
KubeCon
CloudNativeCon
OPEN SOURCE SUMMIT
China 2019

Raft in etcd

Jingyi Hu

Introduction

- Who am I
  - github.com/jingyih
  - A maintainer of etcd (https://github.com/etcd-io/etcd)
- Hopefully would be helpful for
  - Developers who are new to etcd Raft package. This talk provides a starting point for reading the source code.
  - Developers who want to understand the relationship between etcd and Raft. This talk provides a brief description on how etcd uses the Raft package.
  - Developers who want to import and use Raft package in their own project.

Agenda

- Raft Recap
- Raft as part of etcd
- Raft Implementation Details
- Roadmap

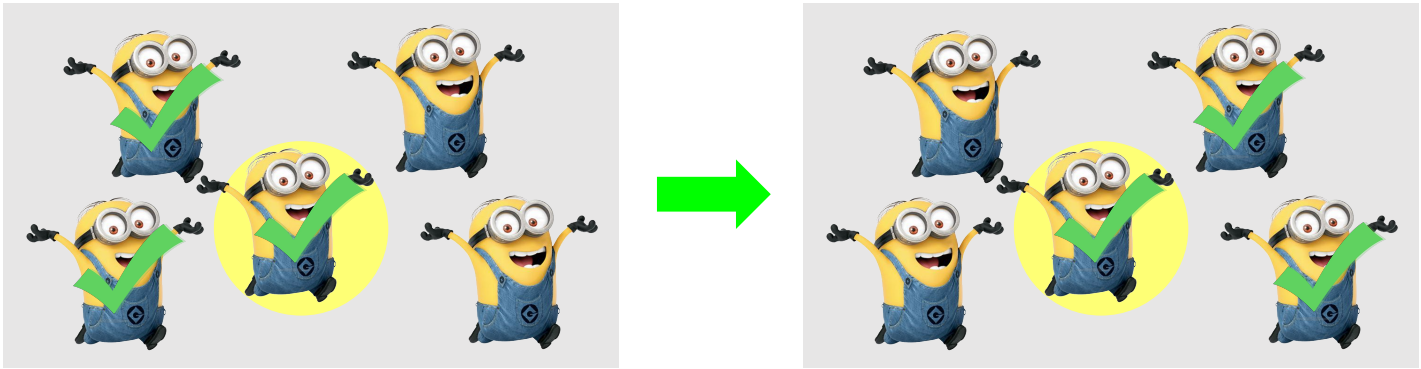
Raft Recap

- Consensus and Quorum
- Replicated state machine
- Leader election
- Log replication

Raft Recap

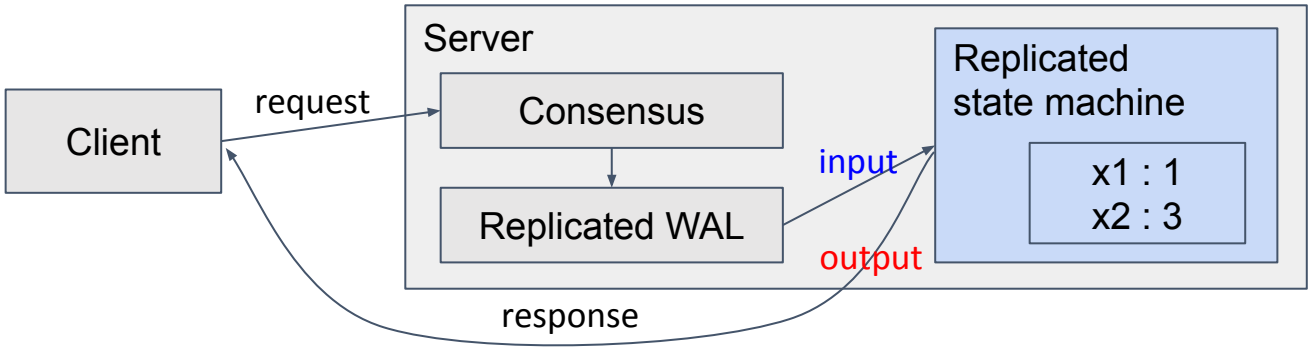
- Consensus and Quorum
  - If Q1 and Q2 are quorum, their intersection is not empty.
  - Cluster can make progress as long as consensus among quorum - high availability / fault tolerance.

(Image source: http://pngimg.com/imgs/heroes/minions/)



Raft Recap

- Replicated state machine
  - Same initial state
  - Same input sequence - replicated write ahead log (WAL)
  - Results in same output and (internal) state transition.



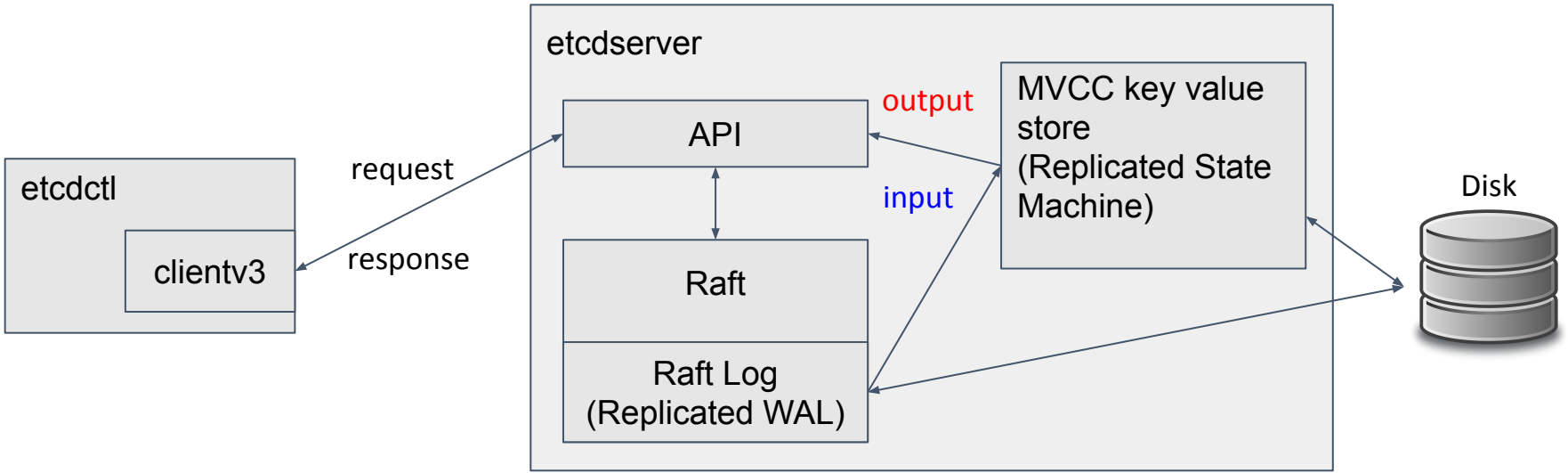
Raft Recap

- Leader election
  - Candidate, Follower, Leader
  - Term
  - Election
  - Heartbeat

Raft Recap

- Log Replication
  - Only leader manages the replicated logs.
  - Leader only append to log.
  - Leader keeps trying to replicate its logs to followers.
  - Committed index
  - Applied index (always smaller than committed index)

Raft in etcd



We will revisit this graph (with more details) after we talked about the Raft implementation.

Raft Implementation

- Minimalistic design for flexibility, deterministic and performance
  - Raft package does not implement network transport between peers
  - Raft package does not implement storage to persist log and state

Raft Implementation

- Raft is modeled as a state machine
  - State
  - Input, output
  - Transition between states

Raft Implementation

- State

```
type raft struct {
   id   uint64
   Term uint64
   Vote uint64
   ...
   raftLog *raftLog // replicated WAL
   ...
   state StateType // Leader, Candidate, Follower
   ...
   step stepFunc
   ...
}


type stepFunc func(r *raft, m pb.Message) error

func stepLeader(r *raft, m pb.Message) error

func stepFollower(r *raft, m pb.Message) error

func stepCandidate(r *raft, m pb.Message) error
```

- Input

```
type Message struct {
   Type              MessageType
                     // e.g. MsgVote, MsgApp
   To                uint64
   From              uint64
   Term              uint64
   LogTerm           uint64
   Index             uint64
   Entries           []Entry // raft log entries
   ...
}
```

Raft Implementation

- Output

```
type Ready struct {
   pb.HardState
   // {Term, Vote, Committed Index}, to be saved to
staroge
   ...
   Entries []pb.Entry
   // Raft entries to be saved to storage
   ...
   CommittedEntries []pb.Entry
   // Raft entries ready to be applied to
   // replicated state machine
   ...
   Messages []pb.Message
   // messages to be sent to peers
   ...
}
```

- State Transition

```
// Step advances the Raft state machine using the given
message.
Step(ctx context.Context, msg pb.Message) error


func (r *raft) Step(m pb.Message) error {
   ...
   // Common logic for leader, candidate, follower:
   // 1. Handle m.Term, which may result in leader stepping
down to a follower
   // 2. Handle campaign and voting

   r.step(m)
   // stepLeader
   // stepCandidate
   // stepFollower
}
```

Raft Implementation

- Example: Propose a change to replicated state machine (KV store)

A

```
Message {
    Type:    pb.MsgProp,
    Entries: []pb.Entry{{Data: data},
}
```

Follower

```
func (n *node) Propose(ctx context.Context, data []byte)
error {
    return n.stepWait(ctx,
        pb.Message{
            Type:    pb.MsgProp,
            Entries: []pb.Entry{{Data: data}},
        })
}
```

Raft Implementation

- Example: Propose a change to replicated state machine (KV store)

A

```
Message {
    Type:    pb.MsgProp,
    Entries: []pb.Entry{{Data: data},
}
```

Follower

B

```
Message {
    Type:    pb.MsgProp,
    To:      r.lead,
    From:    r.id,
    Entries: []pb.Entry{{Data: data},
}
```

```
func stepFollower(r *raft, m pb.Message) error {
    switch m.Type {
    ...
    case pb.MsgProp:
        if r.lead == None {
            return ErrProposalDropped
        }
        m.To = r.lead
        r.send(m)
    }
}


func (r *raft) send(m pb.Message) {
    m.From = r.id
    r.msgs = append(r.msgs, m)
}


func newReady(r *raft, ...) Ready {
    rd := Ready{
        Messages: r.msgs,
    }
    return rd
}
```

Raft Implementation

- Example: Propose a change to replicated state machine (KV store)

B

```
Message {
    Type:    pb.MsgProp,
    To:      r.lead,
    From:    r.id,
    Entries: []pb.Entry{{Data: data},
}
```

Leader

C

```
Message {
    Type:    pb.MsgApp,
    To:      peer 1 id,
    From:    r.id,
    Entries: ents,
    Commit:
r.raftLog.committed,
    ...
}
```

D

```
Message {
    Type:    pb.MsgApp,
    To:      peer 2 id,
    From:    r.id,
    Entries: ents,
    Commit:
r.raftLog.committed,
    ...
}
```

```
func stepLeader(r *raft, m pb.Message) error {
    switch m.Type {
    ...
    case pb.MsgProp:
        if !r.appendEntry(m.Entries...) {
            return ErrProposalDropped
        }
        r.bcastAppend()
    }
}


For each peer:
func (r *raft) maybeSendAppend(to uint64, ...) bool {
    m := pb.Message {
        Type:    pb.MsgApp,
        To:      to, // peer id
        Entries: ents, // from last matched entry to latest
        Commit:  r.raftLog.committed,
        ...
    }

    r.send(m)
}
```

Raft Implementation

- Example: Propose a change to replicated state machine (KV store)

C

```
Message {
    Type:    pb.MsgApp,
    To:      peer 1 id,
    From:    r.id,
    Entries: ents,
    Commit:  r.raftLog.committed,
    ...
}
```

Follower

E

```
Message {
    Type:    pb.MsgAppResp,
    To:      lead id,
    From:    r.id,
    Index:   mlastIndex,
}
```

```
func stepFollwer(r *raft, m pb.Message) error {
    switch m.Type {
    ...
    case pb.MsgApp:
        ...
        r.lead = m.From
        r.handleAppendEntries(m)
    }
}

func (r *raft) handleAppendEntries(m pb.Message) {
    ... // do not need to append if already has more updated
log entries, and report my progress to leader
    if mlastIndex, ok := r.raftLog.maybeAppend(m.Index,
m.LogTerm, m.Commit, m.Entries...); ok {
        r.send(pb.Message{
            To:    m.From, // this is lead id
            Type:  pb.MsgAppResp,
            Index: mlastIndex,
        })
    } else {...} // reject the log entries from leader
}
```

Raft Implementation

- Example: Propose a change to replicated state machine (KV store)

E

```
Message {
    Type:    pb.MsgAppResp,
    To:      lead id,
    From:    r.id,
    Index:   mlastIndex,
}
```

Leader

F, G

```
Message {
    Type:    pb.MsgApp,
    To:      peer 1 id,
    From:    r.id,
    Entries: ents,
    Commit:  r.raftLog.committed,
    // Commit is updated
    ...
}
```

```
func stepLeader(r *raft, m pb.Message) error {
    switch m.Type {
    ...
    case pb.MsgAppResp:
        if m.Reject {...} else {
            ...
            if pr.maybeUpdate(m.Index) {
                r.bcastAppend()
            }
            ...
        }
    }
}

func (r *raft) handleAppendEntries(m pb.Message) {
    ...
    // maybeAppend may advance the committed index in
follower's raft log
    if mlastIndex, ok := r.raftLog.maybeAppend(m.Index,
m.LogTerm, m.Commit, m.Entries...); ok {...}
    ...
}
```

Raft Implementation

- Example: Propose a change to replicated state machine (KV store)

F

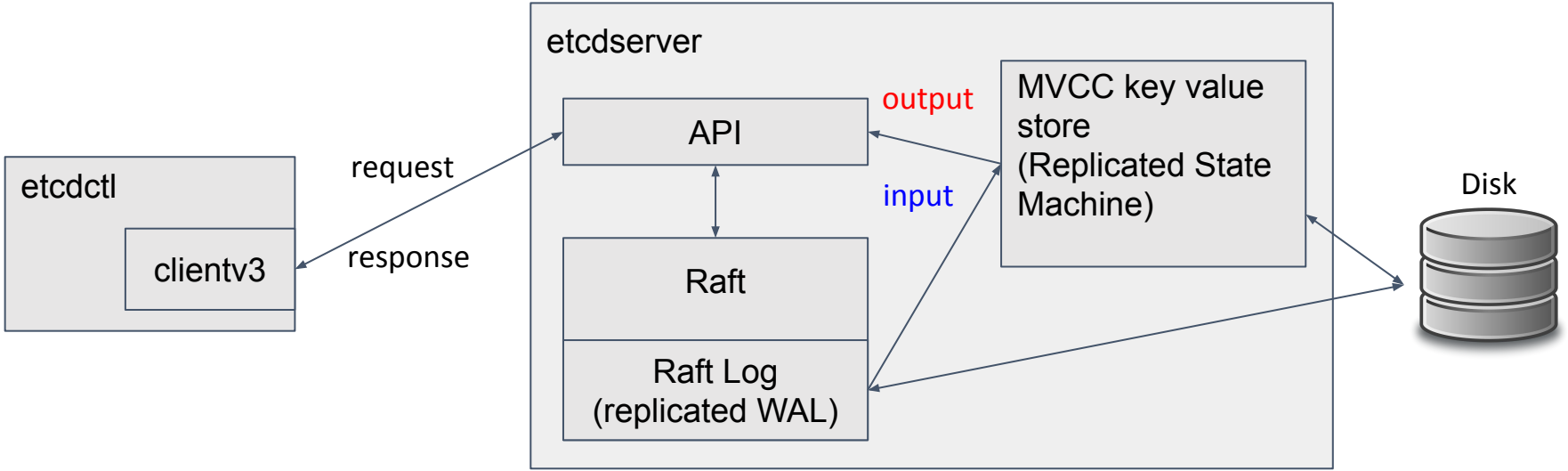
```
Message {
    Type:    pb.MsgApp,
    To:      peer 1 id,
    From:    r.id,
    Entries: ents,
    Commit:  r.raftLog.committed,
    // Commit is updated
    ...
}
```

Follower

```
Ready {
    Messages: // Messages includes a message
of type pb.MsgAppResp to leader
    CommittedEntries: // new committed
entries as the result of maybeAppend()
    ...
}
```

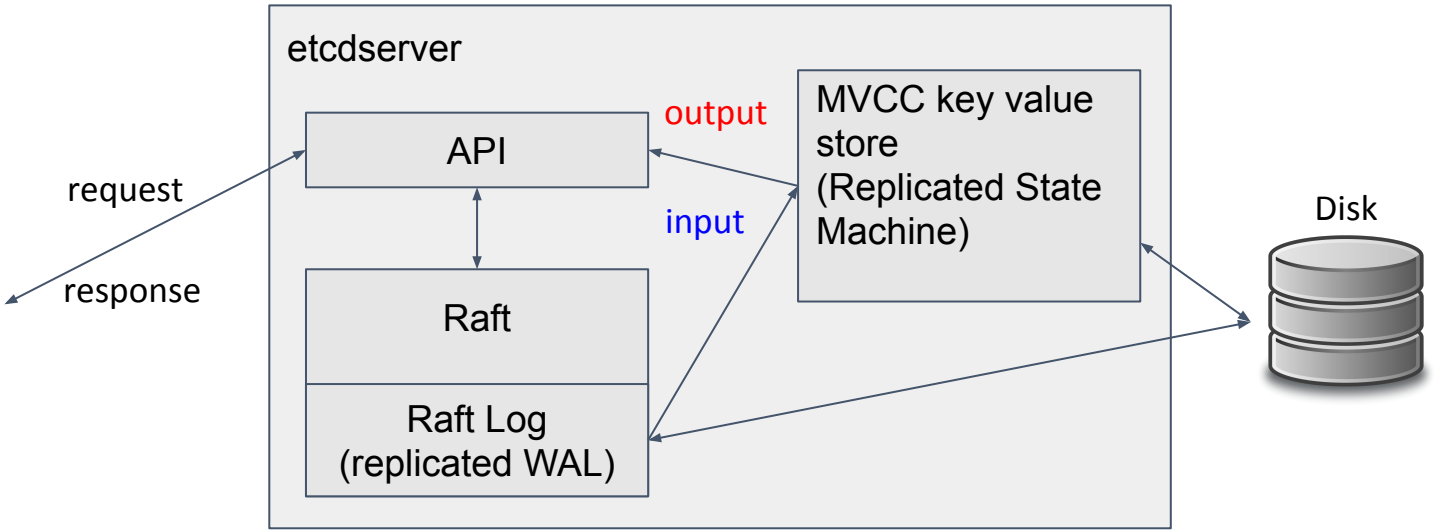
```
// Recap: the output of Raft state machine

type Ready struct {
    pb.HardState
    // Term, Vote, Committed Index
    ...
    Entries []pb.Entry
    // Raft entries to be saved to storage
    ...
    CommittedEntries []pb.Entry
    // Raft entries ready to be applied to
    // replicated state machine
    ...
    Messages []pb.Message
    // messages to be sent to peers
    ...
}
```

Raft in etcd

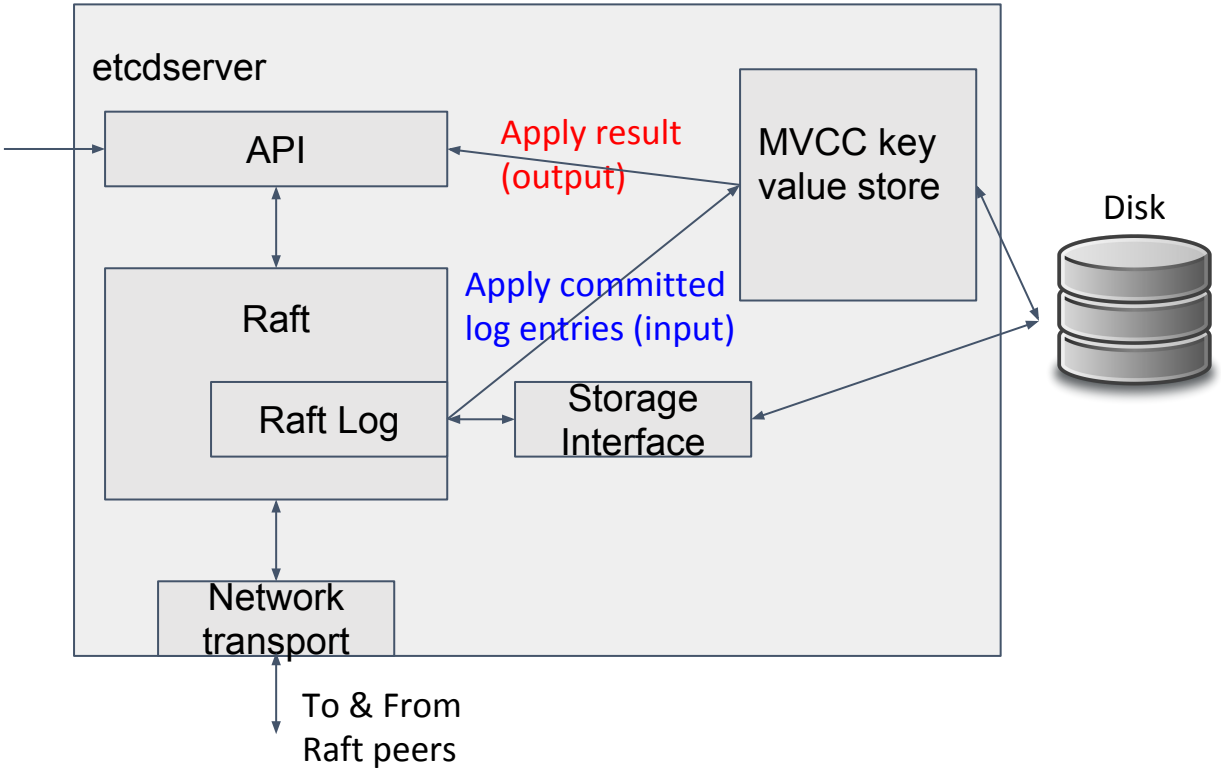


We will revisit this graph (with more details) after we talked about the Raft implementation.

Raft in etcd

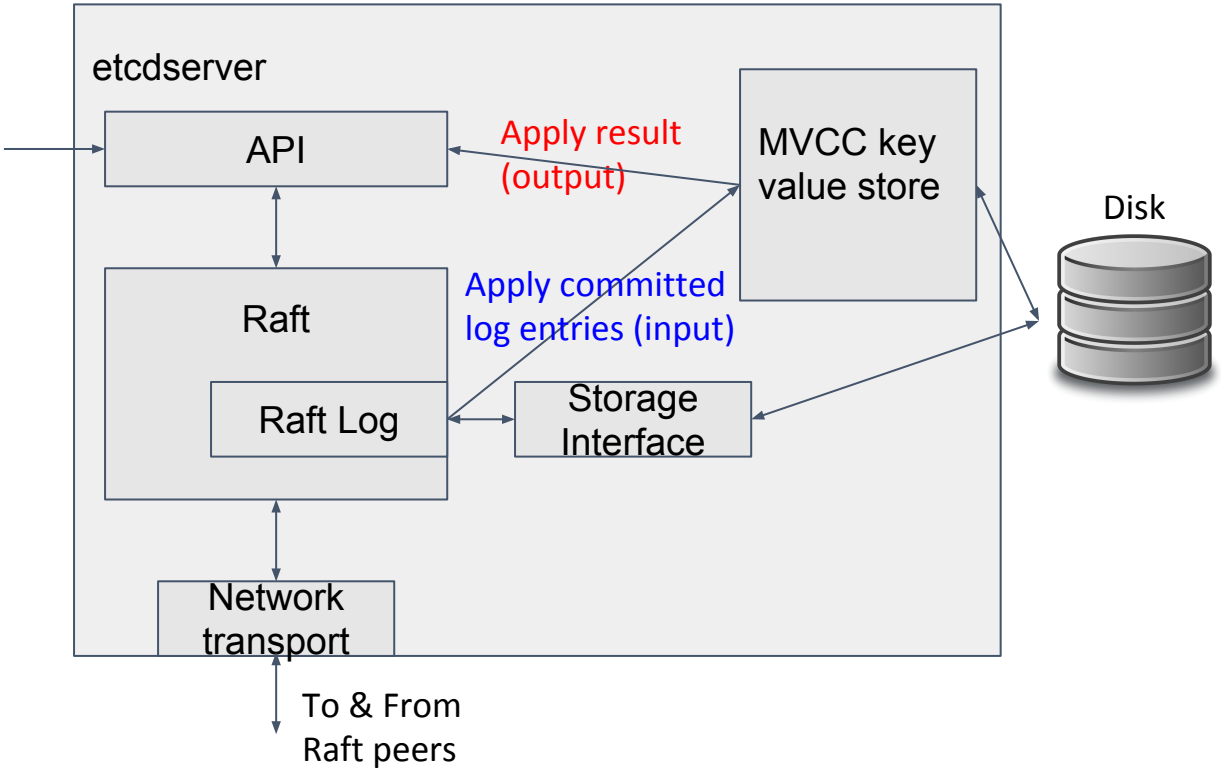

etcdserver
API
output
input
MVCC key value store (Replicated State Machine)
request
response
Raft
Raft Log (replicated WAL)
Disk

Revisit: Raft in etcd



Revisit: Raft in etcd

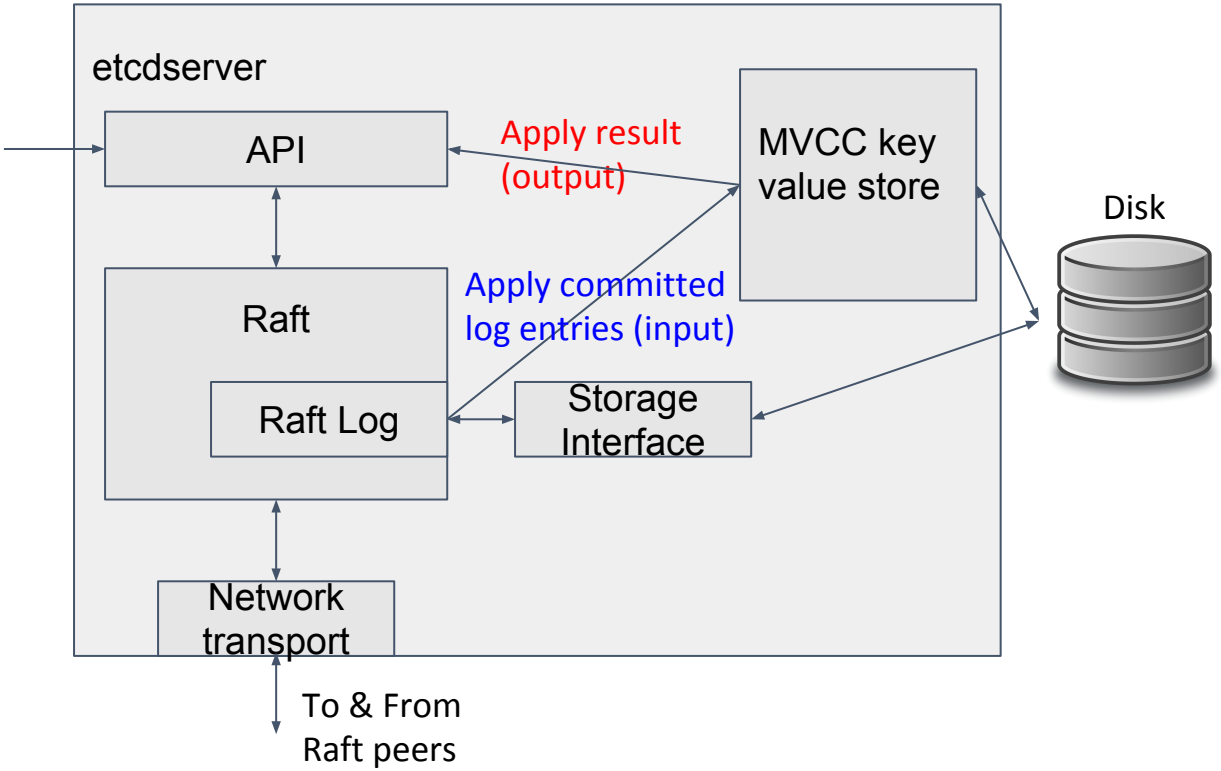

```
// Server's handling loop

for {
    select {
    ...
    case rd := <-r.Ready():
        r.storage.Save(rd.HardState,
rd.Entries, rd.Snapshot)
        r.transport.Send(rd.Messages)
        s.Apply(rd.CommittedEntries)
        ...
    }
}
```



Revisit: Raft in etcd


```
// Server's handling loop

for {
    select {
    ...
    case rd := <-r.Ready():
        r.storage.Save(rd.HardState,
rd.Entries, rd.Snapshot)
        r.transport.Send(rd.Messages)
        s.Apply(rd.CommittedEntries)
        ...
    }
}
```

```
// Request lifecycle

1. Send proposal to Raft
r.Propose(ctx, data)

2. If successfully committed, data
will appear in rd.CommittedEntries

3. Apply committed entries to MVCC

4. Return apply result to client
```



Roadmap

- Raft learner / non-voting member feature in etcd v3.4.
  - Design Doc:

    https://github.com/etcd-io/etcd/blob/master/docs/server-learner.rst
  - Implementation: PR #10725, #10727, #10730
- Joint consensus for Raft membership reconfiguration?
  - https://github.com/etcd-io/etcd/issues/7625


KubeCon
CloudNativeCon
OPEN SOURCE SUMMIT
China 2019
Thanks!
Q&A