
CERN

quarkdb

A highly-available backend for the EOS namespace

Georgios Bitzes

The motivation for quarkdb

- The current solution for the EOS namespace stores *all metadata in-memory*

- This method has reached its *scalability limits*
  - Production instances requiring special machines with hundreds of gigabytes of RAM
  - All metadata must be loaded into memory on boot, often taking *+1h*

Project goals

- A database able to hold large amounts of data
  - in the order of TBs
- Redis protocol with a small subset of redis commands supported
  - mostly string, hash, and set operations
- *High availability*

quarkdb design

- *rocksdb* as the storage backend, a key-value store by Facebook
- Translation of redis commands into rocksdb key-value transactions
- *Raft consensus algorithm* for replication and high-availability

rocksdb

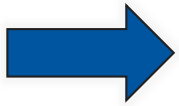
- Persistent key-value store
  - log-structured merge tree in the back
- Embeddable: link to your own binary, and you have a database
- Open-source with a permissive license (BSD), actively developed by Facebook
- Designed for and proven to hold datasets larger than RAM size
- Optimized specifically for SSD storage

Redis command translation

Redis command

rocksdb

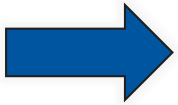
HSET myhash field contents



Key descriptor: “**dmyhash**” =>
“**This key is a hash, current size is 5**”

“**bmyhash#field**” => “**contents**”

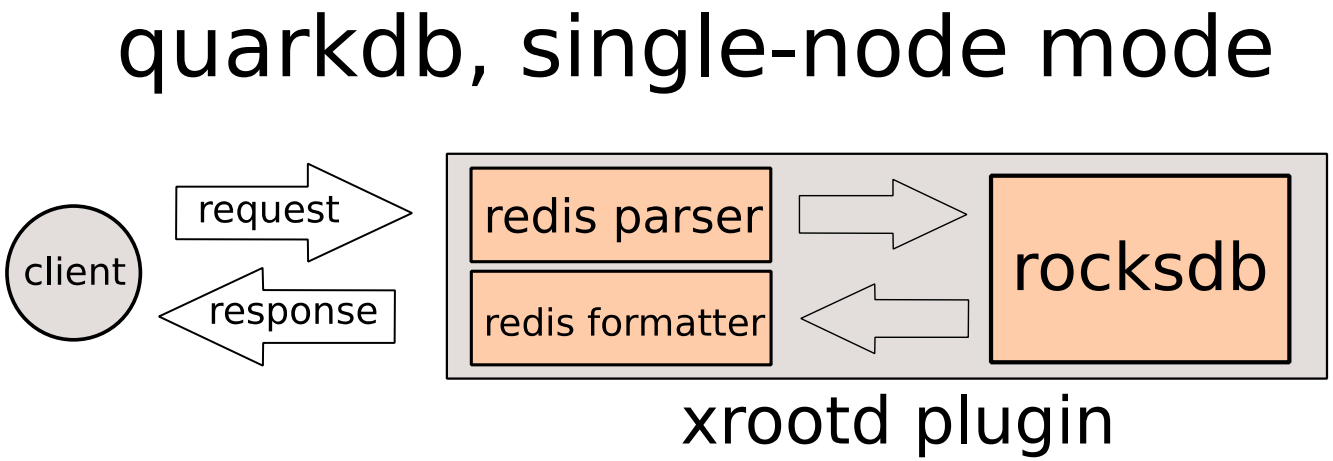
SADD myset element



Key descriptor: “**dmyset**” =>
“**This key is a set, current size is 8**”

“**cmyset#element**” => “**1**”

Standalone mode overview



The need for high-availability

- eos has become critical for data at CERN
- MGM loss means long downtime, great disruption
- Ideally:
  - Transparent failover, *no service interruption*
  - No single point of failure, now possible since database is separate from MGM

Replication

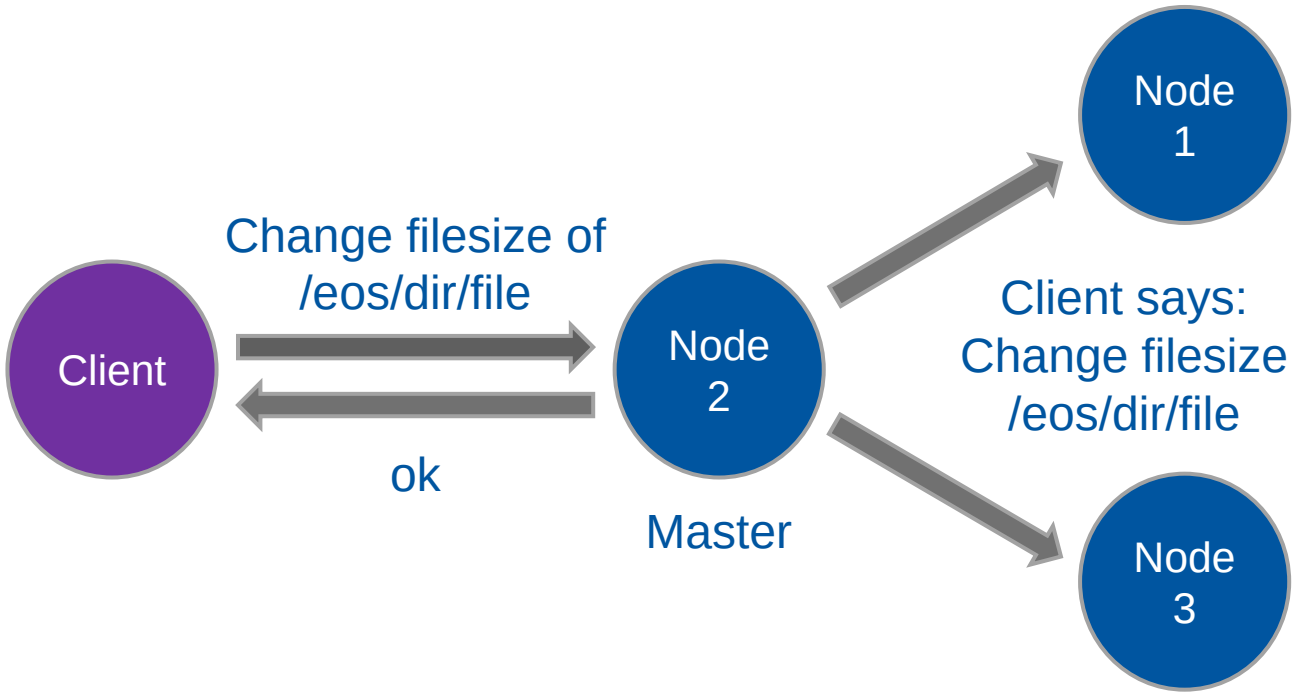
- Need to cluster *multiple replicated nodes* for fault tolerance
- Very tricky to ensure *safety* and *consistency* in a *distributed database*
  - Nodes could fail **in the middle** of receiving updates
  - **Network partitions:** different nodes might have conflicting views
- A solution to this problem already exists; distributed consensus

Distributed consensus

- A way to have multiple nodes agree on something
- Several algorithms and methods exist
- We picked the Raft consensus algorithm – offers *strong consistency* semantics

Master – slave replication

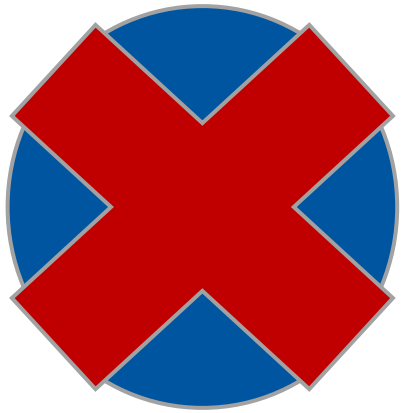
One of the nodes is elected to become the *master* (or *leader*)

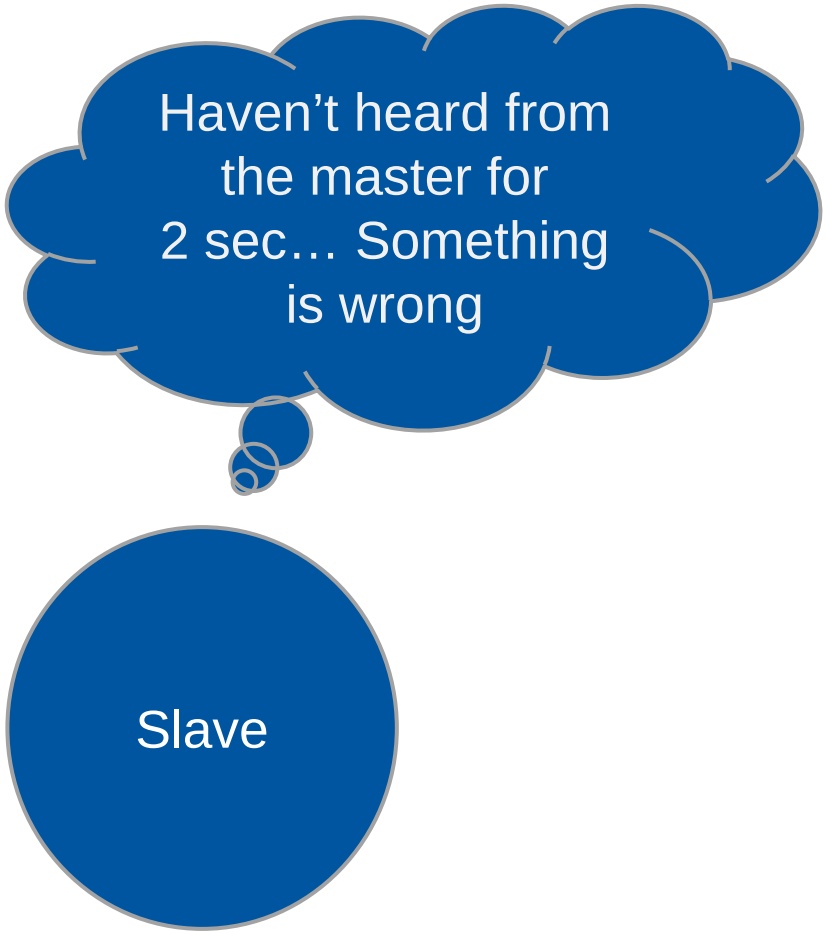


Master election

- The master sends regular heartbeats to all slaves
- If a slave stops receiving heartbeats, it assumes master failure and triggers an **election**
- An election is won if a node receives positive votes from at least a **majority** of the cluster

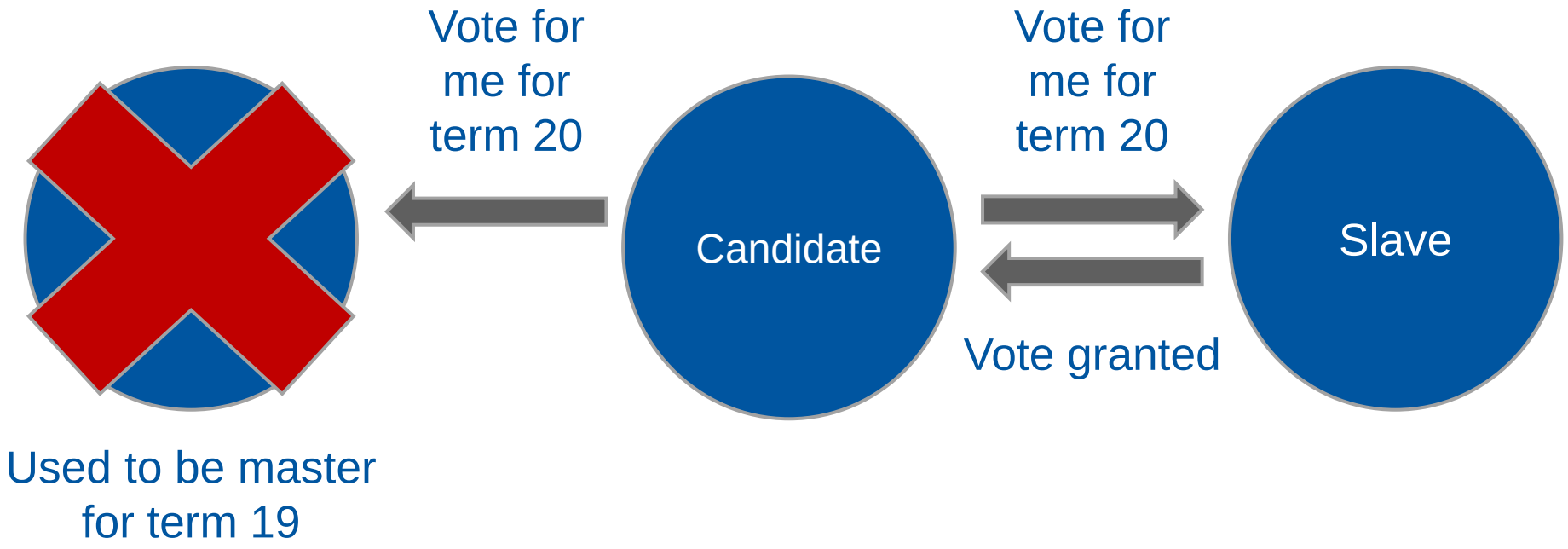
Heartbeats




Haven’t heard from the master for 2 sec… Something is wrong
Slave

Master election (2)

A successful election: 2 out of 3 nodes agree on the new master

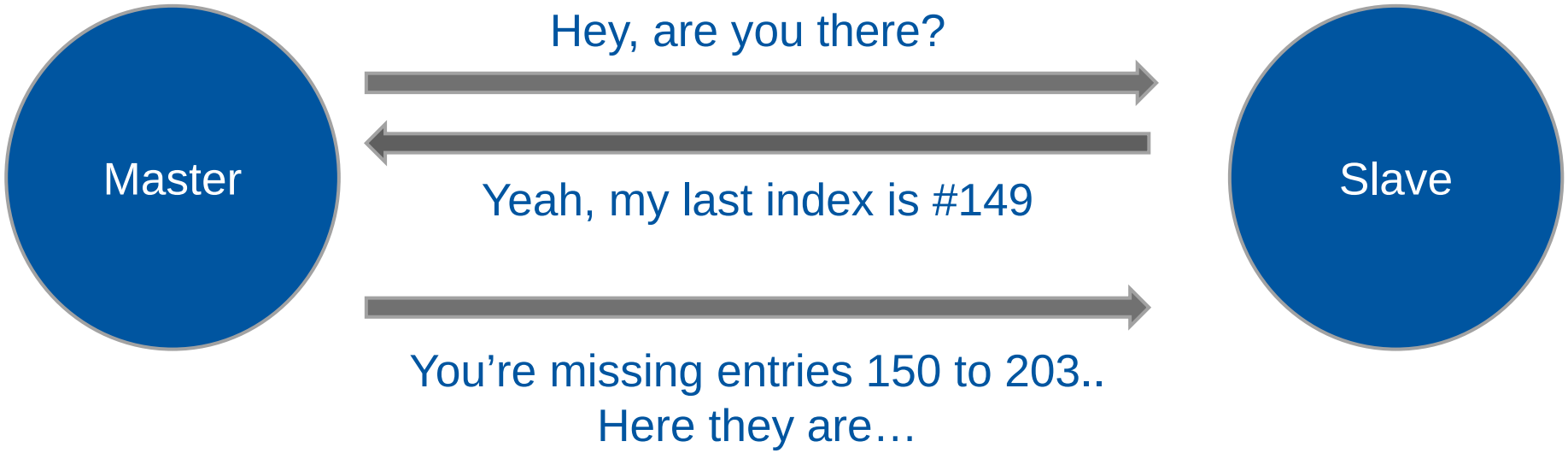


Log replication

- One of the slaves goes offline for 10 minutes – how to bring it up-to-date?
- Record *all writes* into an indexed log, and replicate it

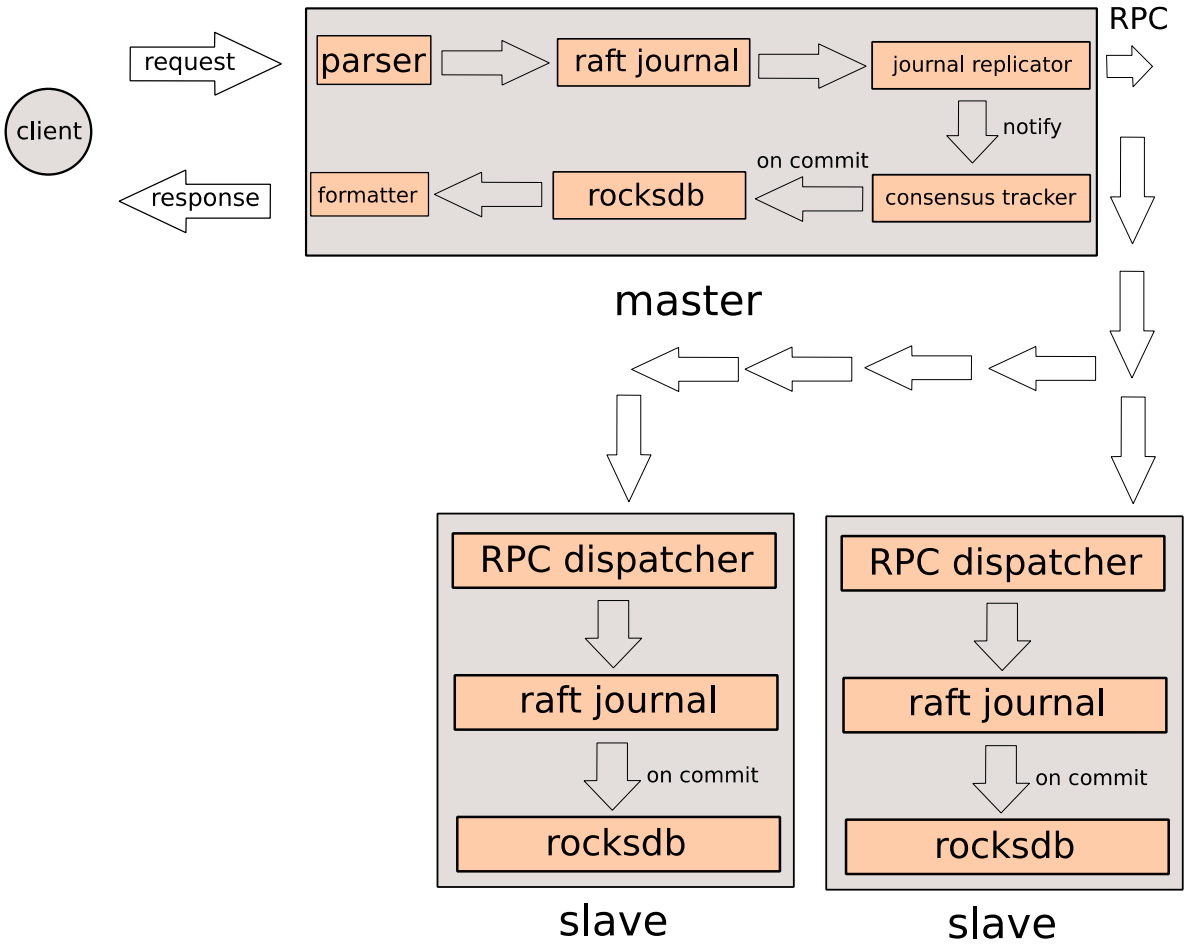
Index	Term	Contents
0	1	SET food pizza
1	1	SET language c++
2	1	SET food pickles
3	5	SET answer_to_life 42
…		….

Log replication (2)



quarkdb: high-level overview

quarkdb, distributed mode



Consistency guarantees

- quarkdb is a *strongly consistent* database (CP from CAP theorem)

- **Linearizability:** after a client receives an ACK to a write, all future reads (from *any* client) are guaranteed to return *that value, or a future one*.
  - *even if the master crashes right after the ACK*

What's been done so far

- Implemented the *raft algorithm* – replication, master election
- Distributed mode is already *fully functional*
- *Membership changes* – ability to add / remove nodes on-the-fly
- *Trimming* of the raft journal, so it doesn't grow indefinitely
- Automatic "resilvering" – bringing a node that just joined the cluster up-to-date

Stability and testing

- Extensive testing: unit, functional
  - Running the test suite stresses all quarkdb components and capabilities, including:
    - ✓ redis protocol parsing
    - ✓ master election machinery
    - ✓ Journal entry replication
    - ✓ Resolving conflicting journal entries
    - ✓ client request servicing
    - ✓ Pipelined writes over multiple connections
    - ✓ Journal trimming
    - ✓ Membership updates

Thanks

- https://gitlab.cern.ch/eos/quarkdb
- Current status: **~8k** lines of code
  - including tests, tools
  - excluding dependencies

- Raft paper
  - https://raft.github.io/raft.pdf
- Raft visualization
  - https://thesecretlivesofdata.com/raft/

Questions, comments?