Creating A Distributed Round Robin Scheduler with Etcd

A developer learns some things about distributed systems and reliability

Eric Chlebek
Software Developer, Sensu

About Me

https://github.com/echlebek (https://github.com/echlebek)

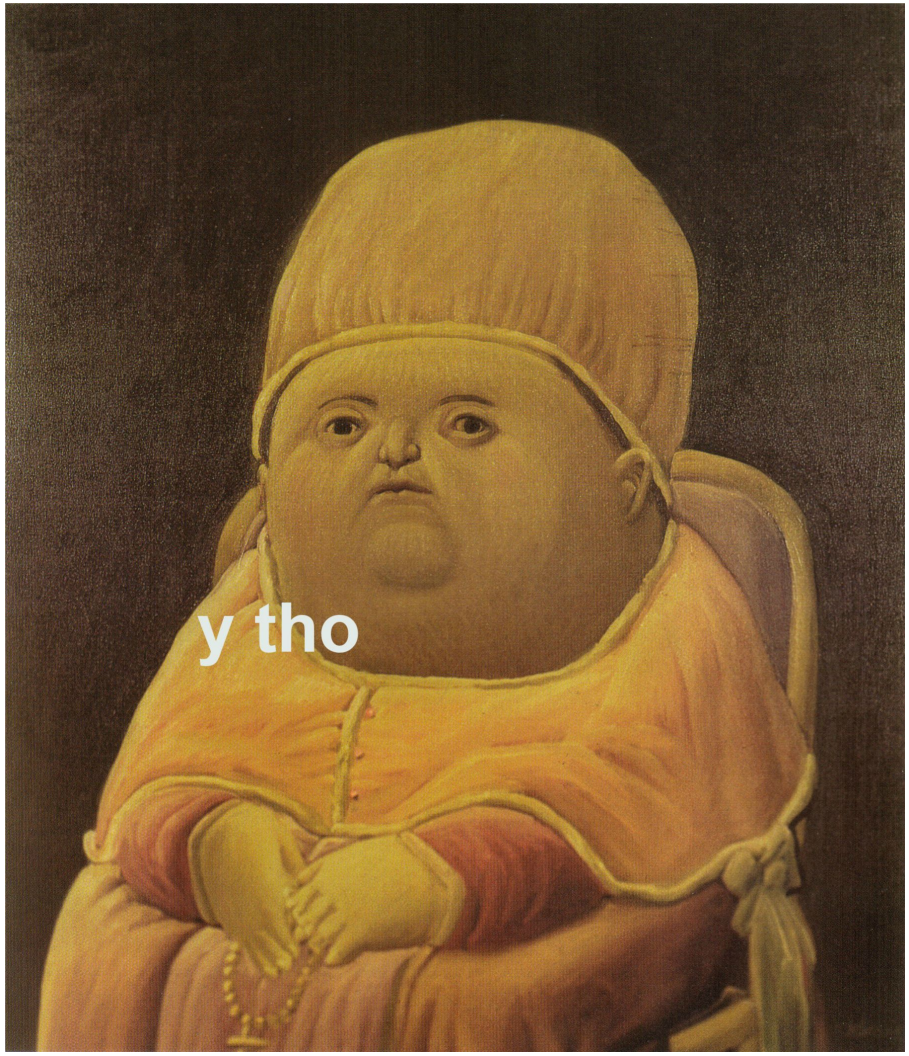
https://lisa19-etcd.herokuapp.com/talk.slide#1 (https://lisa19-etcd.herokuapp.com/talk.slide#1)

- Works @ Sensu on the Sensu Go monitoring project.
- Experience in HPC, Bioinformatics, Ad-tech, Systems Monitoring.

2

Introduction

Introduction



4

Introduction

- Sensu is a monitoring framework for heterogeneous systems.
- For the purposes of this talk, Sensu is a scheduler for executing host-based checks on subscriber nodes.
- Round robin scheduling is one of the key features of Sensu's scheduler.

5

Introduction

- By default, all systems execute their subscribed checks at every scheduling interval.
- Some use cases are better suited a round-robin mode of scheduling (load-balanced websites, network switches)
- Classic Sensu relied on RabbitMQ for round-robin scheduling, and Redis for state.

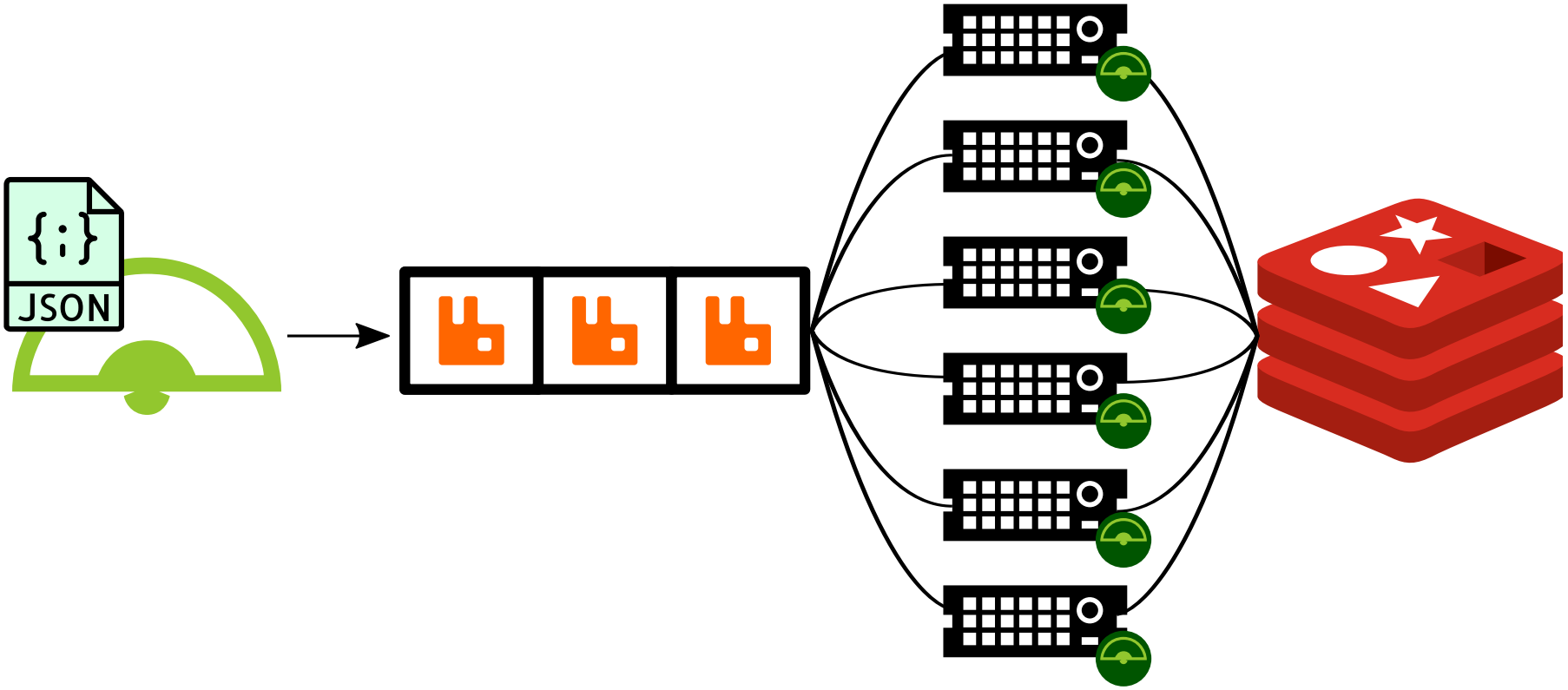
6

Introduction

- Classic Sensu uses RabbitMQ as a broker to distribute tasks to clients.
- Single leader responsible for sending tasks to the queue.
- Randomized consumption of tasks by clients.
- In clustered/HA scenario, failure of the leader could be troublesome.

7

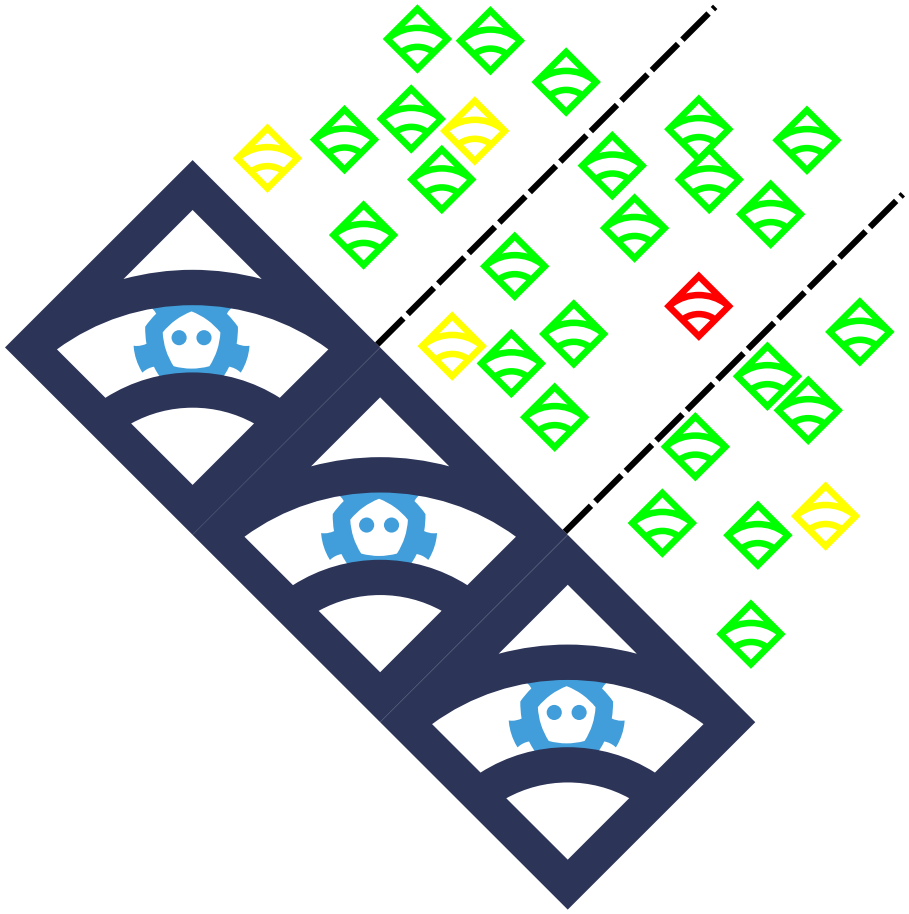
Introduction



Classic Sensu Architecture

8

Introduction



Sensu Go Architecture

9

Introduction

- Sensu Go is built around etcd, and does not support RabbitMQ.
- I needed to come up with a model for round robin scheduling on etcd.
- I didn't want to follow the single leader pattern.
- I wanted round-robin scheduling to be as reliable as the store itself.
- Need to tolerate the loss of either scheduler or worker nodes.
- Would be nice to have a stable ordering of execution for round robin workers.

10

Introduction

A distributed, reliable key-value store for the most critical data of a distributed system (etcd.io)

- Etcd is a distributed key-value database that uses Raft consensus.
- Written in Go. No Java or C components.
- Cross platform. Works on most Go compilation targets.
- Uses the BoltDB embedded database, optimized for SSDs.
- Uses gRPC as a transport, efficient RPC communication between peers and clients.

11

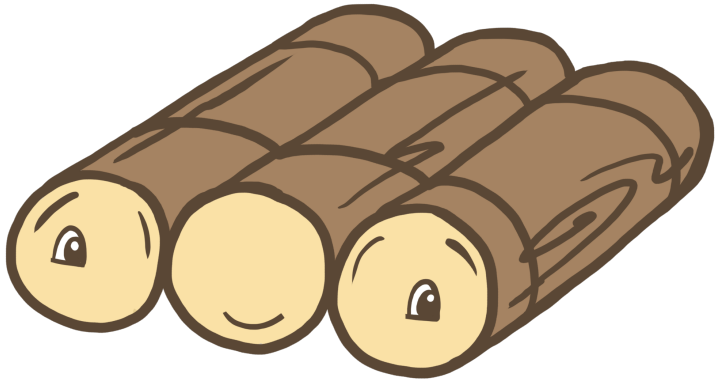
Why etcd?

- Strongly consistent distributed key-value store.
- Benchmarked at tens of thousands of transactions per second.
- Can survive the loss of (n / 2) - 1 members.
- MVCC transaction model.
- Can be embedded in Go applications, no need for external dependency.
- Our goal was to enable a straightforward clustering story for Sensu Go.

12

Raft

Raft Consensus Algorithm



14

Raft Consensus Algorithm

- Created by Diego Ongaro and John Ousterhout at Stanford.
- Their goal was to replace Paxos (Leslie Lamport).
- Designed to maximize understandability.
- An algorithm for managing a replicated state machine and log.

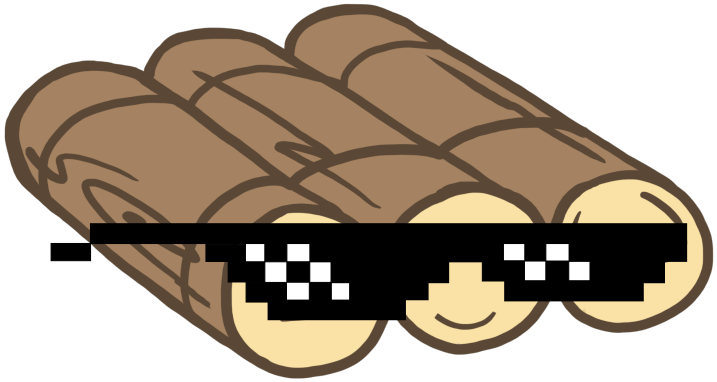
15

Raft Consensus Algorithm

- Why is it called raft?
- A raft is several logs tied together...
- A replicated log...

16

Raft Consensus Algorithm



17

Raft Consensus Algorithm

- Raft is a consensus algorithm that is designed to maximize understandability.
- The algorithm manages a replicated state machine and log.
- Equivalent to Multi-Paxos, in power and efficiency.
- All algorithms of this class require a heartbeat, so raft has one too.
- Timeouts determine if a member is no longer alive.

18

Raft

What is a log?

- A log is an append-only data structure.

What is a state machine?

- A state machine is a mathematical model for computation. It takes its input from a log.

How does this apply to raft?

- Each raft member has a state machine that consumes the replicated log. Because the log is guaranteed to be the same, the state machines will produce the same outputs.

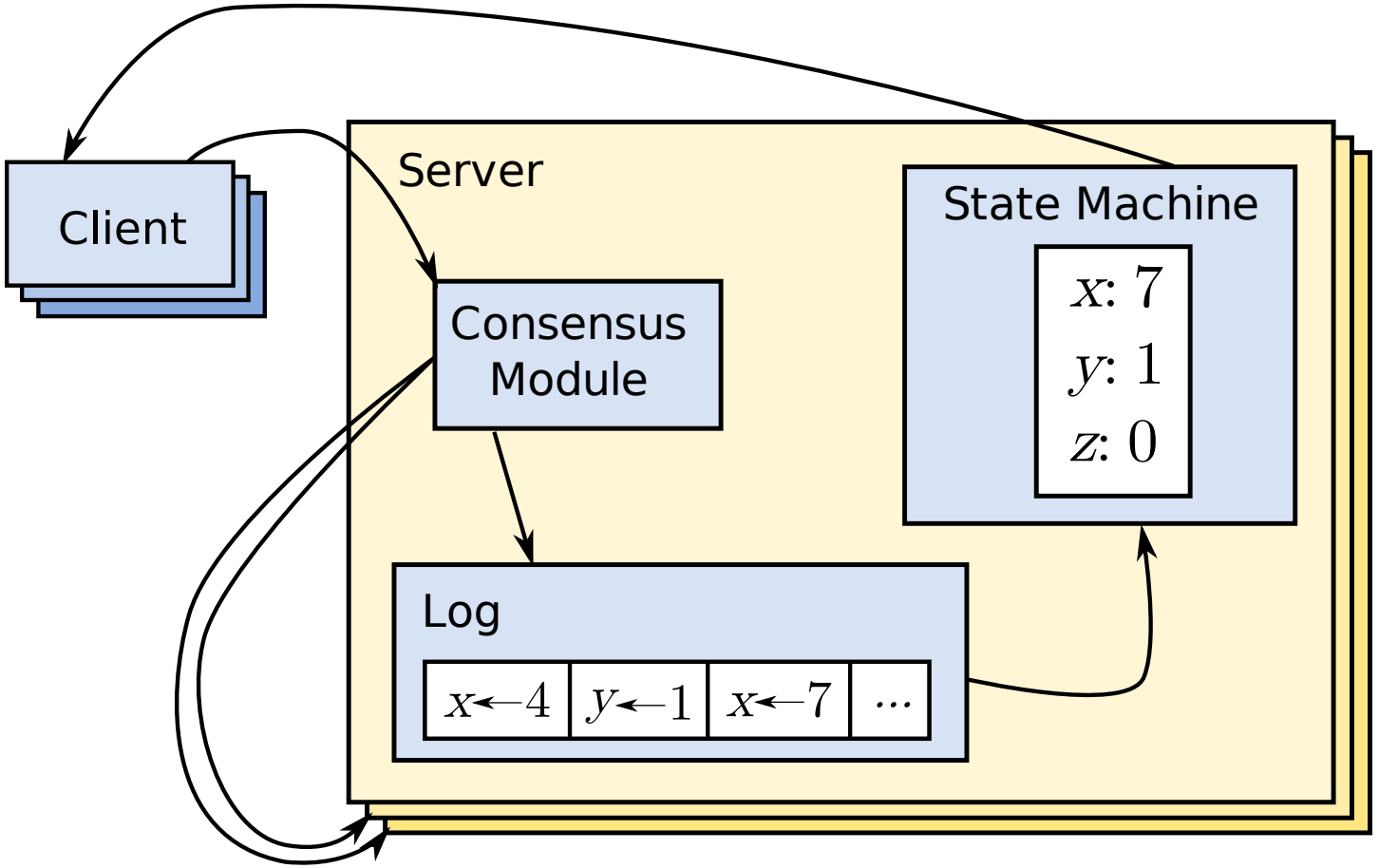
19

Raft

- Raft members are always in one of three states: leader, follower, candidate.
- Elections are used to determine the class of each member.
- If a follower has not seen a heartbeat for a long time, it establishes itself as a candidate and initiates an election.
- The result of the election process is that the follower will become the leader, or another cluster member will become the leader, or a timeout will occur.

20

Raft



Replicated state machine architecture

21

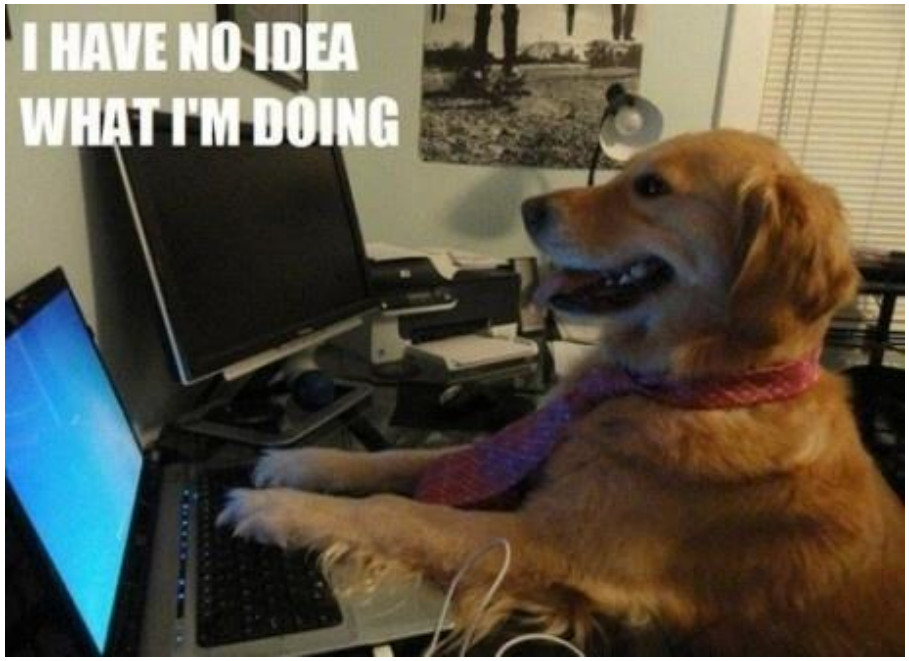
Raft

- Consensus algorithms guarantee safety (even with network delays, partitions, and packet loss, message duplication, and reordering).
- They are functional, AKA available, as long as a majority of their members are working and can communicate with one another.
- They do not depend on timing to ensure consistency in their logs. Bad clocks can cause availability problems at worst.

22

Raft

- Raft has become more popular than Paxos, as it is easier to understand, and implement.
- Few people succeed in understanding Paxos, and it requires great effort to do so.
- Even seasoned researchers struggle at understanding Paxos.



23

Raft

- Raft implements consensus by electing a leader, and making that leader responsible for managing the replicated log.
- The leader accepts log entries from clients, replicates them to followers, and tells them when it is safe to apply the logs to their state machines.
- Because the leader has the sole responsibility for managing the replicated log, it is free to append to the log in any way it likes.

24

Raft

- Raft clusters are available as long as a majority of the members are working.
- All raft cluster sizes are odd numbers. (1, 3, 5, 7, 9)
- If 4 machines are members of a raft cluster, the cluster size is at least 5.

25

Raft

- When more than (N/2 - 1) raft members fail, the cluster becomes unavailable.
- In a net split, the minority partition will not be available.
- This is essential to guarantee raft's correctness property.

26

Raft

- The raft algorithm describes an infinitely growing log.
- Infinitely growing logs don't work so well in practice...

27

Raft

- Any useful implementation of raft requires some sort of log compaction.
- Many raft and paxos systems use snapshotting to deal with log compaction.
- Snapshotting can be implemented in various ways.
- After a snapshot, the log history to a certain point is compacted into a single entry.

28

Raft Key Takeaways

- Correctly implemented, a store built on raft will always be consistent.
- If more than half of a raft cluster fails, the service becomes unavailable.
- If a raft cluster is split in two, the smaller half becomes unavailable, while the larger half remains available, as long as it has a sufficient number of nodes.

29

Back to etcd

etcd API

What does the etcd API offer?

- Key-value storage (range, put, delete)
- Multi-version concurrency control
- Transactions (single round trip)
- Leases
- Watchers

31

KV Storage

32

KV Storage

33

MVCC

34

Transactions

- etcd's transactions are a single round-trip
- that means you can't read back a value, and then operate on it, transactionally
- but you can execute comparisons server-side

35

Transactions

```
func main() {
    ctx := context.Background()

    client := newClient()
    defer client.Close()

    if _, err := client.Put(ctx, "foo", "bar"); err != nil {
        log.Fatal(err)
    }

    _, err := client.Txn(ctx).If(
        etcd.Compare(etcd.Value("foo"), "=", "bar"),
    ).Then(
        etcd.OpPut("frob", "true"),
    ).Else(
        etcd.OpPut("frob", "false"),
    ).Commit()

    if err != nil {
        log.Fatal(err)
    }

    resp, err := client.Get(ctx, "frob")
    if err != nil {
        log.Fatal(err)
    }
```

36

```
    fmt.Println(string(resp.Kvs[0].Value))
}
```

Leases

37

Swiss Alps

38

Leases (Keepalives)

39

Leases (Keepalives)

- When combined with keepalives, leases offer a powerful primitive for creating etcd database triggers.
- In Sensu, leases are used for implementing vigilance control. When agents haven't been heard from for a long enough period, a leased key expires, which alerts the backend to the presence of failure.
- Leased keys can be combined with watchers for some interesting control flow constructs.

40

Watchers

41

Leases and Watchers Together

42

Leases and Watchers Together

Mt. Baker

44

Lease and Watchers Together

- Allows creating a distributed trigger
- Semi-durable; can survive cluster downtime, but watch events are dropped if nobody is watching
- Forms the basis of a round-robin ring

45

Round-robin ring

- A round-robin ring is a circular list of workers.
- The ring tracks which worker is the next to receive work.
- On a configurable interval, the workers travel around the ring, waiting for their turn to work.
- Unlike a token ring, the workers are not responsible for passing tokens to keep the ring mechanism working.

46

Round-robin ring

- The round-robin ring is operated by one or more schedulers.
- Any scheduler can add or remove workers from the ring.
- The schedulers compete to advance the ring to the next position. (first write wins, lock-free)
- When a worker's turn to work comes up, every scheduler is notified. (watcher)

47

Round-robin ring

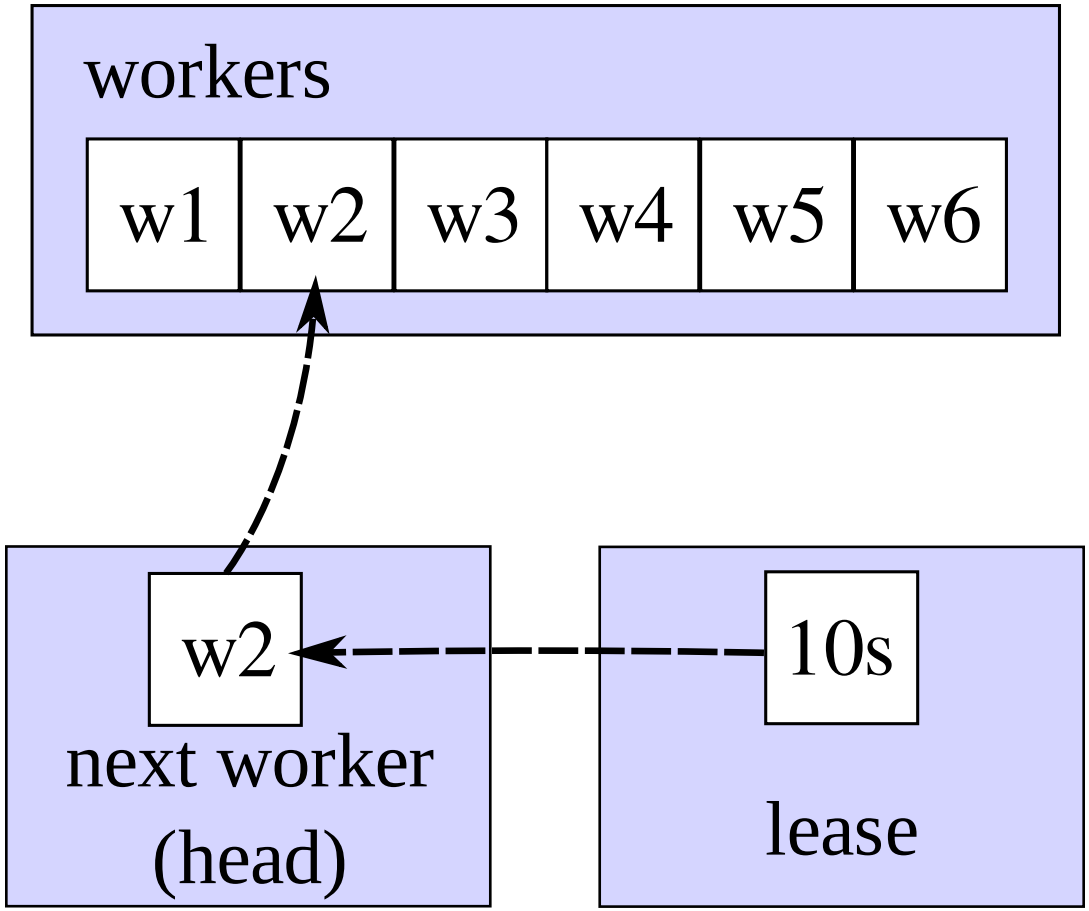
- Workers in the ring are leased; if not kept alive, they will expire.
- This prevents the ring from containing workers that have failed. (eventually)
- Ring is lexicographically ordered, like etcd keys.
- The next worker to work is stored under a "next" key.
- If the schedulers notice that a worker has expired, and would have been the next to work, they compete to advance the ring.

48

Round-robin ring

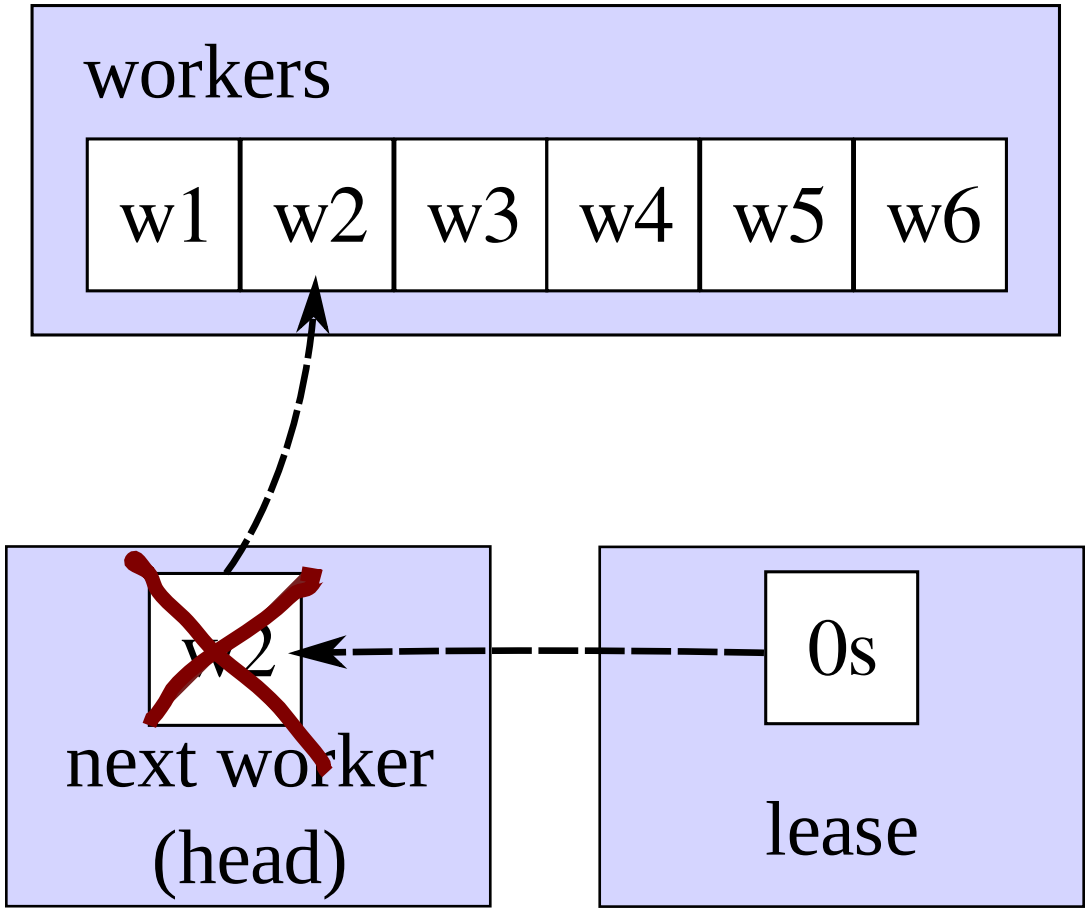
- Schedulers can fail, need at least one to keep scheduling working.
- If all schedulers fail, ring state is maintained.
- etcd servers can fail, ring will keep working as long as a majority are healthy.
- If etcd loses availability, ring state is maintained until restart.

Round-robin ring



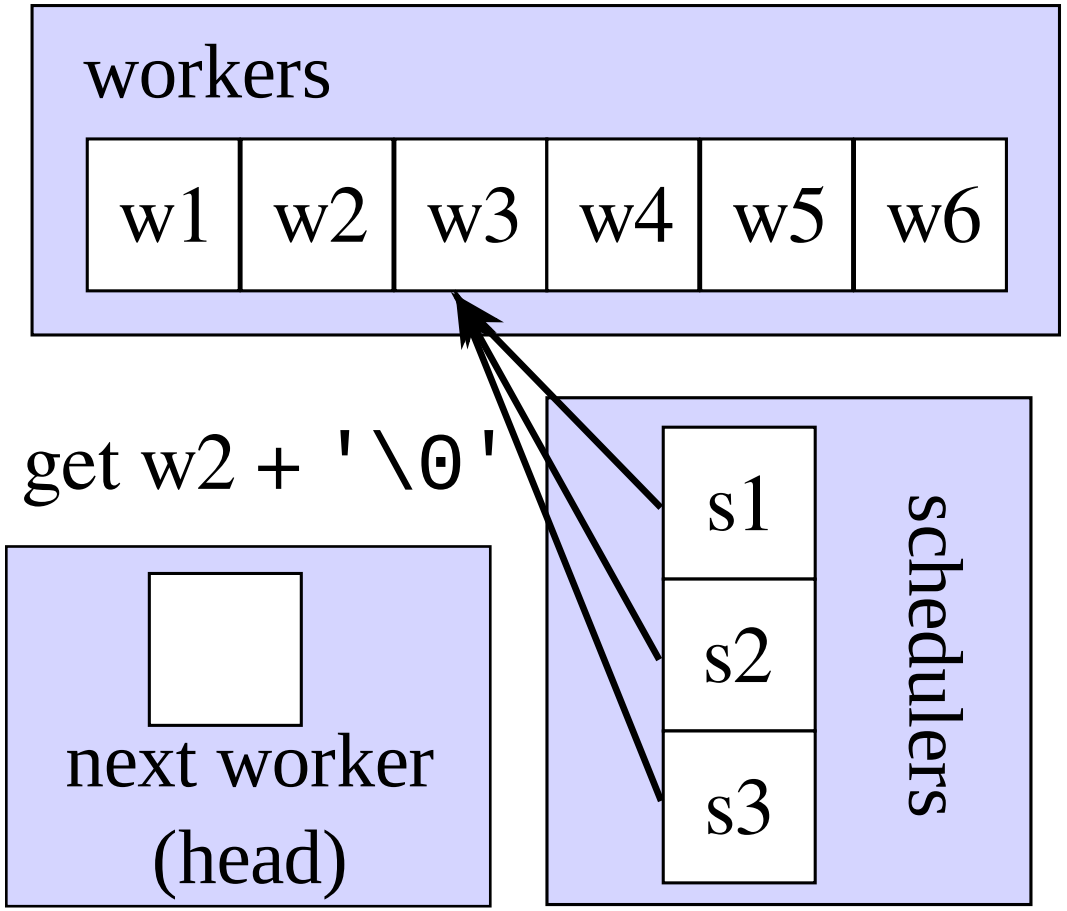
50

Round-robin ring



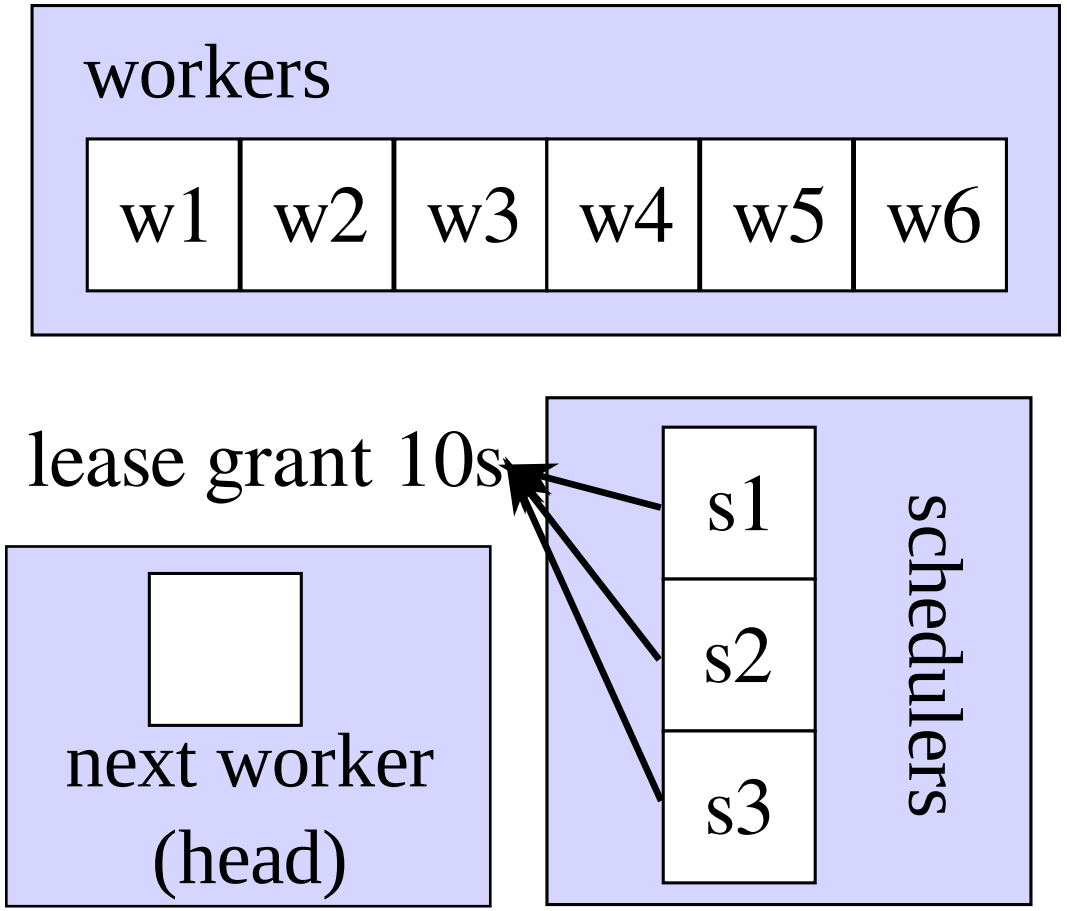
51

Round-robin ring



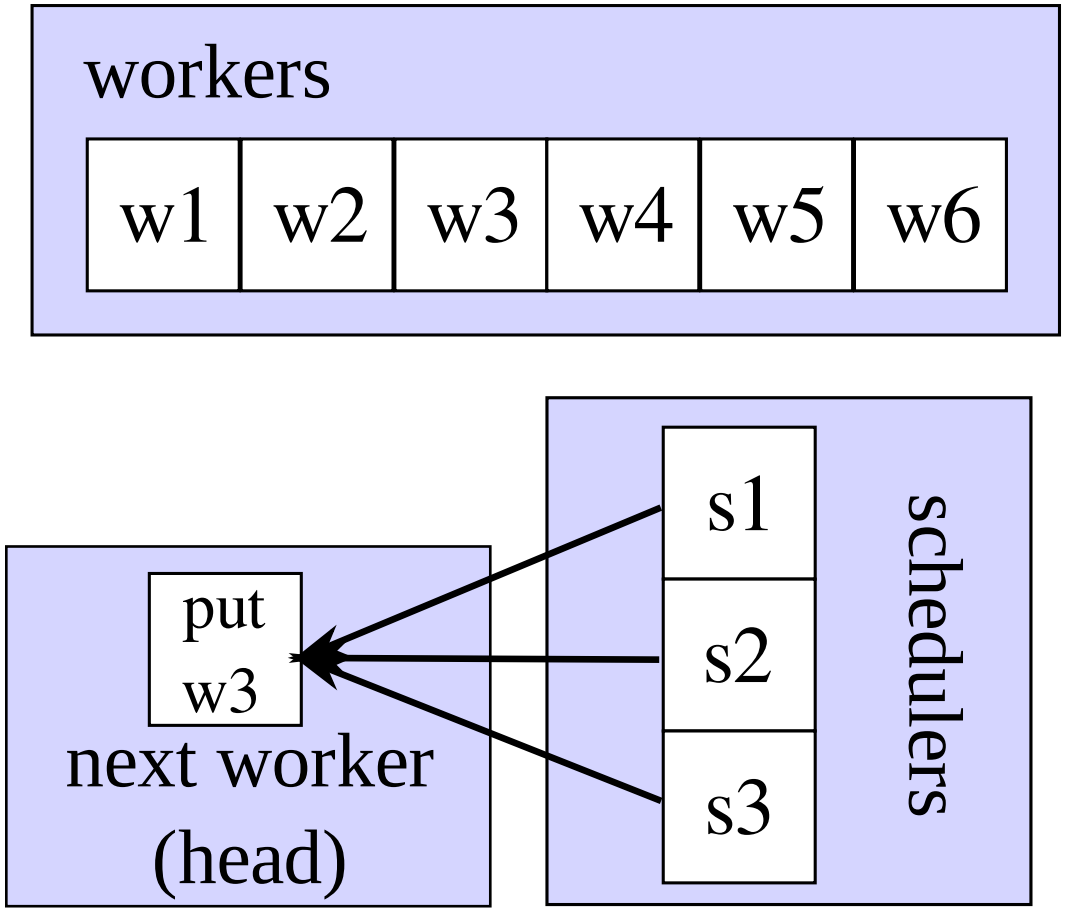
52

Round-robin ring



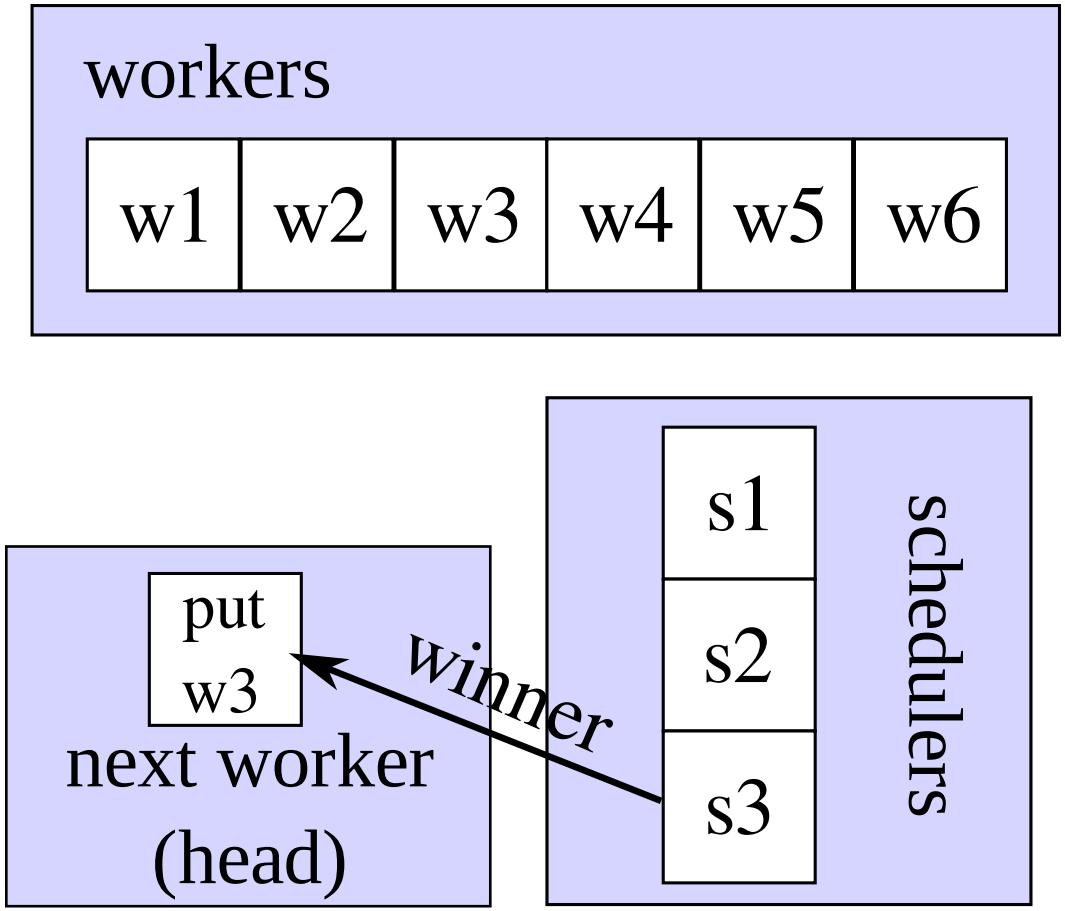
53

Round-robin ring



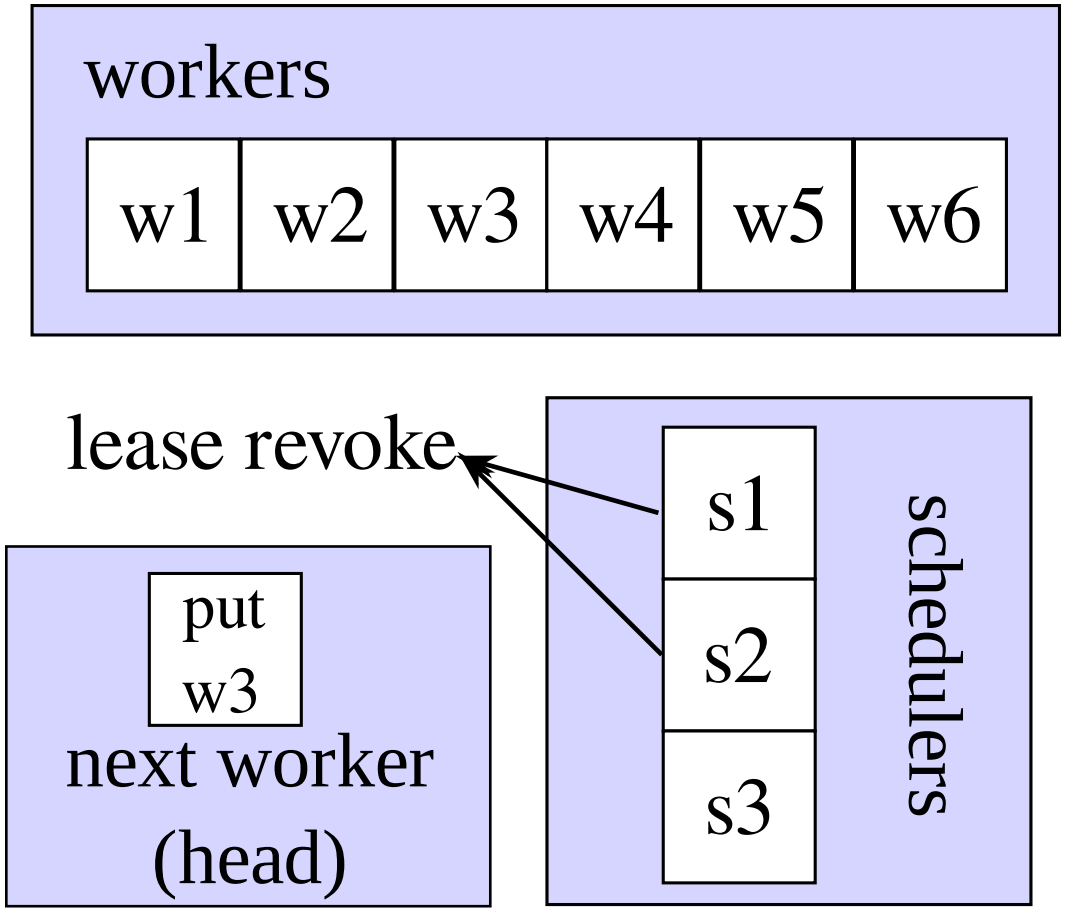
54

Round-robin ring



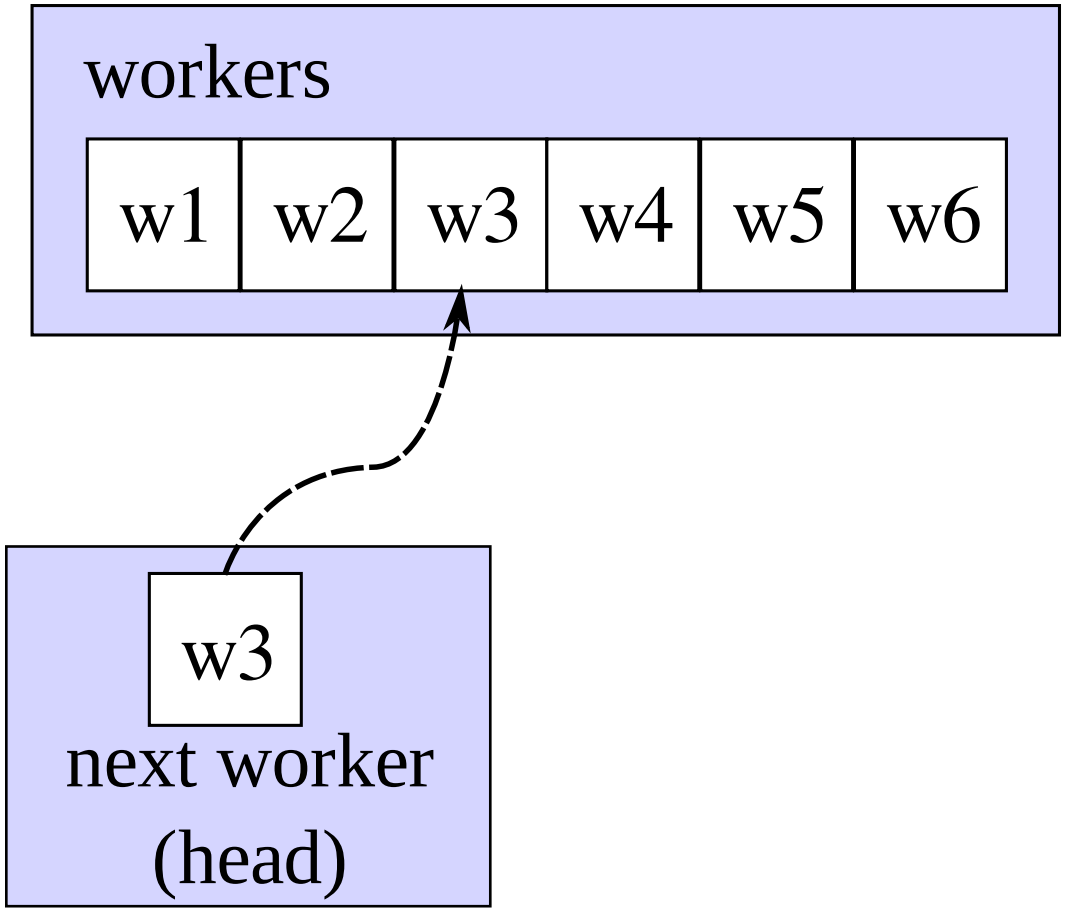
55

Round-robin ring



56

Round-robin ring



57

Testing

- Difficult to unit-test this library, eventually gave up.
- etcd interfaces are not easy to mock out.
- Created integration tests that run reasonably quickly.
- Easy to set up multiple etcd instances in a single Go process.

Testing

- Early versions of the ring were not very successful!
- The first version lacked synchronization between the schedulers, and had unsolvable concurrency bugs.
- Moving the trigger mechanism into etcd, using leases, solved the synchronization problem.

59

Testing

- etcd failures.
- Scheduler failures.
- Worker failures.
- Any combination of the above.

60

What have I learned?

- The feature-set of etcd is quite interesting, and has some surprisingly powerful primitives.
- Testing complex data structures, and coordination routines, remains tricky. I am still learning how to best approach this.
- Lease expirations are remarkably un-performant. (Linear scan of all leases for every expiration, solved in etcd 3.4)

61

"Why didn't you use a regular database?"

- It's complicated.
- I would always rather use Postgres. It's awesome, and has a richer model for transactions.
- Sometimes working within particular constraints can result in an interesting outcome.

62

"Why didn't you use the single-leader pattern, but with etcd?"

- I was fearful of handling leader failure correctly.
- I believed that etcd provided the primitives for building a distributed, coordinated data structure.
- I perceived the single-leader pattern to be less reliable.
- It seemed more fun to do it this way.

63

"Does it scale?"

- I think so... more testing needed is needed here.
- etcd watchers can scale surprisingly high with gRPC proxy - 1M watch events per second with 20 proxies.
- etcd should be able to service hundreds of scheduler nodes, maybe even thousands.
- A single scheduler should be able to handle thousands or tens of thousands of workers.

64

Open-source implementation

Sensu's round-robin ring is available as a Go library under an MIT license.

https://godoc.org/github.com/sensu/sensu-go/backend/ringv2 (https://godoc.org/github.com/sensu/sensu-go/backend/ringv2)

65

Thank you

Eric Chlebek
Software Developer, Sensu
eric@sensu.io (mailto:eric@sensu.io)
http://sensu.io (http://sensu.io)
@EricChlebek (http://twitter.com/EricChlebek)