
CERN

EOS namespace using XRootD with Redis plugin

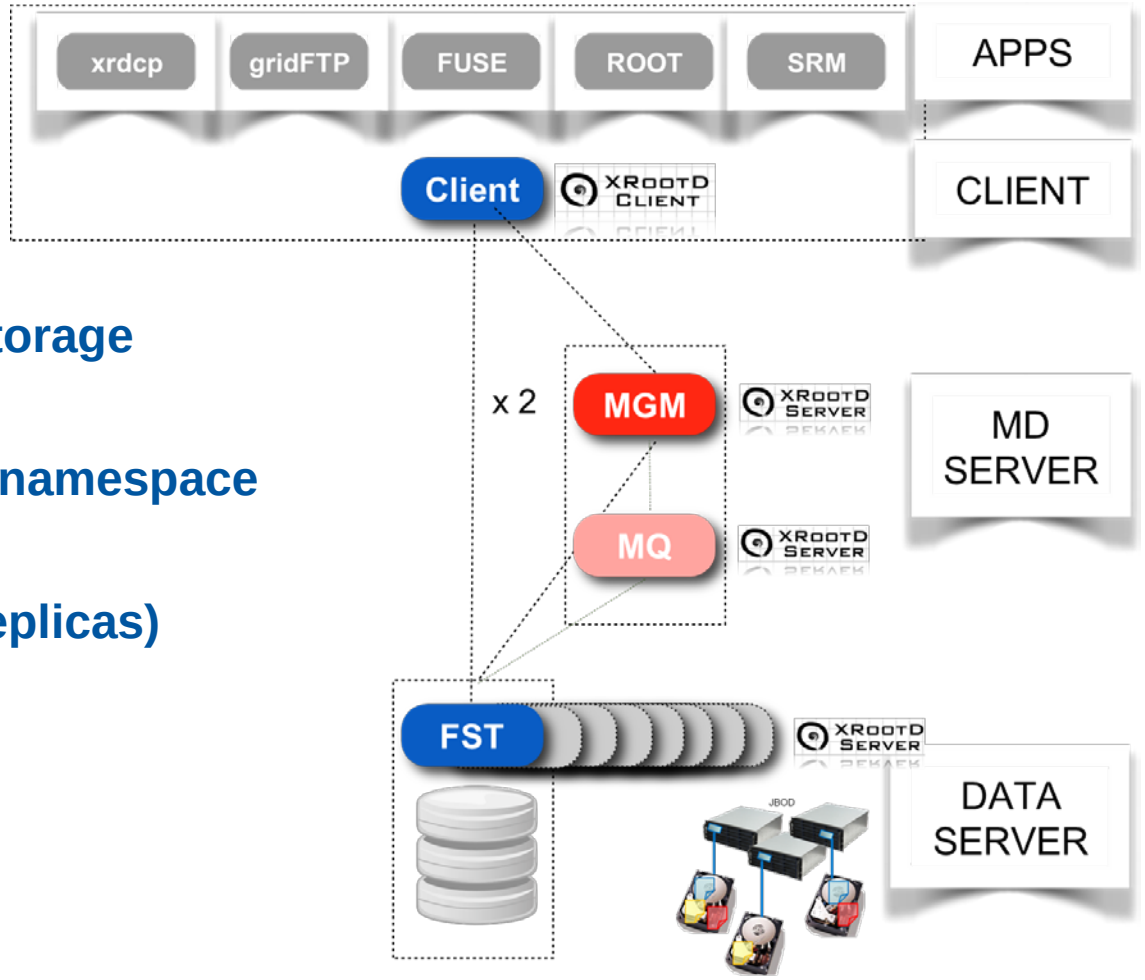
Elvin Sindrilaru on behalf of the
EOS team and IT Storage Group

XRootD Workshop Tokyo

Outline

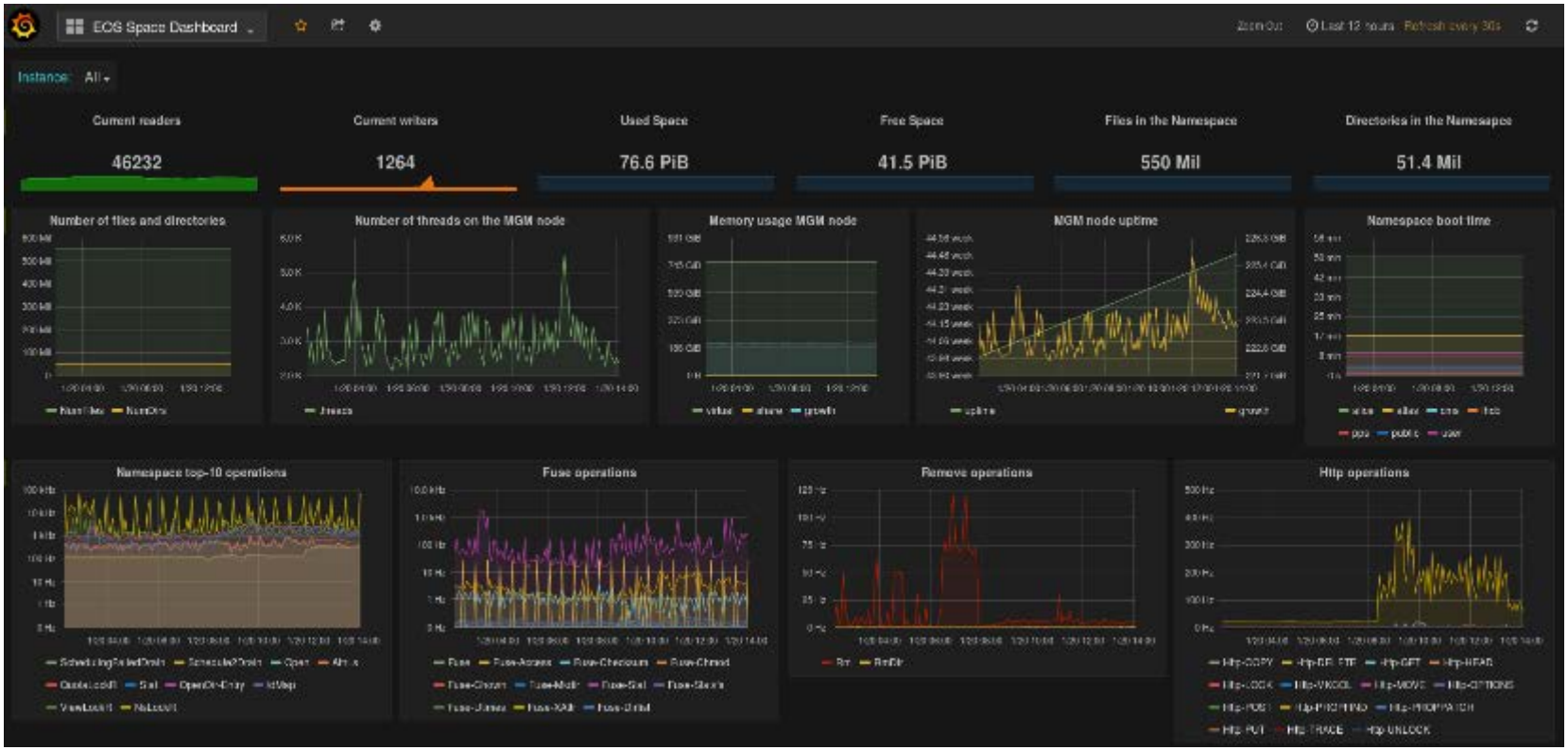
- EOS architecture
- Namespace architecture and bottlenecks
- Namespace on Redis
- XRootD with Redis and rocksdb as backend
- Ensuring HA using Raft
- Plans for the future

EOS architecture



- Disk only physics file storage
- In memory hierarchical namespace
- File layouts (default 2 replicas)
- Physics data & others
- Low latency access

What scale are we talking about?



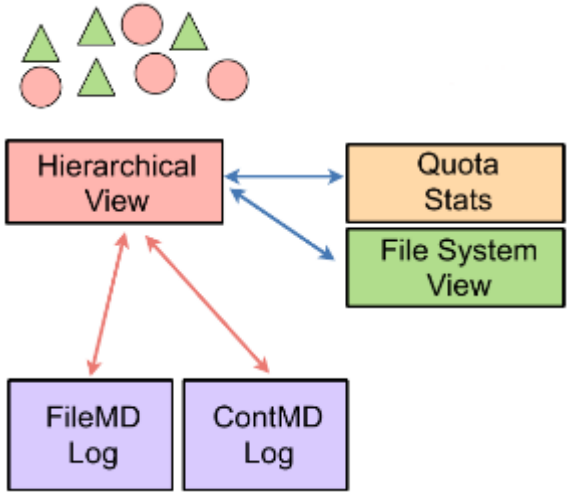
- Biggest instance **ALICE**
  - **~ 375M files** and ~ **100K directories**
  - Using ~**390GB** of RAM
  - Boot time **~60 min**

What is the EOS namespace?

- C++ library used by the EOS MGM node
- Provides API for dealing with hierarchical collections of files
- It's single threaded with fast in-memory operations

- **Filesystem elements**
  - Containers & files

- **Views**
  - Aggregate info about filesystem elem.
  - E.g QuotaView, FileSystemView etc.



- **Persistence objects**
  - Objects responsible for reading and storing filesystem elements
  - Implemented as binary change-logs

Namespace architectures pros/cons

- **Pros:**
  - Using hashes all in memory → extremely fast
  - Every change is logged → low risk of data loss
  - Views rebuilt at each boot → high consistency

- **Cons:**
  - For big instances it requires **a lot** of RAM
  - Booting the namespace from the change log takes long

What's the target?

- Still fast and consistent
- Scale-out the namespace → avoid one machine's memory limitation
- Reduce the boot time i.e. service down time
  - By **partitioning the namespace** i.e. faster boot time for each partition
  - Use master – slave **replication** to ensure high availability

EOS Namespace Interface

- Prepare the setting for different namespace implementations
- Abstract a **Namespace Interface** to avoid modifying other parts of the code

- **EOS citrine 4.0.***

  - **Plugin manager** – able not only to dynamically load but also stack plugins if necessary

  - **libEosNsInMemory.so** – the original in-memory namespace implementation

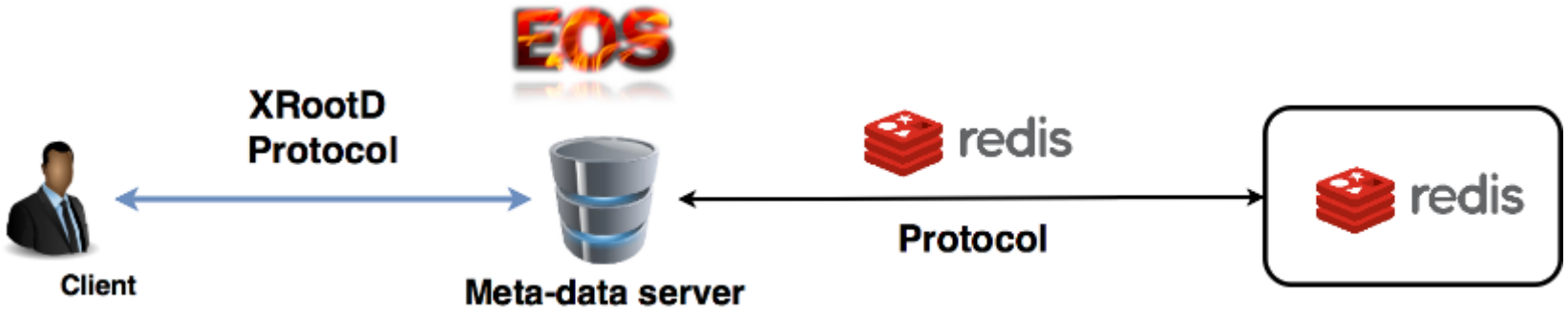
  - **libEosNsOnRados.so** – not existing but can be imagined based on CEPH

  - **libEosNsOnFilesystem.so** – not existing based on a Linux filesystem

Why Redis?

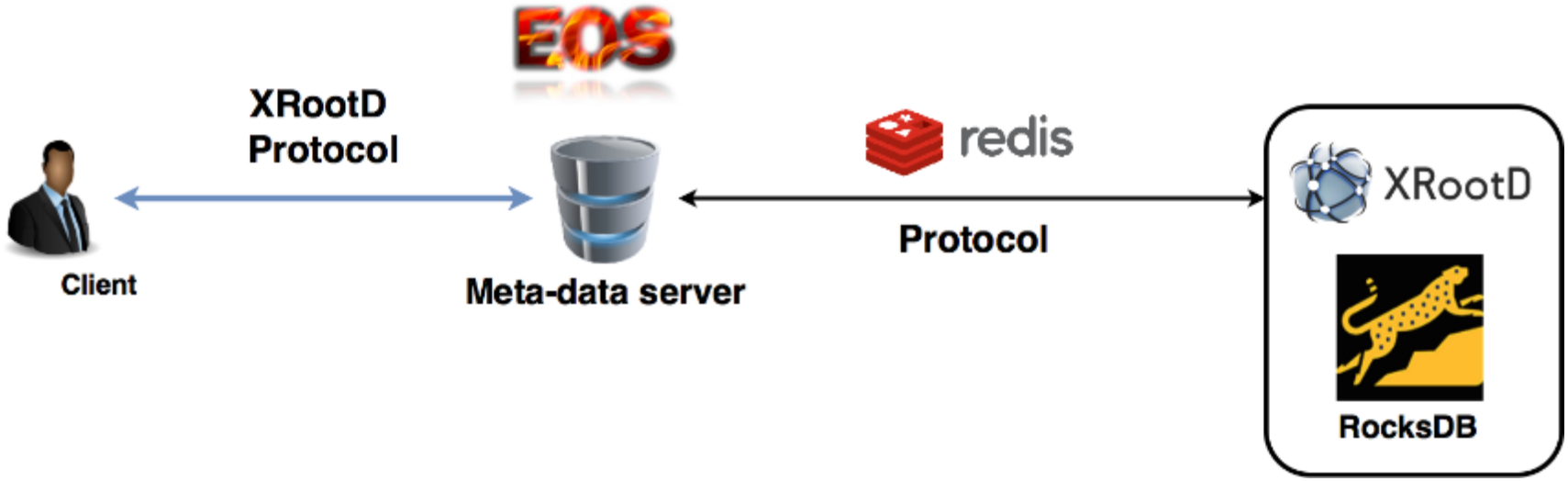
- **Redis** – in-memory **data structure store**
- Separate data from the application logic and user interface
- Supports various data structures: strings, hashes, lists, sets, sorted sets etc.

- Namespace implementation: **libEosOnRedis.so**
- **Light-weight EOS MGM** node that can easily be restarted or updated



XRootD and Redis

- Replace Redis backend with XRootD
- Implemented as an XRootD **protocol plugin** – to be contributed upstream
- XRootD can use **RocksDB** as persistent key-value store



rocksdb

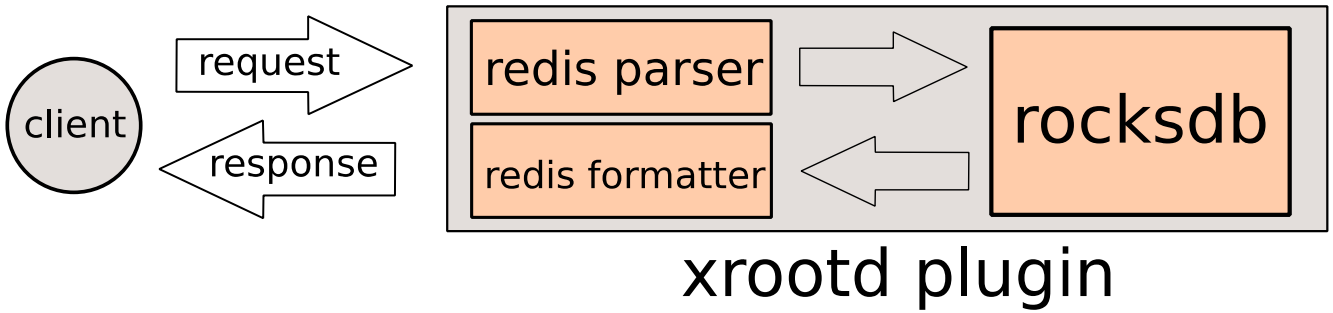
- **Persistent key-value store**
  - uses log-structured merge trees
- **Embeddable**: link to your own binary, and you have a database
- Open-source with a permissive license (BSD), actively developed by Facebook
- Designed for and proven to hold datasets **larger than RAM size**
- Optimized specifically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

rocksdb with a Redis interface

- Implemented a subset of the redis protocol with rocksdb as backend – **XRootD protocol plugin**

- **Preliminary benchmarks:** +100k of *pipelined* writes / sec. on an SSD machine (single-node, non-replicated mode)

quarkdb, single-node mode



The need for high-availability

- **EOS** has become **critical** for data at CERN
- In an event of MGM loss (unrecoverable disk corruption)
  - Long downtime, loss of recent writes

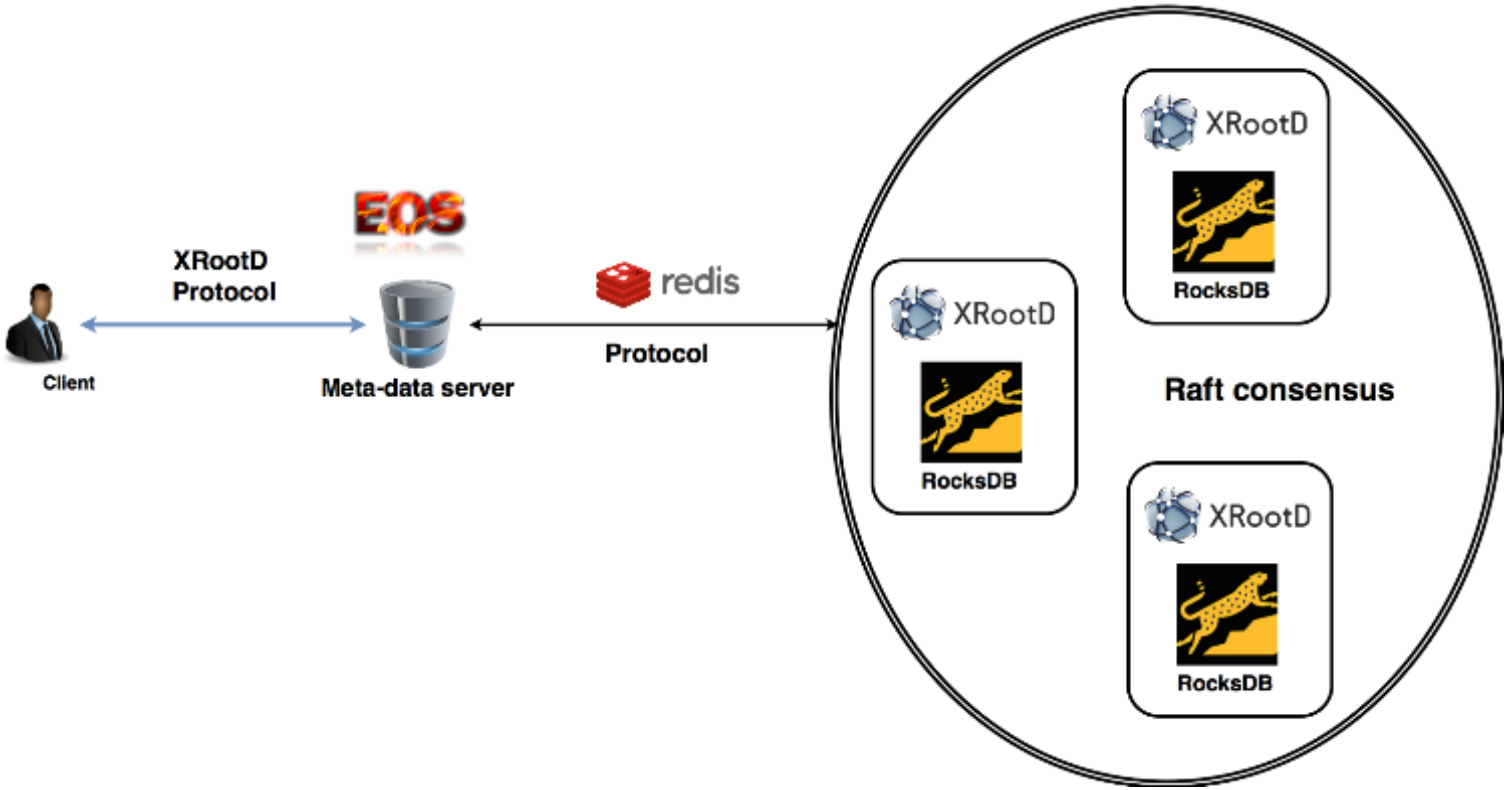
- **Ideally**:
  - Transparent failover, no service interruption
  - The database becomes **separate** from the MGM

High availability

- 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 could mean different nodes have conflicting views of the database
  - How to select which node receives writes? What happens when it fails?
- **Solution** to this problem already exists: raft (or paxos) algorithm

Namespace HA

- Ensure high-availability using the **Raft consensus algorithm**



Raft algorithm

- A single node is selected to be the master
- All writes from clients are sent to the master

- **Semi-synchronous replication**: client receives “OK” as soon as the write is replicated on a majority, or *quorum* of nodes

- Cluster is available as long as a quorum of nodes are functional
  - floor(n/2)+1: 2 out of 3, 3 out of 5, 4 out of 7

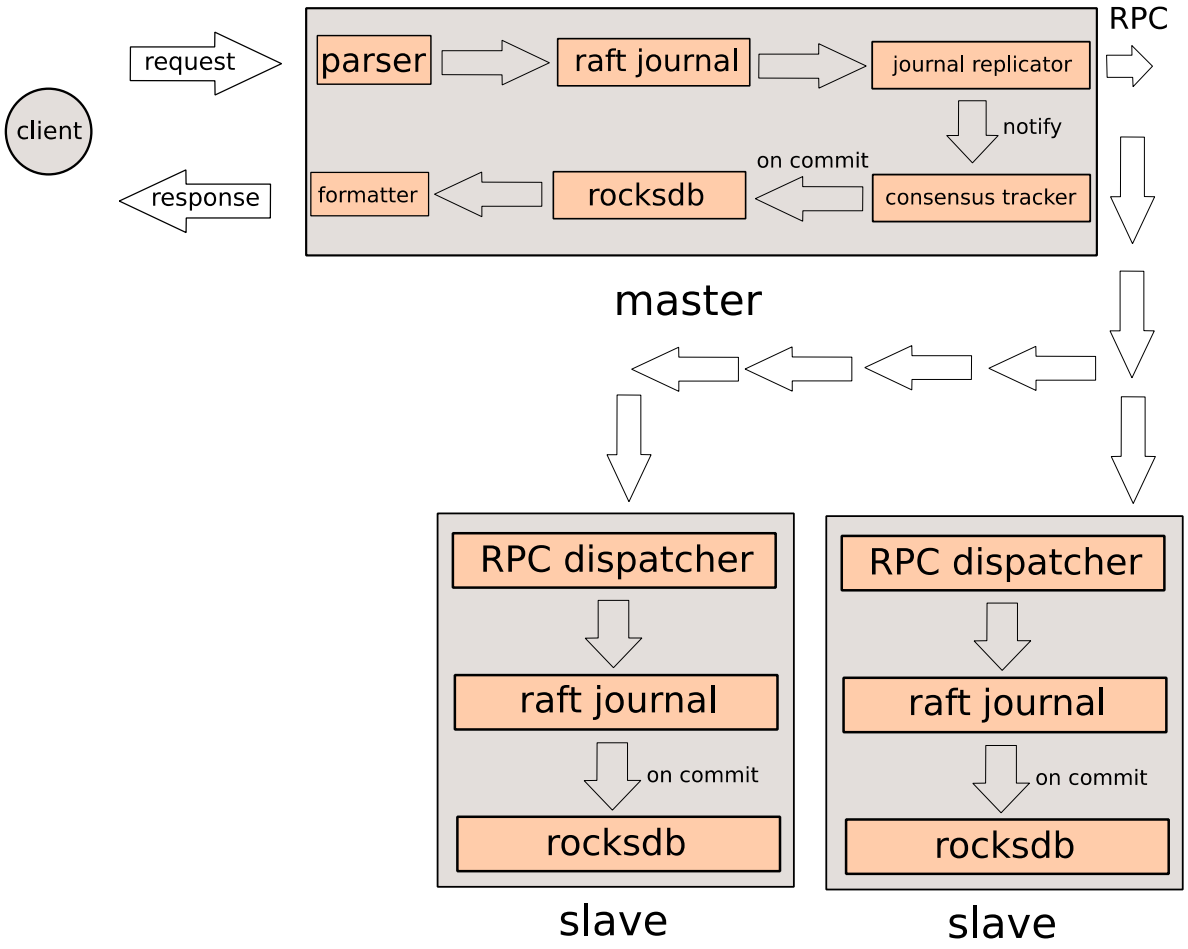
Raft algorithm (2)

- All writes go through the raft log: a **linear journal replicated** by every node
- A journal entry is irrevocably committed once **replicated on a quorum of nodes**
- On failure of the master node and assuming we still have a quorum:
  - The remaining slaves hold an **election** on who becomes the next master
  - The election machinery ensures that only nodes which have *all* the committed entries can succeed in becoming a master

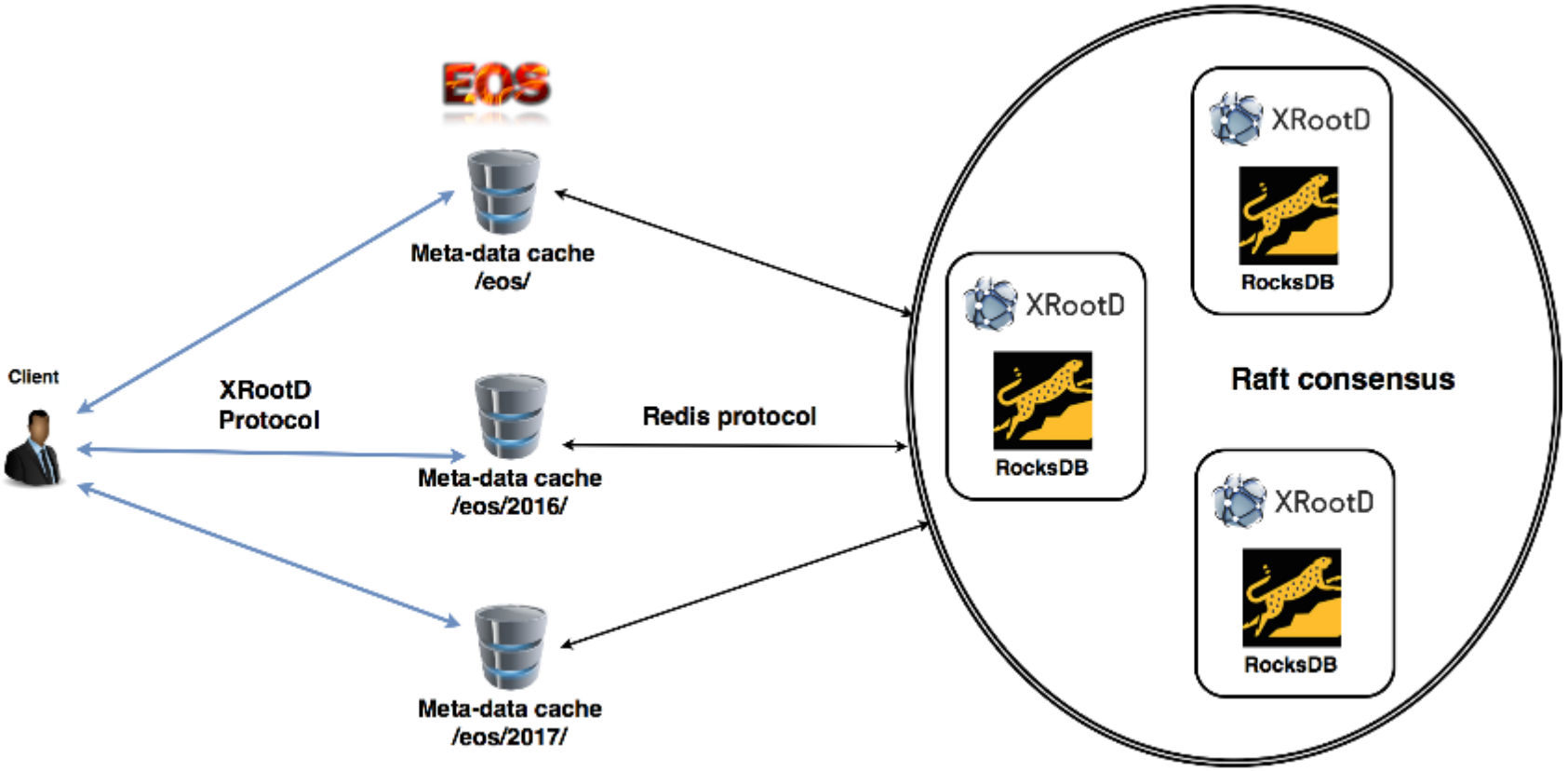
- **Preliminary benchmarks:** 10-20khz *pipelined* writes in distributed mode
- More on Raft
  - https://raft.github.io/
  - http://thesecretlivesofdata.com/raft/

quarkdb: high-level overview

quarkdb, distributed mode



EOS NS – the future



Summary

- **EOS Namespace**
  - Learn from XRootD's design
    - Use **plug-ins**
    - Define a stable **interface**
  - Need to meet **scalability and growth** demands
  - Use the **Redis protocol** for basic key-value operations
  - Prototype on top of **Redis/XRootD** and HA using **Raft**
  - Currently being deployed on pre-production instances using **CentOS7**

www.cern.ch