Dacian: Variable Consistency and Bounded Staleness in Synchronously Replicated Systems

Ryan Johnson
University of Michigan
ryantj@umich.edu

Victor Hao
University of Michigan
vhao@umich.edu

Abstract

Distributed storage systems that support variable consistency have become the norm to fit the varying needs of a diverse set of applications. Existing solutions handle variable consistency by directing relaxed consistency requests to replicas that are replicated asynchronously or by routing requests with slightly higher consistency requirements through a primary. Dacian is a replicated key-value store which aims to provide better performance by instead performing replication synchronously and supplying clients with more recent information with which to direct variable consistency requests. In this paper, we describe the API and design of Dacian and present an evaluation of its performance on variable consistency workloads with regards to both request latency and throughput.

1 Introduction

Distributed storage systems form the foundation for many modern applications from e-commerce sites to social networking and beyond. As a result, the data for these applications is spread across multiple machines and replicated to ensure maximum availability and reliability. Depending on the scale of the application, these machines may be located in the same data center or across the globe in multiple data centers. As the scale of the storage solution increases, end-to-end latency of a client request can become adversely affected by the communications between storage nodes as required by the consistency protocol. Thus, it becomes important to offer different levels of consistency to a client when latency is more important.

Read performance in such scenarios depends heavily on the level of consistency required. For example, a strongly consistent read typically must be served by a primary replica, or must go to multiple other replicas usually comprising a quorum. This can be especially expensive if the primary is not co-located in the same data center as the client making the request. In contrast, a read requiring only eventual consistency might read from any single peer containing the requested data, drastically reducing the end-to-end latency of the request and the load on the system as a whole. Of course, a range of other consistency guarantees exist between strong and eventual, the specifics of which we cover later on in Section 3.1. In fact, many real world applications are able to take advantage of variable consistency transactions so long as they are made available to them.

A common example of an application which could take advantage of variable consistency is a shopping cart service on an e-commerce website. A shopping cart is able to tolerate occasional inconsistency during use, as it is not critical that every added or removed item shows up in the cart. At checkout time, the user can verify the contents and make any changes necessary before finalizing the purchase. Eventual consistency may suffice for this case, but it is not hard to imagine that a customer may become frustrated if the cart does not reflect their recent actions. Instead, it might be worthwhile for the shopping cart application to make use of read-my-write consistency where the most recent modification to the cart by the user is guaranteed to be reflected in all future reads. The application might decide to use a synthesis of these consistency levels in its requests depending on network conditions or other factors known to the client.

Another example application which may benefit from variable consistency is a search engine. A search engine typically prefers to return as up to date information as possible to the user, but may be able to get away with requesting older data for uncommon searches where the results are unlikely to have changed. Depending on internal information on search popularity, the engine might want to set some bound on how old the data returned from the storage system can be. In this case, bounded staleness requests would allow the application to set a time limit on the staleness of returned results.

For searches relating to breaking news, the time limit could be set within the realm of seconds (or the search engine may simply choose strong consistency) while for searches where the results are expected to remain relatively stable the bound might be set around tens of minutes or more.

It is clear that variable consistency offerings are important for modern distributed storage systems, and developments relating to this area have strong implications for future performance gains across a diverse set of applications. To this end, our paper makes the following contributions:

- We present a protocol that enables various consistency levels in a distributed, synchronously replicated key-value store.
- We develop an implementation of this protocol, called Dacian, based on Raft and describe how clients can use it to obtain their desired consistency level.
- We assess the performance of the system, considering the impact on both latency and throughput.

The next section discusses related work and some other existing distributed storage solutions. Section 3 outlines the overarching design of Dacian and the rationale for the choices that were made. Section 4 covers the key implementation details of Dacian that give it its more desirable properties. Section 5 presents an evaluation of the performance of Dacian under workloads of varying consistency levels and Section 6 concludes our findings.

2 Related work

The inspiration for Dacian comes at least in part from many preexisting distributed storage systems and the literature regarding them. At this point in time, there most likely exists some distributed storage system which provides almost any conceivable combination of features while also providing the ability for applications to make trade offs in consistency, availability, and performance in any way that best suits their needs. Just some examples of these systems include Amazon's Dynamo [4], Google's Spanner [3] and BigTable [2], Microsoft's Azure Cosmos, and Apache's Cassandra [5]. Dacian borrows some of its design from these predecessors, including its variable consistency model and its *get,put* interface. Another closely related system is Microsoft's Pileus [7], which is also similar to other variable consistency systems but mainly improves on the ways in which applications may specify their consistency requirements in a service level agreement.

Our approach differs from existing variable consistency systems in that prior work has mostly focused on improving request latencies while our designs also target improvements in overall system throughput. Additionally, existing systems typically replicate their data to other data sites asynchronously from the master over some period of time, usually on the order of minutes. Client requests which have relaxed consistency are usually directed towards these asynchronous sites to be served. To our knowledge, a system which takes advantage of synchronous replication sites to respond to variable consistency requests has not been published before. Finally, existing systems tend to apply variable consistency in the context of improving multiple-datacenter deployments, but do not concern themselves with applying the same concepts within a single datacenter. Our designs work towards improving latency and throughput for both inter and intra-datacenter requests.

3 Design

Dacian is a distributed, synchronously replicated key-value store that offers clients the ability to perform *get* requests with variable levels of consistency.

3.1 Client API

The exposed API is similar to a traditional key-value store, but with a few modifications. The API is as follows:

put(key, value) $\rightarrow$ *redirect, version, peers*
get(key, version, bounded) $\rightarrow$ *redirect, value, version, peers*

put requests follow the traditional format, but provide additional information in their responses. The *redirect* field in a *put* response indicates that the operation did not succeed, but another server may be able to handle the request. The *version* field is the version of the key that was written by this *put* request. The *peers* field is a list of servers that have received this version of the key, or newer.

get requests include two new fields, compared to the traditional format. *version* is the minimum acceptable version of the key. The client decides on this version ahead of time, and includes the version in its request. *bounded* is a flag which designates that the desired *version* is not necessarily a previous version of the key, but rather an arbitrary point in time. By setting the *bounded* flag, the client is asking for a version of the key that existed after the specified *version* in the *get* request.

get responses are similar to *put* responses, but include the retrieved value. They similarly include a redirect if the initial server was not able to handle the request, as well as the version of the key that was retrieved and any

peers that are known to have this version of the key, or newer.

We will now describe how the client can use this interface to choose different consistency levels for *get* requests.

3.2 Consistency Levels

Dacian supports the six consistency levels, similar to those outlined in Pileus [7], when performing *get* requests: linearizable, causal, monotonic, read-your-writes, eventual, and bounded staleness. These consistency levels can be selected based on the *version* field of the request. Some consistency levels consider the actions of multiple clients, while others only consider the actions of the current client.

3.2.1 Linearziable

Linearizable *get* requests always return the newest version of the key that was written by a previous *put* request from any client. In this scenario, the *version* is unused, as the client does not know the latest version ahead of time.

3.2.2 Causal

Causal *get* requests return a version of the key that is at least as new as any *put*s to the same key that the client has observed. These causally preceding *put*s may or may not have been performed by other clients, and can be observed directly by making the *put* request or indirectly by performing a *get* request.

3.2.3 Monotonic

Monotonic *get* requests return a version of the key that is at least as new as any previous *get*s to the same key. This ensures that the client will always see the same or newer values when reading keys.

3.2.4 Read-your-writes

Read-your-writes *get* requests return a version of the key that is at least as new as any previous *put*s to the same key. This ensures that the client will always see the affects of its previous *put*s.

3.2.5 Eventual

Eventual *get* requests offer few guarantees on the returned version of the key. Over time if there are no new *put*s to the key, then *get* requests will eventually return the latest version; however, there are no guarantees on how long this will take.

3.2.6 Bounded

Bounded *get* requests offer a loose bound on how stale the returned version of a key can be. If there have been no *put*s to the key in the last t seconds, then the latest version of the key is returned, otherwise any version written in the last t seconds can be returned.

3.3 Replication

Unlike *get*s, *put*s are always strongly-consistent and are performed at a quorum of replicas. This ensures that *put*s will always be seen by *get* requests that either read from an up-to-date replica or read from a quorum of replicas.

In contrast to existing variable-consistency systems, such as Pileus [7], that replicate operations asynchronously on the order of minutes, Dacian replicates operations synchronously. This provides two advantages: the first is that replicas are kept up-to-date as much as possible so that these replicas have a better chance of being able to serve a *get* request with a specified version. The second is that by replicating synchronously, the system has more information available when responding to the client's *put* request. This allows the client to immediately know which replicas have received the *put* request. This is conveyed in the *peers* field in the *put* response.

While peers included in a *put* response have only received the operation and are not guaranteed to have applied it, as we will show in Section 5, we observed that this was not an issue in practice.

4 Implementation

Dacian is a distributed key-value store built on top of brpc [8] and braft [1]. It provides the previously described *get* and *put* API to clients, and clients can use the API to issue *get* requests with their desired consistency level. We will first briefly discuss Raft, the underlying replication protocol.

4.1 Raft

Raft is a consensus algorithm that relies on Paxos to elect a single leader [6]. This leader is the authority for a given term, and replicates log entries to followers using an *AppendEntry* RPC. Followers are replicas that are not the leader for the current term, and they will accept any *AppendEntry* RPCs from a peer they believe to be the current leader.

Traditionally, only the leader can serve read and write requests in a Raft cluster. When the leader receives an operation that it needs to replicate to its peers, it first appends the entry to its own log, then it sends *AppendEntry* RPCs containing the operation it is trying to replicate to

its peers. When the leader receives successful responses from a quorum of replicas, it commits the operation to its own log and sends commit messages to the followers. The leader asynchronously sends commit messages to its followers as it responds to the request.

Raft monitors the health of the leader by sending heartbeats from the leader to followers, typically every 150-300ms. If a follower does not receive a heartbeat after a timeout, it starts a leader election using Paxos. We will now describe how we build off of Raft to provide the key-value interface we describe.

4.2 Put Requests

As mentioned previously, *put* requests follow the traditional format, yet include extra information in the response. Additionally, *put* requests are strongly-consistent. In the case of Raft, this means that *put* requests are always initiated at the leader; they cannot be initiated at followers. The leader performs the following sequence of steps when it receives a *put* request.

1. ***Check leadership status:*** if the replica that receives the *put* request does not believe it is the current leader, it will redirect the client to the replica it believes is the leader.
2. ***Assign a version:*** the leader uses its local clock to get the current time in milliseconds since the epoch. The leader then uses this as the version for the *put* request.
3. ***Append to the leader's log:*** the leader assigns the *put* request the earliest empty slot in its log.
4. ***Replicate log entry to followers:*** the leader then sends an *AppendEntry* RPC to the other replicas, which includes the key, value, version, and log location. The leader then waits for a quorum of successful responses.
5. ***Commit and send reponse:*** the leader then commits the log entry locally, and sends commit messages asynchronously to its followers. When the leader responds to the *put* request, it will include any peers who participated in the quorum in the *peers* field of the response.

4.3 Get Requests

get requests differ from the traditional format in that they include a *version* field that specifies the minimum desired version of the requested key. *get* requests can also contain a *bounded* flag which indicates that the *version* specified is a timestamp in the past and returned versions of the key should not be staler than that timestamp. Because potentially older versions of a key are acceptable

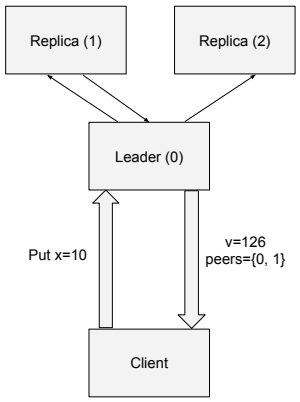

Figure 1: Forwarding of quorum information in a *put* response. Peer identifiers are shown in parentheses. Only replica 1 responds in time to the *AppendEntries* request to be included in the *put* response.

depending on the supplied *version*, any replica can attempt to service a *get* request, unlike in a traditional Raft implementation. To do this, replicas not only store a key-value mapping, but also the current version for each key. Replicas perform the following sequence of steps when they receive a *get* request:

1. ***Check for linearizability:*** if the *get* request desires linearizability, i.e. the *version* is -1, and the replica does not believe it is the current leader, it will redirect the client to the replica it believes is the leader.
2. ***Check for bounded staleness:*** if the *get* request specifies the *bounded* flag, the replica will check to see if the last time it synced with the primary is greater than or equal to *version*. If that is not the case, the replica will similarly redirect the client to the replica it believes is the leader.
3. ***Check version requirements:*** if the request specifies a minimum acceptable *version*, the replica will check the version it has for that key is greater than or equal to the desired *version*. If that is not the case, the replica will again redirect the client to the leader.
4. ***Send response:*** if the previous checks pass, the replica is sufficiently up-to-date to satisfy the client's request. It will send back to the client the *value* and *version* of the key it has, and it will also add itself to the *peers* list, like in *put* responses.

In checking for bounded staleness, the replica checks the last time it synced with the primary. To track this information, we have added a timestamp to Raft's heartbeat messages which is set to the local time of the leader. When followers receive a heartbeat from the leader, they

update their latest sync time. If the Raft implementation uses a TCP connection from the leader to each follower, then after receiving the heartbeat and processing previous *AppendEntries* RPCs, the replica is known to be synced with the leader at the given timestamp.

Additionally, because Raft sends a heartbeat every 150-300ms, bounded staleness requests can be specified at the granularity of 0.5 seconds, barring any issues due to clock drift. We will now discuss how the client picks the *version* that it sends in its *get* requests and what state it stores.

4.4 Client Sessions

When clients submit *get* requests, they must specify a minimum desired *version*, unless they want strong consistency. To do this, clients must determine the appropriate *version* for the corresponding key and consistency level. This requires each client maintain some state, which we will call the client's session. A session stores the following information:

- A set of replicas
- The current leader of the replicas
- A mapping from key to the last version received from a *get* request to that key, which we will denote *vget*(*key*)
- A mapping from key to the last version received from a *put* request to that key, which we will denote *vput*(*key*)
- A mapping from key to a set of replicas and their last known version for that key, which we will denote *vpeer*(*key, peer*)

It is expected that, within a session, a client has only one outstanding request at a time, and clients use the information stored in their session to achieve one of the six supported consistency levels. Figure 2. shows how to calculate *version* using the client's session and the desired consistency level.

Consistency Level	Version
Linearizable	-1
Causal	*max*(*vget*(*key*), *vput*(*key*))
Monotonic	*vget*(*key*)
Read-your-writes	*vput*(*key*)
Eventual	0
Bounded	*now*() - *t*

Figure 2: Consistency levels and their corresponding *version* requirements.

When a client receives a *get* or *put* response, it will update the respective *vget*(*key*) or *vput*(*key*) with the returned *version*. The client will also update any *vpeer*(*key, peer*) for the peers in the *peers* field of the response with the returned *version*. These updates are monotonically increasing, so the *version* for a given key stored in the client's session will never decrease.

5 Evaluation

5.1 Latency and Throughput

For our first experiment, we used three servers with 15,000 concurrent clients making requests to the servers. The clients varied the percentage of *get* requests that requested eventual consistency, as opposed to linearizability. We found that the throughput increased and latency decreased as the percentage of eventually consistent *get*s increased. As shown in Figure 3, at the extremes, the eventually consistent *get*s had over twice the throughput of the linearizable requests, and latency similarly halved.

Figure 3: Impact of eventual consistency on latency (blue) and throughput (red).

This result is expected as replicas other than the leader are able to serve *get* requests, which improves throughput.

5.2 Redirected Gets

As mentioned previously, *put* responses contain speculative information about which replicas have a specific version of a key. It is possible for a client to make a *put* request, receive a response indicating that a specific replica has the written version, and then send a *get* request to that replica for the given version before the replica has a chance to commit the result to its log.

To test how often this occurs, we designed an experiment where 4,800 clients made *get* and *put* requests concurrently. Each client had an equal chance to send a *put* or a read-your-writes *get* on the previously *put* version of a key.

Out of 4,231,097 *get*s, only 31 were redirected. Due to the low probability of this occurrence, we have con-

cluded that sending the speculative *peer* information in *put* responses is performant.

5.3 Mock Social Media Workload

In the next experiment, we set up a mock social media workload using our key-value store. We used the same three replica configuration and compared a system that performs only linearizable *get*s with a system that performs variable consistent *get*s as outlined in Figure 4. We estimated what consistency different functionality of a social media website would require, and approximated how many *get* requests would require each form of consistency.

Consistency Level	Percentage	Use-case
Linearizable	10%	Permissions
Causal	20%	Comments
Monotonic	20%	Top posts
Read-your-writes	10%	My posts
Eventual	10%	Like-count
Bounded	30%	Recent posts

Figure 4: Example consistency levels for a mock social media site. Percentages are synthetic.

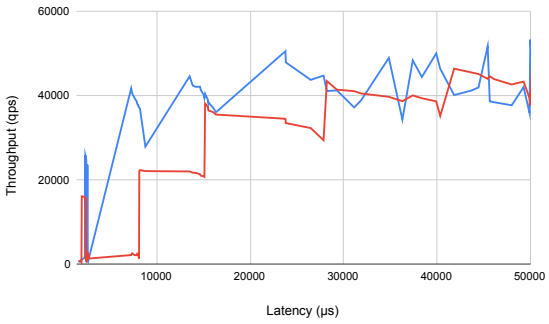

Figure 5: Latency vs. throughput for a mock social media website workload using both strongly consistent *get*s (red) and variable consistency *get*s (blue).

From Figure 5, we can see that the variable consistent system performs better in terms of latency and throughput when compared to the strongly consistent system.

6 Conclusion

To summarize, Dacian incorporates and expands on ideas from preexisting distributed storage systems to implement variable consistency requests with better performance. Instead of simply directing relaxed consistency requests to asynchronous replicas and relying on the master for slightly stronger consistencies, Dacian takes advantage of quorum information to allow clients to direct such requests to synchronous replicas, avoiding the master in most cases. We have shown that this approach is performant, as in our tests throughput increased and latency decreased when the fraction of eventual consistency requests increased. Additionally, separate tests showed that only a small fraction of read-my-write requests were redirected to the master when contention on a given key by other clients was low, indicating that synchronously returning information on the peers containing written data to the client improves performance.

7 Division of Work

When working on the design and implementation of the system, we pair-programmed and worked synchronously. We worked asynchronously on the paper, and reviewed it synchronously. Overall, we split the work evenly between the two of us.

References

[1] BAIDU. braft. https://github.com/baidu/braft, 2019.

[2] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 205–218.

[3] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., WOODFORD, D., SAITO, Y., TAYLOR, C., SZYMANIAK, M., AND WANG, R. Spanner: Google's globally-distributed database. In *OSDI* (2012).

[4] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, Association for Computing Machinery, p. 205–220.

[5] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[6] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[7] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based service level agreements for cloud storage. SOSP '13, Association for Computing Machinery, p. 309–324.

[8] THE APACHE SOFTWARE FOUNDATION. better rpc. https://github.com/apache/incubator-brpc, 2018.