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

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

Classic Sensu Architecture
Introduction

Sensu Go Architecture
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.
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.
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.
Raft
Raft Consensus Algorithm
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.
Raft Consensus Algorithm

- Why is it called raft?
- A raft is several logs tied together...
- A replicated log...
Raft Consensus Algorithm
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.
Рафт

Что такое лог?

- Лог — это структура данных, добавляющая данные.

Что такое состояние машины?

- Состояние машины — математическая модель вычисления. Она принимает на вход данные из лога.

Как это применимо к рафту?

- Каждый участник рафта имеет машину состояния, которая потребляет лог, который является реплицированным. Продолжение рафа...
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.
Raft

Replicated state machine architecture
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.
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.
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.
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.
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.
Raft

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

What does the etcd API offer?

• Key-value storage (range, put, delete)
• Multi-version concurrency control
• Transactions (single round trip)
• Leases
• Watchers
KV Storage
KV Storage
MVCC
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
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)
    }
}
fmt.Println(string(resp.Kvs[0].Value))
}
Leases
Swiss Alps
Leases (Keepalives)
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.
Watchers
Leases and Watchers Together
Leases and Watchers Together
Mt. Baker
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
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.
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)
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.
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

- Workers: w1, w2, w3, w4, w5, w6
- Next worker (head): w2
- Lease: 10s
Round-robin ring

workers

w1  w2  w3  w4  w5  w6

next worker
(head)

0s

lease
Round-robin ring

workers

w1  w2  w3  w4  w5  w6

get w2 + ' \0 '

next worker (head)

s1

s2

s3

schedulers
Round-robin ring

workers

w1  w2  w3  w4  w5  w6

lease grant 10s

next worker (head)

s1  s2  s3

.schedulers
Round-robin ring

workers

w1  w2  w3  w4  w5  w6

put

next worker
(head)

s1
s2
s3

schedulers
Round-robin ring

workers

w1   w2   w3   w4   w5   w6

put
w3

next worker
(head)

winner

s1
s2
s3

schedulers

Creating A Distributed Round Robin Scheduler with Etcd

https://lisa19-etcd.herokuapp.com/talk.slide#1
Round-robin ring

workers

w1  w2  w3  w4  w5  w6

lease revoke

put

w3

next worker (head)

s1

s2

s3

collectors
Round-robin ring

workers

w1  w2  w3  w4  w5  w6

next worker (head)

w3
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.
Testing

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