Append-only Datastore

Mingyan Zhao Steven Tung Kevin Krakauer

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. We do not provide overwrite and delete operations. This simple interface guarantees properties making client implementation simple: the relative order of data for a given key is consistent and each client communicates only with a single node.

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 and distributed logging applications.

1 Introduction

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

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. Followers provide linearizability for client appends on a single key. That is, all clients writing to the same key of the same follower will see linearized writes.

3. 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. Because of (2), writes directly to the same follower are linearizable.

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:

- **Chat** - Chat applications are inherently append-only. Users in different parts of the globe expect near-instantaneous (only a few

seconds) latency. Users in the same building, however, expect to receive messages instantaneously. Think of a global company: teammates send messages and links to nearby teammates, expecting them to arrive immediately. Consider a chatroom occupied by a team in North American and a team in Asia. Each team member sees their local peers' messages immediately and in exactly the order that they are sent. It's acceptable for the a message from the other team to take a bit longer and appear slightly out of order in the logged chat.

- **Distributed logging** - Consider a distributed application running in multiple data centers on multiple continents. The application is monitored in real time in each datacenter. Logs are write-only, and critical for monitoring. By using the append-only datastore for logging, the monitoring service:
  - Continues operating when a datacenter is partitioned from the rest of the system.
  - Receives logs with extremely low latency from the nearby monitored application.
  - Has an eventually consistent log that can be stored for later analysis and debugging.

2 Design

Nodes in the append-only datastore are classified in a simple hierarchy as seen in TODO(add the architecture diagram). A single *leader* directs multiple *followers*. Clients connect to and communicate with a nearby follower, likely running in the same datacenter, to minimize latency. Our design tolerates arbitrary partitions between nodes and ensures an eventually consistent view of data.

Clients are exposed the following API by followers:

- `append(key, data)` - Appends `data` to the existing data for `key`. Guarantees that `data` appears after any data that the client already read for `key`.
- `get(key)` - Gets the data for `key`, expressed as a list of indexed values. For example, the returned data will be the result of writes 1, 3, and 5, expressed as $\{d_1, d_3, d_5\}$. Subsequent `get` requests will always return d_1, d_3, and d_5 in the same relative order, but other writes may be prepended, appended, or interspersed.

2.1 Follower

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

- Handing `append` and `get` requests from clients.
- Ordering appends.
- Updating the leader about appended data.
- Receiving the eventual, global ordering of data from the leader and imposing it on data.
- Syncing with other followers to communicate updates and orderings.

Each follower stores the entire datastore locally in memory. When clients issue `get(k)`

requests, these are served from the in-memory datastore to minimize response latency. It also causes the follower to ask the leader for the most up-to-date data for `k`, ensuring that frequently accessed data is kept relatively fresh.

`append` requests are also fast, limited by the speed of a local disk write. When a client appends data to key `k`, the follower writes that data - along with its index - to local disk before returning to the client. This ensures that data is not lost upon crashing and rebooting. For increased fault tolerance, the append should be written to multiple disks locally or to a remote disk as well. 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.

As soon as the append is written to disk, the follower (concurrently with replying to the client) sends an `update` request to the leader containing the key being appended to and the index that the follower has assigned the appended data. The leader resolves and orders the append with any other concurrent appends from different followers (see next section), and responds.

There are two types of responses. The first - and likely most common - case is a simple `SUCCESS` response. This indicates that the append completed successfully and the follower has up-to-date data for that key. The second is a `NEEDS_UPDATE` response. This indicates that the data was appended, but another `update` preempted it. In this scenario, the follower changes the locally stored index of the appended data, and sends a `sync` request another follower indicated by the leader's response.

Followers use the `sync` request to get data they are missing, effectively "patching the holes" in their local datastore. This ensures the eventually consistent property of the system. `sync` requests also signal what data the sender has to the recipient, enabling the recipient to also patch holes in its local datastore.

2.1.1 Common Case

When updates do not overlap, the follower handles `append` requests as follows:

1. Write the append to local storage along with the index `i` the follower believes applies to the append.
2. Respond to the client that the request was successful.
3. Concurrently with (2), send an `update` to the leader that an append occurred on key `k` at index `i`.
4. The leader responds with `SUCCESS`.
5. The follower updates local disk to indicate that the append has been committed at index `i` by the leader.

The data for key `k` has simply been appended to, changing from $\{d_a, d_b, d_c, ..., d_{i-1}\}$ to $\{d_a, d_b, d_c, ..., d_{i-1}, d_i\}$, where $a < b < c < i$. Note that there may be data "holes" between a, b, c, and i not yet filled in by either a `get` or `sync`.

2.1.2 Concurrent Updates

When concurrent updates occur:

1. The same as steps 1-3 above.
2. The leader responds with `NEEDS_UPDATE`.
3. The follower updates local disk to indicate that the append has been committed at index `i‘` by the leader.

4. The follower sends a `sync` request to the follower that appended the data at index `i‘-1`, as indicated in the leader's response.

5. The follower receives and stores on local disk the data it was missing, including what the leader deemed to be the "winner" of index `i`.

The data for key `k` in this case is an append and a reordering, changing from $\{d_a, d_b, d_c, ..., d_{i-1}\}$ to $\{d_a, d_b, d_c, ..., d_{i-1}, d_i, d_{i'}\}$, where $a < b < c < i$ and as above may have more holes to fill.

It is possible for a chain of `sync` requests to form when more than 2 followers append a key at the same time. If follower a starts syncing with follower b, which is already waiting on follower c to respond to a `sync`, b will send what is knows to a. a then retries to get missing data, at which point b waits to respond until it has new data to send.

2.1.3 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.

2.2 Leader

The leader is responsible for globally ordering writes for each key. Like GFS's master [2], it only stores metadata rather than the datastore itself. This decreases network utilization, as data itself is never sent. Also, the burden on disks is lighter per update, increasing writes per second and the average lifetime of each disk. Like GFS, the simplicity of a single master simplifies our implementation and the rest of our design, both of which increase the overall stability of the system.

The leader answers `update` requests from followers. These requests contain the key being appended to, the index at which the follower is trying to append, and a nonce to prevent erroneous double-appends. The leader authoritatively orders appends by keeping a map of key to current index. If a mapping exists from key k to index i, an update specifying a write to index $i+1$ increments i and returns `SUCCESS`. The leader writes all map updates to disk before responding.

An `update` to $i' < i$ indicates that the sender is not synced to index i. In this case the leader increments its index and reassigns the append the index of $i+1$. It communicates this back to the follower, and tells it to sync with the follower that originated the append at index i.

Along with the current index, the leader's map also contains the follower that originated the append at that index. This allows followers to be directed at the freshest information whether

2.2.1 Broadcasting Updates

Although the system as described is eventually consistent, it may lead to unacceptably stale data. An updated value might get synced between followers a and b, but follower c would miss the update entirely. If a value is updated and 3 days later follower c syncs via a `get` or `append`, c's client will get 3-day-old data.

To prevent excessive data staleness, the leader is able to broadcast updates to every follower. Depending on the workload, broadcasting every

update may slow down the system and excessively consume bandwidth. Thus users can pass a flag specifying whether they want broadcasts for every update or periodic broadcasts. New broadcast behavior is easy to add, as it requires implementing a single-method Go interface that can make use of the same utility functions we used to write our broadcasters.

2.2.2 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).

2.3 Clients

The append-only datastore presents clients with a simple interface consisting of the `get` and `append` API described above. The consistency model guarantees that clients:

- Can read their own appends immediately.
- Never see data reordered relatively.
- Communicate only with a fast, nearby follower, and no other nodes in the system.
- Can always read and write, even when their follower is partitioned from the rest of the system.

The sacrifices made are:

- Consistency is eventual, not stronger.
- We do not support random writes.
- The system continues to work but may provide stale data in the event of a leader failure.

3 Evaluation

The system is written entirely 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. The cluster consisted of:

- A leader in Seoul, Korea.
- A follower and client in Frankfurt, Germany.
- A follower and client in Los Angeles, USA.
- A follower and client in São Paulo, Brazil.

3.1 Workload 1

We should probably read and write a lot.

3.2 Workload 2

Let's see what happens if we kill a follower.

4 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 have to 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] also provides eventual delivery of large quantities of data (specifically logs), but contains a publish/subscription mechanism.

5 Future Work

We could greatly increase throughput by providing a client library (rather than a raw GRPC interface) that is aware of the indeces or latest index it holds. This would enable followers to selectively return only the few missing 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 tailed 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,"