Scalable and Consistent Client Caching on Distributed Key-value Stores

Charles Xu
Author
charles.xu@duke.edu

Jeffrey S. Chase
Advisor
chase@cs.duke.edu

Thesis submitted in fulfillment of the requirements
for graduation with distinction in the Department of
Electrical and Computer Engineering of Duke University

April 2017

Contents

Abstract **3**

1 Introduction **3**

2 Background **4**
2.1 Underlying Key-Value Store 4
2.2 Consistency, Scalability and Causality 4
2.3 Assumptions 5

3 Design **5**
Consistency 7
3.1 Open RPC as Session Lease 7
3.2 Session Lease Only with Read Quorum 7
Scalability 8
3.3 Imprecise Notification using Prefix of Key 8
3.4 Idle Client Suspension and Fast Recovery 9
3.5 Invalidation v. Update Notification 9
3.6 Prefetch and Eviction 9
Causality 10
3.7 Causal Order of Updates using Logical Time 10

4 Application **11**

5 Implementation **11**

6 Evaluation **12**
6.1 Volume Size and Number of Notifications 12
6.2 Recovery Speed Compared to Polling 13
6.3 Invalidation v. Update 13

7 Conclusion **14**

References **15**

Abstract

We propose a client cache design for the distributed key-value stores that are sharded and replicated using state machine replication on an NRW quorum without a primary. The caching scheme is scalable and supports two consistency models—linearizability of writes on each key, or a causal order of writes on all keys. Applications exhibit different access patterns in the size of values stored on the remote, the range of key space utilized, and the frequency of writes. Configurable parameters in our design allow each application to balance among maintained states, generated traffic, and response latency according to its distinctive pattern. Through implementation and simulation, the cache performance is evaluated and quantitative results summarized.

1 Introduction

In large-scale storage services, caching has been a popular technique to reduce the response latency and increase the system throughput. In a distributed setting, however, writes to the remote storage impose significant challenges to any caching scheme in terms of consistency, scalability, and causality. Clients who have cached a stale copy of the value must learn of such updates in a way that scales and preserves the order of dependency.

Polling with TTL (Time-to-live) has been used as the caching mechanism in many systems, such as the Web and DNS (Domain Name Service). Every object is cached with an expiration time, upon which the client polls the object again. This approach is not ideal. Setting the TTL too long risks serving stale data, but setting it too short essentially does not cache the object and risks congesting the network with polling requests. Clients also have to keep track of expiration on a per-item basis, which does not scale with growing number of cache entries.

Addressing this issue, recent work on client caching for replicated stores could be generalized to two paradigms. One follows the gossip/epidemic/anti-entropy approach [12], which is scalable but provides only eventual consistency, a much weaker model than what we propose. The other is pubsub in primary-backup replication [6, 9], which is strongly consistent and is informative to our design. However, we would like to go further to address the case of symmetric replication with quorum-based protocol and no primary.

The server-driven publish-subscribe caching scheme that we propose has the server notify stale clients when subsequent writes commit. It ensures consistency, scales well, and allows causal constraint. Consistency helps predict the outcomes of concurrent events and therefore defines correctness and ensures safety. Scalability becomes challenging, because states about clients are maintained on the server to ensure timely notifications to the right clients. Further, the distributed key-value store is sharded and replicated, and thus so are the server states. Sharding and replication also make causality difficult, which enforces a partial order of the global events to observe the happen-before relationship [10].

Multiple techniques are devised to reduce the amount of server states while preserving consistency. Logical time is used to achieve causality should the application opts in. A prototype has been built using Node.js and deployed on AWS EC2. Through simulations, the tradeoffs of different configurations are investigated. On recovering stale

caches, our caching scheme spends 80% less time than does standard polling design.

2 Background

2.1 Underlying Key-Value Store

Sharding The key space is partitioned into shards such that accesses to keys on different shards could be processed concurrently and the system throughput therefore magnified. The total number of shards is dynamically elastic. Techniques such as consistent hashing [1] enable the system to find the right shard for any given key, even across view change.

Replication Each shard is also replicated on multiple replicas using state machine approach, which assumes that each operation is deterministic and that each replica executes the sequence of operations in the same order. Consensus is achieved using NRW quorum voting [2]. It states that for a replica set of size N, the read quorum consists of R replicas and W for write quorum. All reads and writes are sent to and applied on the N replicas, but the value of a read is known once R replicas have responded, and a write is committed if W or more replicas acknowledged. If $W+R > N$, then reads and writes from the NRW quorum is always consistent, because the read quorum and the write quorum must overlap on at least one replica. It means that for any read quorum, at least one replica has seen the latest write and therefore the read quorum always returns the latest value.

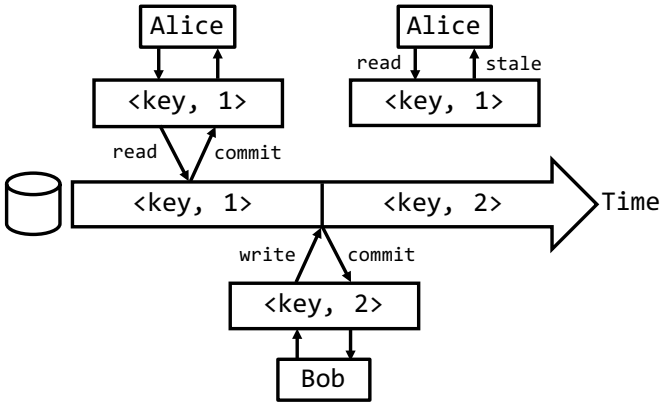
Client as Coordinator Unlike the primary-backup replication where clients only talk to the primary who is elected by the quorum to collect the votes from the replica sets for each operation[3, 4, 5], our key-value store asks each client to be the *coordinator* for all of its requests, read or write. The word coordinator is carefully chosen to differentiate from a single primary because there are multiple coordinators in our system, as there are multiple clients. It is assumed that clients, as well as replicas of the key-value stores, are spatially separated and that the transmission delay significantly affects the system performance. By acting as the coordinator for itself, the client removes the extra round trip to the primary.

2.2 Consistency, Scalability and Causality

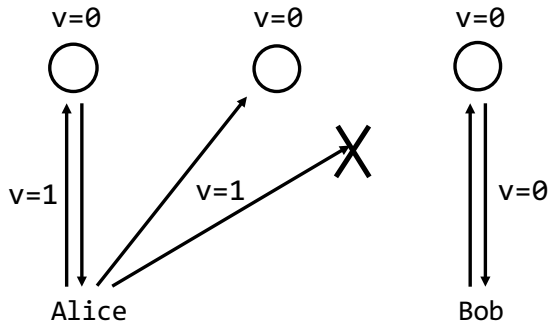
A *consistent* cache is consistent with the remote storage and serves its client the latest value on a given key. The word consistency by itself is used in such narrow sense for clarity. A more general concept, *consistency models* define the ordering of events at concurrent sites by multiple actors. There is a spectrum of consistency models, from weak to strong. Our design introduces and supports two of those (§3).

Staleness causes inconsistent reads. It could exist on either the client cache due to multiple clients or the distributed key-value store due to multiple replicas, as shown respectively in Figure 1(a) and 1(b).

To fight staleness and retain consistency, solutions can be categorized as either client polling or server notification. In either approach, consistency and scalability are conflicting objectives. In the former case, a low polling frequency may cache stale data, but a high polling frequency risks congesting the network with refresh requests. In the latter case, the server maintains states about clients, the amount of which grows as



(a) Alice and Bob start with cold caches. Alice first reads the value to her cache. Then Bob updates the value. Later, when Alice reads again, a cache hit returns the stale value, unaware of the update by Bob.



(b) Replicas start with value 0. Alice writes into all replicas and received 2 acknowledgements so the new value is committed. Assuming $R = 1, W = 2, N = 3$, Bob reads from one replica and might get stale data, since $R + W \not> N$.

Figure 1: Examples of Staleness in (a) Client Cache and (b) Replicated Store

more clients join. Server states may be reduced at the cost of more network traffic, which is yet another scalability issue. Our objective, therefore, is to expand the utility frontier and to achieve the balance of the two. Depending on the application, such balance might shift and the system settings are configurable to accommodate different scenarios.

Causality is defined as a happened-before relationship. In the context of the key-value store, users might store pointers to other keys in the value. Therefore, if Alice writes to key k_1, and then Bob writes to key k_2 with value that references k_1, then we say k_1 causally precedes k_2, and everyone has to learn about the write by Alice before the write by Bob, because otherwise referencing k_1 from k_2 might return stale value.

2.3 Assumptions

At best, consistency on the client cache could only be as strong as that of the key-value store. We assume that the key-value store is strongly consistent and focus on maintaining consistency with caching. Applications above our system are assumed to be read-intensive, so optimizations are to focus on minimizing read latency to improve average latency. Secure and reliable channels are assumed to ensure in-order exactly-once tamper-free delivery in peer-to-peer communications.

3 Design

The design follows a server-driven publish-subscribe approach. Clients who have cached a particular key-value pair are called *subscribers* to that key, and the server will notify these subscribers about subsequent updates on that key. The *client* refers to the client library or stub, so the cache itself is an implementation detail whose existence remains transparent to the application, as shown in Figure 2.

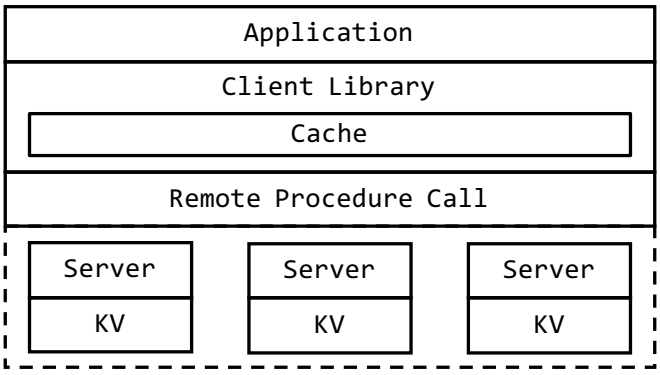


Figure 2: An Overview of Server-driven Publish-Subscribe Cache Design

The basic approach to the pubsub design is to have the server maintain a *subscription set* that tracks who has cached what. The subscription set introduces several key challenges to scalability and consistency. The amount of subscription records grows as the number of clients increases and the range active keys expands. The granularity at which subscription sets are maintained is one of the key factors in scalability, but any compression must also ensure timely and accurate notifications to protect consistency. As clients come and go, the server must know when it is safe to reclaim resources and how to recover stale caches when clients reconnect. Clients must distinguish server failure from no committed writes, as both cases surface as no server notification received. Given the store is replicated, so are server states. We look for ways to retain the same set of information without replicating the server states on the entire replica sets. Since the store is also sharded, messages from different shards arrive at the client in a random order and causality is, therefore, difficult.

We approach these challenges with the following. Session leases are used to detect server failure from no server notifications. Each client maintains sessions only with replicas in the read quorum, so states are only replicated on the read quorum and session renewal traffic reduced. The server may choose either invalidation or update as the notification to subscribers, where the new value is attached to an update but not to an invalidation. Keys are grouped into volumes, and subscription sets are managed per-volume, so notifications are generated per-volume too. Every write is time-stamped, and a single timestamp proves to encapsulate the states in the entire client cache and enables fast recovery. Logical time is used to preserve the causal order of notifications when delivered to the upper-layer application.

The cache supports two consistency models:

Processor Consistency [7] All committed writes to the same key are observed in the same sequential order by all participants.

Causal Consistency [10] All committed writes to the key-value store are observed in an order that preserve the happen-before relationships (causality).

The causal consistency is a stronger model than processor consistency, since causal updates on a given key suffice to give the later but not vice versa. Performance,

however, will suffer from higher latency for the stronger model. In the following, we start by addressing both consistency and scalability and then extend to causality.

Consistency

3.1 Open RPC as Session Lease

As opposed to tracking expirations on every object returned by the server, we manage client-side freshness with a single session with the server [8]. While the session is maintained, the client cache is consistent with the key-value store, because the session ensures connectivity between the client and server, and connectivity ensures delivery of server notifications. A session is established when the server grants a session lease to the client. Any session lease comes with an expiration time. The client maintains the session by periodically renewing the session lease. The renewal of session lease serves as liveness detection, which works even when the communication channel between the client and the server is interrupted. The reason is that the client and the server have agreed on the duration of the lease. Hence, when the session expires, both the client and the server know and agree that the session ends. Resources allocated on behalf of the client may then be safely reclaimed.

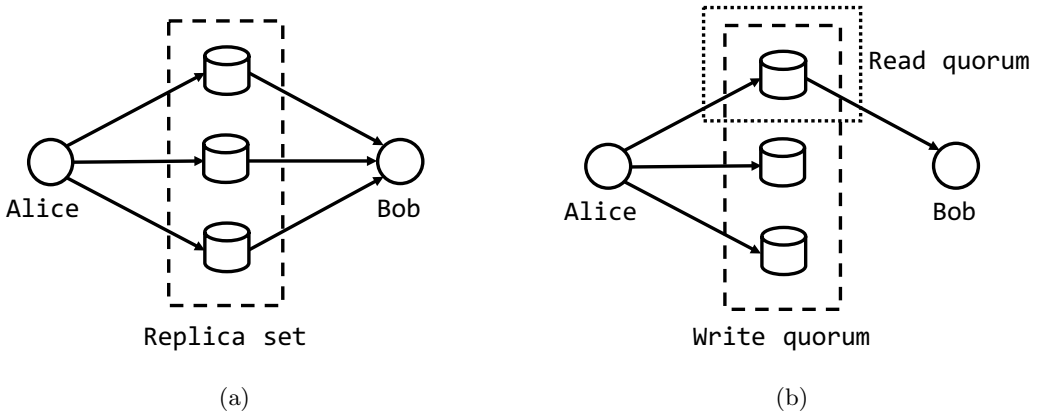
Once the session expires, consistency is no longer guaranteed. Reads are consistent if the server fails (so no writes may be committed) but not so if the network is partitioned. In an asynchronous network such as the internet, these two cases are indistinguishable [3, 4, 5]. After the session expires, the cache notifies its application and continues serving reads to provide higher availability.

We use timed open RPC (Remote Procedure Call) as session leases. The client invokes on the server an RPC, which stays open until either server responds or the call times out. The server may respond with a set of updates/invalidations to the client cache, or simply return an ACK before the timeout if no writes are committed on the key-value store. Any response by the server closes the RPC, and if so, the client will immediately issue another. The session expires if the server fails to respond or response fails to reach the client by the timeout. The advantage of open RPC is that it merges lease renewal and invalidation messages, so traffic is reduced relative to sending separate messages [9].

3.2 Session Lease Only with Read Quorum

Recall that the key-value store is replicated with the client being the coordinator for itself. For consistent read and write from the replica set, one obvious approach is to have the client maintain a session with each replica, as seen in Figure 3(a). The problem of such design arises when the number of replicas is large. Leases and notifications from each replica are replicated and transmitted independently, creating superfluous traffic across the network.

This situation is avoided if the client maintains sessions only with replicas in its read quorum as shown in Figure 3(b). Recall that the key-value store uses NRW quorum consensus (§2.1), so consistency is maintained as long as the read and the write quorums overlap, because at least one of the read replicas knows the latest committed write and will notify subscribers whose read quorums it participates in. Our system is read-intensive, so R is configured to be much less than W to optimize for lower read latency.



(a) (b)

Figure 3: Alice's write on key k is committed on every replica in the N quorum. Assume Bob has cached a stale entry on k. Two Schemes to Achieve Consistent Read by Session Leasing with (a) Quorum N and (b) Quorum R. Each replica in the respective quorum then independently notifies Bob about Alice's write.

The example in Figure 3(b) is configured as $N = 3$, $R = 1$, $W = 3$, commonly referred to as ROWA (read one, write all).

The client retains a *preference list* of read replicas, which is sorted by the transmission delay from the client to each replica. When one of the sessions times out, the client establishes a new one with the first replica on its preference list who are not in the last read quorum. Given the order of the preference list, the read quorum is always chosen such that the average latency is minimized.

The new problem with maintaining sessions with only the read quorum is that the subscription sets are no longer replicated on all N replicas. A new replica that just joined the read quorum does not know the client cache states. The client needs to inform the new replica about its cache to help the replica reconstruct the subscription sets. Using techniques in §3.3 and §3.4, reconstruction is fast and scalable.

To mitigate the period of inconsistency gap between one replica failure and new replica discovery, one may maintain R+1 sessions, namely one standby session, such that any single fault is tolerated by seamlessly failing over to the standby replica.

Scalability

3.3 Imprecise Notification using Prefix of Key

Maintaining subscription sets on a per-key basis does not scale since the key space may be huge. One mitigation is to group keys with the same prefix into one volume and manage the subscription sets on a per-volume basis. Clients are subscribed to volumes. Any update on a key k in the volume v generates notifications to all clients subscribed to v. Indeed, it is possible that some clients in v do not cache on k, and if so the client library simply ignores this notification. Notifications as such are imprecise but accurate, in that all clients who do cache on k are notified.

The volume size is configurable by adjusting the length of prefix used to identify a volume. The tradeoff is that larger volumes reduce the number of states that servers must maintain but also create more unnecessary notifications. We explore such tradeoff

through evaluation §6.

3.4 Idle Client Suspension and Fast Recovery

Idle clients remain in the subscription sets, and the server still grants leases and delivers updates to them. Idleness could be detected by the client library when the application running above has not read or written any keys for a long time. If the application is idle, the client will stop renewing leases from the server. After the session expires, the server may safely reclaim resources allocated to the idle client [11].

When the idle or partitioned client reconnects, its stale cache requires recovery procedure upon session reestablishment. Purging the entire cache is not ideal, because applications are read-intensive and writes are less frequent, so most entries in cache are likely to be fresh.

We achieve fast recovery using the timestamp of the last server response before the session expired. Along with a set of prefixes of keys in cache, that single timestamp t captures the entire client cache state. For each prefix p that the stale client has cached, the server returns from the volume identified by p all committed updates whose timestamps are later than t. As shown in §3.7, the stale client must have known all updates to the keys it has cached if those updates are issued before t. Everything after t, the client missed. Knowledge of writes committed after t is sufficient to recover consistency. The recovery response from the server follows the technique of imprecise notification (see §3.3).

3.5 Invalidation v. Update Notification

Our cache design supports both *invalidation* and *update* as forms of server notifications to subscribers. The difference is that the updated values are attached in an update message but not in an invalidation. The advantage of sending updates is that clients save another round trip to the server to read the new value (transmission delays are assumed high). However, if the value is large in size (therefore high propagation delay) and if the probability of another access is low for that key, then invalidations give lower average latency and scale better. The choice between the two is configurable and applications may choose based on their access patterns.

3.6 Prefetch and Eviction

Applications seldom read just one key-value pair. The value might reference another key, whose value might reference another key. Such reference chains create a special type of locality where linked key-value pairs are likely to be accessed soon. To further optimize the average latency, the client may recursively prefetch other key-value pairs that are linked by the ones already in cache.

The cache is bounded in size. Upon eviction, the client checks if there remains in cache other keys with the same prefix as the evicted key. For quick look up, the client may track prefixes in cache using the hierarchical hash tables in the form of `Map<prefix, Map<key, value>>`. When a prefix maps to an empty key-value table, then the client informs the server who then unsubscribes the client from the corresponding volume given the prefix.

Causality

3.7 Causal Order of Updates using Logical Time

To commit on the key-value store an update on key k or on key q that references k, the server needs to verify that the issuing client has seen the latest value on k. This consistency constraint prevents version branching and achieves Processor Consistency. Its importance can be seen in the following scenario.

Consider the case where multiple clients issue concurrent updates on the same key, as in Figure 4.

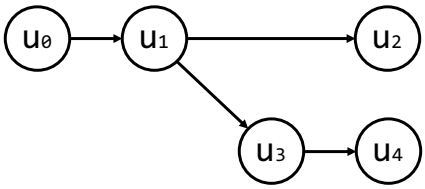


Figure 4: Concurrent Updates with Version Branching. We are speaking of *concurrency* in a rather strict sense. The client who applies u_2 onto u_1 has not seen u_3 or u_4, and the client who issues u_3 and u_4 has not seen u_2. Therefore, u_4 does not causally precede u_2 and u_2 does not causally precedes u_4. We say that u_2 and u_4 are concurrent. So are u_2 and u_3.

Suppose that u_3 and u_2 are concurrent updates by different clients and that u_3 is issued slightly after u_2. It is possible that u_3 arrives at the server earlier than u_2 does. If u_3 is committed by the time u_2 arrives, then u_3 is the latest write that u_2 does not know, so the server must reject u_2 to protect consistency. Notice that for concurrent updates as such, any ordering is consistent, but committed updates could never roll back, so the first write wins forever [12].

Another challenging scenario is that causality must be preserved during the construction of a reference chain. If Alice issues an update u_1, and then Bob issues u_2 that points to u_1, then all other clients must learn u_1 before u_2. If they learn u_2 first without knowing u_1, then referencing u_1 is either not found or stale at best. The dependency here is that u_1 causally precedes u_2, which requires Causal Consistency.

Given only one shard, the consistency constraint at the start of this section suffices to preserve causality. Bob knows about u_1 before he issues u_2. It must be that u_1 is delivered to or polled by Bob, which means u_1 has been committed on the key-value store. It follows that other clients either have already received an update about u_1, or u_1 and u_2 together are delivered in the next exchange. It is never the case that u_1 arrives at a later update than u_2.

The problem is worse when u_1 and u_2 are stored on different shards, as illustrated in Figure 5. Causality is retained if the client only delivers an update u when u is stable, i.e. all updates that causally precede u are received. Let $[u_1, u_2, ..., u_i]$ be a stream of updates on one shard of the key-value store. Say in the latest exchange, the shard replies with the update u_i. It means the client has already known all the updates from that shard up to u_{i-1}, because if not, u_{i-1} will be part of the server response. Likewise, u_{i+1} has not been applied on the key-value store just yet, because if so, u_{i+1} will in the response as well. We consider u stable if all the timestamps of the latest responses from all shards are later than that of u. Nothing that happens later than u could possibly cause u, and everything earlier than u the client has already known.

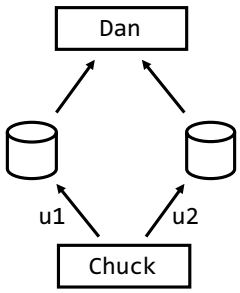


Figure 5: Sharded Key-value Store Given Random Order of Messaging. Here, Chuck writes both u_1 and u_2 and knows u_1 causally precedes u_2. Even if Chuck waits until u_1 committed and then issues u_2, Dan could still receive u_2 before u_1.

The subscript for each update u is a form of logical time, which defines a total order of events that satisfies the partial ordering constraint by causality. It is worth pointing out that such order, albeit deterministic, is rather arbitrary and is not the only one that preserves causality [12]. The causality captured using the logical time also relies on the information exposed to the system. Causal exchanges outside of the system may be considered concurrent by the system. The problem of incomplete observation is a difficult one and is beyond the scope of this paper to discuss it in any detail. An attempt at this problem using physical time and clock synchronizing algorithm is described in [10].

4 Application

Many systems of critical services for a secured internet can be seen as applications of key-value stores, examples including Secure Border Gateway Protocol (S-BGP), Domain Name System Security Extensions (DNSSEC), Certificate Authority as Public Key Infrastructure (CA-PKI), etc [13]. They store values in the form of certificates. A certificate is signed with the private key of its issuer, so its content is safe from being tampered [14]. The primary task of applications as such is to validate a given assertion about a chain of certificates. The goal of the client cache is to minimize the total time it takes for such validation. Causality is desirable to ensure correctness of validation.

5 Implementation

We build and open-source[1] a prototype of the cache design we propose. The tech stack is summarized below.

Aspect	**Technology**
Server	Node.js + Express
Client	Node.js
API	RESTful JSON
Deployment	AWS EC2

[1]Public repository hosted on Github: `https://github.com/charleschengxu/safeclientcache`

Both the server and client library are implemented using Node.js for its native support for JSON so no explicit marshalling/serializing/pickling/flattening is needed to issue an RPC. The RPC interface is RESTful in JSON, independent of the backend server framework or language, so one could easily integrate with any existing production services with the interface intact. RESTful API runs on HTTP, which is an application layer protocol and ensures exactly-once semantics for peer-to-peer communication. Express is a web server framework on Node.js that simplifies URL routing and API binding. We use AWS EC2 (us-west-2, Oregon, US) as our VPS (virtual private server) and by deploying the key-value store on the remote, we are able to obtain more realistic performance measurements.

6 Evaluation

6.1 Volume Size and Number of Notifications

The prefix of each key is used to group keys into volumes for scalability. The length of the prefix is configurable by the application, and the tradeoffs are focused on the size of the subscription sets maintained on each server and the number of notifications sent by the server given subsequent updates.

The results shown in Figure 6 is obtained using the following test bench. It first spins up the server and then sequentially starts one client at a time. Each client is asked to perform 1,000 operations, 20% of which are writes and the rest are reads. Subsequent writes generate notifications to other clients, and the server counts the total number of such messages. Values stored on the key-value store are 32-bit integers. We used updates instead invalidation for server notifications. The keys used by all clients are hashed using SHA1 for the more uniformed distribution.

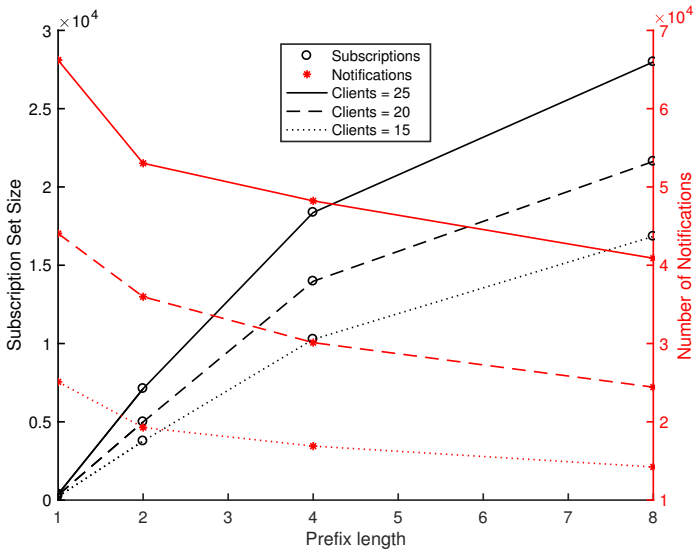


Figure 6: Scalability as Functions of Prefix Length and Number of Clients

Given shorter prefix length, keys are partitioned into larger volumes, so the subscription sets are effectively compressed. At the same time, larger volumes lead to more

imprecise notifications so the total number of messages is larger for shorter prefixes. In this particular case, when the prefix length shrinks from 8 to 2, the total number of messages sent rises by about only 30% while the size of subscription sets drops by about 80%. Depending on the number of clients, the write frequency, and network bandwidth, different applications may set the prefix length to suit their needs.

6.2 Recovery Speed Compared to Polling

We compare the performance of our pubsub design to that of traditional polling, results shown in Figure 7.

We use prefix length of 3 for volume keys and assume 20% of the client cache is stale before recovery. The polling procedure that we implement is to send the server keys that the client knows and the server replies with values for those keys, and finally the client updates corresponding keys in cache if its value is stale.

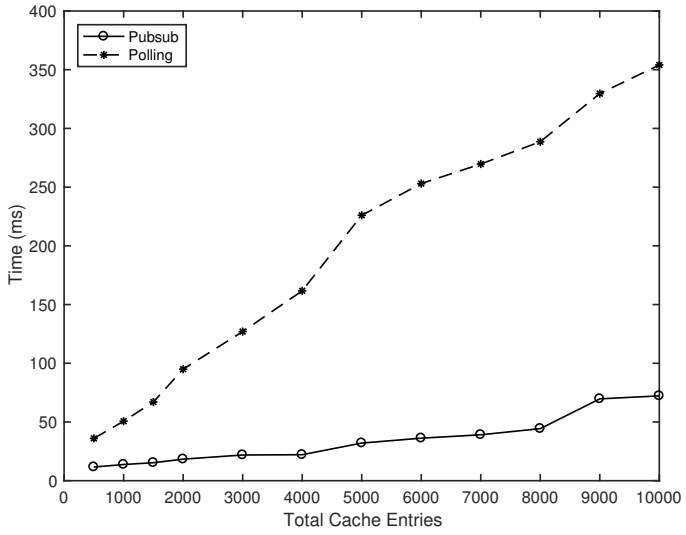


Figure 7: The Time It Takes to Recover a Stale Client Cache

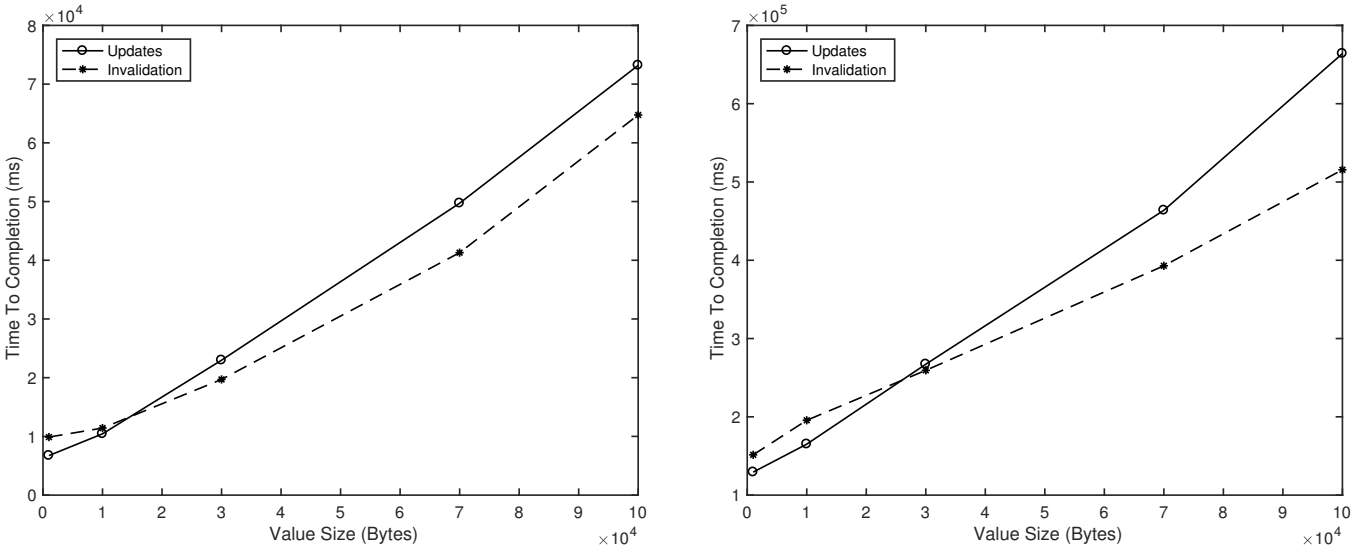
Apparently, the pubsub design performs better. It is worth pointing out the step-like latency in both cache designs. The server maintains an in-memory hashmap as a mock of the key-value store. The jumps result from the multi-layer physical cache on the machine, since lower cache layers have larger sizes and higher latencies.

6.3 Invalidation v. Update

The tradeoffs of implementing server notifications as either invalidations or updates are discussed in §3.5, and are explored further here. The simulation provides us a peek into how value sizes and network latency affect the performance of this cache design.

We assume values are of equal size and use prefix length of 10 for volume keys. Five clients are created at once doing random read/write on the same 1,000 keys to populate their cache. Then, they repeat the same procedure to access the same set of

keys again, and we time the second run to see how the cache may help, given different value size and propagation delay, as results shown in Figure 8.



(a) Server and Clients Connected to the Same Router

(b) Server in Oregon and Clients in North Carolina

Figure 8: Time to Completion given Various Value Sizes

In either case, we see the two lines crossing. Updates outperform invalidation when the value size is small, and the reverse is true when value size is large. This observation is what we expect. The reason is that when the value size is large, propagation delay starts to dominate transmission delay and sending large blobs actually slows the system down, especially when the updated values have a low chance of being read again.

7 Conclusion

In this paper we have described a cache design for the distributed key-value stores that are sharded and replicated. Consistency, scalability, and causality issues are addressed with specific design proposals. We have implemented such caching scheme and deployed on AWS EC2 to evaluate its performance and the tradeoffs of various configurations, which demonstrates its flexibility to support and optimize for different access patterns. Future work includes supporting distributed transactions on top of the client cache.

References

[1] D. Karger, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. May 1997.

[2] G. DeCandia, et al. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*. 41(6):205–220. October 2007.

[3] B. Oki, et al. Viewstamped replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. *ACM Symposium on Principles of distributed computing*. 8–17. August 1988.

[4] L. Lamport. Paxos made simple. *ACM SIGACT News*. 32(4):18–25. December 2001.

[5] D. Ongaro, et al. In search of an understandable consensus algorithm. *USENIX Annual Technical Conference*. June 2014.

[6] A. Adya, et al. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD*. 23-34. May 1995.

[7] R. C. Steinke, et al. A unified theory of shared memory consistency. *Journal of the ACM*. 51(5): 800849. September 2004.

[8] P Hunt, et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. *InUSENIX*. (8)11. June 2010.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *Operating Systems Design and Implementation*. November 2006.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*. 21(7):558–565. July 1978.

[11] Y. Jian, et al. Engineering Web Cache Consistency. *ACM Transactions on Internet Technology*, 2(3):224–259, August 2002.

[12] D. B. Terry, et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings of the fifteenth ACM symposium on Operating systems principles*. 172–182. December 1995.

[13] Q. Cao, et al. Certificate Linking and Caching for Logical Trust. *n. p.* November 2016. `arxiv.org/ftp/arxiv/papers/1701/1701.06562.pdf`.

[14] R. L. Rivest, et al. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*. 21(2):120–126. February 1978.