

A. Partitioning and Replication

To cope with the very large data sets, it is necessary to *partition* them across multiple servers. This has the advantage of both increasing storage capacity as well as providing better throughput under heavy load. To this end, Amazon's Dynamo uses consistent hashing with different placing strategies across its servers[2]. A similar strategy is employed by COPS[2] [3].

In order to provide failure resilience and durability, the data set needs to be *replicated* across multiple datacenters. This also allows for workload distribution, reducing operation latencies and providing better performance overall. When scaling systems over a wide geographical area, it is important to provide localized replicas to overcome communication latencies. The downside, however, is that the replicas need to be kept synchronized, a task that is far from trivial. According to the CAP theorem [4], in order to provide tolerance to network partitions (e.g., link failures, power outages, etc.), a careful trade-off between consistency and availability must be made.

B. A scalable causally consistent data-store

Ideally, a data store system would scale linearly with the amount of resources, while still providing availability,

Proposal submitted to committee: June 10th, 2014; Candidacy exam date: June 17th, 2014; Candidacy exam committee: Prof. Rachid Guerraoui, Prof. Willy Zwaenepoel, Prof. Katerina Argyraki.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(B. Falsafi) (signature)

[1] Service Level Agreements
[2] Cluster of Order Preserving Servers

low latency and partition tolerance. For causally consistent systems, there is the added complexity required to enforce causality. The replication process needs to install updates while respecting their causal order.

In the case of COPS, this is done using metadata to track dependencies across update operations. Each datacenter keeps a copy of the entire data set, which is partitioned over several servers using a consistent hashing scheme. A client would connect to its local datacenter and issue queries and updates. While queries are handled locally by the corresponding partition, updates are first written to stable storage, and then they are dispatched to other replicas. Update propagation is done in parallel in a asynchronous manner, with each partition communicating directly with its correspondents in other datacenters. Since partitions are disjunct, certain items may have dependencies on other partitions. A partition may need to issue *dependency check requests* to the other partitions in its datacenter before it can install a received update.

We find that the remote dependency check messages saturate the servers in a data center, reducing operation throughput. What is more, an increase in partitions can cause a proportional increase of checks per update operation. We build upon the existing design to reduce the amount of remote dependency checks to at most *one per update*, allowing the number of partitions to be increased without impacting intra-datacenter message traffic. The new mechanism achieves a near-linear increase in throughput with respect to the number of partitions. The trade-off is that our system delays making newer versions of an object visible until they are fully replicated. We argue that this is acceptable for the mentioned use cases (e.g., social networks, news sites, etc.).

II. Consistency Models

Strong Consistency. The linearizability (or strong consistency) model dictates that the propagation and installation of updates between replicas should happen atomically. All replicas need to report the same view over the data store at all times. One solution to enforce this is to have the system serialize update operations across replicas, and to block until each completes. While this is very desirable from a consistency and durability stand-point, its downside is that service availability is impacted when there is a failure, and operation throughput suffers as the number of replicas grows[1].

Eventual Consistency. This model provides very weak consistency guarantees: the only invariant is that all replicas will, after an arbitrary period of time, reach the same view of the data set. In contrast to linearizability, eventual consistency provides high availability and high throughput. Update propagation is usually done in parallel, and in case of failures, each replica can continue serving clients independently. A significant problem, however, is that replicas need to reach a consensus regarding concurrent updates which conflict. Performing *conflict resolution* is a difficult problem, and usually it is left up to the application. Sometimes a data store level solution can be used (e.g., last-write-wins), but this is often infeasible. One example is Amazon's Shopping Cart: in the case of conflicts, "add to cart" operations are prioritized over "remove from cart" operations. The end result is that, while removed items can resurface, added items are never discarded[2].

Furthermore, some classes of applications (such as social networks or news sites) require partial ordering of updates. In order to enforce this on top of an eventually consistent system, each particular application would need significant additional implementation effort.

Causal(+) Consistency. As a middle ground between the previously mentioned solutions, causal consistency provides stronger guarantees than eventual consistency, while still maintaining high throughput and availability. This model guarantees that across all replicas the data view respect a partial *causal* ordering. For example, let's assume a user named Alice reads an article on her favourite news site, and posts a review. Given this situation, her review only makes sense when the article she's replying to is also displayed (i.e., her post is *causally dependent* on the article), however there might be replicas that have not received said article yet.

An important concept is the notion of *potential causality* between operations. The following three rules define potential causality (denoted $\rightsquigarrow$):

- **Execution Thread** – if a and b are two operations in a single thread of execution and a *happens before* b, then $a \rightsquigarrow b$.
- **Gets From** – a is a `put()` operation and b is a `get()` operation, then $a \rightsquigarrow b$ (b reads the context created by a).
- **Transitivity** – $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

In addition to this, the *causal+ consistency* model (described by [3]), also guarantees that, as with eventual consistency, all replicas will eventually converge to a common view. This is one of the strongest models that does not sacrifice availability under network partitions.

III. State of the art

A. Chain Replication

This paper describes the basic functionality desirable for a key-value storage system. It supports storing binary objects (blobs) together with *query* and *update* operations. The interface also ensures that these operations fail-fast, returning an error to the application when a problem is encountered in the system. All operations are executed in a sequential order across the replicas, achieving **strong consistency**. The system is designed to offer high throughput and availability on *fail-stop* servers.

Protocol. The nodes are organized in a *chain* structure, similar to a queue implemented using a simply linked-list. There is a single HEAD node, a single TAIL node, and several intermediate replica nodes. Each node keeps a copy of the data set, and for each of the objects it also maintains a queue of pending operations. Updates can only be issued to the HEAD node, which forwards them down the chain to the TAIL. Once the TAIL has committed the update to stable storage, it sends a confirmation to the client. Query operations, on the other hand, can only be issued to the TAIL node, which generates replies. Since the TAIL is the only node to handle queries, and it is also the last node to install an update, it is guaranteed that a

new version of an object is only visible after it has been *fully replicated*. The data set is partitioned into multiple *volumes* which are placed on separate chains. A single node can take part in multiple chains, with, potentially, different roles.

Failure handling. From a client's perspective, a request to the system that is lost is indistinguishable from a request received, but ignored. This can sometimes happen, for example, when there is a failure and the chain needs to be reconstructed. An advantage of this approach is that it does not expose the client to new failure modes in the case of transient server failures.

Chain replication uses a designated Master server (which is transparently replicated) to detect failures and issue reconfiguration requests. There are 3 possible failure scenarios:

1) failure of the HEAD node
2) failure of the TAIL node
3) failure of any intermediate node.

In case 1, update operations become unavailable, while query operations are still possible. Once the Master node detects the failure, it simply changes the HEAD to the next node in the chain. In case 2, no operations are possible, since only the TAIL node can reply. Similar to case 1, once the failure is detected, the previous node in the chain becomes the new TAIL. Both these cases take 2 message delays to recover. In case 3, the predecessor p and successor s of a failed node need to establish which of the updates that p has forwarded have reached s. This process requires 4 message delays, during which update operations are unavailable, but query operations are unaffected.

This failure handling schema is a type of *Primary/Backup* approach, with the Primary role shared by both the HEAD and TAIL. This provides better load distribution, especially for balanced query/update workloads. The downside is that update operation latency increases proportionally to the number of nodes in a chain.

Simulation experiments. The network used for experiments simulates infinite bandwidth (very high by comparison to the traffic generated by the system), with a latency of 1ms per message. This scenario is comparable to that found in a modern datacenter. Chain replication is configured with a single chain and 2, 3, and 10 replicas, respectively. The query and update latency times are fixed to 5ms and 50 ms, and the time for each intermediate node to commit an update to stable storage is set to 20ms. Given this setup, for 3 replicas, it takes 94ms to perform an update, and only 7ms to perform a query. For a scenario with 25 clients, it is shown that chain replication achieves better throughput by up to 50% compared to a Primary/Backup implementation, for workloads with up to 20% update operations. When the proportion of updates is increased, the 2 systems converge to the same value, with the mention that chain replication never fares worse.

When partitioning the data set over multiple chains, the throughput of chain replication becomes the same as the one of Primary/Backup. This is observed for a number of 5000 volumes. It is important to note the impact of update heavy workloads, with a $\sim$24x decrease in throughput between query-only and update-only request sets.

Discussion. Chain replication offers better availability than traditional Primary/Backup protocols, and under low network latency conditions is well suited for intra-datacenter replication. Given the strong consistency guarantees it offers, it provides a good foundation for a reliable storage fabric. COPS[3] uses it as a local replication strategy in order to mitigate partition server failures.

On the other hand, due to the fact that the update latency is proportional to the latency between replicas, it is not suitable for replication across a wide geographic area. Given that failure detection is done by a centralized authority, recovery would not be possible if one of the partitions is cut off from the Master.

B. Amazon's Dynamo

Dynamo is Amazon's highly-decentralized, highly-available geo-replicated data storage system. Because of the numerous types of applications that rely on it, there are very high reliability constraints, the system being required to handle huge workloads (i.e., hundreds of thousands of simultaneous connections). At the same time, it strives to have very low latency, with around 200ms per operation measured in the 99.9th percentile. Such a tight constraint is required when dealing with stringent SLAs for each application running on top of the system. In order to achieve these requirements, the system focuses on availability, rather than consistency.

The system stores all data as opaque binary objects that use a unique 128 bit ID generated from the *key* of an item. A client can access an item using the `get()` and `put()` primitives. Multiple versions of each item are stored, and each item also has metadata associated with it (e.g., version information, etc.). Sometimes concurrent updates of the same item can cause conflicting versions to appear. Since Dynamo needs to provide an *always writeable* store, conflict resolution takes place at read time. When such a conflict is detected, a notification is pushed to the application, prompting for a resolution. This approach respects the end-to-end principle[5] by allowing each application to choose the manner in which to resolve the conflict. There is also support for data-store level reconciliation, such as last-write-wins. Furthermore, a set of configurable parameters is provided, with which applications can use to tune the behavior of the system to meet their requirements.

Design considerations. The design of the Dynamo system is based on the following considerations:

- **Incremental scalability** – adding a new node should increase storage space and throughput, but the operation itself should have minimal impact on availability
- **Symmetry / Decentralization** – all nodes are peers and share equal responsibilities, without any need for a centralized authority
- **Heterogeneity** – each node should handle work based on its capabilities

In order to achieve these goals, multiple techniques are employed:

- **Partitioning** – consistent hashing is used to handle data distribution, while a *zero-hop DHT* is used for lookup.

This solution requires more routing information on each node, provides better latency than multi-hop routing alternatives.

- **High Availability** – Dynamo is designed to always allow write operations, so multiple versions of an item can appear, uniquely identified using *vector clocks*.
- **Temporary Failure Handling** – operations use a subset of the available nodes which is healthy in order to perform operations (Sloppy Quorum). In case a replica goes down, updates destined for it are handed to another live host which keeps them until the replica returns (Hinted Handoff).
- **Permanent Failure Recovery** – Merkle trees are used to reduce the amount of data exchanged at replica recovery time.
- **Group Membership with Failure Detection** – a gossip based protocol is used for host join/leave operations and for node failure detection. Nodes cannot arbitrarily join or leave the system; they are added manually by an administrator instead.

Partitioning and Replication. Each node gets assigned a random position in the consistent hash, being responsible for all the keys before its relative position. In order to achieve better load distribution, a node can be assigned multiple positions in the hash, these being called *virtual nodes*. Since the partitions' size is relative to the number of nodes, any change in membership requires reconfiguring all the members. Dynamo uses a fixed partition size based on the expected number of nodes, thus *decoupling data partitioning from partition placement*. Therefore, when a new node joins or leaves the system, it is no longer required that all the other nodes reconfigure themselves. Instead, the new node simply "steals" items from existing nodes, while a node leaving distributes the items randomly among the remaining nodes.

There are three configurable parameters offered by Dynamo: the number of replicas used for a data item (N), and the number of hosts that need to complete a read (R) or write (W) operation, respectively, for that operation to be considered successful.

Each individual item is committed to a *coordinator* and $N-1$ other nodes. The replicas are chosen to be successors of the coordinator in the consistent hashing scheme. A *preference list* is created for each item, describing the set of non-virtual nodes that store it. Out of this list, a subset of healthy nodes is used for most operations, and the size of the subset is configurable. When a replica goes down, updates for it are handed off to another healthy host, with a "hint" of which was the initial destination of that update. The receiving healthy host waits and periodically checks if the previous node has recovered. When this happens, it forwards it all the updates missed. Since this approach may not always be sufficient, the fall-back solution is to use Merkle trees in order to efficiently detect inconsistencies between item versions and to transfer only the data subset that differs from a healthy replica.

Versioning and Conflict Detection. Dynamo uses *vector clocks* to detect multiple conflicting versions of an item. A vector clock is represented as a list of (HOST, COUNT) tuples, and is stored as part of an item's metadata. A `put()` operation must refer to the version of the item that it is superseding. This information is usually obtained by a previous `get()` operation. Each item also maintains a Lamport clock which is incremented at each local `put()` operation. When committing a new version to local storage, it is associated a new vector clock that will contain the tuple ($host_{id}$, $host_{clock}$). A conflict is detected when the update is sent to the other replicas: the received version's clock is compared to the latest installed version's clock. Vector clocks allow *partial ordering* of operations that are causally related. If a replica cannot establish this order, it yields that the versions are probably concurrent and they conflict. Otherwise, the older versions are simply discarded.

For example, let's assume that we have an item A replicated across replicas x, y and z. The item is first written at replica x, and it is given the vector clock $A_{v0}[(x_{id}, 1)]$. Next, A is updated concurrently at both y and z, creating 2 new versions: $A_{v1}[(x_{id}, 1), (y_{id}, 1)]$ and $A_{v2}[(x_{id}, 1), (z_{id}, 1)]$. Comparing these 2 versions it is impossible to establish an order, because A_{v1}'s clock for y is greater than A_{v2}'s, but A_{v2}'s clock for z is greater then A_{v1}'s.

One concern is that the size of the clocks grows proportionally with the number of servers handling the operations. This number is fairly limited, since most users will connect to the same replica for the duration of a session. Still, vector clocks are pruned and elements older than a specific threshold are discarded.

Executing Operations. In order for a client to access the system, it needs a way to identify which host it should use as a *coordinator*. Dynamo provides two ways of doing this: One alternative is to use a separate **load balancer** service, which assigns each client to a certain host. The advantage of this approach is that it does not require additional effort on the client side. On the other hand, having to contact the load balancer service introduces an extra hop, which increases overall operation latency. The other solution is to have the client use a **specialized library** which is aware of the partitioning mechanism used by Dynamo and can decide which host to use by itself. This provides lower operation latency, but may not be acceptable for all applications.

As previously mentioned, Dynamo also makes available the N, R and W knobs. The later two dictate the number of hosts required to perform a *read* or *write* operation in order for that operation to be considered successful. Usually, these values are chosen such that $R + W > N$, but $R < N$ and $W < N$. More precisely, these values establish the *quorum* needed to perform operations. This offers better flexibility, since choosing the right balance of these values impacts **data freshness**. For example, if an application requires support for intensive read workloads that require always fresh values, it can set R to 1 and W to N. However, the trade-off in this case will be that the system will not be able to perform updates if less than N healthy replicas remain. Further, a larger value of W will lead to higher durability, since an update will be replicated on multiple hosts, but it will also impact the latency of such operations. In addition to this, the system becomes unavailable for the respective operations if less than R or W healthy replicas remain. A typical combination used in

production is (N, R, W) = (3, 2, 2).

Experiments. Dynamo has been tested in a production environment using heavy workloads and different types of applications. It is important to note that, while the strict SLAs required operation latency times of 200-300ms in the 99.9th percentile, the average operation latency of the system was actually lower by an order of magnitude (i.e., $\sim$ 20ms).

Another important insight is the degree of version divergence that can occur. Intuitively, the more conflicting versions exist, the harder it is to reconcile them. Over a 24 hour profiling period, it was found that 99.94% of the users of the Shopping Cart service saw exactly 1 version (no conflicts). Only 0.00057% saw 2 versions, and around 0.00009% saw 4 versions. It is further argued that in a system processing primarily human generated requests there is little to no chance of divergence, and, therefore, that conflict resolution happens sufficiently rarely as to not produce performance degradation.

A key observation, however, is that when several hundreds of hosts are used, the Zero-Hop DHT design would become very large and would impact performance. It is argued that in this case, employing a constant-time DHT system can mitigate the problem.

Discussion. Dynamo is very well suited for large online systems, providing the "always-on" experience desired. Its main strength is, in fact, its highly configurable system which allows each application to make its own trade-off between durability and availability. Furthermore, the consistent hashing scheme allows for even load distribution, contributing to the system's ability to scale linearly with the number of servers. Also, it makes a case for applying the end-to-end principle in designing a middleware system.

However, the downside to this approach is that updates are not completely ordered, and each application is required to detect if inconsistencies have surfaced. As discussed before, this pushes the burden of detecting and handling such problems to the application. In certain cases, this requires complex solutions to be implemented in order to ensure proper functionality.

C. COPS

The COPS system is designed to be geo-replicated and provides causal consistency guarantees, while maintaining high throughput and availability. The data set is partitioned across multiple servers using consistent hashing, similar to Amazon's Dynamo. The paper defines the concept of ALPS systems (***A****vailability*, *Low* ***L****atency*, ***P****artitions* and ***S****calability*) and makes a case for *causal+ consistency* as the strongest consistency model applicable to such systems. It builds upon existing systems which used operation log exchange to enforce causality. Further, it uses a layer of local replication enforced by chain replication to handle intra-datacenter failures. Because all the causality information is captured by item metadata, it is also resilient to entire datacenter failures.

The interface exposed to the user is similar to Dynamo's (`get()`/`put()`), but also has added support for *read transactions*. The semantics of this operation entail that given a list of item identifiers, the read operations will be executed *atomically*. There is an extra call available (`getTx(<list of ids>)`) which returns a list of items corresponding to the provided keys, with the guarantee that they belong to the same data set *snapshot*. Due to the implementation differences, the standard COPS implementation does not offer read transaction support, and maintains a *single version* data view. The COPS-GT implementation provides support for read transactions and keeps a *multiple version* view of the store.

Since the system provides causal+ consistency, it employs extra metadata information in order to detect potential conflicts. More specifically, it sends the previous item's version with the `put()` call, allowing the local partition server to identify conflicting concurrent updates. The conflict handling strategy is the same one as Dynamo's: the application is notified and asked to provide a handler.

Dependency tracking and checking. Each item is assigned a version number, such that if two items x and y of versions i and j, respectively are causally dependent, then $i < j$. Version numbers are generated based on a *Lamport timestamp* and the ID of the originating node. Each client employs a library which keeps context information: the dependencies of each operation. When an item is retrieved using `get()`, it is added to the list of potential dependencies of that client session (*Gets From* clause). When an item is written using `put()`, the client library sends the dependency list to the server, clears it, and then adds the returned version information of the written object to the list (*Execution Thread* clause). However, under read-heavy workloads, the dependency list can become large quickly, so only the *nearest dependencies* are tracked. For example, if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then it is safe to discard a as a dependency for c, since it is guaranteed by the *Transitivity* clause. However, COPS-GT requires *all dependencies*, not just the nearest ones, to be recorded in order to provide a consistent snapshot. To further prevent the dependency list from becoming very large, dependencies are also *garbage collected* once they are fully replicated.

A client issues `put()` operations to its local datacenter, which assigns it a new version, and then commits it to stable storage. The update is then placed in a replication queue, and is further sent to all the other replicas. When a replica receives a remote update, it needs to make sure that all its dependencies are installed before it. Since the dependent items can be on other partitions in the same datacenter, the server may need to issue a blocking **remote dependency check** call. That is, it will contact the partition responsible for the dependent item, and it will wait until the dependency is satisfied. During this time, the partition server will still process other user and replication requests.

Read Transactions. COPS-GT provides the convenient mechanism of transactions which guarantees that multiple reads are executed atomically. This is useful in avoiding time-to-check-time-to-use race conditions. The system accomplishes this using only two rounds of parallel partition server querying. The trade-off is that it requires full dependency information in order to achieve this. In the first round, all the requested keys are queried and their values, along with their version information are retrieved. After this, it is verified if any of the dependencies are found amongst the retrieved keys,

and if so, the version of these dependencies is compared to the version of the keys. Should it be found that any dependencies are newer, a second round of queries is done to retrieve the required versions.

Since transactions require more dependency information, it is important to garbage collect these values so as to keep both the client-side context, as well as the server-side metadata small.

- **Dependency Garbage Collection** First of all, dependencies of fully replicated items are not useful, so it is safe to remove them from the client context. The system tracks when an update has been fully replicated, and when it is, it signals the client. Since all transactions are upper-bounded to `trans_time` seconds (5 seconds in practice), the client waits this additional time, and then removes all fully replicated dependencies.
- **Version Garbage Collection** Further, it is only necessary to keep older versions that might still be needed for a potential ongoing transaction. Since transaction times are bounded, it is safe to discard any item version older than `trans_time` + a `delta` (to account for clock skew).

Evaluation. COPS is evaluated using a deployment of 2 datacenters, each having between 1 and 16 servers. Each server is accessed by 1024 clients which issue query and update operations with a ratio of 1:16, and the objects are of 1 byte each. With 16 servers / datacenter a $\sim$12x increase in throughput is observed in comparison to using just 1 server / datacenter. When the number of clients is increased to 32K, the overall throughput drops by 15%, but the increase obtained with 16 servers remains $\sim$ 12x. However, this value drops to $\sim$ 10x when the object size is increased to 1 kilobyte.

Discussion. COPS implements a highly available, geo-replicated data store that guarantees causal consistency. In addition, it provides support for read transactions, which are very useful in mitigating data race conditions.

However, while it does manage to scale well with the number of servers in a datacenter, it does not achieve linear scalability, with respect to operation throughput. Furthermore, even with the extensive garbage collection, the metadata can still increase proportionally with the number of servers, especially for heavy read workloads. These are typical to social networks; the Twitter homepage loads, by default, 20 posts.

We will further present an analysis of this problem, and a new design that achieves improved throughput and scalability with the number of servers in a datacenter.

IV. Our Approach

Given the sub-linear scalability, in terms of throughput, obtained by COPS, we decided to analyze its behavior compared to eventual consistency. We used a C++ implementation of COPS that also provided eventual consistency support, by not tracking any dependencies. We setup an experiment that exposes the systems to a workload that stresses dependency tracking, by executing a read from each partition, then executing a write to a random remote partition. This means that for each replicated update, a replica will need to consult all other local partitions. We replicate each partition 3 times and vary the number of partitions between 1 and 5. As we can see in Figure 1, COPS scales worse than expected, with a throughput rate $\sim$ 40% lower than that of eventual consistency.

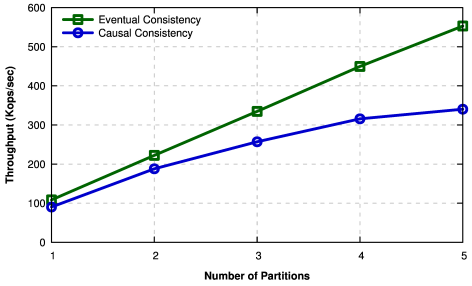

Fig. 1. Throughput comparison of COPS against eventual consistency

One principal bottleneck is the number of remote dependency checks required in the case of mostly-read workloads. In order to reduce this number, we build our system based on the following observation: *Gets From* (or *external*) dependencies need to be checked because replicas of the same partition can have different views of the data set.

A. Intuition

We propose the following solution: only make updates **visible** after they have been **fully replicated**, while allowing the update propagation process to be done in parallel, just like in COPS. It is important to note that it is still necessary to track *Execution Thread* (or *internal*) dependencies. Since only the *nearest dependency* is important, each update will need *at most one* remote dependency check. Once all dependencies are satisfied, the update can be safely committed to stable storage. It is only made visible after it has been fully replicated.

This design, however, requires an efficient way of detecting full replication of an update. A possible approach is to simply have all replicas report to the source replica when they have installed the update, and have the stable replica confirm full replication, similarly to a *two-phase commit protocol*. Unfortunately this approach requires broadcasts over multiple datacenters, and would not scale for our intended purpose. An improved idea uses the fact that each server keeps a logical clock which, together with the source node id, is used to identify the version of a data item. If replicas exchanged information about which remote updates they have installed, then it would be possible to compute the most recent fully replicated update. We call this technique "*trading update visibility latency for throughput*".

We can foresee two downsides to this approach:

- a client might not be able to immediately see its own updates reflected locally; he may read staler values instead.
- in order for full replication to be confirmed, all replicas need to exchange update information. This would mean that the system would no longer be able to install fresher values if a replica would fail. It would still be able to process requests, and all updates would be installed in the remaining healthy datacenters, but the data returned would be stale.

We tackle these issues individually: we provide *multiple update spaces*, which allow each client to see his own changes locally as soon as they are installed. Clients would only have access to their own *private* update space, until the update is fully replicated. When this happens, said update is moved into the *global* update space, where it can be accessed by anyone.

In order to tolerate datacenter failures, we envision employing a strategy which is able to detect such failures and remove the faulty replicas from the information exchange. Using a system such as PAXOS[6] would enable the system to recover from such outages with little data becoming stale.

B. Protocol

We briefly describe the details of the protocol that provides scalable causal consistency by making updates visible only when they are fully replicated.

Replication confirmation. Each partition replica n maintains a *version vector* VV_n with the property that $VV_n[i]$ is the version of the latest committed update received from replica i. In addition to this, a *replicated version vector* RVV_n is also kept, having the property that $RVV_n[i]$ is the version of the latest update that has been fully replicated. We work under the assumption that the data store holds multiple versions (similar to COPS-GT).

We allow each replica to broadcast its version vector VV_n, and to receive the version vectors of all other replicas. This operation is done periodically, and the interval can be set arbitrarily, depending on the required freshness of the data, as well as the overall latency between datacenters. We call this operation a "*Replica Heartbeat*". Upon receiving a remote version vector, each host updates its RVV: $RVV_n[i] = max(RVV_n[i], min(VV_j, 0 \leq j < N \ and \ j \neq n)), 0 \leq i < M$, where N is the total number of replicas in the system.

For an item A, A_{rid} is the id of its source replica, and A_{lc} is its logical timestamp (i.e., the version identifier). When a `get()` request is received, the server will return the newest item version that respects the following condition: $RVV[A_{rid}] >= A_{lc}$.

Multiple update spaces. We assume that all the users of the system have a unique identifier U_{id}, and we store the id of a user in the metadata of a item (A_{uid}). In order to keep updates private from other users until the time of full replication, the previous condition simply becomes: $RVV[A_{rid}] >= A_{lc} or A_{uid} = U_{id}$.

Garbage collecting old versions. As in the case of COPS-GT, it is necessary to discard old versions that are no longer needed to keep the size of the data store low. Since the `get()` call requires the system to walk the list of item version from newest to oldest, once a viable fully replicated value is found, all values older than it can be pruned. This is done by a separate background thread, so as to not impact performance.

C. Experiments

We implemented our *replication confirmation* protocol and we ran the same experiment presented at the beginning of the section. Figure 2 shows that operation throughput is now nearly identical to that of regular eventual consistency. What is more, our system scales nearly linearly with the number of servers in each datacenter. Finally, since each item only needs to check its nearest dependencies, the size of the metadata becomes constant, regardless of how many partitions are used in the system.

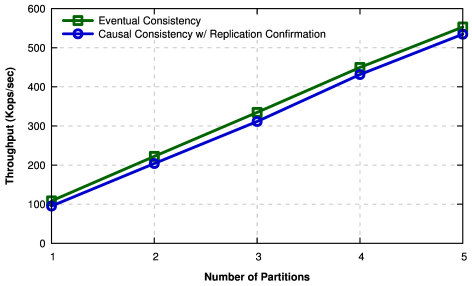

Fig. 2. Throughput comparison of COPS using replication confirmation against eventual consistency

V. Conclusion and future work

Lloyd et al. [3] claim that Causal+ consistency is the strongest consistency model that can be achieved in large scale ALPS systems. We provide a new mechanism that scales nearly linearly with the number of partitions and achieves high throughput and availability. The trade-off is that update visibility can be delayed, depending on the replica heartbeat interval and the network latency between datacenters. However, we argue that this is acceptable for applications such as social networks, where such delays would be unnoticeable to users.

Further, we intend to thoroughly evaluate the impact of our technique on the freshness of data, in order to assess if there are workloads which would expose stale values. Also, the system can be extended to offer read transaction support (such as COPS-GT), potentially, without any extra metdata. We also intend to evaluate the impact of transient datacenter failures, the time required for our system to recover, and the percentage of data that becomes stale during this time.

References

[1] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability." in *OSDI*, vol. 4, 2004, p. 91–104.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, vol. 7, 2007, p. 205–220.

[3] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, p. 401–416.

[4] E. A. Brewer, "Towards robust distributed systems", PODC, 2000.

[5] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, p. 277–288, 1984.

[6] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.