Append-only Datastore

Mingyan Zhao
myzh@stanford.edu

Steven Tung
steven.tung@stanford.edu

Kevin Krakauer
kevinkrakauer@gmail.com

Abstract

We built an eventually consistent, append-only datastore focused on low read/write latency and high availability. In the common case, clients interact only with nearby nodes for extremely low latency. Data is always appended to the store for a given key and the relative order of observed data is maintained.

The datastore is eventually consistent. Data is stored in memory for low latency and in local disk for safety. Nodes can operate when disconnected from the system, preserving availability in the face of total and long-term partitioning. We believe this system will be useful in chat, social media, and distributed logging applications.

1 Introduction

As organizations increasingly move to the cloud, applications are designed from the ground up with a distributed architecture (*microservices*). Despite numerous advantages, distributed application design requires addressing latency and partitioning concerns that monolithic applications do not have.

Our *append-only datastore* addresses latency and partitioning concerns for append-only workloads. It is designed to run as a service distributed globally across multiple data centers. By explicitly distinguishing between a *leader* node and *follower* nodes, we gain several advantages over other eventually consistent systems:

1. Clients communicate only with their nearest follower for extremely low latency.
2. Eventual consistency is minimally disruptive to clients. A client may read the data for key k and receive data consisting of 2 writes' data: $\{d_1, d_2\}$. Write x with data d_x sent through another follower preserves the ordering of d_1 and d_2, so the system eventually returns $\{d_x, d_1, d_2\}$, $\{d_1, d_x, d_2\}$, or $\{d_1, d_2, d_x\}$ when read.

We also gain the advantages of some more traditionally eventually consistent systems, such as great partition tolerance [1]. Together, these properties are highly desirable for many append-only applications. Consider the following:

- **Distributed logging** - Consider a monitoring service using the append-only datastore for logs in real time. The service writes and retrieves logs with extremely low latency from the nearby data center. Also it is provided an eventually consistent log that contains data from all of the deployments.

- **Chat Application** - Consider a chatroom occupied by a team in North America and a team in Asia. Each team member sees their local peers' messages immediately, and all messages eventually.

2 Related Work

Amazon's Dynamo [1] supports always-writable semantics, high partition tolerance, and laser-focuses on low-latency operation. Unlike our system, Dynamo is a key-value store. It also exposes a great deal of complexity to developers, who must tailor their use of Dynamo such that conflicts are resolvable and must manually implement conflict resolution in clients. Our system does not allow for conflicts. Google's GFS [2] is also optimized for append-heavy workloads and uses a single master for simplicity and intelligent coordination. It supports random writes as well. However, it is explicitly optimized for non-latency-sensitive applications, and a single read can require multiple hops (the GFS master and chunkserver). LinkedIn's Kafka [3] provides eventual delivery of large quantities of data, but running under a publish/subscribe mechanism.

3 Design

Nodes in the append-only datastore are classified in a simple hierarchy as seen in Figure 1. A single *leader* directs multiple *followers*. Clients connect to and communicate with a nearby follower, likely running in the same data center, to minimize latency. Our design tolerates arbitrary partitions between nodes and ensures an eventually consistent view of data. Clients are exposed to the following API by followers:

- `append(key, data)` - Appends `data` to the existing data for `key`. Guarantees that `data` will be committed eventually if it is written to local disk.
- `get(key)` - Gets the data for `key`, represented as a list of values. Later calls to `get` return data in the same relative order, but other data may be interleaved or appended.

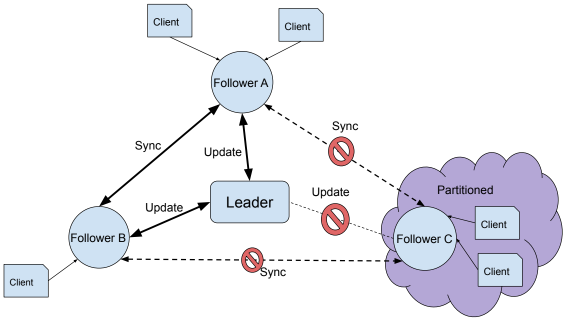


Figure 1: Architecture

3.1 Leader

The leader is responsible for globally ordering writes for each key. Like GFS's master [2], it only stores metadata rather than the value itself. This decreases network usage. Also, the burden on disks is lighter per update, increasing writes per second and the average lifetime of each disk. The simplicity of a single master simplifies our implementation and the rest of our design, which increases the overall stability of the system.

3.1.1 Index number

The leader maintains the latest index number for all keys. Index numbers are globally unique and monotonically increasing. Once generated, each is mapped to a list of values on the requesting follower. If another follower needs that value, it simply needs the index number to fetch the data.

3.1.2 Update

Upon receiving an `Update` request, the leader advances the index number by one and updates the follower information to the incoming one.

The `Leader` decides whether the requesting follower needs to sync up with other followers. If a mapping exists from key k to index i, an `Update` carrying the same index i does not need a Sync (i still increments by 1). This may happen when 1) a key created on the leader for the first time, or 2) the requesting follower is the same as the recorded follower for the key.

An `Update` to $i' < i$ indicates that the sender does not have the latest data. In this case the leader signals the follower to sync with the follower in its records that originated the append at index i. This may happen when two followers are append data to the same key concurrently. The common, non-concurrent case is shown in Figure 2.

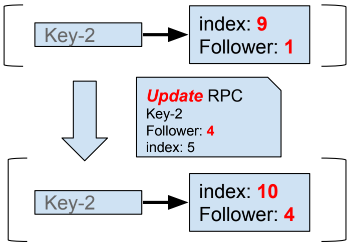


Figure 2: Update procedure

3.1.3 Broadcasting index numbers

The leader broadcasts ensure the eventual consistency of the system. Each appended value gets replicated and synced immediately between the requesting followers and in-record follower, but all of the other followers do not know it until the leader broadcasts the latest index number.

Depending on the workload and use cases, we provide different ways of broadcasting. One is triggered per-request, which is expected to provide lower consistency latency but may only be used in a lower QPS scenario, since the broadcasting may cause a great amount of communication between the followers. The other one is triggered periodically, which is expected to be used in most cases. The leader broadcasts the updated keys and their index numbers repeatedly at a small, set interval.

3.1.4 Leader Fault Tolerance

Because the leader writes all updates to disk, it tolerates crashes and reboots. For greater fault tolerance, it should write to multiple disks or a remote disk as well.

The system continues to function when the leader is partitioned, but followers and thus clients do not receive updates from other followers. As long as the partition is eventually healed, the system will propagate information correctly and self-heal. If for some reason there is a permanent partition, it must be manually worked around by changing the cluster configuration (i.e. the set of nodes in the system).

3.2 Follower

The follower is our most critical component. Its responsibilities include:

- Handing `append` and `get` requests.
- Managing data.
- Updating the leader about appended data.
- Syncing with other followers to communicate updates and orderings.

3.2.1 Data structure

Followers replicate the entire datastore in memory to minimize reading and writing latency. The data structure is described in Figure 3.

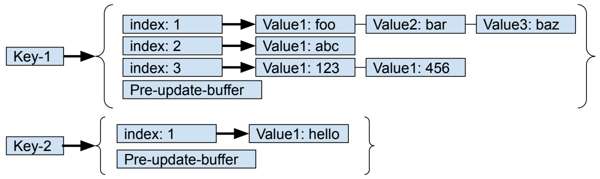


Figure 3: Data procedure on Follower

3.2.2 Client Interfaces

Append(key, data): Upon reviving an `Append` request, the follower writes the data to memory first and then writes to a local disk for data safety. It returns immediately after the data is written to local disk. For increased fault tolerance, the append should be written to multiple disks locally or to a remote disk. Note that throughout this section when we refer to updating local storage, the in-memory store is also updated immediately after data is written to disk.

The follower always appends data to the pre-update-buffer first, then issues an `Update` if the pre-update-buffer is not empty. If it receives an `Update` response with a index number, it creates a new index entry and moves all of the data from pre-update-buffer to the new entry.

Get(key): When clients issue a `Get` request, they are served from the in-memory datastore. We guarantees that the data read by the clients are correct, but not necessarily yet globally consistent. The follower builds up a list with all of the data from each index as well as everything from the pre-update-buffer.

3.2.3 Synchronization

Followers use the `Sync` request to get data they missed. This ensures the eventual consistency of the system. `Sync` requests also signal what data the sender has to the recipient, enabling the recipient to create new index entries in its local datastore. The `Sync` request is issued when a follower receives an `Update` response from the leader indicating it needs synchronization.

Since a follower may missing multiple indices of data, a `Sync` request may carry a list of indices that it is asking for. However the recipient may also be missing some of the indices if there is a large amount of `Sync` requests in flight simultaneously. To mitigate this, the `Sync` response only replies with indices and data the recipient already has, so that the issuer can accept what is returned and do another `Sync` asking for what is left. Eventually, each follower synchronizes with and achieves global consistency.

Combined with the `Update` requests, Figure 4 demonstrates a message flow of synchronization in common case.

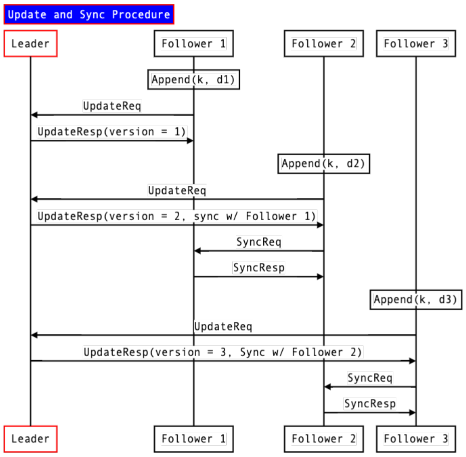


Figure 4: Update and Sync message flow

3.2.4 Follower Fault Tolerance

Both `Update` and `Sync` requests are retried in case of network failure. If a follower reboots during a `update`, it can inspect the disk at boot and see that it has appended data locally that has not been confirmed by the leader. If a follower reboots during a `sync`, it will have stale data until a `get` or `append` causes it to update. This keeps the system simple while retaining its consistency properties.

4 Evaluation

The system is written in Go and can be run manually or distributed and run with Docker. Nodes communicate via gRPC.

We ran tests with a cluster of Google Compute Engine virtual machines. Google Cloud platform provides internal IP communication between VM instances, which means request between servers do not go through the public Internet and are thus fast. We deployed our system in the following locations.

- One leader in Iowa, Central USA.
- One follower and one client in each of the following five locations
  - Iowa, central USA.
  - Los Angeles, west USA.
  - South Carolina, east USA.
  - London, west Europe.
  - Tokyo, northeast Asia.

In the performance test, each client in the performance test sends 120,000 `Append` against 10 keys randomly, so that in total there are 600,000 values in the datastore globally. After all of the `Appends` are done, each client then retrieves the data by issuing `Get` for all of the 10 keys every 0.5 second periodically.

4.1 Append Request Performance

We measures the `Append` request latency in Table 1. The result shows that `Append` is extremely fast, about 7.5 milliseconds on average. The 600,000 requests can be done in a little less than 1 second due to appending only involving communication with a single node.

Location	S. C.	Tokyo	Iowa	London	L.A.
1 Req	8.06	7.91	8.23	7.82	7.94
600k Req	967.62	948.72	988.00	938.75	952.92

Table 1: Append request latency in millisecond (S.C. for South Carolina, L.A. for Los Angeles)

4.2 Eventual consistency

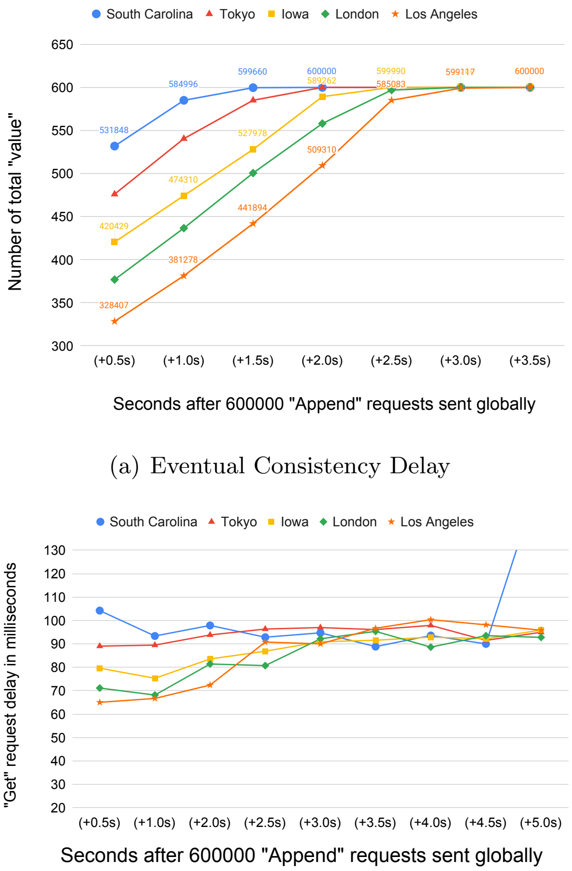
One critical result is the eventual consistency, since this is what we trade off for the low latency read and write. The result is shown in Figure 5. After 0.5 seconds, all of the followers have already begun syncing with each other and received some of the data from each other. The start point is different since the clients were started sequentially. After 2.0 seconds, the follower deployed in South Carolina finished synchronization and received all of the 600,000 values. After 3.5 seconds, all of the follower held all of the data and the system reached eventual consistency.

4.3 Get Request Performance

The result of the `Get` is shown in Figure 5. Latency starts differently because the Follower held different amount of data. After 3.5 seconds, the latency of different Followers converged to between 90 milliseconds and 100 milliseconds. Note that the implementation of `Get` could be optimized to achieve better performance.

5 Future Work

We could greatly increase throughput by providing a client library (rather than a raw GRPC interface) that is aware of the indices or latest index it holds. This would enable followers to



(a) Eventual Consistency Delay

(b) Get Request Latency.

Figure 5: Performance Test Result

selectively return only the few pieces of data a client is missing rather than all the data.

To increase fault tolerance, we have discussed making each follower and the leader their own small cluster of consensus nodes. This would greatly increase fault tolerance and, while it may complicate the system, would not complicate clients or the protocols via which nodes communicate.

Lastly, there may be cases where followers fail or are partitioned from their nearby clients. In these cases, we would like to explore whether clients can fall back to another follower.

6 Conclusions

We built a datastore to provide low latency and high availability for append-only data storage. While this work is tailored to a specific set of workloads, there are numerous applications that can benefit from this approach.

Our system can be used to support distributed services such as distributed monitoring and chat applications with high performance. Importantly, it exposes a simple interface allowing for simple clients. We believe that this makes it a useful tool in building distributed and microservice systems.

References

[1] DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels, “Dynamo: Amazon’s highly available key-value store,”

[2] Ghemawat, Gobioff, and Leung, “The google file system,”

[3] Kreps, Narkhede, and Rao, “Kafka: a distributed messaging system for log processing,”