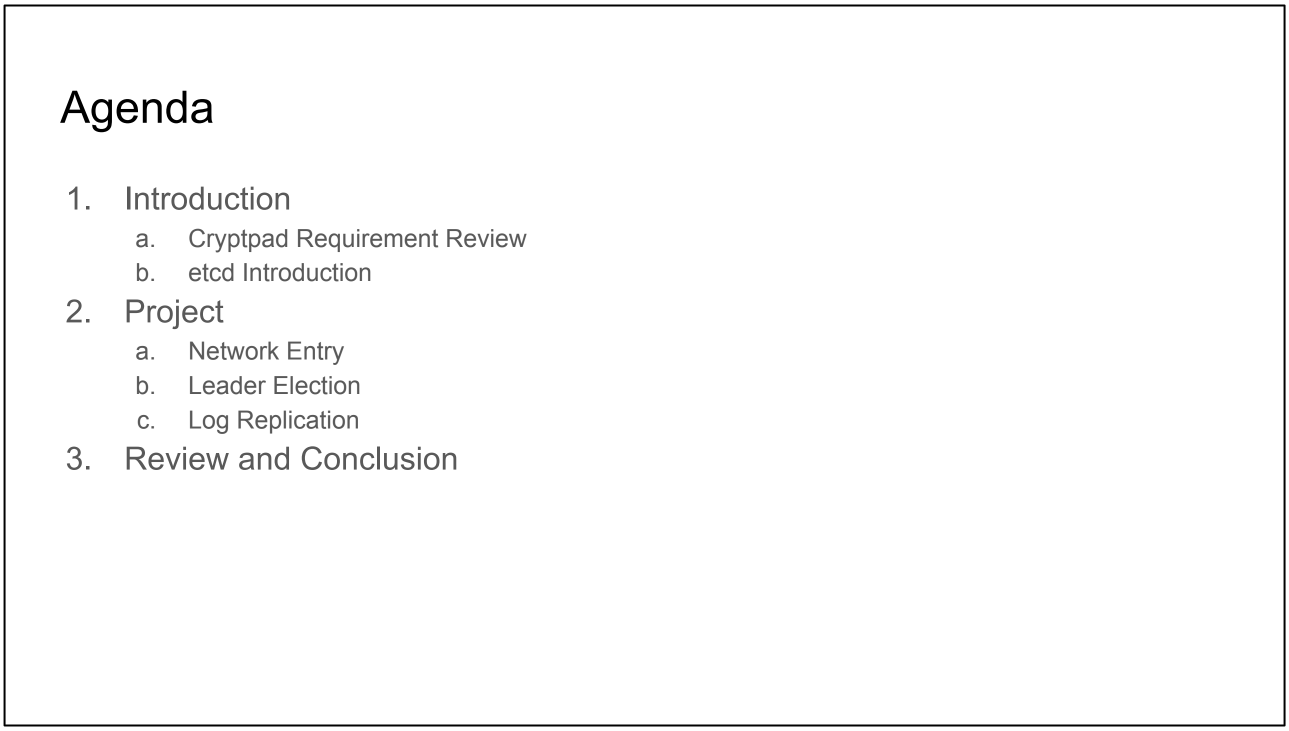
Toy Distributed Key/Value Store

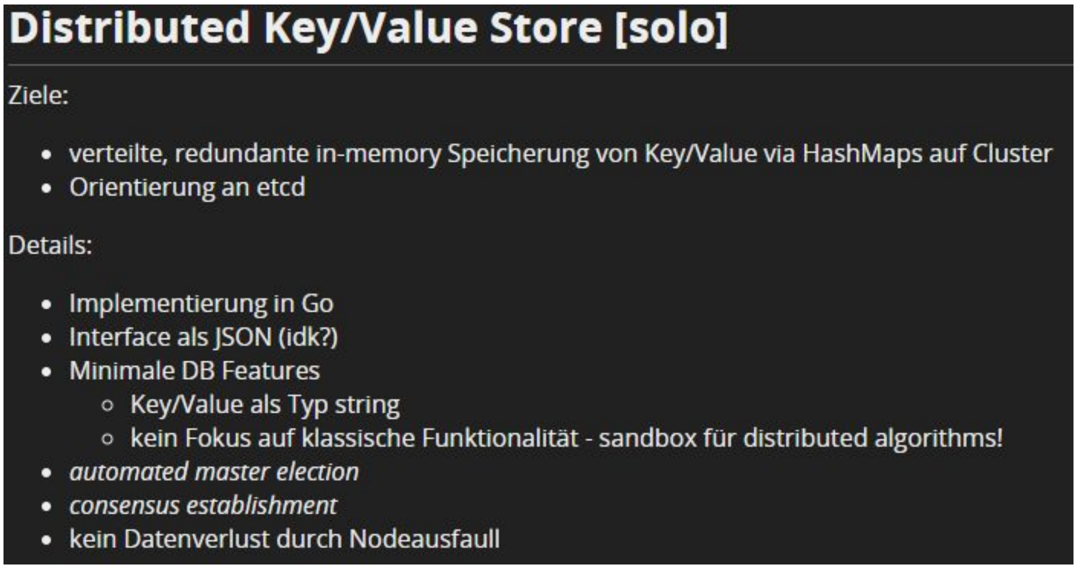
Jonas Heinrich, TINF18B2



Agenda

1. Introduction
   a. Cryptpad Requirement Review
   b. etcd Introduction
2. Project
   a. Network Entry
   b. Leader Election
   c. Log Replication
3. Review and Conclusion

- Project will be a mix of theoretical slides and a look into the code
- I hope to introduce the relevant parts at the relevant time to keep a good flow of information by using Sidenotes



Distributed Key/Value Store [solo]

Ziele:

- verteilte, redundante in-memory Speicherung von Key/Value via HashMaps auf Cluster
- Orientierung an etcd

Details:

- Implementierung in Go
- Interface als JSON (idk?)
- Minimale DB Features
  - Key/Value als Typ string
  - kein Fokus auf klassische Funktionalität - sandbox für distributed algorithms!
- *automated master election*
- *consensus establishment*
- kein Datenverlust durch Nodeausfaull

Cryptpad Review of Requirements

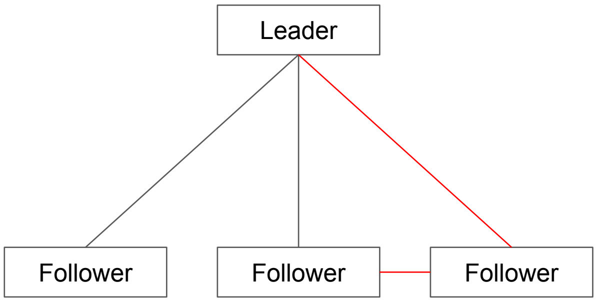
etcd

***"etcd** is a strongly consistent, **distributed key-value store** that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of machines. It **gracefully handles leader election**s during network partitions and **can tolerate machine failure**, even in the leader node."* - https://etcd.io/

- Written in Golang
- Backend for service discovery and cluster state/configuration for Kubernetes
- Communication is handled by the RAFT consensus algorithm

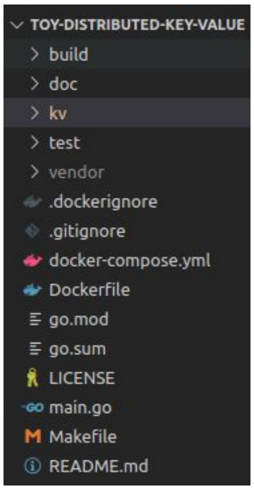
Goal of this project is to build a toy version of etcd

Network Entry



- Follower requires the address of one network member
- Calls the `/register` function of that node
    - If leader: 200, will now receive heart beats, updates, etc.
    - If follower: 301, redirect to Leader

Side Note: Project Structure

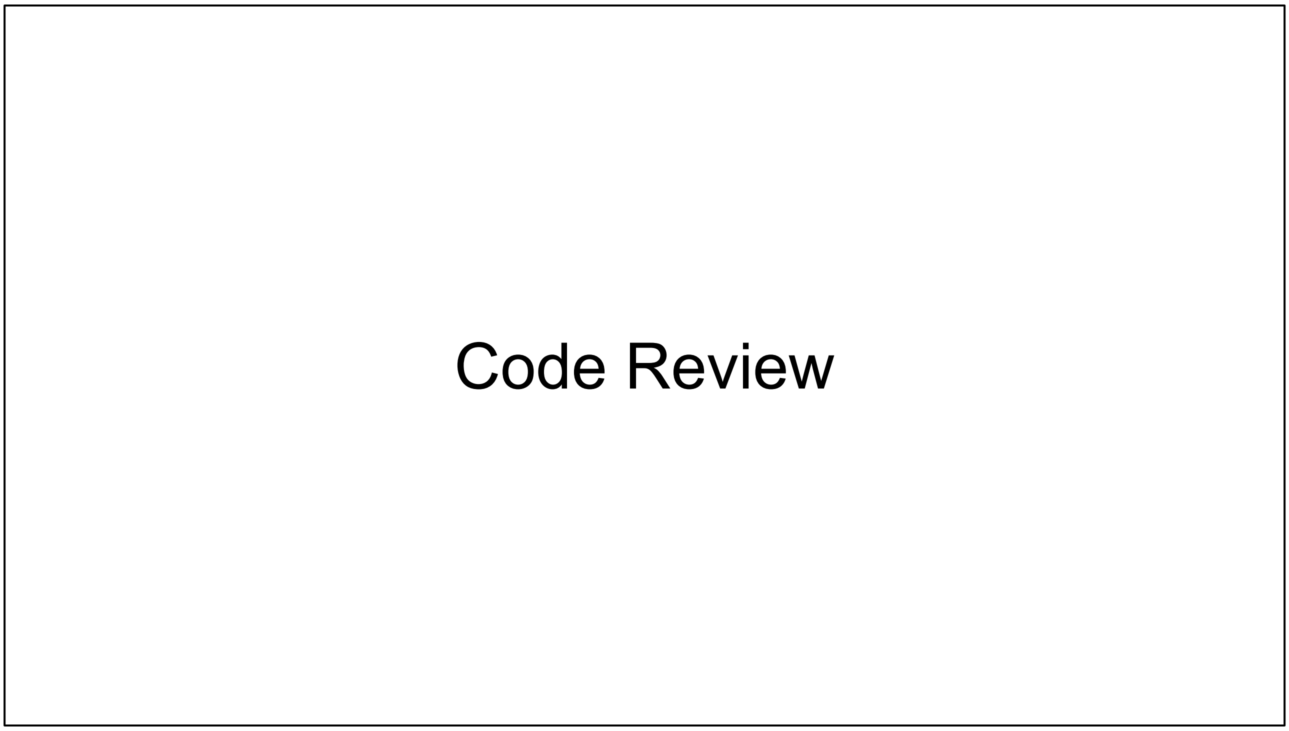

```
TOY-DISTRIBUTED-KEY-VALUE
  build
  doc
  kv
  test
  vendor
  .dockerignore
  .gitignore
  docker-compose.yml
  Dockerfile
  go.mod
  go.sum
  LICENSE
  main.go
  Makefile
  README.md
```

Besides the obvious:

- Each node is encapsulated by a Docker Container
- Each node is part of a Docker Network (controlled by Docker-Compose)
- `kv`: contains the source code of leader and follower
- `test`: contains test code for another container

Since I want to take peeks into the code, a basic understanding of the project structure seems to be helpful.

The test code is part of another container. It can tell the other containers to execute specific commands to get better testability. Such commands include self-destruct, state-dumping, registration, ..

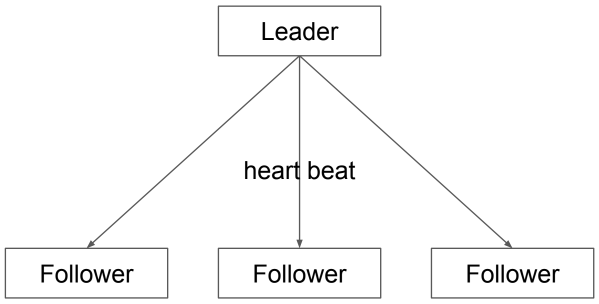


Code Review

- Command line interface will not be shown in this demonstration to save time

- Registration
    - `kv.go`
        - `Start`: In release mode, as soon as the service is started the node tries to connect to the leader
        - `register`: Call the node with the given entry point to register
            - If it is the leader node: all is fine
            - If it is not the leader node: Get the leader ip and retry
    - `routes.go`
        - `handleRegister`:
            - If not leader: respond with leader ip
            - Else: Append to own follower ip addresses
- Route: /register

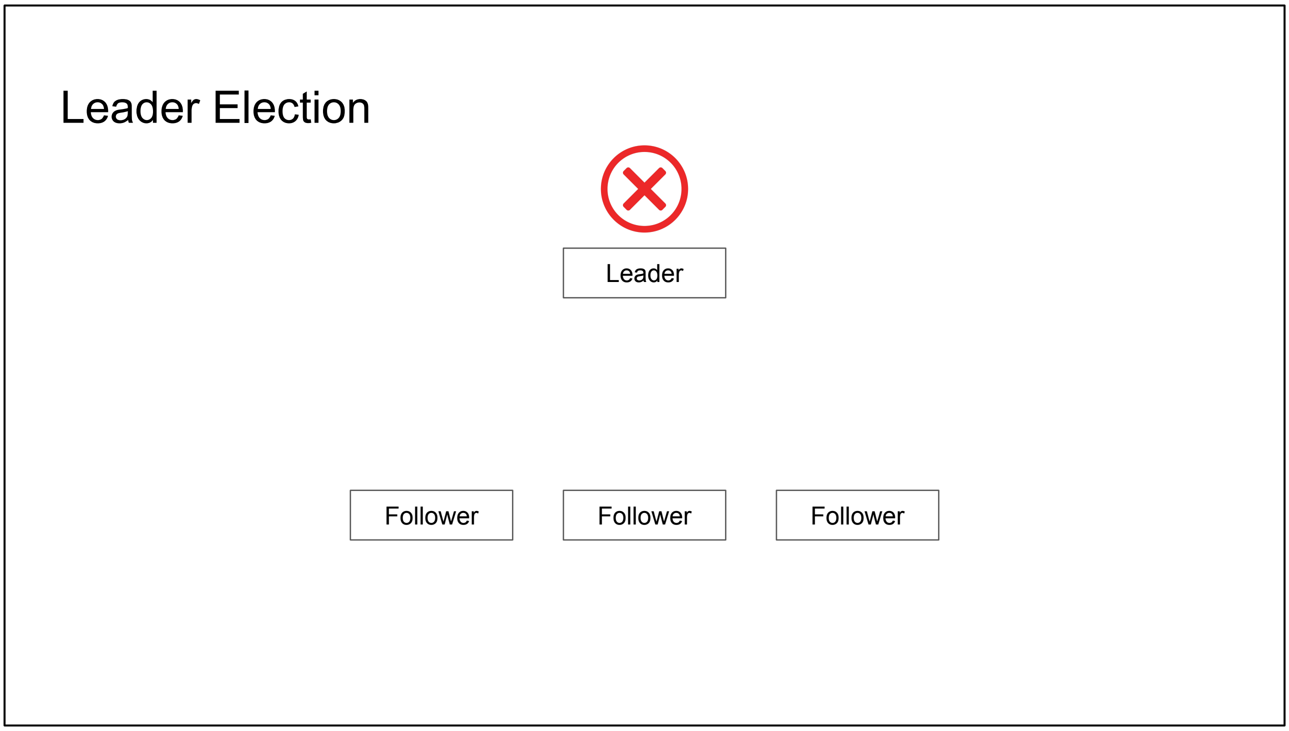
Leader Election



- Term: A term is an arbitrary period of time on the server for which a new leader needs to be elected

- Suppose we have a network with 1 leader and three followers
- How do the followers detect if the leader fails and even know of one another?
    - Regular heartbeat from leader to followers
    - Heart beat contains information on the current term, all follower addresses, etc.
    - Could be optimized

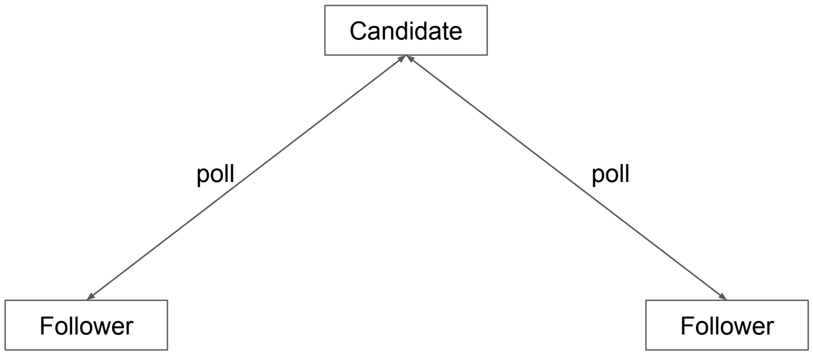
- If the leader node fails, a new one should be selected



Leader Election

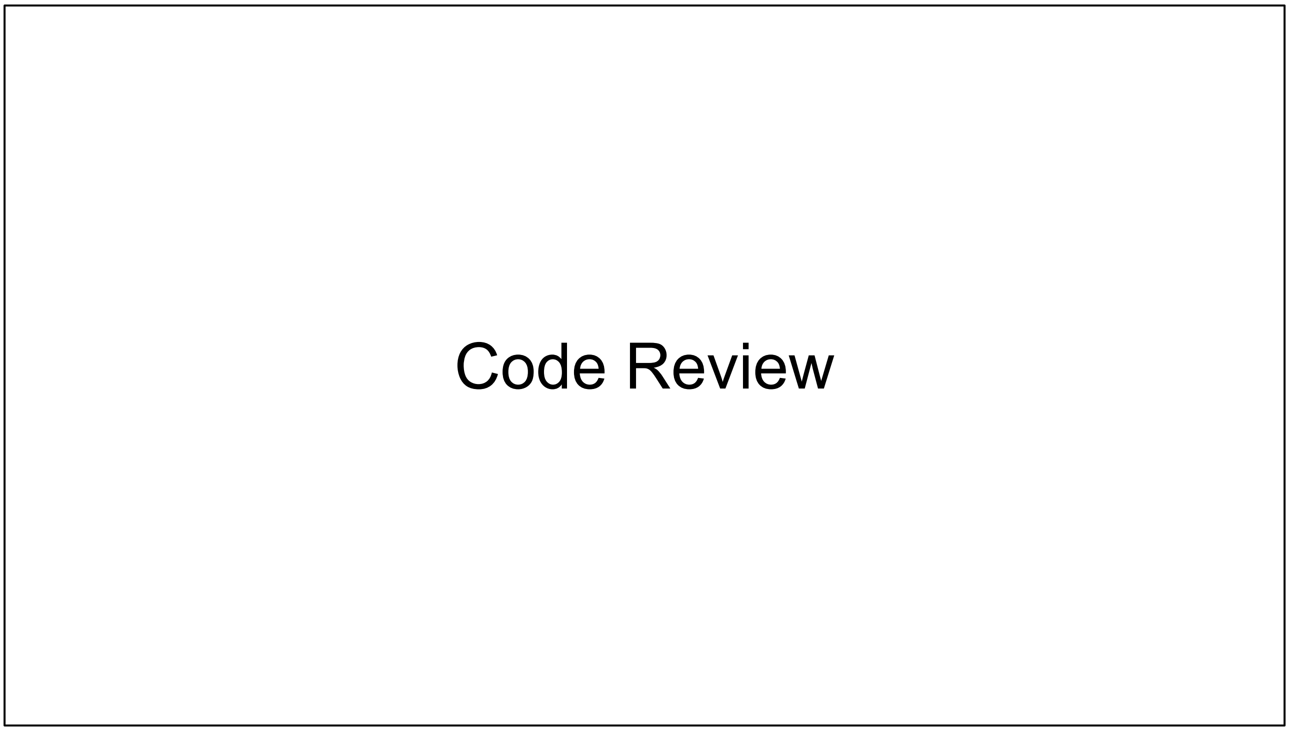
- Each of the followers has an internal loop that checks if the leader’s heartbeat arrived in time
- As soon as the leader seems to be absent
  - Increases term number
  - Makes itself a candidate

Leader Election



- Each candidate node requests a poll response from all other followers: newLeaderAddress, termNumber
- If 50% + 1 of the votes: election is won
  - Broadcast result to other followers
- Else
  - Election lost
    - either the check will trigger again, leading to a new election
    - Or another leader won and has broadcast its status

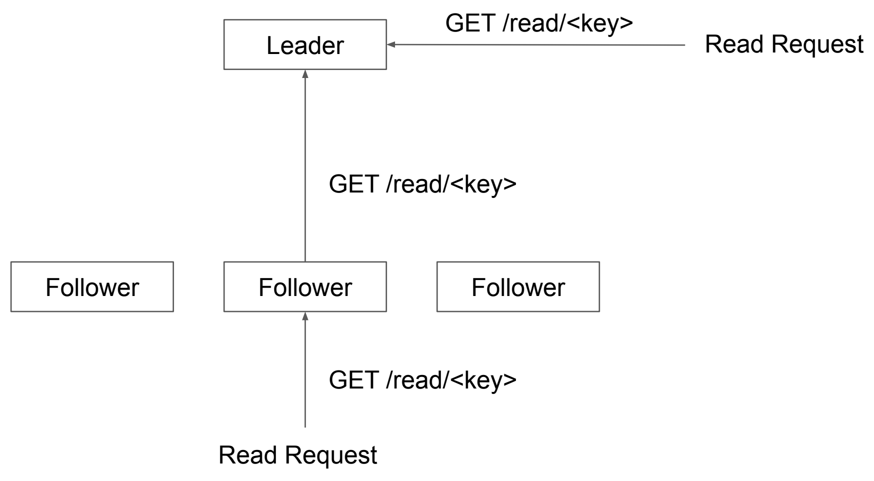
- Two questions should enter your mind:
  - Why are not all followers candidates?
    - Check for leader is randomized, therefore one will start sooner than the others
    - Hopefully it completes the election before the others even realize it
  - If there are multiple candidates, how does a follower decide?
    - First come, first serve (therefore lowest latency should win)
    - One vote per term, first candidate request wins
- Candidates do not have to communicate, they know if their votes received a majority



Code Review

- Leader Election
  - Switch between `kv.go` and `routes.go` (two screens)
    - `heartBeat`
      - Routine entered by leader
    - `handleHeartBeat`
      - Resets the leader death check
    - `checkLeader`
      - In each follower: check if leader is alive
    - `runPoll`
      - Runs election by asking each follower for its vote
        - `handlePoll`
      - If majority, election is won, broadcast leader update
    - `broadcastLeaderUpdate`
      - Tell each node who’s boss
        - `handleLeaderUpdate`

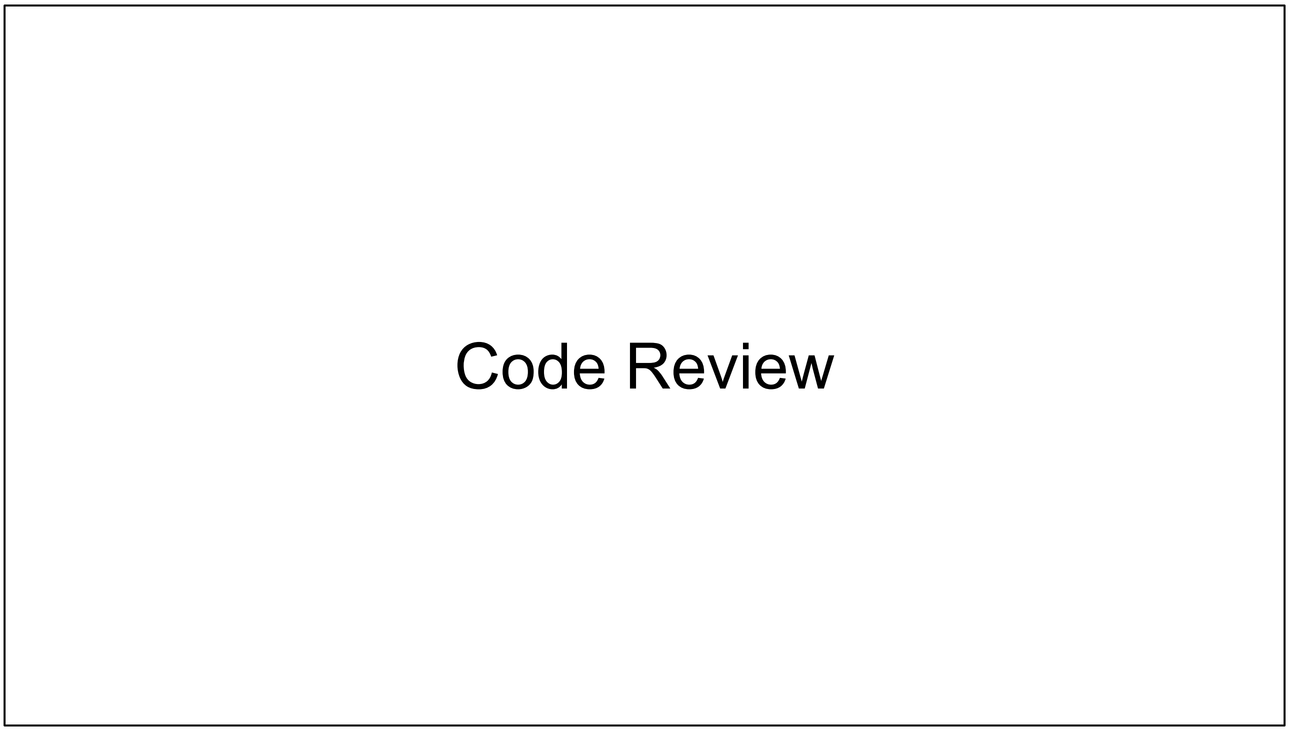
Read



- If a read request is directed to the leader, it will reply immediately
- A read request to a follower is proxied to the leader node

- An application therefore only has to know the address of a follower

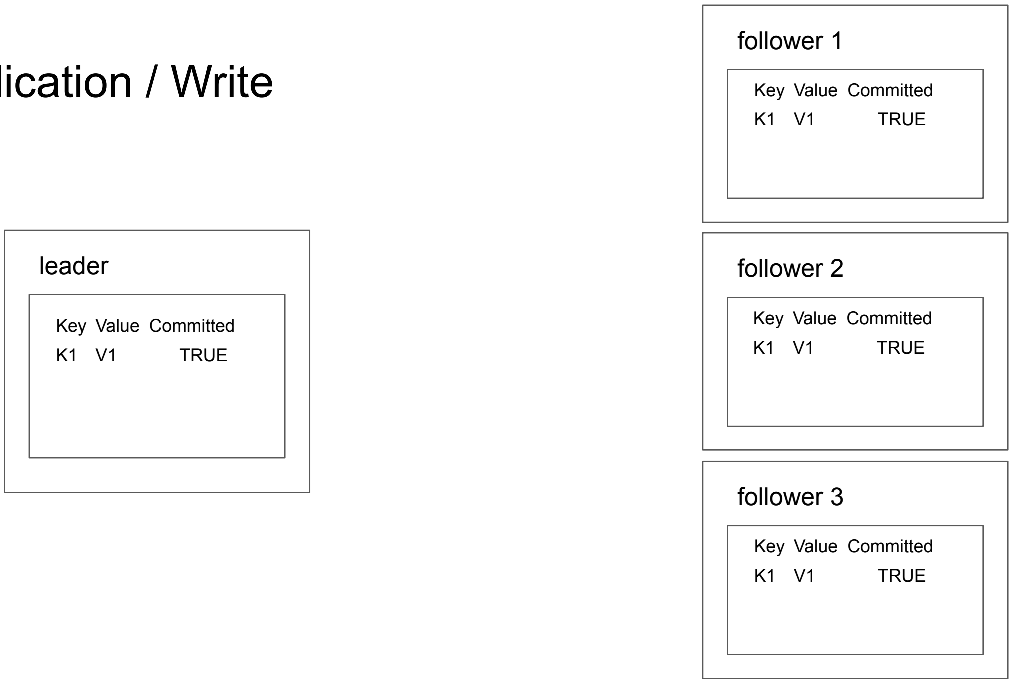
- Why does the follower not reply immediately? It cannot guarantee that the value is up to date (as we will see on writing)
- An optimization that I did not perform could be to only check if the hash is current to save on bandwidth costs for large objects



Code Review

- Read
  - `routes.go`
    - `handleRead`
      - If leader: reply immediately
      - If follower: proxy request and reply

Log Replication / Write



leader

Key	Value	Committed
K1	V1	TRUE

follower 1

Key	Value	Committed
K1	V1	TRUE

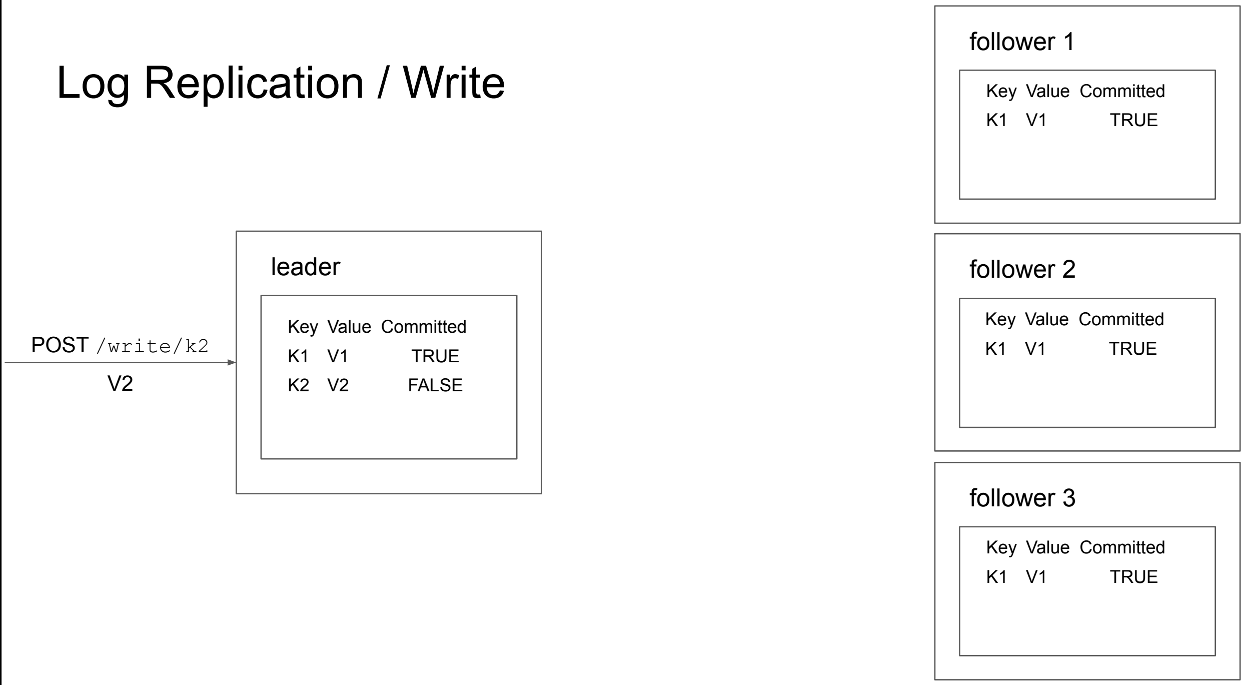
follower 2

Key	Value	Committed
K1	V1	TRUE

follower 3

Key	Value	Committed
K1	V1	TRUE

Log Replication / Write



leader

Key	Value	Committed
K1	V1	TRUE
K2	V2	FALSE

follower 1

Key	Value	Committed
K1	V1	TRUE

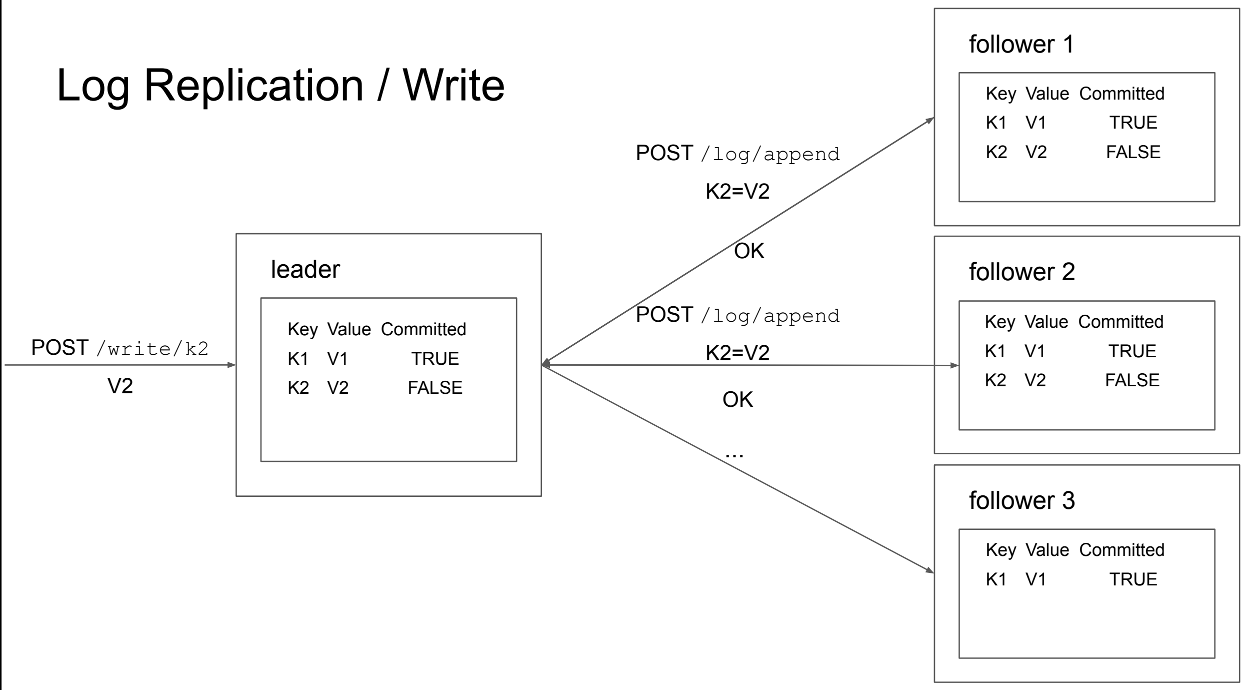
follower 2

Key	Value	Committed
K1	V1	TRUE

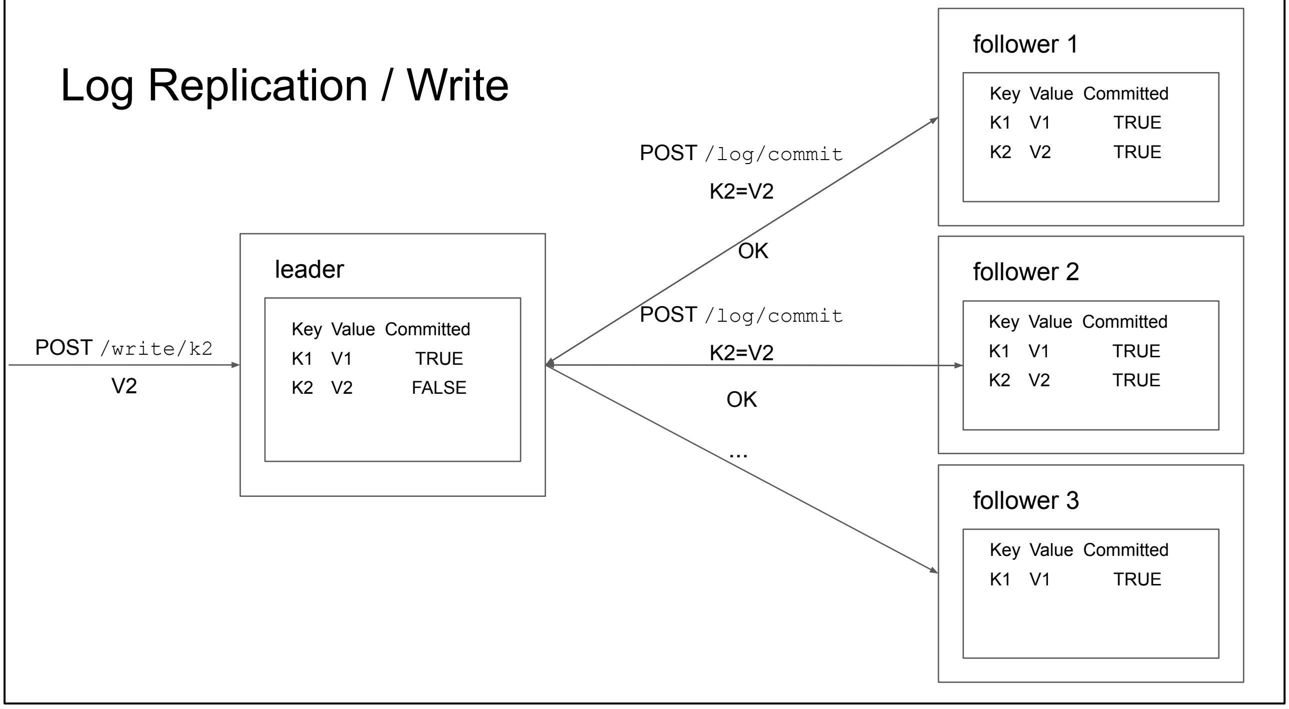
follower 3

Key	Value	Committed
K1	V1	TRUE

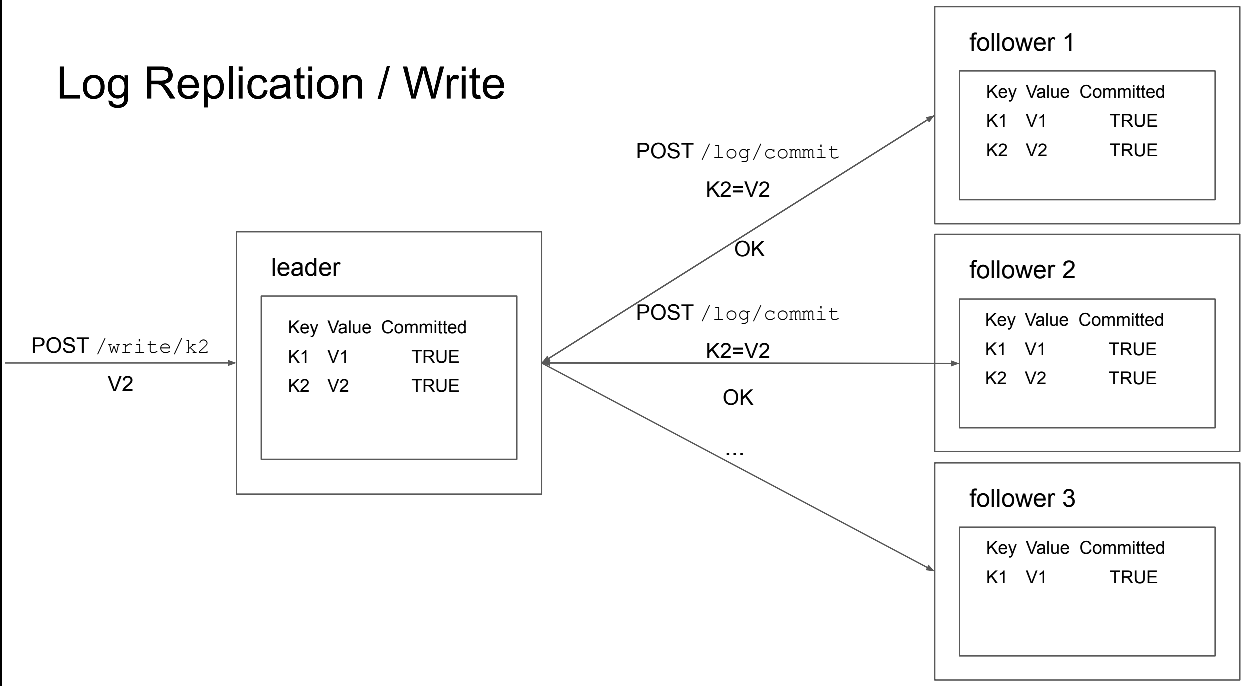
Log Replication / Write



Log Replication / Write



Log Replication / Write



leader

Key	Value	Committed
K1	V1	TRUE
K2	V2	TRUE

follower 1

Key	Value	Committed
K1	V1	TRUE
K2	V2	TRUE

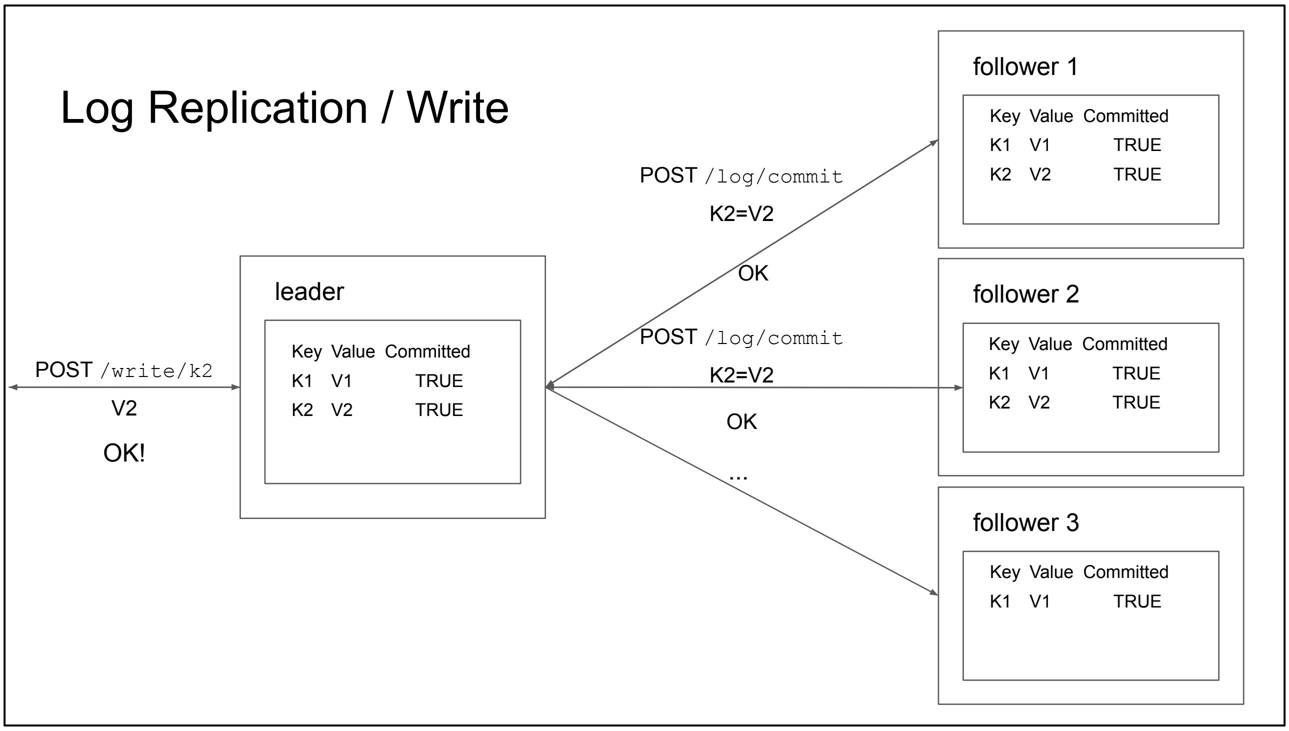
follower 2

Key	Value	Committed
K1	V1	TRUE
K2	V2	TRUE

follower 3

Key	Value	Committed
K1	V1	TRUE

Log Replication / Write



- An addition for the leader election process is in order!
    - If a follower receives a poll request with a last log hash smaller than their own, it will vote no
    - Follower three will be a candidate, but not win the election
    - Upon election, the leader has to resync all of the logs between nodes (speed up follower 3, delete uncommitted changes)

Next Steps

- Delete implementation
- More testing (leader election and log replication especially)
- Try in a cluster environment
- Benchmarking

Review and Conclusion

- Main assumptions:
  - broadcastTime << electionTimeout << failureTime
  - only one node failure at a time
  - All nodes are truthful
- Leader node can become bottleneck of cluster
  - Heartbeats, Log Replication, Read Requests

- Keeping distributed state up-to-date is hard with strong guarantees!
- Testing a distributed system is a pain
- Good for heavy-read, low-write workloads with a reasonable number of nodes

References

- https://etcd.io/
- https://github.com/etcd-io/etcd
- https://en.wikipedia.org/wiki/Container_Linux#ETCD
- https://raft.github.io/ (contains links to the papers)
- https://en.wikipedia.org/wiki/Raft_(algorithm)

My code is available on GitHub.

Thank you