VJAB: A Distributed Fault-Tolerant Strongly-Consistent Key-Value Store

Austin Byers
The University of Chicago

Victor Jiao
The University of Chicago

Abstract

We took single synod paxos and bullshitted it into a multi-paxos environment until things seemed to work! Hooray!

1 Introduction

One of the simplest data storage paradigms is the *key-value store*, also commonly thought of as a *dictionary*. Data values are stored and retrieved using a *key* that uniquely identifies them. Key-value stores have found success at every scale from on-disk single-site storage [6] to Internet-wide highly-available retail storage systems [2].

There are many potential benefits to a key-value store that is *distributed* across multiple nodes, including more storage, fault-tolerance, replication, and high availability. There is no perfect or one-size-fits-all solution; systems designers must think about what properties they want their distributed key-value store to have.

We start with the observation that single-site key-value stores (e.g. [6]) are simple, consistent, and easy to use. Their main drawback is the possibility of failure - if the data becomes corrupted or the machine fails for any reason, the application is permanently unavailable. Our goal, then, is to use a distributed system to provide the same benefits as a single-site key-value store (in particular, strong consistency) but with an additional level of fault-tolerance.

In this paper, we introduce VJAB, a distributed key-value store with `get` and `set` operations that are guaranteed to be strongly consistent. Consistency is achieved using so-called *Multi-Paxos* [7] [8]. VJAB handles nodes which fail intermittently (fail-recover) or permanently (fail-stop) as well as network partitions. (We do not consider Byzantine failures [9]). VJAB is guaranteed to be available so long as a majority of the nodes are alive and responding on the same network partition (perfect availability is not possible by the CAP theorem [5]). Similarly, while progress is impossible to *guarantee* in an asynchronous consensus protocol [4], VJAB makes this possibility extremely unlikely.

2 Multi-Paxos Algorithm

In this section, we describe the Multi-Paxos consensus algorithm [7] [8] at the core of VJAB. The basic idea behind the Paxos algorithms is to require a quorum (majority) before any decision can be committed. Since any two majorities must have at least one node in common, every majority is guaranteed to contain at least one node who accepted the most recent proposal.

2.1 Single-Synod Paxos

We first provide a brief overview of the single-synod Paxos algorithm, which is used to reach consensus around a single value. Suppose we have a set of *proposers* $P_1, \ldots, P_n$ and *acceptors* $A_1, \ldots, A_n$. Proposers may at any time propose a value. The value can be anything for which the application requires consensus (the value for a given key, the current leader, etc). The algorithm proceeds as follows (taken directly from [8]):

Phase 1 (establishing a proposal):

1. The proposer sends a **`PREPARE`** request to every acceptor along with a proposal number n. n must be higher than *any* previously chosen proposal number (including those chosen by other proposers).
2. If an acceptor receives a `PREPARE` request with number n greater than that of any `PREPARE` request to which it has already responded, it responds with a **`PROMISE`** not to accept any more proposals numbered less than n along with the highest-numbered

proposal (if any) and corresponding value it has already accepted.

Phase 2 (accepting a value):

1. If the proposer receives a PROMISE from a majority of acceptors, then it sends an **ACCEPT** request to every acceptor for a proposal numbered n with a value v , where v is the value of the highest-numbered proposal among the responses. If the responses reported no prior values, the proposer may choose any value for v.

2. If an acceptor receives an ACCEPT request for a proposal numbered n, the proposal is **ACCEPTED** unless it has already responded to a PREPARE request having a number greater than n.

2.2 The Distinguished Proposer

The aforementioned single-synod Paxos algorithm is used to reach agreement for a single proposal. However, in a strongly consistent key-value store, we essentially need to reach agreement for every get and set operation. While we could run Single-Synod Paxos for each such operation, this creates considerable message overhead.

Instead, we note that Phase 1 is only necessary if there are multiple proposers. A single active proposer need not extract PROMISEs, since there is no risk of being overridden by other proposals. Thus, a simple optimization is to first elect a *distinguished proposer*, or *leader*. Once a leader has been established, all get and set operations can be routed to them and they can jump straight to Phase 2 for each operation.

Leader election is just another example of the consensus problem - a perfect application of single-synod Paxos! Specifically, if a node receives a get or set request and does not know who the current leader is, they initiate a new proposal of the form "I propose to be the distinguished proposer until time T."

This proposal, if accepted, grants the newly distinguished proposer a *leader lease*. Until the lease expires, no other node is allowed to issue any proposals. This means that, in the absence of external monitoring or failure detection techniques, a leader which goes down at the very beginning of its lease will cause VJAB to be completely unavailable until the lease expires. Thus, a lease which is too long risks large periods of unavailability, while a lease which is too short may actually result in *more* messages than single-synod Paxos (e.g. if every get/set requires both a leader election and a Phase 2 commit). We set the lease to 10 seconds.

Note that the use of a lease assumes that nodes have loosely-synchronized clocks. This can be maintained using a time synchronization protocol like NTP and adding small delays to the leader election process to account for clock drift. However, correctness is never violated even if there are multiple leaders due to large clock drift. The presence of multiple proposers causes Multi-Paxos to degrade gracefully into the single-synod Paxos approach, and correctness is always maintained.

```
ForwardRequestToLeader(Node n, Msg m):

  if n.leader == null or
  n.leaderExpiration <= time() then
    {Leader unknown or expired - propose ourselves!}
    n.PROPOSE(leader=n)
    n.waitForMajorityResponse()
    {Find promise with highest proposal num}
    prior ← argmax(n.promises)

    if time() <= prior.expiration then
      {There is another active leader}
      n.leader = prior.leader
    else
      {Assert ourselves as the new leader}
      T ← time() + LEASE_DURATION
      n.ACCEPT(leader=n, expiration=T)
      n.waitForMajorityResponse()
      if n.majorityAccepted() then
        n.leader = n
      else
        return ERROR
      end if
    end if
  end if
  n.leader.HandleRequest(m)
```

Figure 1: When a node receives a get or set request, it is required to forward the message to the leader. If the leader is unknown or expired, the node will propose itself to be the new leader. In that case, either it learns about the existing leader from the returned PROMISEs, or it fully establishes itself as the leader. Note that this is the logical algorithm; the actual implementation is event-driven, there is no waitForMajority().

3 Implementation

We first describe the leader election process and its expected behavior under various failure conditions. Then we explain the exact procedure for get and set routines.

3.1 Leader Election

Whenever a node receives a `get` or `set` request, it is required to forward the request to the current leader. Thus, the first step in every `get` or `set` operation is to determine who the leader is (see Figure 1).

Every node N keeps track of the most recent leader L it is aware of and the corresponding lease expiration time T. Upon receiving a `get`/`set` request, the node checks if (a) L is defined and (b) the current time is less than T. If so, L is still the active leader, and N immediately forwards the request to L. The leader is responsible for delivering the final result to the end-user.

If N has never received a request before or the most recent lease has expired, then N issues a proposal to be the new leader. If there is already a leader with an active lease, N will learn about it from the PROMISE messages. Otherwise, N finishes the Paxos algorithm and establishes itself as the new leader.

3.2 Gets

Every get or set operation increments a *sequence number*, but the proposal number stays the same for a given election cycle.

Once the leader has been determined, they are responsible for processing the request. `ACCEPT` operations will have one of three subtypes: `LEADER`, `GET` or `SET`. The leader asks all acceptors to `ACCEPT-GET` the given key. The value will be contained in the ACCEPTED responses.

Much like a promise, the leader can simply choose the value that has the highest proposal and sequence number, since having a majority response guarantees that the latest value will be contained in the responses.

An optimization (which we do not yet implement), is to cache the GET requests so the leader need not re-ask the group if it already knows it has the most recent value.

3.3 Sets

Sets are very similar to gets. The leader asks everyone to ACCEPT, and then waits for a majority. Each acceptor is responsible for committing.

4 Test Cases

We have several test cases designed to test our implementation and code. Out of those, there are a few that showcase our algorithm well.

```
basic-test.chi
--------------
 1 | start -n node-1
 2 | start -n node-2
 3 | start -n node-3
 4 | start -n node-4
 5 | wait -t 1
 6 | set -n node-1 -k A -v 11
 7 | wait -t 1
 8 | get -n node-1 -k A
 9 | wait -t 0.3
10 | get -n node-2 -k A
11 | wait -t 0.3
12 | get -n node-3 -k A
13 | wait -t 0.3
14 | get -n node-4 -k A
```

This starts the nodes, and then tests setting one value and accessing that value across all nodes.

In line 6, when node-1 gets the set request, it does not know who is the leader is, so it makes a proposal for itself to be leader, while queueing the 'set' request. Then when it becomes the leader, it makes the set request with the latter two "Accept" and "Accepted" messages of the shortened leader-version of multi-paxos.

Then, for the get requests in lines 8 to 11, all the nodes know who the leader is, and then forward their requests to the leader to answer. The leader asks everyone for what they think the value is, and takes the majority of that. This is to deal with the case that we have out-of-date values.

```
fail-recover.chi
----------------
 1 | start -n node-1
 2 | start -n node-2
 3 | start -n node-3
 4 | start -n node-4
 5 | wait -t 1
 6 | fail -n node-2
 7 | wait -t 1
 8 | set -n node-1 -k A -v 11
 9 | wait -t 1
10 | recover -n node-2
11 | wait -t 1
12 | get -n node-2 -k A
13 | wait -t 0.3
14 | get -n node-3 -k A
15 | wait -t 0.3
16 | get -n node-4 -k A
```

This tests our code for the case where one of our nodes are down before we make a set. Since only one (out of three acceptors) are down, node-1 in line 8 is still able to get a majority quorum to pass itself as the leader, and then make the set as the leader. In lines 13 and 14 when node-3 and node-4 (who were awake the whole time)

receive get requests, they simply forward those to the leader, node-1, for processing.

However, when node-2 gets the get request, it's not aware of a leader, then tries to become the leader, then is sent reject messages by the other nodes telling it who the current actual leader is, and then gives up.

```
leader-election-with-fail.chi
-----------------------------
 1 | start -n node-1
 2 | start -n node-2
 3 | start -n node-3
 4 | start -n node-4
 5 | set -n node-1 -k A -v 100
 6 | wait -t 1
 7 | fail -n node-1
 8 | wait -t 10
 9 | set -n node-2 -k A -v 200
10 | wait -t 1
11 | recover -n node-1
12 | wait -t 1
13 | set -n node-1 -k A -v 300
```

This test case tests our implementation for handling leader elections and leases.

First, node-1 elects itself as the first leader when it receives a set request. Then, it goes down. After its lease expires, node-2 in line 9 becomes the leader (after a set request) and establishes itself with the awake nodes, node-3 and node-4.

In line 11, node-1 recovers, and thinks that there is no leader, and so proposes itself as the leader. However, the lease for node-2 has not yet expired, so the other nodes tell it in a reject message that node-2 is the leader, and the set fails.

5 Conclusion

While working on implementing Multi-Paxos, we learned a lot about Paxos and about working within the field of distributed systems and debugging these systems, as well as doing such in Python.

It seems that the base reasons for why base Paxos works is based on three sections - usage of a combination of two 2PC cycles, where the first filters for the most recent successful update, and the second requires a majority to update.

To actually store a value, Paxos relies on the second half of the algorithm to ensure that a majority of the acceptors accept the value.

This ensures that gets, which relies on the first half of the algorithm, work nicely. Gets are different from sets because while you want to make sure a majority of nodes agree with you on a set, you want to look at all the nodes and simply take the most recently updated value for the get. Because every two majorities in a group share at least one node, we know that when we reach a majority amount of promises, we know we MUST have seen the most recently passed value (since that was accepted by a majority of notes).

This seems really basic. However, to optimize this further, and for more complex, industry level situations, there are a huge number of tweaks to Paxos. For example, there are lots more potential roles one can add - roles to learn from nodes, nodes to handle specific gets or sets, nodes to manage other nodes... You could cross paxos with byzantine generals, and then have specific nodes to handle specific combinations of nodes in both cases. The list goes on. In addition, to optimize for specific cases when specific patterns of gets or sets are known, many other variants of paxos can be made, in many papers such as Paxos Made Complex, and more.

Python, although a top choice as an 'easy' coding language, can quickly turn out to be difficult while coding. Its lack of a formal type system and a formal class structure / header means that it's really easy to have typos or set variables to things they shouldn't be, only to have them blow up in your face in obscure ways as either user-facing programs or internal routers experience errors. So in a real-life system, it's possible that a language like C++ or Java would be better for implementing a distributed algorithm, not just because it would be faster, but also because of the built in type security.

6 Acknowledgments

Online blog posts were very helpful for understanding the basic idea behind Multi-Paxos [3] and leader election [1].

References

[1] BEISKE, K. Leader election, why should I care?, September 2013. https://www.elastic.co/blog/found-leader-election-in-general#paxos.

[2] 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 *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 205–220.

[3] FENG, A. Paxos/multi-paxos algorithm, December 2011. http://amberonrails.

com/paxosmulti-paxos-algorithm/#multi-paxos.

[4] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[5] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*, 2 (2002), 51–59.

[6] GOOGLE. LevelDB. https://github.com/google/leveldb.

[7] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998), 133–169.

[8] LAMPORT, L. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[9] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 3 (1982), 382–401.