Distributed Systems && Go

Connor Zanin

/usr/bin/whoami

- Connor Zanin
- Live in Boulder
- Work for Newstore, Inc.
- Presented GoMR, 2020

- https://connorzanin.com
- https://github.com/cnnrznn/raft
- https://github.com/cnnrznn/fake-etcd

Goal

1. Introduce DS concepts and Raft
2. Show how DS + Go == success
3. Demo the system

Distributed Systems

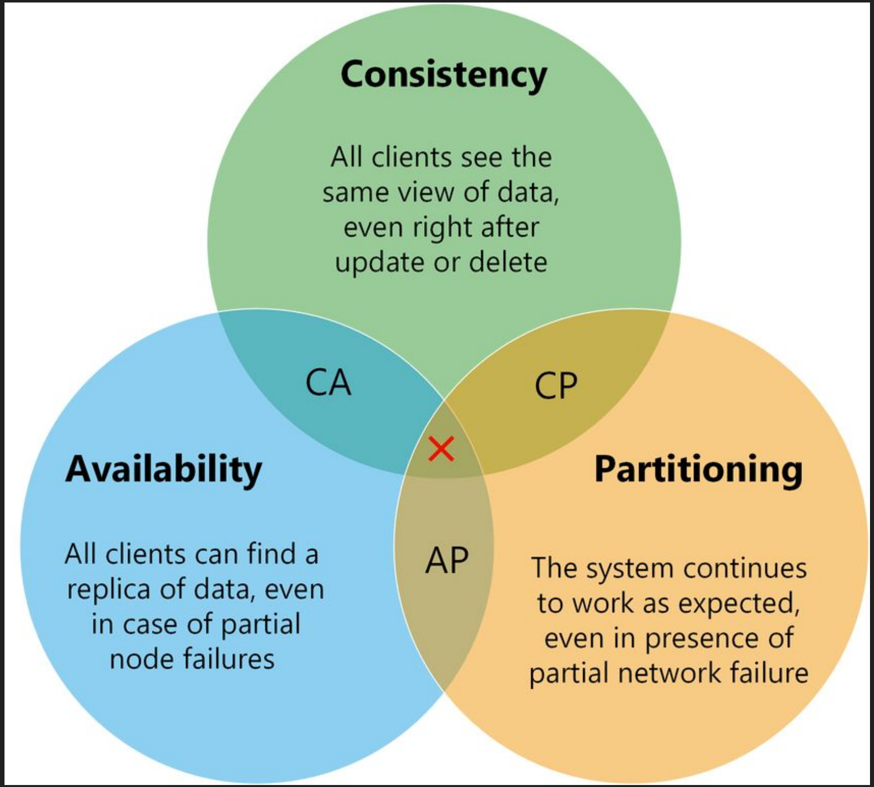
What is a distributed system?

- Set of machines connected over a network
- Working to some common goal

Distributed systems problems

- Partial failure
- Fault tolerance + recovery
- Synchronization
  - Clocks
  - State machines
  - Order of events
- Vulnerable to CAP (consistency, availability, partitioning)
- **Consensus**

CAP Theorem


Consistency
All clients see the same view of data, even right after update or delete
CA
CP
Availability
All clients can find a replica of data, even in case of partial node failures
AP
Partitioning
The system continues to work as expected, even in presence of partial network failure

Raft High-Level

What is Raft?

- Consensus protocol
- Replicated Log
- Crash Fault Tolerant (CFT) (fail-stop)
    - All nodes follow the protocol or crash
    - To tolerate *f* failures, need *2f+1* nodes
- Leader-Follower protocol (asymmetric)
- “Committed” log entries survive

Uses

- In-memory key-value stores (etcd)
- Distributed configs (Zookeeper, Consul)
- Pub/sub, message queues
- Distributed file systems (GFS)
- Any in-memory fault-tolerant cache

Raft Protocol States

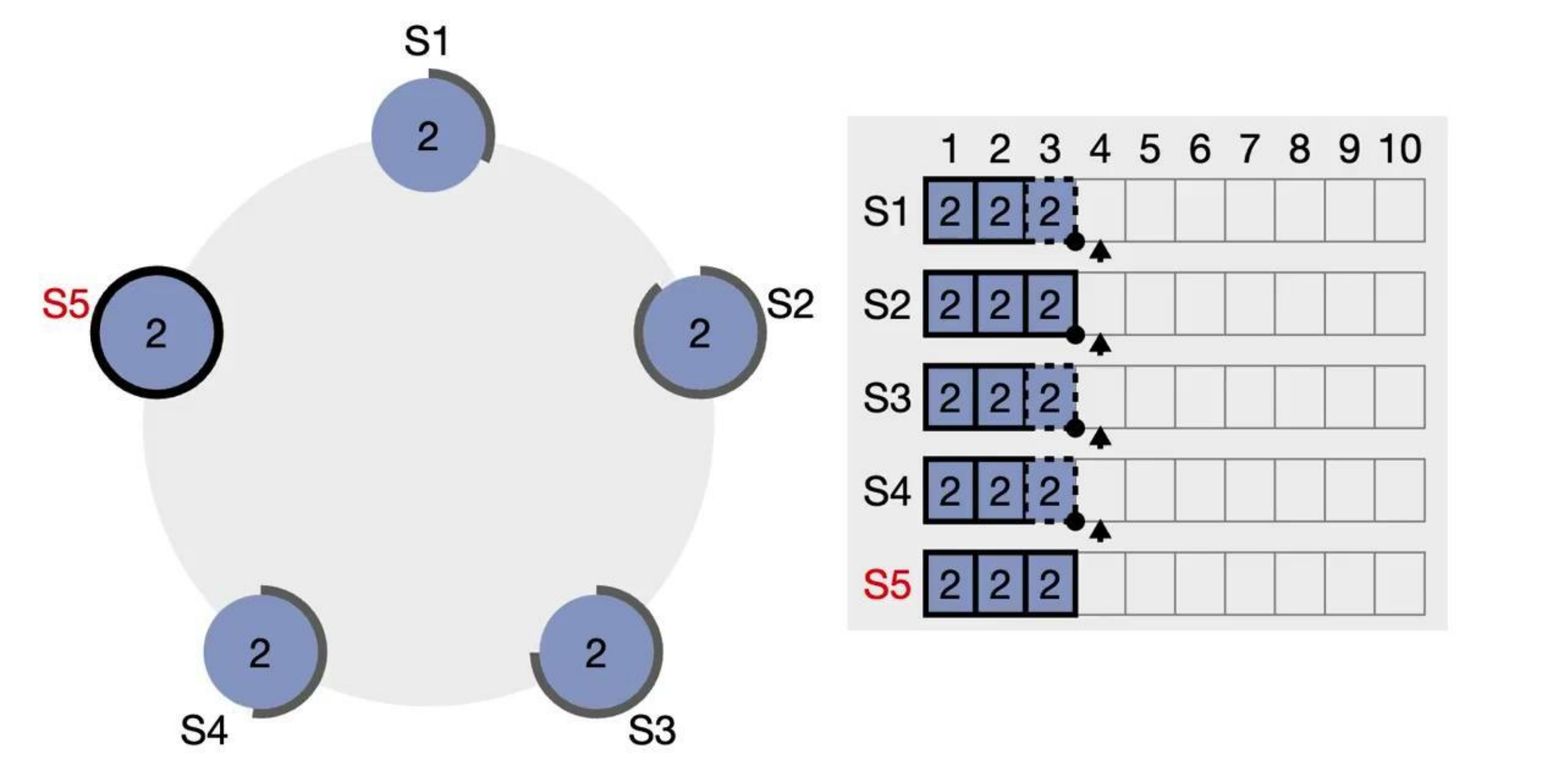
- Normal operation
  - System accepts log entries from clients
  - Log entries are replicated
- Leader election
  - Leader is detected to have failed
  - Remaining nodes vote on a new leader

Raft Node States

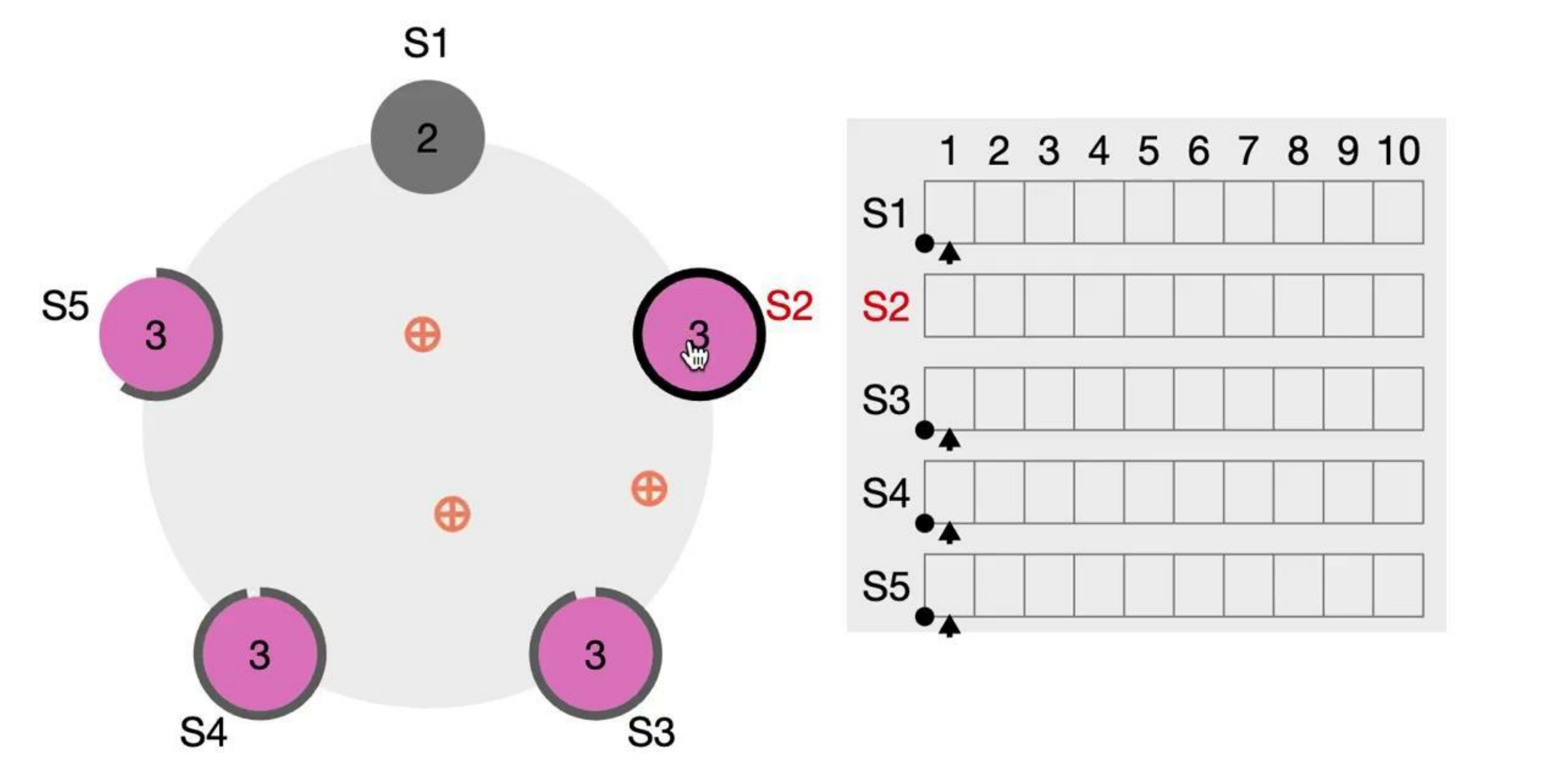
At any time a node is either a

- Leader
  - Accepts user input and replicates to followers
- Follower
  - Listens for leader heartbeats
  - Replicates log entries
  - Detect leader failure
- Candidate
  - Claim leadership for term i+1
  - Requests votes from other nodes

Normal Operation



Leader Election



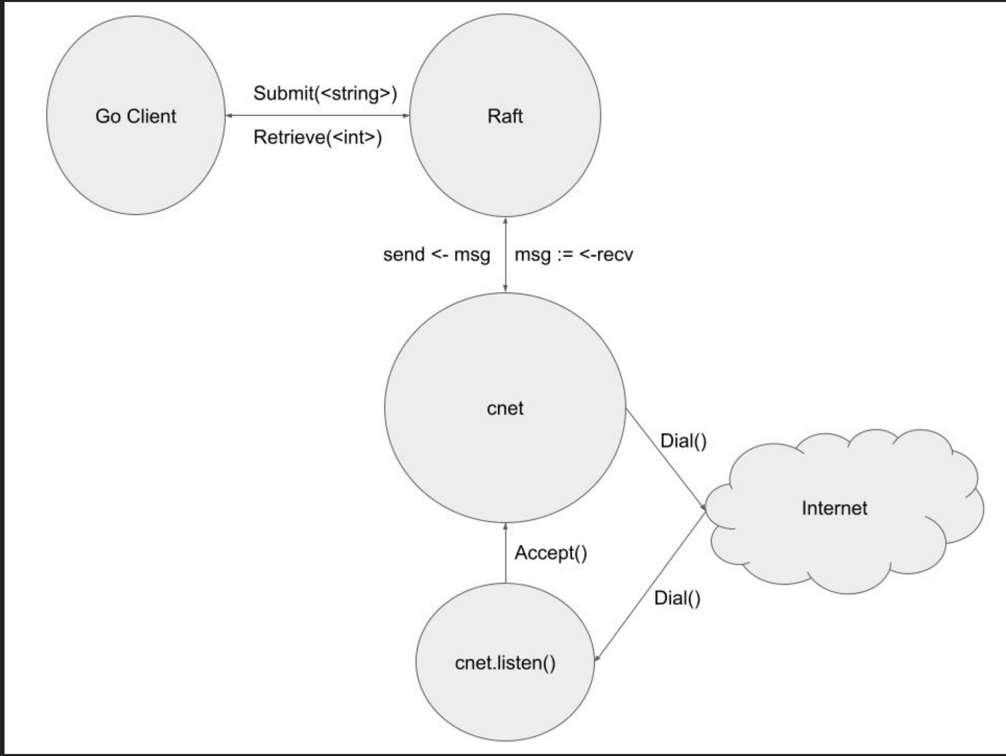
Design and Implementation

Implementation

- Design for a distributed system
- Interesting + key code segments

Design

- Circle == goroutine



Client API

Why does this block?

```go
func (r *Raft) Submit(msg string) Result {
	r.input <- Entry{
		Msg: msg,
		Id:  uuid.New(),
	}

	return <-r.output
}

func (r *Raft) Retrieve(start int) []Entry {
	start = max(start, 1)
	return r.log[start : r.commitIndex+1]
}
```

Run() goroutine

- Same select per role
- Leader & Follower iteration speed

```go
	for {
		r.scanForAwaiting()

		var timeout time.Duration
		var callback func(chan cnet.PeerMsg)
		switch r.role {
		case Leader:
			timeout = 100 * time.Millisecond
			callback = r.sendAppendMsg
		case Follower, Candidate:
			timeout = time.Duration(rand.Intn(500)+500) * time.Millisecond
			callback = r.becomeCandidate
		}

		select {
		// receive client command
		case entry := <-r.input:
			r.handleInput(entry, send)
		// handle append responses
		case am := <-appendChan:
			r.handleAppendMsg(am, send)
		// handle election
		case lm := <-leaderChan:
			r.handleLeaderMsg(lm, send)
		// send regular updates faster than heartbeat timeout
		case <-time.After(timeout):
			callback(send)
		}
	}
```

Message routing

```go
func route(recv chan cnet.PeerMsg, appendChan chan AppendMsg, leaderChan chan LeaderMsg) {
	for {
		// Read messages from recv
		pm := <-recv

		// Parse payload
		// Forward message to correct channel
		switch pm.Type {
		case LEADER:
			var lm LeaderMsg
			err := json.Unmarshal(pm.Msg, &lm)
			if err != nil {
				fmt.Println(err)
				continue
			}
			leaderChan <- lm
		case APPEND:
			var am AppendMsg
			err := json.Unmarshal(pm.Msg, &am)
			if err != nil {
				fmt.Println(err)
				continue
			}
			appendChan <- am
		}
	}
}
```

CNET - Simple networking

- Payload agnostic
- Runs in its own goroutine
- Failure agnostic
- Room for optimization

```go
type MessageType int

type PeerMsg struct {
	Src, Dst string
	Msg      []byte
	Type     MessageType
}
```

```go
func (n *Network) Run(send, recv chan PeerMsg) {
	connChan := make(chan net.Conn, 100)
	go n.listen(connChan)

	for {
		select {
		case conn := <-connChan:
			pm, err := recvMsg(conn)
			conn.Close()
			if err != nil {
				fmt.Println(err)
				continue
			}
			recv <- *pm
		case pm := <-send:
			n.sendMsg(pm)
		}
	}
}
```

Demos

Demo 1 - ./httpraft

- Code in network send() for delaying messages
- See the protocol updating in realtime

Demo 2 - ./fake-etcd

1. API definition
2. Data store definition
3. Interaction with the raft lib

Want to contribute?

- Log modularity
  - Rip it out and give it its own interface
  - DB, files, in-memory
- Leader-forwarding
  - Forward a request to the current leader on behalf of the client
- Network optimization
  - TCP → UDP

Summary

- Raft is a protocol for maintaining a replicated log
- Raft is resilient to crash-faults (fail-stop)
- General approach for DS software design using Go
- Demonstrated ease-of-use with `fake-etcd`