Distributed Consensus and the Raft Algorithm

Concepts of Programming Languages
22 November 2018

Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim

About this Lecture: Some Thoughts

- Goal of this lecture is to look at a non trivial example in Go.
- The problem we look at, is language agnostic.
- It can be implemented in almost every language.
- It combines concurrent programming and network programming.
- It contains non trivial logic.
- It is a real world example with high relevance.

2

What is Distributed Consensus?

Distributed consensus is the problem how to achieve consistency in distributed systems. Distributed consensus protocols can be used for distributed databases which should stay consistent:

- The system stays consistent even if some nodes goes down
- The system stays consistent if a network partition occurs (but could get unavailable)
- The system never responds with wrong or inconsistent data (different responses from nodes)

3

The CAP Theorem

In a distributed system the following requirements can be met:

- **Consistency** - Data is the same across the cluster, so you can read or write from/to any node and get the same data.
- **Availability** - The system stays available even if one or more member goes down
- **Partition Tolerance** - If parts of the system loose connection to other parts, the system stays alive

Pick Two: Only two requirements can be met -> CP, AP are typical (CA does not exists)

codahale.com/you-cant-sacrifice-partition-tolerance/ (https://codahale.com/you-cant-sacrifice-partition-tolerance/)

Securing C and P

- All known algorithms use the concept of a quorum to detect network partition problems
- Only the partition with the majority of nodes stay alive to ensure consistency
- All known algorithms use the concept of a Master or Leader node to control replication
- All known algorithms use a replicated log and implement a two phase commit

Two Phase Commit

Phases:

- Prepare Phase - Data is persisted on all members. Data is not visible but stored.
- Commit Phase - Persisted data will be loaded into memory. Data gets visible.
- This is typically implemented with an append only log (transaction log) and a state machine which reads the log entries and loads them into memory.

6

What is Raft?

- Raft is a protocol for distributed consensus
- Raft is designed to be easy understandable
- Raft predecessor Paxos was highly complex
- Most Paxos implementations are buggy or academic
- Raft was developed 2014 in a phd thesis at Stanford University
- Zookeeper ZAB is an alternative but more complex solution

The Raft Paper (https://raft.github.io/raft.pdf)

The ZAB paper (https://github.com/lshmouse/reading-papers/blob/master/distributed-system/Zab:%20High-performance%20broadcast%20for%0Aprimary-backup%20systems.pdf)

The Paxos Paper (https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf) 7

Who uses Raft?

- ectd - The Cloud Native key value store -> Part of Kubernetes!
- Docker Swarm - Docker in cluster mode
- dgraph - A Scalable, Distributed, Low Latency, High Throughput Graph Database
- tikv - A Distributed transactional key value database powered by Rust and Raft
- swarmkit - A toolkit for orchestrating distributed systems at any scale.
- chain core - Software for operating permissioned, multi-asset blockchain networks

The Raft Algorithm

- Raft is based on a replicated state machine with the states: **FOLLOWER**, **CANDIDATE** and **LEADER**
- All nodes start in the FOLLOWER state
- The leader is responsible for sending heartbeat message to all members
- The leader is responsible to replicate data to all other nodes (2 phase commit)
- If a member does not receive a leader message in a random timeout interval, a election starts
- During an election a candidate sends vote messages to all cluster members
- The election is won, if a quorum (>50% = n/2 + 1) of members respond with OK

9

Raft in Action

Raft Explanation (http://thesecretlivesofdata.com/raft/)

The Raftscope Visualization (https://raft.github.io/raftscope/index.html)

The Raft State Model

- There is only one Leader which is responsible for consistency and replication

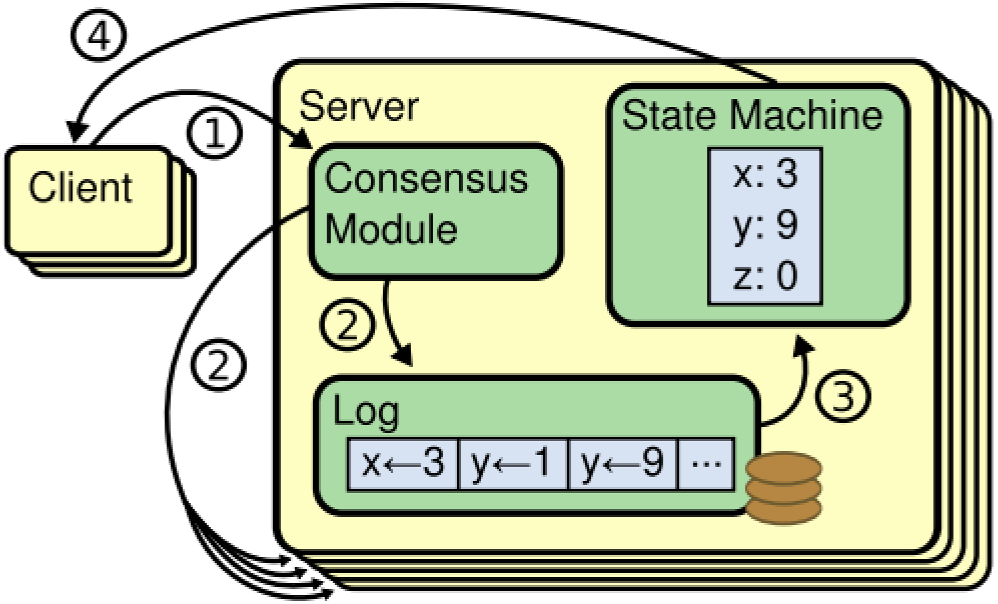

Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

Implementing Raft with Go

The most prominent implementation is the Raft implementation of Etcd (Part of Kubernetes)

- github.com/etcd-io/etcd/tree/master/raft (https://github.com/etcd-io/etcd/tree/master/raft)

This implementation is highly optimized and abstracts from the Raft paper

Other implementations

- github.com/hashicorp/raft (https://github.com/hashicorp/raft)
- github.com/cloudflare/go-raft (https://github.com/cloudflare/go-raft)
- Read the Raft paper for specification (https://raft.github.io/raft.pdf)

Is it possible to build your own Raft implementation?

Requirements and Decisions

- We want to stay as close a possible on the original specification
- We want to make a Raft cluster local testable (as single process)
- We want to use gRPC as middleware

Step I - Defining a KV Store API

```go
// Copyright 2018 Johannes Weigend
// Licensed under the Apache License, Version 2.0

// Package kvstore contains the KeyValueStore API.
package kvstore

// KVStore is the business interface to a distributed key value store.
type KVStore interface {

    // Sets a value for a given key.
    SetString(key, value string)

    // GetString returns the value for a given key.
    GetString(key string) string
}
```

- This is the business API for setting and getting data in/out or Raft cluster

Step II - The Raft RPC Interface : AppendEntries

```
type NodeRPC interface {
```

```
    // AppendEntries is used by the Leader to replicate logs and it is  used as heartbeat.
    // The Leader will call the AppendEntries method on all nodes in the cluster.
    // Arguments
    // - term         : leaders term
    // - leaderId     : leadersId to redirect calls to leader
    // - prevLogIndex : Index of log entry immediately processing
    // - prevLogTerm  : Term of the prevLogIndex entry
    // - entries      : log entries to store (empty for heartbeat)
    // - leaderCommit : leaders commit index
    // Results
    // - term         : current termin (for leader, update itself)
    // - success      : true if follower contained entry matching prevLogIndex and prevLogTerm
    AppendEntries(term, leaderID, prevLogIndex, prevLogTerm int, entries []string,
        leaderCommit int) (int, bool)
```

- The interface could be easily proxied with gRPC or run locally without proxy

15

Step II - The Raft RPC Interface : RequestVote

```
type NodeRPC interface {
```

```
    // RequestVote is called by candidates to gather votes.
    // It returns the current term to update the candidate
    // It returns true when the candidate received vote.
    // Arguments
    // - term         : candidates term
    // - candidateID  : candidate requesting vote
    // - lastLogIndex : index of candidates last log entry
    // - lastLogTerm  : term of candidates last log entry
    // Results
    // - term         : the current term, for candidate to update itself
    // - voteGranted  : true means candidate received vote
    RequestVote(term, candidateID, lastLogIndex, lastLogTerm int) (int, bool)
}
```

16

Step III - Implement the Raft Interfaces in a Server/Node

```
type Node struct {
    id             int
    statemachine   *Statemachine
    replicatedLog  *ReplicatedLog
    electionTimer  timercontrol // runs only if the node is FOLLOWER or CANDIDATE
    heartbeatTimer timercontrol // runs only if the node is in LEADER state
    currentTerm    int
    votedFor       *int
    cluster        *Cluster // our cluster
    stopped        bool     // helper to simulate stopped nodes
    mutex          sync.Mutex
}
```

Step IV - Write Tests

```go
func TestElection(t *testing.T) {

    n1 := NewNode(0)
    n2 := NewNode(1)
    n3 := NewNode(2)

    nodes := []*Node{n1, n2, n3}
    cluster := NewCluster(nodes)
    defer cluster.StopAll()

    cluster.StartAll()

    time.Sleep(4000 * time.Millisecond)

    ok, err := cluster.Check()
    if !ok {
        t.Error(err)
    }

    cluster.StopAll()
}
```

18

Be Creative!

- Write your own Raft Implementation!

More information (https://youtu.be/vYp4LYbnnW8?t=4)

19

Summary

- Go is an excellent choice to implement an algorithm like Raft
- You can implement the Raft specification with less than 1000 Loc
- You can learn from the Etcd implementation, which is on Github
- Building a production safe implementation is anyway hard in any language

20

Thank you

Johannes Weigend (QAware GmbH)
University of Applied Sciences Rosenheim
johannes.weigend@qaware.de (mailto:johannes.weigend@qaware.de)
http://www.qaware.de (http://www.qaware.de)
@johannesweigend (http://twitter.com/johannesweigend)