Distributed Systems - Practical Assignment

Barr Israel
321620049

Introduction

The system I implemented is an in-memory distributed string only key-value store that maintains a consistent state via a replicated log, the replicated log is implemented using Multi-Paxos, which runs a Paxos instance for each "commit", a commit consists of one or more key-value store write requests. The key-value store supports listing all keys, reading and writing key-value pairs, and conditionally storing based on revision numbers(similar to compare-exchange but only given revision instead of full value).

The network of applications used in the project consist of:

- an etcd server for storing configuration values, maintain an alive server list, and facilitating leader selection via compare-exchange transactions.
- The main server application written in Go, implementing the distributed key-value store by storing the current database and log state, handling client HTTP request, communicating with the etcd server as necessary, and communicating with other server peers via gRPC to run Paxos and add to the log.
- a demo application written in Rust, implementing a network based document editor(like Google Docs), communicating with the server application instances via HTTP to read and write the documents.

In the end of the report there are graphs illustrating the main communication paths the various components are part of for various actions.

Code Architecture

The main server application consists of a few packages, each with its own responsibilities:

- cluster: responsible for all communications between the cluster servers and for holding the database and Paxos instances state.
- etcd: responsible for all communications with the etcd server.
- httpserver: responsible for running and serving the HTTP server, handling client requests to write and read into the database
- config: responsible for reading configuration values from the command line
- util: utility functions

Running instructions and per file responsibility descriptions are found in the file tree in the README.md file.

HTTP API

Method	Path	Description	Available Query Parameters	Return Example	Consistency
GET	/keys	gets a list of all existing keys and their revisions	**linearized**: if present, the request will be linearized, otherwise it will be sequentially consistent **omit-deleted**: if present, the list will omit deleted keys, otherwise it will contain deleted keys(and the revision they were deleted on)	[{“key”:“key1”, “revision”:2}, {“key”:“key2”, “revision”:3}]	Linearized if query parameter **linearized** present, otherwise sequentially conssistent
GET	/keys/{key}	gets the current value and revision of {key}	**linearized**: if present, the request will be linearized, otherwise it will be sequentially consistent **revision-only**: if present, the reply will only contain a revision number and not a value,	{“value”:“lorem ipsum”, “revision”:7}	Linearized if query parameter **linearized** present, otherwise sequentially conssistent
PUT	/keys/{key}	writes (conditionally) the body of the request as the value for {key}	**async**: if present, the request will return an empty reply immidietly and perform the write asynchronously **revision**: if present, the given revision will be used to do a conditional write(detailed below)	{“success”:true, “revision”:3}	Linearized
DELETE	/keys/{key}	deletes (conditionally) the value for {key}	The same as PUT / keys/{key}	{“success”:false, “revision”:4}	Linearized

Deletion remains

A deletion of a key will not fully delete it from the database, instead it will leave it with an empty value(distinct from an empty string) and a new revision number.
A deletion is functionally identical to writing an empty value(which is not possible with a normal PUT /keys/{key})

Revisions

Each write/delete sets a new revision number to the key.
Revision numbers are increasing, but not 1 by 1, and it is possible for 2 keys to have the same revision number.
Non-existent keys have a revision of `0`.
In more details:
2 concurrent writes/deletes to different keys may cause them to set the same revision number.
2 concurrent writes/deletes to the same key may cause the revision to be updated twice, so they may return different revision numbers, and the one with the higher revision reflects the current state of the database.
2 non-concurrent writes/deletes to any 2 keys will always have the revision of the first write be smaller than the revision of the second write.

Conditional Writes/Deletes

If a revision is present on a write/delete, it will cause it to become conditional.
A conditional write/delete will only be applied to the database if the current revision value is equal to the given revision value.
This is similar to a compare-exchange operation, but it only compares the revision value, which means sending the old value is unnecessary.
If the write/delete succeeded, it will return `{"success":true,"revision":<new revision>}`, and if not, it will return `{"success":false,"revision":<current revision"}`
A non-conditional write/delete will always succeed and return `{"success":true,"revision":<new revision>}`.

Configurable Parameters

The etcd server holds relevant configuration values that are loaded by each peer:

- `paxos_member_count`: the amount of total members in the cluster
- `paxos_max_req_per_round`: the maximum amount of requests that may be fulfilled in a single commit, used to limit the size of messages in an individual Paxos instance and to prevent an unbound loop of reading requests without fulfilling them.
- `paxos_cleanup_threshold`: the maximum amount of committed Paxos instances to hold before initiating active log compaction.
- `paxos_retry_milliseconds`: amount of milliseconds to wait before retrying a failed gRPC request.
- `etcd_lease_ttl_seconds`: amount of seconds before a lease expires if not renewed
- `paxos_artifical_dela_milliseconds`: optional artificial delay added to all acceptor requests, used to test the system since otherwise Paxos instances are committed too fast.

Implementation Details

Leases

Each alive peer holds a lease in the etcd server and keeps it valid with a heartbeat coroutine.
The “alive” flag is set under the lease, and the current leader has a “leader” value set using its lease.
When a peer crashes, the lease expires and both of these are deleted(“leader” if the crashed peer was the leader).

Leader Selection

The leader is selected using a compare-exchange-like transaction on the etcd server.
There is no active listening to a crashed leader, but instead peers will discover a crashed leader when they fail to communicate with the peer they currently believe is the leader(or they reach it and it replies that it is not really the leader, likely due to a crash-recovery).
When a peer discovers the leader crashed, they contact the etcd leader to check the current leader, and if there is no current leader, they attempt to promote themselves to be the leader using a compare-exchange-like transaction.

Paxos

Multi-Paxos

My Multi-Paxos implementation adds a “Paxos ID” field to all relevant gRPC requests in order to identify the Paxos instance the request belongs to.
Paxos instances are always commited(and eventually deleted) in the order of their Paxos ID, which increases 1 by 1.

Paxos Algorithm Implementation

My implementation of Paxos is a variation of the prepare-promise-accept-accepted algorithm shown in class, with two notable differences:

ACCEPTED messages do not contain the selected proposal.
Instead, if an acceptor receives an ACCEPTED message for a round they do not have a proposal set for, they send a “Fill In Request” to the acceptor that sent them the ACCEPTED message, which replies with the missing proposal.
ACCEPTED messages are only sent after an ACCEPT message, which do have a proposal attached, so every acceptor that sent an ACCEPTED must have a proposal set for that round.
It is possible for an acceptor to crash before supplying the missing proposal, but because an instance will be committed only after more than $\frac{n}{2}$ acceptors have sent accept, and because there will never be more than $\frac{n}{2}$ crashes at the same time(which extends to $\frac{n}{2}$ crashes in the same round with the recovery detailed later), it is impossible for all “Fill In Requests” to fail before the instance needs to be committed.

Because all write requests go through the leader, in the first round all acceptors are expected to return an empty proposal, which allows the leader to tell them to accept the requests that are currently pending on the leader.
If the leader can’t accept in the first round, that means it receives a proposal consisting of writes the previous leader tried to commit(but it crashed), so instead it adopts those and continues with them(like the original algorithm).
Note: because write requests are retried when a leader crashes and are sent to the new leader, it is possible for identical requests to appear in both the proposal from the previous leader and in the leaders pending writes, and in rare cases it might be committed twice, but from the log’s point of view, they are different writes, and the database implementation can handle this duplication.

Leader vs Acceptor

All alive peers are always listening to acceptor related requests and respond accordingly.
Acceptors are completely reactive, and do not perform any action that isn't in response to an incoming gRPC message(e.g sending ACCEPTED when receiving a valid ACCEPT or commiting when receiving enough ACCEPTED).
At any given time, under normal conditions at most one peer will run one instance of the leader algorithm in a loop. This is enforced with the leader flag in etcd and a local flag in each peer.
It is possible for a leader to be demoted while still running the loop, and it will realize it was demoted and abort the loop eventually.
When any peer promotes itself to be the leader, it starts the leader algorithm loop.

Recovery

My implementation of Paxos is fully in-memory, which does not allow immediate recovery of a crashed peer. Instead, a crashed peer recovers by requesting the minimal information required to join the next Paxos instance, and the leader replies when the currently ongoing instance is commited(or immediately if there isn't one).
A recovery state reply consists of:

- The full database state
- The "Committed Paxos ID", which is the ID number of the most recently commited Paxos instance.
- The "Minimum Paxos ID", which is the ID number of the oldest Paxos instance that might still be needed. The recovering peer only keeps it to be aware it might receive ACCEPTED from these, but it will not do anything with them.

Because there might not be a leader at a given moment because it crashed and no one noticed yet, the recovering peer needs a leader to appear, but it can't promote itself until it is recovered, so instead, in order to reach a leader it performs the following in a loop until it recovers:

1. Get the current list of alive peers.
2. Contact each alive peer asking for the current leader.
3. A contacted peer will refresh its leader against the etcd server and become the leader if there isn't one.
4. The contacted peer will reply with the current leader(which might be itself).
5. The recovering peer will contact the leader it received to get the current state.
6. The leader will reply with the current state.

Log Compaction

All peers have both passive and active log compaction.

Passive Log Compaction

When an acceptor receives ACCEPTED from **all** acceptors for a particular Paxos instance on a particular round, it knows all other acceptors already have the proposal and will eventually commit it, so it is allowed to delete it.
It is possible some of the other acceptors have not received ACCEPTED from this acceptor yet, but ACCEPTED messages are sent asynchronously and will eventually reach them, even if this acceptor deletes the Paxos instance.
Note: because Paxos instances are commited and deleted in order, a Paxos instance that can't be deleted yet will prevent any following Paxos instances from being deleted.

Active Log Compaction

When the amount of committed but not deleted Paxos instances crosses some configurable threshold on some acceptor, every time a Paxos instance is marked to be committed, that acceptor will ask **all** other peers for their committed Paxos ID.
Knowing all the committed Paxos IDs allows the acceptor to delete all Paxos instances with a Paxos ID smaller than the minimum committed Paxos ID.

Database and Read/Write Requests

Read Requests

Sequentially Consistent read requests are answered immediately by the peer that received the request.
Linearized read requests are forwarded to the leader, which naturally linearizes them in the order it receives them, and replies between Paxos instances(when there isn't one running) to ensure an unexpected leader switch can't cause a non-linearized execution.

Write Requests

Write requests are always linearized, so they are always passed forward to the leader.
When the leader receives a write request, it adds it to a write request queue along with a reply channel and waits for a Paxos instance that contains the write to reply.

Paxos Instance Committing

When an acceptor receives enough ACCEPTED messages for a round, it commits the Paxos instance.
Committing a paxos instance consists of iterating over the write requests it contains and attempting to apply them individually to the database.
When a write request does not contain a revision, it will always be applied to the database.
When a write request does contain a revision, it will only be applied if it matches the current revision of the same key in the database(non-existent keys have revision 0).
When a write is applied to the database, it receives a revision equal to the ID number of the Paxos instance it came from.

Replying To Write Requests

When the acceptor that is also the leader commits a write, it also compares it to all pending write requests and replies whether it was successful and whats the current revision to all matching pending writes.
It is possible that one committed request will match with multiple pending writes if they are identical.
In that case, if the requests had no revision, they will all get the same reply(successful write), but if they did have a revision, only one of them will be considered successful and the rest will fail.

HTTP Server

The HTTP server is a simple implementation of the API specified above that connects HTTP requests with a function API implemented in the server state

Docker

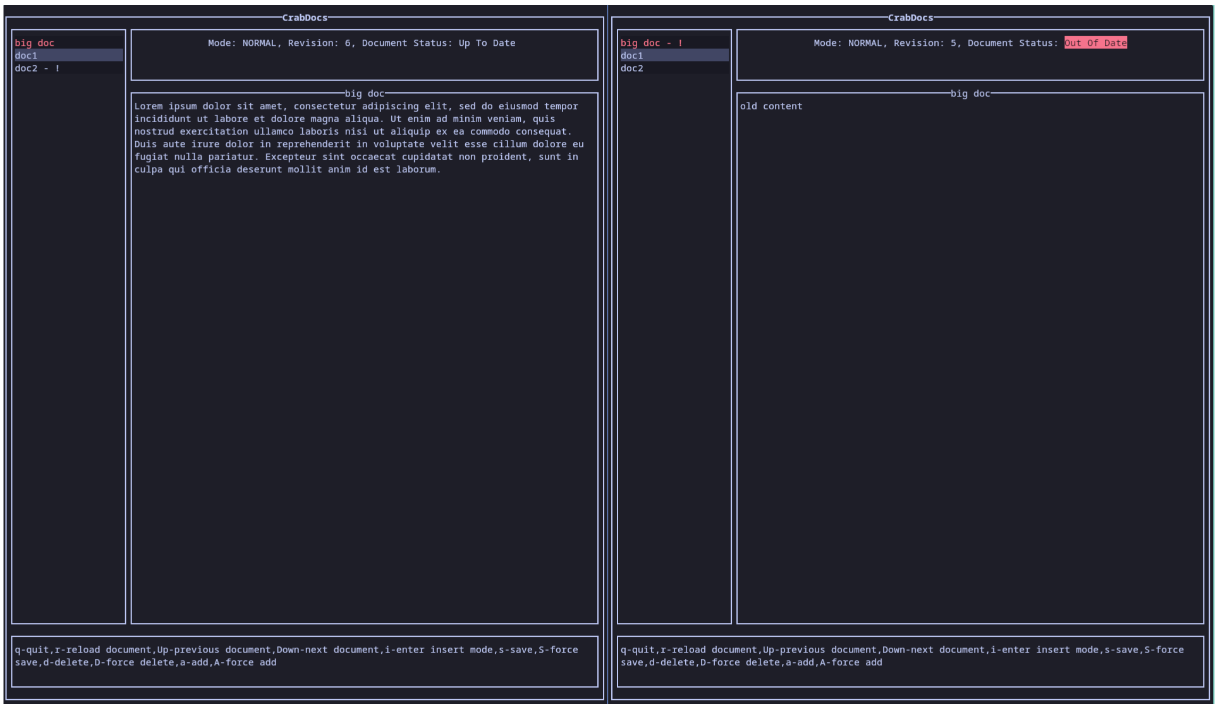
The `etcd/compose.yaml` file defines the following services:

- etcd server: the official etcd container, launches before any other container.
- init container: sets up the configuration on the etcd server, only launches after the etcd server appears as healthy
- 5x server containers: contains the server binary and a minimal httpcheck binary for health checks, launches after the init container shuts down successfully.

Demo

The demo client I implemented for this project is a network based document editor(like Google Docs) written in Rust. It communicates with the server cluster in a round-robin order, retrying with the next server if a request fails.
The client continuously monitor the list of keys, adding them to the sidebar and notifies the user if there are updated documents.
All keys are shown to the user, even deleted ones. Deleted keys are labeled as deleted when the user opens them.
When a document is saved, by default it is written conditionally with the revision that the user started editing the document from, rejecting the save if the revision is out of date. This allows the user to refresh their local copy of the document, or force-save it, writing it unconditionally to the database.
The client also displays to the screen whether the document they are looking at is out of date.

The source code for the demo is under the etc directory.



```
CrabDocs
big doc
doc1
doc2 - !
Mode: NORMAL, Revision: 6, Document Status: Up To Date
big doc
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.
q-quit,r-reload document,Up-previous document,Down-next document,i-enter insert mode,s-save,S-force
save,d-delete,D-force delete,a-add,A-force add
```

```
CrabDocs
big doc - !
doc1
doc2
Mode: NORMAL, Revision: 5, Document Status: Out Of Date
big doc
old content
q-quit,r-reload document,Up-previous document,Down-next document,i-enter insert mode,s-save,S-force
save,d-delete,D-force delete,a-add,A-force add
```

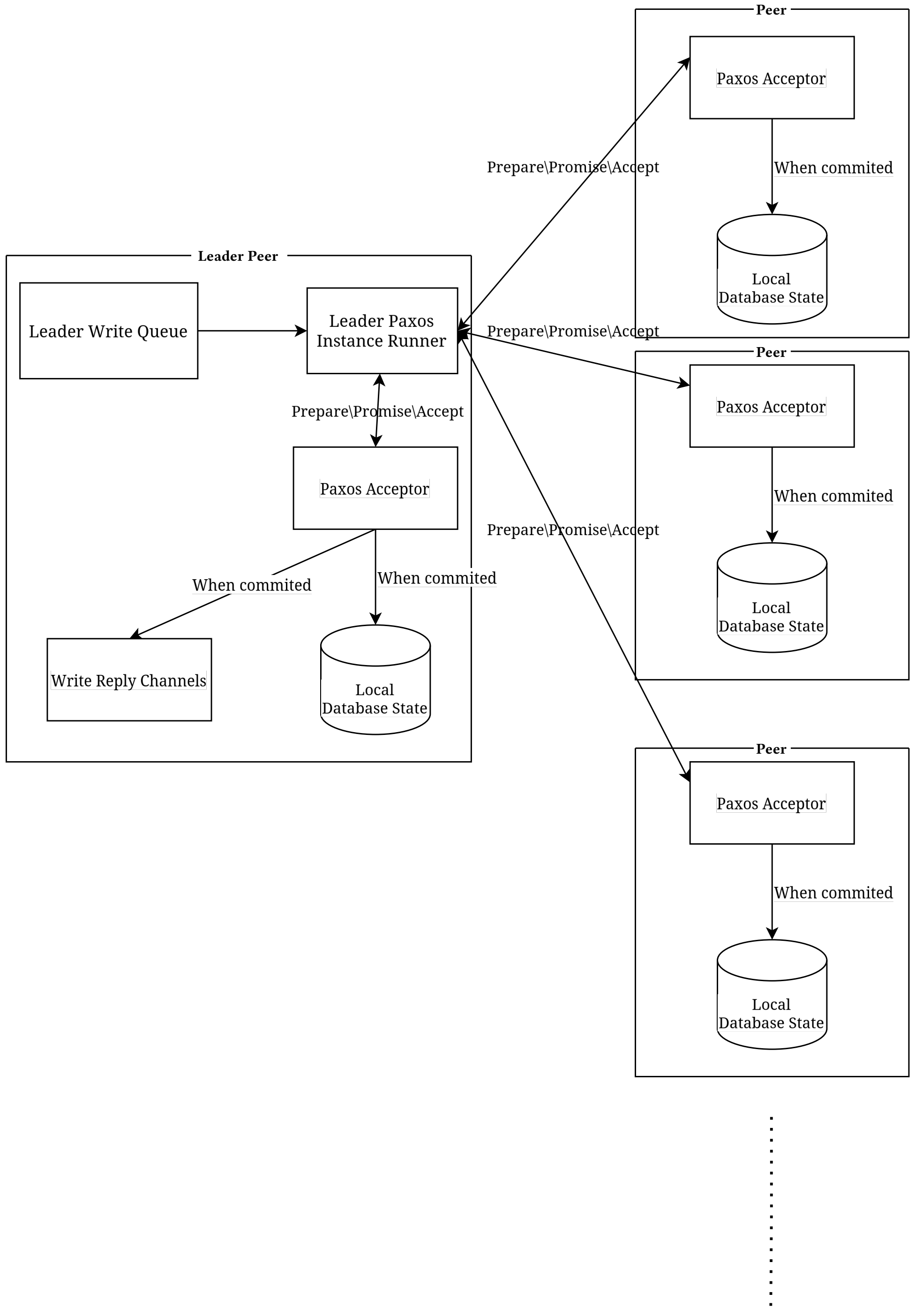
Figure 1: two instances of the demo client running side by side

Communication Graphs

Following are graphs that illustrate some of the communication paths that make up the full system.

Paxos Instance

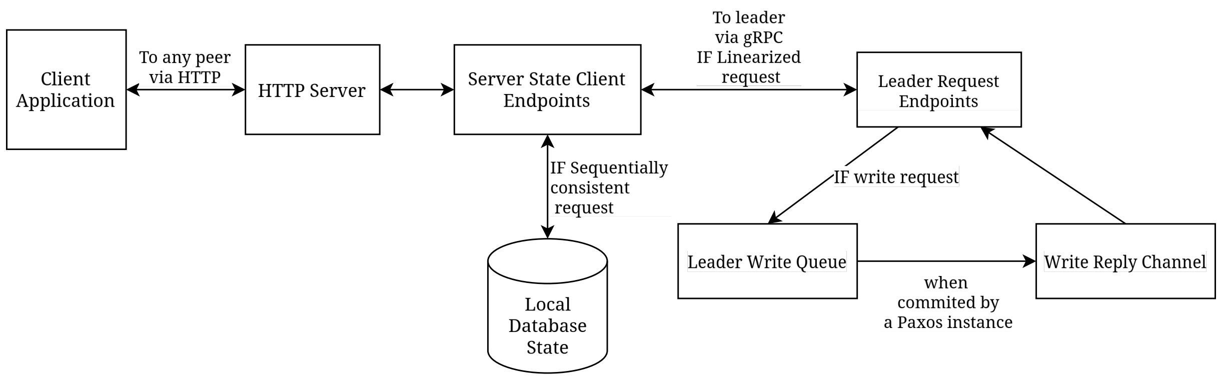
A Paxos instance(one full commit consisting of one or more rounds) consists of the following communications:



ACCEPTED messages pass between all Paxos Acceptors, but removed from the graph for visibility. When a Paxos instance is committed, all Paxos Acceptors apply all the writes it contains into the database, and the acceptor that is also the leader replies to write requests via an attached reply channel.

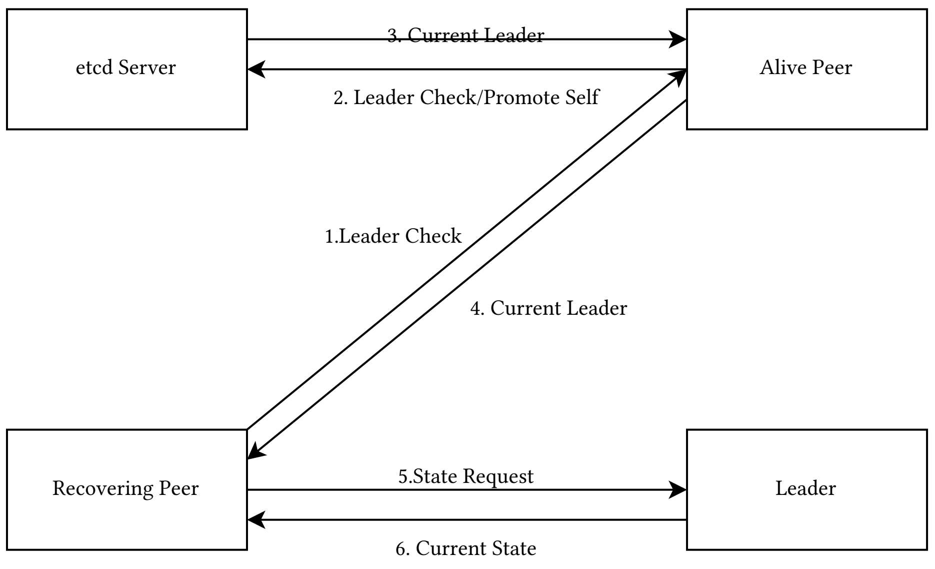
Request Handling

A request from a client goes through the following path:



Sequentially consistent requests(which can only be reads) can be served from the local database state.
Linearized requests(which may be reads or writes) are passed to the leader, which linearizes all requests.
Passing read requests to the leader naturally orders them in the order they are received by the leader. Writes are linearized by the Paxos algorithm and they get a reply when they are committed by some Paxos instance(even if they were rejected due toe a revision mismatch)

Recovery



etcd Server
Alive Peer
3. Current Leader
2. Leader Check/Promote Self
1.Leader Check
4. Current Leader
Recovering Peer
Leader
5.State Request
6. Current State