PushyDB

Jeff Chan, Kenny Lam, Nils Molina, Oliver Song

{jeffchan, kennylam, molina, osong}@mit.edu

https://github.com/jeffchan/6.824

1. Abstract

PushyDB provides a more fully-featured database that exposes RPC calls through a single client library. It builds on the features offered in the previous labs, and primarily adds in a reliable publication-subscription service which delivers updates to clients about keys. Pushy also provides an optional time-to-live on **`PUT`**'s, failure-recovery, and an implemented Multi-Paxos library. We also provide a sample application to demonstrate the functionality of its subscription service.

2. PubSub

We implement a publish-subscribe service which provides reliable, ordered updates to clients on changes to keys to which they have subscribed. The service, also called our `messagebroker` service, provides a framework for publications to be pushed to each client with a correct ordering per key to which they have subscribed. Like the `shardmaster` service, it is implemented as a single fault-tolerant system using Paxos. The semantic guarantee which our design provides is that clients receive in all updates about the period for which they are subscribed in the Paxos log.

2.1 Per-Key Ordering Consistency

In order to provide per-key ordering consistency, the `messagebroker` service maintains a list of client-key pair subscriptions and the versions of the last successfully pushed update to each client per key. Because `shardkv` servers must notify the `messagebroker` service on every processed **`PUT`** request, the per-key versioning happens with the `shardkv` servers, such that any **`PUT`** will increment the version number. New publications to clients will then only occur if the new publication has a version which is one greater than the last successful publication.

2.2 Subscriptions

Though the subscriptions are stored within the `messagebroker` servers, the key-value storage (`shardkv`) servers accept the **`SUBSCRIBE`** and **`UNSUBSCRIBE`** RPC's and act as a relay for clients. Clients who wish to subscribe to a key notify a `shardkv` server responsible for the key of their intent to subscribe, at which point the key-value group does a round of Paxos to agree upon the subscription. Once the Paxos instance containing the subscribe action has been decided, the RPC call can return to the client, even though the subscription has not yet been processed by the `messagebroker` service. This reply simply guarantees that the request will eventually get processed in the correct order of the Paxos log. In order to support out-of-order forwarding from the `shardkv` servers to the `messagebroker`'s, **`SUBSCRIBE`** and **`UNSUBSCRIBE`** requests must also update the version number for a key. As with **`GET`** and **`PUT`**, the **`SUBSCRIBE`** RPC provide at-most-once semantics, because subscriptions may not be idempotent if **`UNSUBSCRIBE`** requests are interleaved or vice-versa. The **`UNSUBSCRIBE`** RPC follows the same pattern as the **`SUBSCRIBE`** RPC.

This design adds the overhead of acting as a temporary proxy for clients to the `shardkv` service, but `shardkv` servers already act as proxies for **`PUT`** requests, and we believe **`SUBSCRIBE`**/**`UNSUBSCRIBE`** requests will be infrequent compared to **`PUT`**'s, so the added responsibility is minimal. This also allows for **`SUBSCRIBE`** and **`UNSUBSCRIBE`** calls to be co-located within the `shardkv` client library, which is the only library that must be distributed to clients. This library does not need to know about the `messagebroker` service or servers, but only about `shardkv` servers and that a service may push notifications once subscribed, see Figure 1. Changes to the `messagebroker` service or server topology will then only affect server-side code on the `shardkv` servers, allowing for redeployments of any service, so long as the API remains the same and the `shardmaster` service is available.

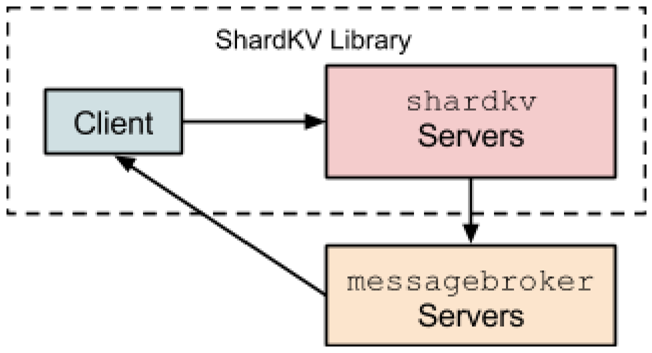

Figure 1. The client only sends RPC's to the `shardkv` service, but may receive updates from the `messagebroker` service.

2.3 Non-Blocking Notifications

For every **PUT** that is processed from the Paxos log, each storage server begins a go-routine to notify the `messagebroker` service. This go-routine is only ended when a `messagebroker` replies that it has successfully accepted the message, and messages are only accepted by the `messagebroker`'s in order of increasing version number. This gives our design the major benefit of being asynchronous with respect to the notifications. See Figure 2. However, this means that every storage server will individually pass along the same notification to a `messagebroker`. These notifications are filtered out by the service, but this adds to the number of RPC's which are called. On a real deployment, a `messagebroker` within the same datacenter would be responsible for local groups, reducing network congestion.

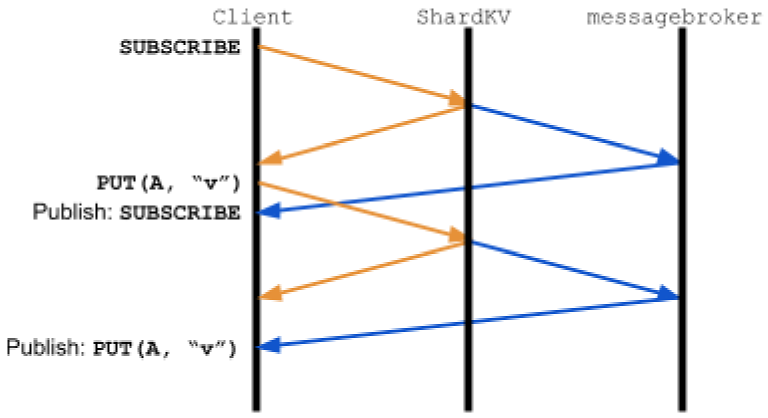

Figure 2. RPC messages are forwarded from `shardkv` servers to `messagebroker` servers such that they are not blocking normal operations.

3. Expiry

Another feature which we provide for is the temporary storage of values on a key. This feature can be useful for Pushy to act as a cache or as a buffer for large values to be passed from client to client. When a client calls the **PUT** RPC, they can specify an amount of time they intend the value to last within the system, or a time to live (TTL). Every server which is the recipient of a client RPC, whether **GET** or **PUT**, will propose a local read of its own clock along with the intended operation. Replicas use the timestamps from the Paxos log when applying an operation, instead of reading its own clock. When applying a **PUT** with a expiration, server sets expiration time to the given TTL + proposed timestamp from the Paxos log. When applying a **GET**, server checks the expiration time of the key against the timestamp from Paxos log. In this way, values expire consistently regardless of shard transfers or failures. From the perspective of the client of the original RPC, values accurately expire after the TTL interval.

4. Multi-Paxos

PushyDB also provides the option to use Multi-Paxos. We implemented Multi-Paxos, conducted benchmarks, and ultimately chose to remain with regular Paxos. Our implementation of Multi-Paxos follows a simple approach to leader election: the paxos instance with the highest ID that isn't dead is the leader. All paxos instances ping all other paxos instances regularly, and instances that do not ping in 2 ping intervals are declared dead. When a leader is elected, only that leader proposes. The leader does a single round of prepare, and from then on only does accept rounds. With these optimizations, we found 7-30% performance increases in concurrent RPC benchmarks, as can be seen in Figure 3.

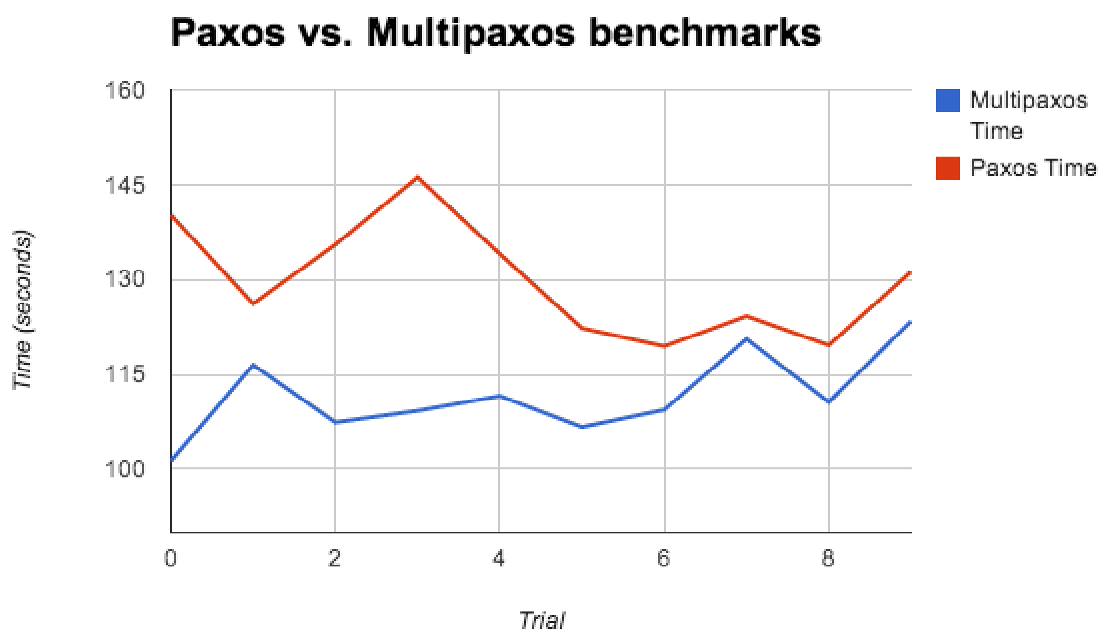

Figure 3. Paxos vs. Multi-Paxos performance in concurrent RPC benchmarks. Multi-Paxos performs better in all cases.

However, there were drawbacks to our Multi-Paxos implementation. We found Multi-Paxos to use five times the amount of RPCs (benchmarked with `TestRPCCount`) on average as a result of implementing the `Push` goroutine to drive lagging non-leading instances to catch up. Our Multi-Paxos implementation also introduced bugs, which would manifest only when used with other systems. Because of these drawbacks, we decided to use regular Paxos in our final product.

5. Application

To demonstrate PushyDB, we created two demos. The first demo subscribes to HTTP post requests from a PushyDB service. The second demo is a full sample application built in Go using PushyDB.

5.1 Subscription service

The PushyDB subscription service forwards updates on subscribed keys to a specific HTTP endpoint. This setup is advantageous because the developer can use any technology stack / language. It would even work with a static page running Javascript. The drawback here is all of your interactions with the database must be through HTTP requests, which are more cumbersome than having a direct database connection. In this case, we created a Meteor application that shows updates to a key in real time.

5.2 Sample web app

The PushyDB sample web app is a simple application built on Martini, a minimal web framework for Go. Its structure is depicted in Figure 4. Because we use Go, we are able to make direct PushyDB calls, as well as listen directly to the `Receive` channel. For any other language, we would have to write database bindings. From the web backend, we receive and push updates to the web frontend using a web socket. The demo application allows anyone on the website to update to a key in the database and receive pushed updates in real time.

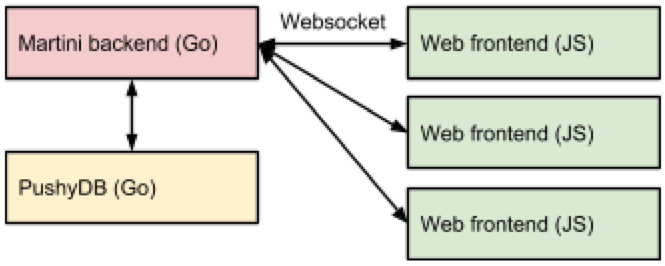

Figure 3. PushyDB sample web app structure.

6. Persistence

We persist the Paxos log to recover servers from crashes, so long as the disk contents are not lost. Every time a Paxos server updates an instance (a `Propose`, an `Accept`, or a `Decided`), it saves a copy of the instance to disk. The `highestDone` value, which represents the instance at which the Paxos log was garbage collected, is also persisted. On recovery, we read the disk for persisted data, copy them into memory, and restore the state of the Paxos log.

6.1 LevelDB

We use LevelDB [1], an on-disk key-value store, to persist data. LevelDB allows storage of arbitrary byte arrays, handles synchronous writes, and supports batched operations. It is designed to be high performant and supports data access from a single process. We chose to use LevelDB mainly for its performance and support for synchronous writes, along with write-ahead logging to preserve the integrity of data [2].

We did however run into three bugs when using `goleveldb` [3], a third-party Go wrapper around LevelDB. The first bug we encountered was that we could not close connections to databases properly, which resulted in failing test cases. Instead, we were still able to emulate separate databases by sharing a single database for all servers, and prefixing all keys with the appropriate server name. The second bug related to incorrect behavior when using synchronous writes. We continued by disabling the option, but this configuration no longer tolerates machine crashes during a write. Finally, `goleveldb` occasionally threw `nil pointer` panics when compacting data, and we were unable to identify the underlying source. We plan to report these bugs to the author of `goleveldb`.

7. Future Work

We believe the feature set given can provide much utility to developers, but continued work would improve several of the features. First, with more time, we would be able to fix remaining bugs with Multi-Paxos. Second, alternative embodiments of Paxos are also available, and though we implemented two variants, EPaxos could reduce the latency of the underlying agreement protocol on which all consensus in our system relies. We believe that this would provide many gains in our system, since non-conflicting operations could be processed in parallel much more efficiently. Third, the failure recovery mechanism could benefit from snapshots of the database state instead of replaying an entire Paxos log. Fourth, the `messagebroker` service could be modified to automatically accept all incoming notifications from the `shardkv` servers and put the notifications into a buffer. The `messagebroker`'s would then decide amongst themselves to whom each notification can be published and when. It is unclear if this would positively or negatively affect performance or the RPC count in a real deployment, but would provide for a stronger layer of abstraction. Finally, an alternative (less buggy) disk-interface could be used, such as those listed here [4].

8. Conclusion

The primary goal of PushyDB was to implement a reliable publication-subscription service on top of the key-value service we developed through 6.824 labs. As we demonstrate in the provided standalone application, this feature is useful for any clients who wish to receive in-order updates to keys without periodically pinging the database. We also provide for additional features which we believe may be useful in a database of any kind, such as crash recovery and temporary **`PUT`**'s. Finally, we found improvements to the performance of the paxos protocol by implementing Multi-Paxos, but have experienced integration issues with this variant.

9. References

[1] LevelDB. https://code.google.com/p/leveldb/

[2] Comment by Sanjay Ghemawat, co-writer of LevelDB https://news.ycombinator.com/item?id=2526311

[3] LevelDB Go wrapper library. https://github.com/syndtr/goleveldb/

[4] Status of leveldb http://grokbase.com/t/gg/golang-nuts/143tggs35x/go-nuts-status-of-leveldb