

Implementing Distributed Consensus

 Dan Lüdtkke



What?

- My hobby project of learning about Distributed Consensus
 - I implemented a Paxos variant in Go and learned a lot about reaching consensus
 - A fine selection of some of the mistakes I made

Why?

- I wanted to understand Distributed Consensus
 - Everyone seemed to understand it. Except me.
- I am a hands-on person.
 - Doing \$stuff > Reading about \$stuff

Why talk about it?

- Knowledge sharing

Distributed Consensus

Protocols

- Paxos
 - Multi-Paxos
 - Cheap Paxos
- Raft



- ZooKeeper Atomic Broadcast
- Proof-of-Work Systems
 - Bitcoin
- Lockstep Anti-Cheating



Implementations

- Chubby
 - coarse grained lock service
- etcd
 - a distributed key value store



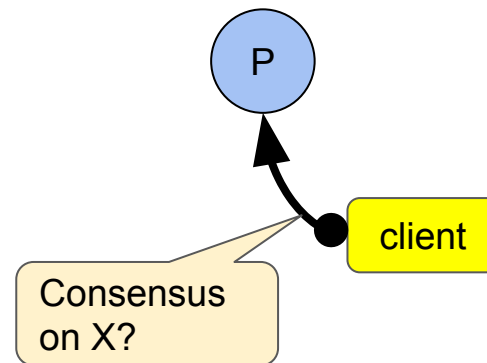
- Apache ZooKeeper
 - a centralized service for maintaining configuration information, naming, providing distributed synchronization



Paxos

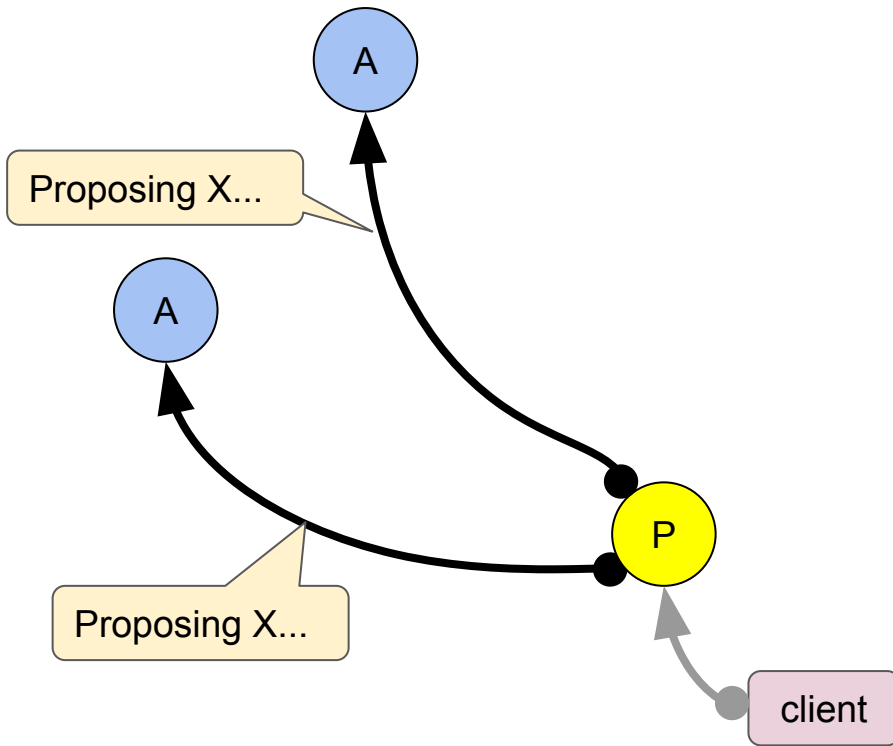
Paxos Roles

- Client
 - Issues request to a *proposer*
 - Waits for response from a *learner*
 - Consensus on value X
 - No consensus on value X
- Acceptor
- Proposer
- Learner
- Leader



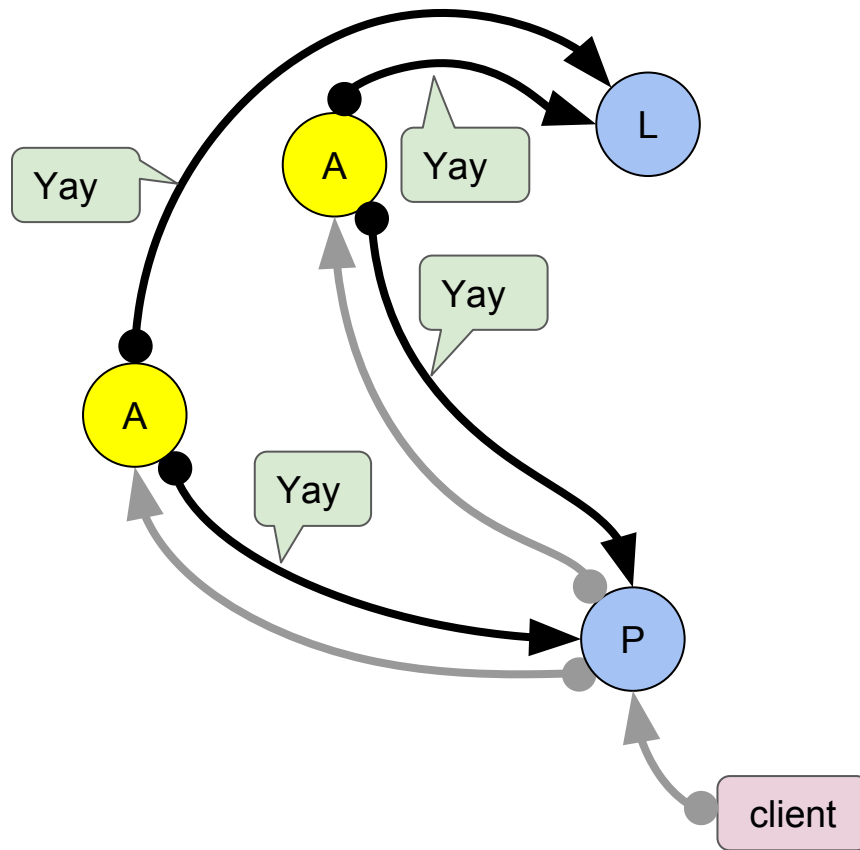
Paxos Roles

- Client
- Proposer (P)
 - Advocates a *client* request
 - Asks acceptors to agree on the proposed value
 - Move the protocol forward when there is conflict
- Acceptor
- Learner
- Leader



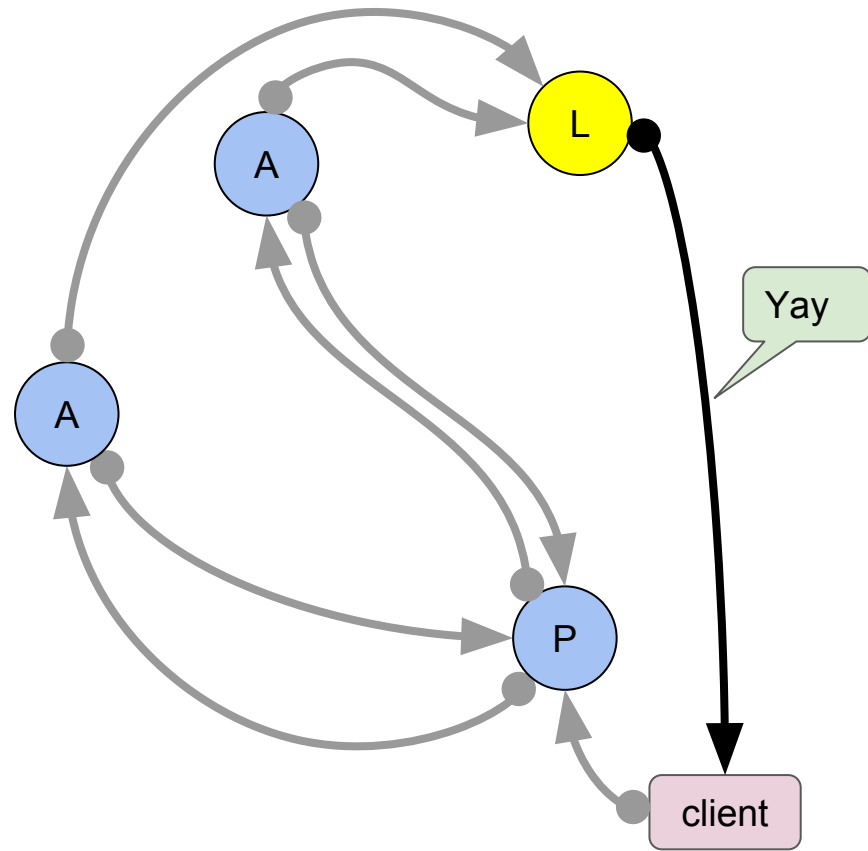
Paxos Roles

- Client
- Proposer (P)
- Acceptor (A)
 - Also called "voter"
 - The Fault-tolerant "memory" of the system
 - Groups of acceptors form a *quorum*
- Learner
- Leader



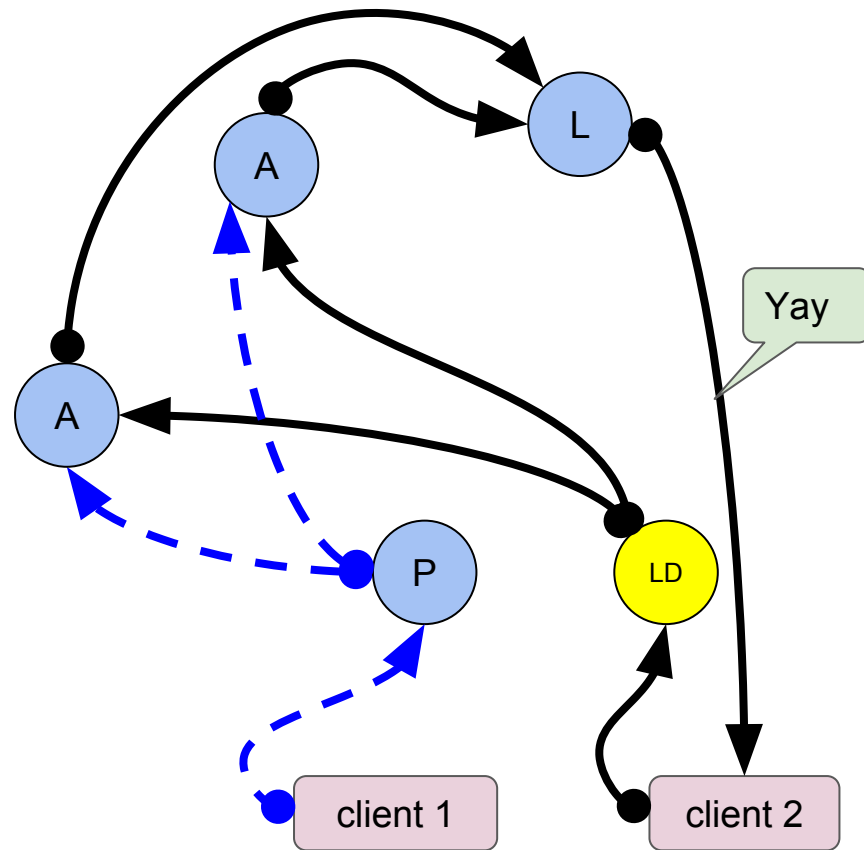
Paxos Roles

- Client
- Proposer (P)
- Acceptor (A)
- Learner (L)
 - Adds replication to the protocol
 - Takes action on learned (agreed on) values
 - E.g. respond to *client*
- Leader



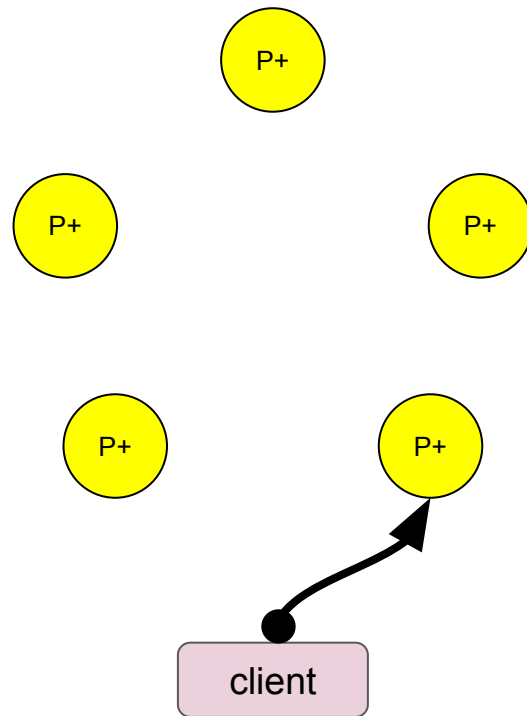
Paxos Roles

- Client
- Proposer (P)
- Acceptor (A)
- Learner (L)
- Leader (LD)
 - Distinguished *proposer*
 - The only *proposer* that can make progress
 - Multiple *proposers* may believe to be leader
 - *Acceptors* decide which one gets a majority



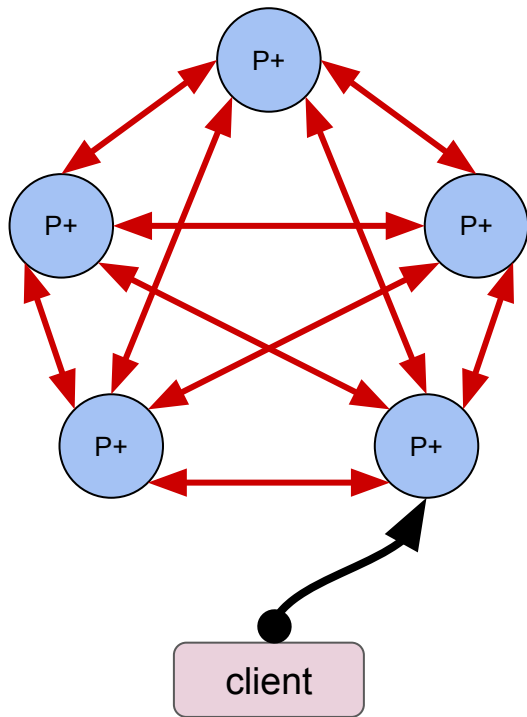
Coalesced Roles

- A single processors can have multiple roles
- P+
 - Proposer
 - Acceptor
 - Learner
- Client talks to any processor
 - Nearest one?
 - Leader?



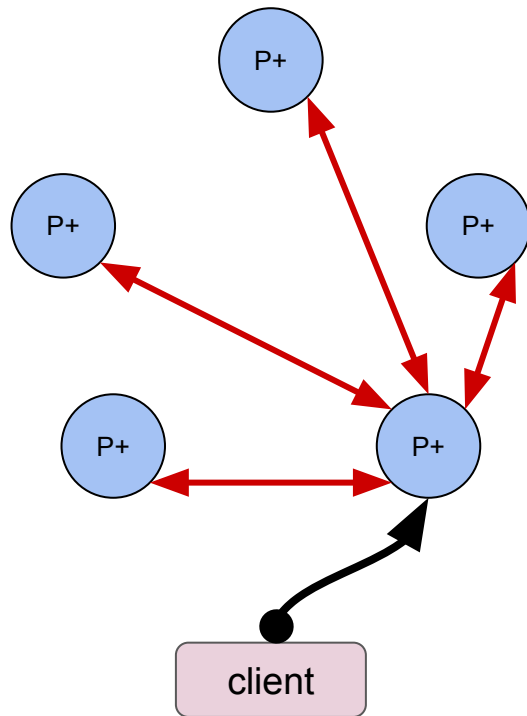
Coalesced Roles at Scale

- P+ system is a complete digraph
 - a directed graph in which every pair of distinct vertices is connected by a pair of unique edges
 - *Everyone talks to everyone*
- Let **n** be the number of processors
 - a.k.a. Quorum Size
- **Connections** = **n** * (**n** - 1)
 - Potential network (TCP) connections

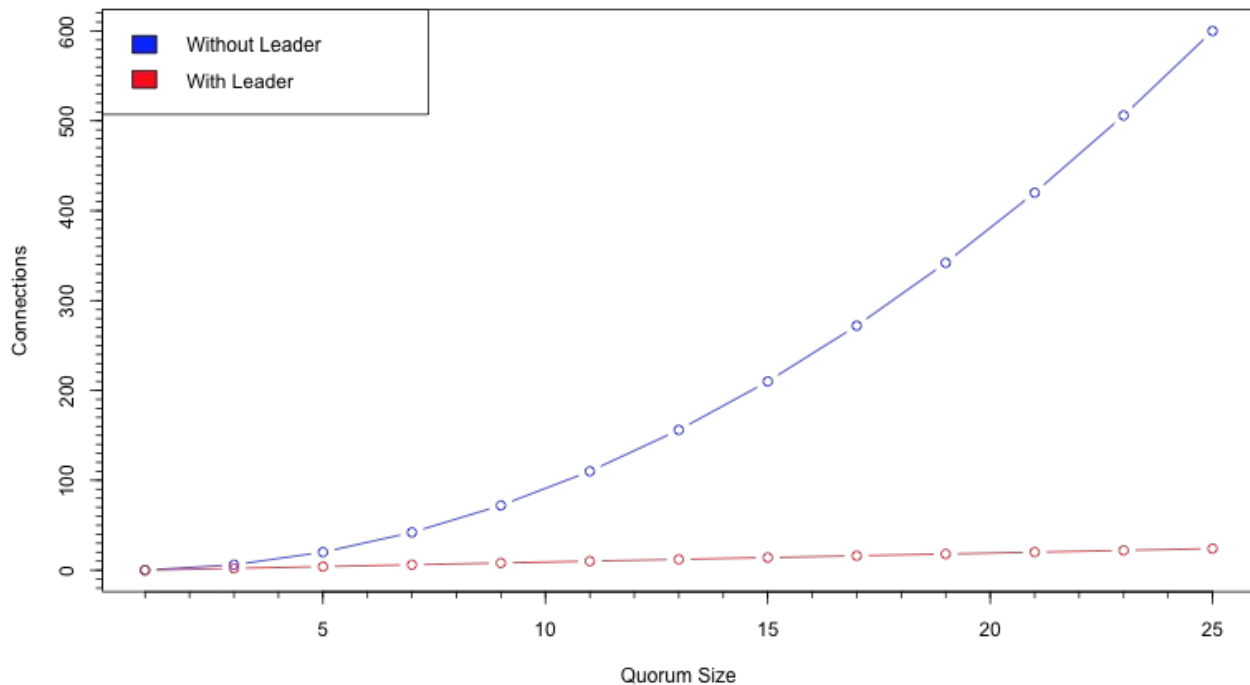


Coalesced Roles with Leader

- P+ system with a leader is a directed graph
 - *Leader talks to everyone else*
- Let **n** be the number of processors
 - a.k.a. Quorum Size
- **Connections** = **n** - 1
 - Network (TCP) connections



Coalesced Roles at Scale



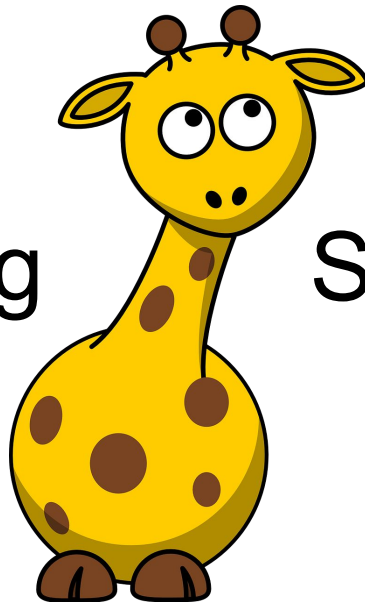
Limitations

- Single consensus
 - Once consensus has been reached no more progress can be made
 - But: Applications can start new Paxos runs
- Multiple proposers may believe to be the leader
 - dueling proposers
 - theoretically infinite duell
 - practically retry-limits and jitter helps
- Standard Paxos not resilient against Byzantine failures
 - Byzantine: Lying or compromised processors
 - Solution: Byzantine Paxos Protocol



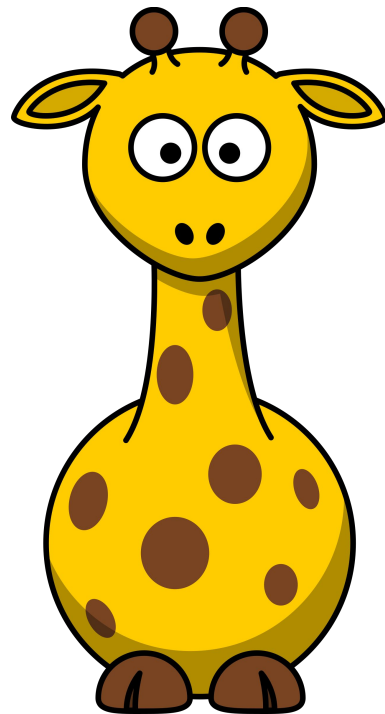
Introducing Skinny

- Paxos-based
- Feature-free
- Educational
- Lock Service



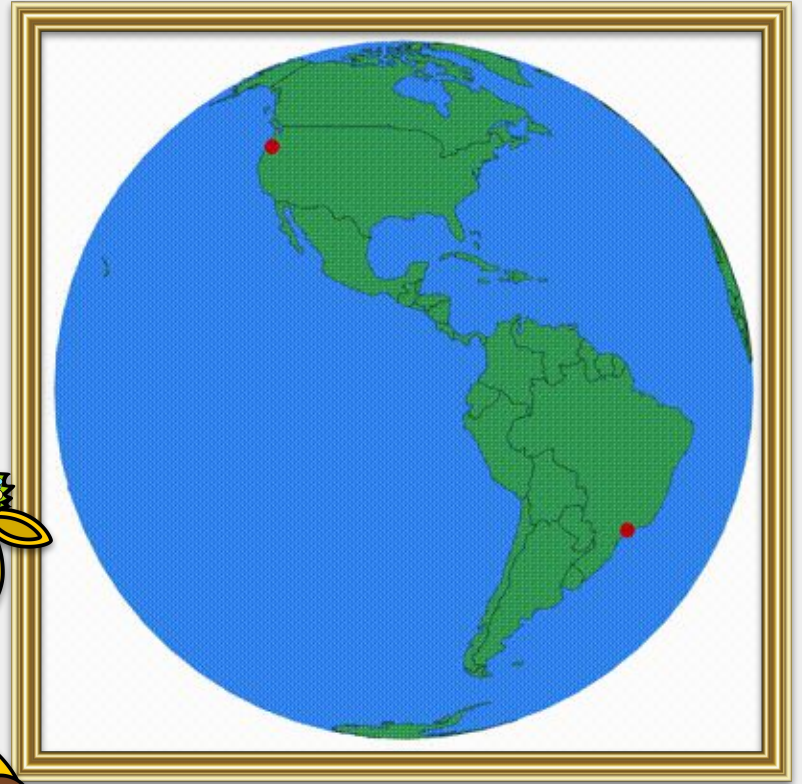
Skinny "Features"

- Easy to understand and observe
- Coalesced Roles
- Single Lock
 - Locks are always advisory!
 - A lock service does not enforce obedience to locks.
- Go
- Protocol Buffers
- gRPC
- Do not use in production!



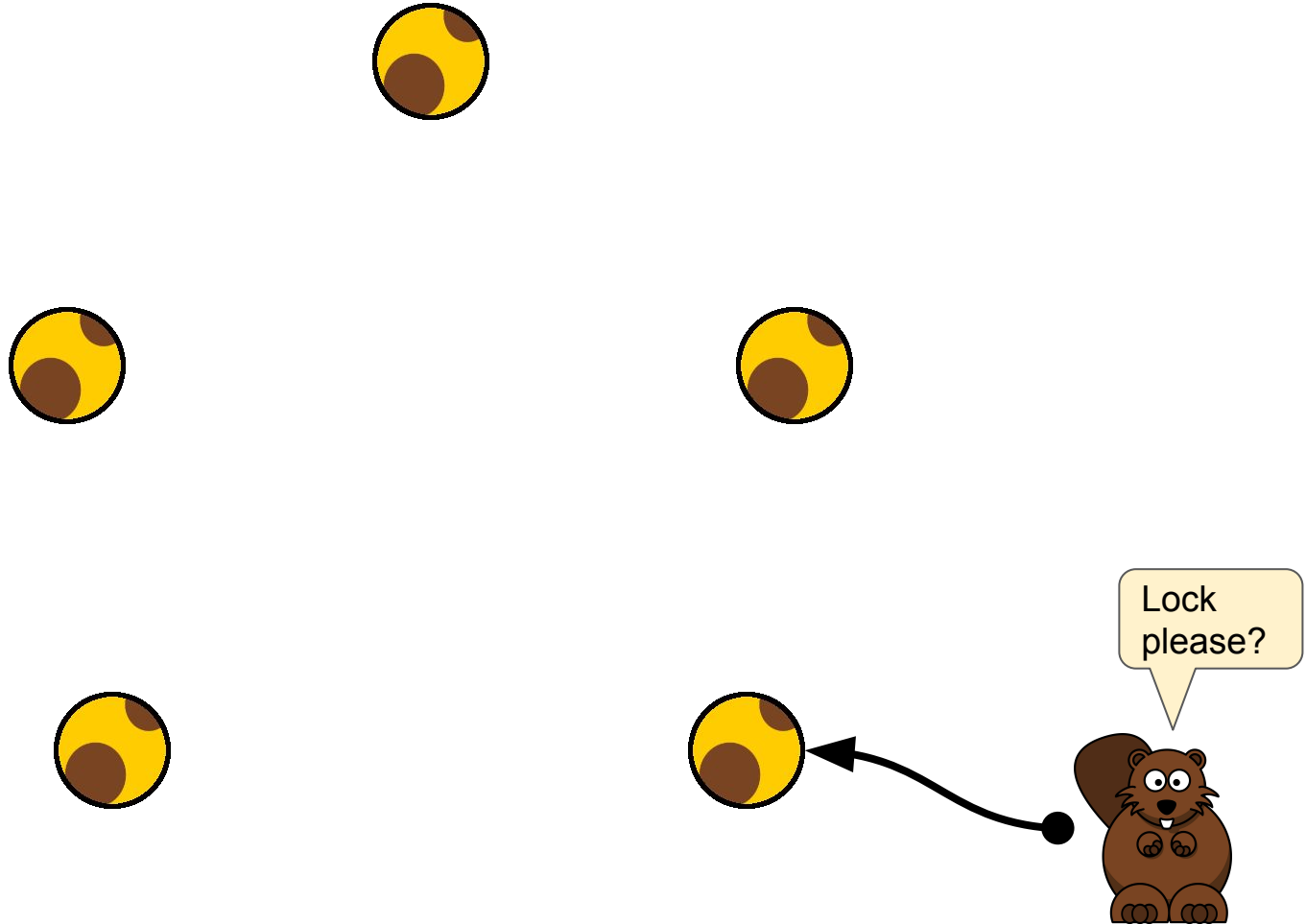
Assuming...

- Oregon
 - North America
- São Paulo
 - South America
- London
 - Europe
- Taiwan
 - Asia
- Sydney
 - Australia

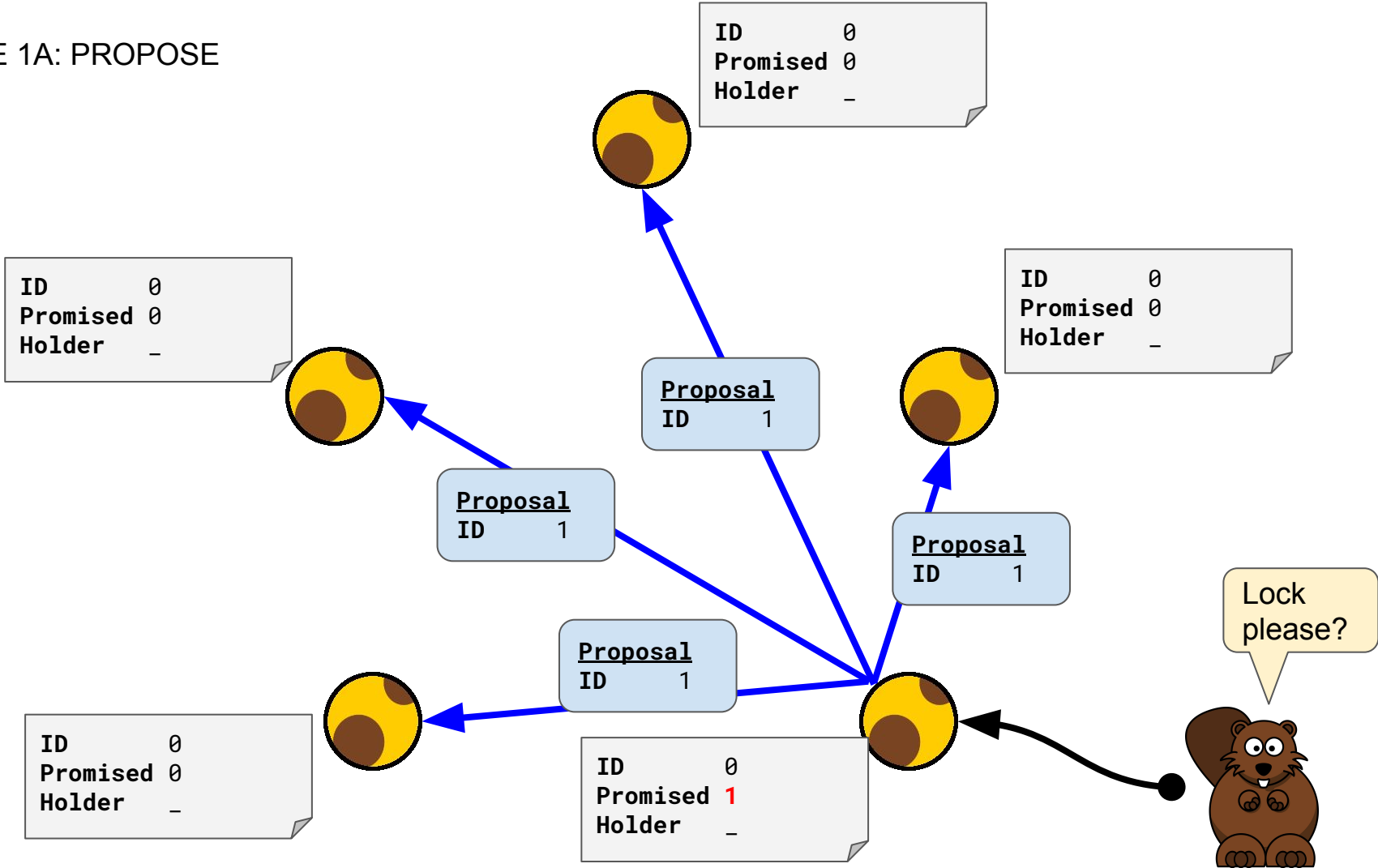


How Skinny reaches consensus

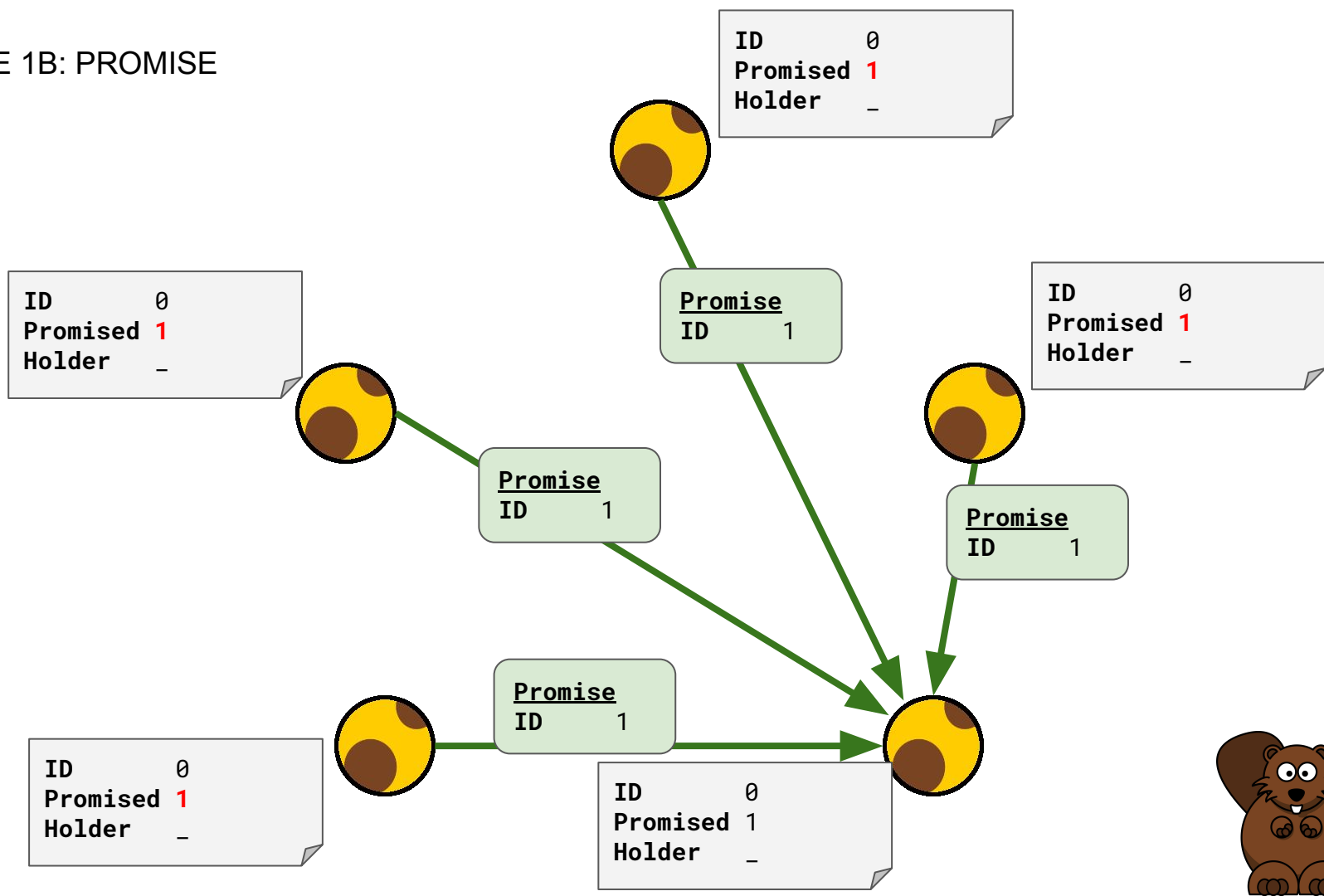
SKINNY QUORUM



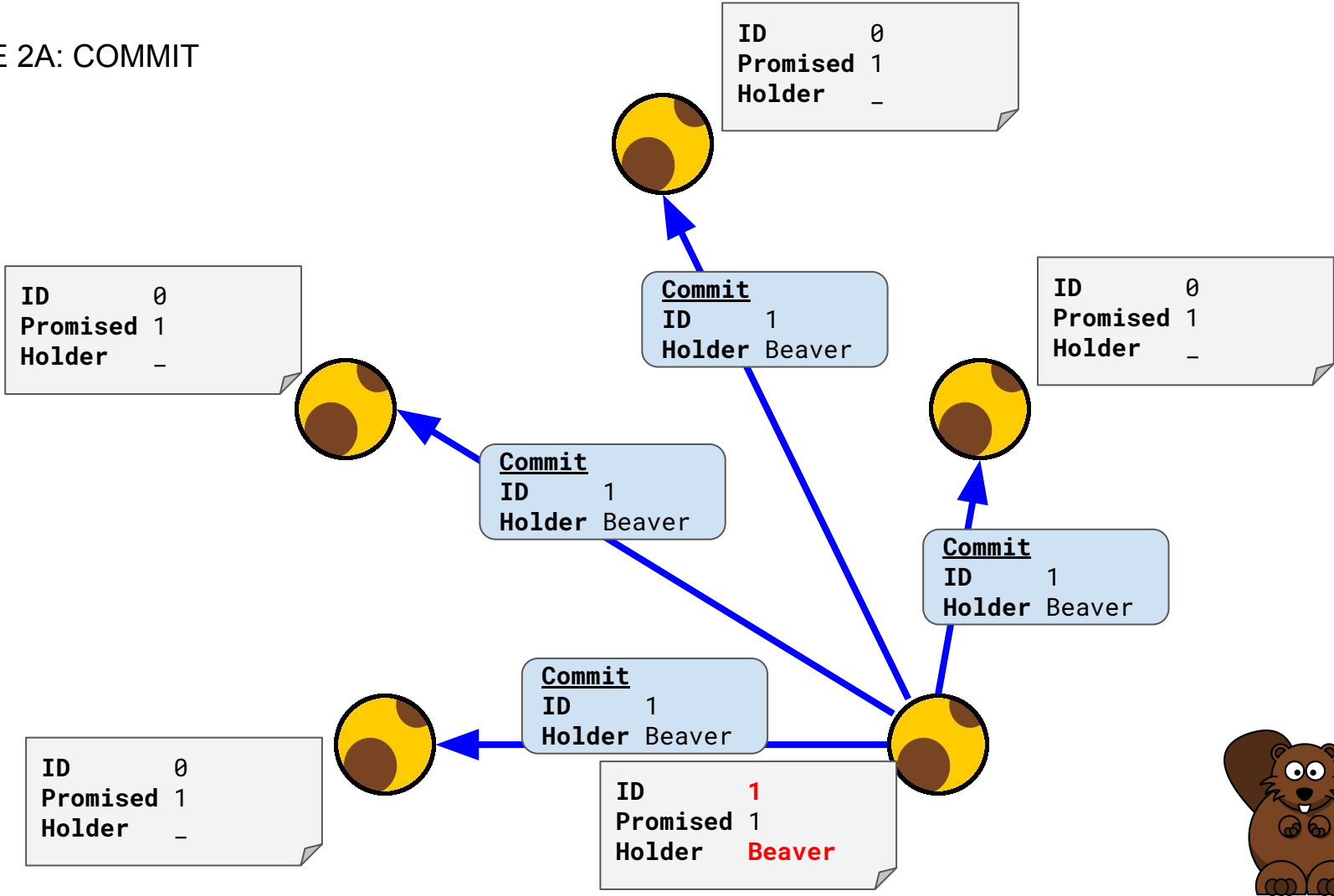
PHASE 1A: PROPOSE



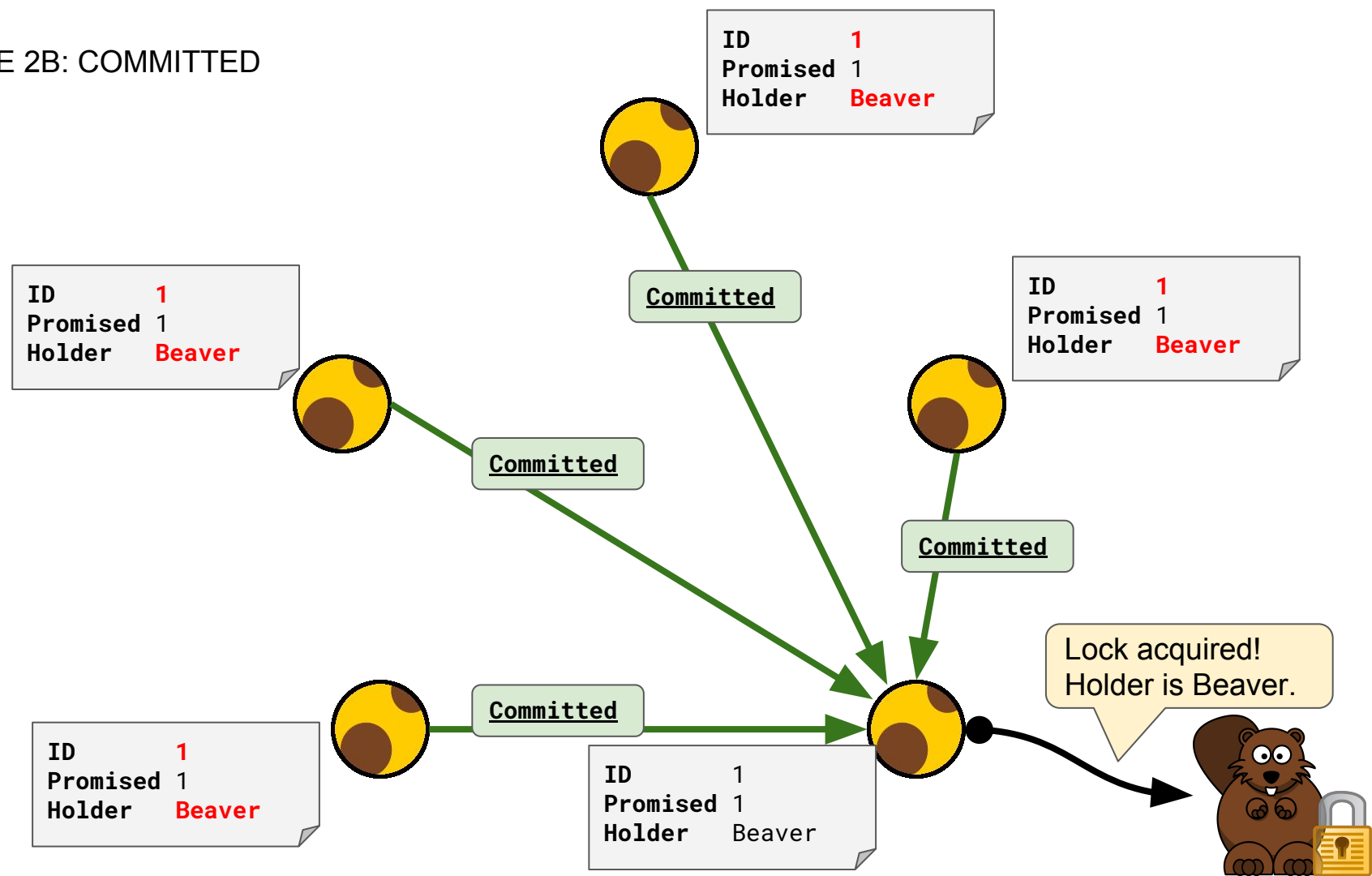
PHASE 1B: PROMISE



PHASE 2A: COMMIT



PHASE 2B: COMMITTED



How Skinny deals with Instance Failure

SCENARIO

| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |



| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |



| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |



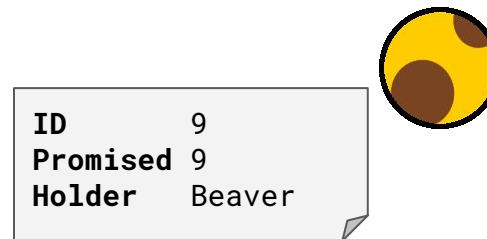
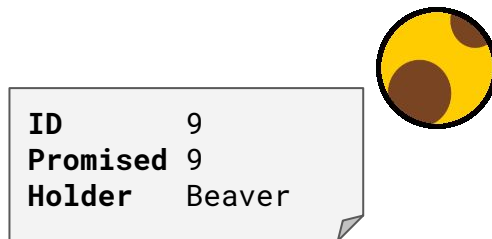
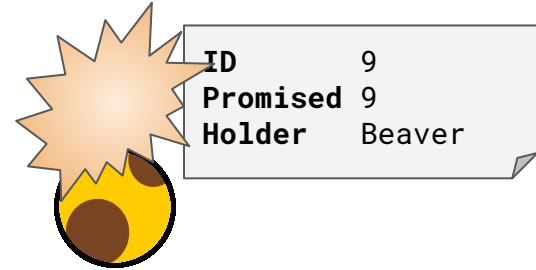
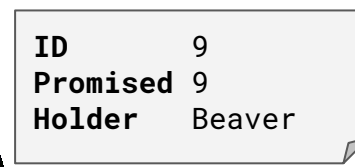
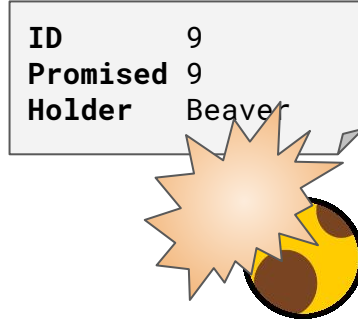
| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |



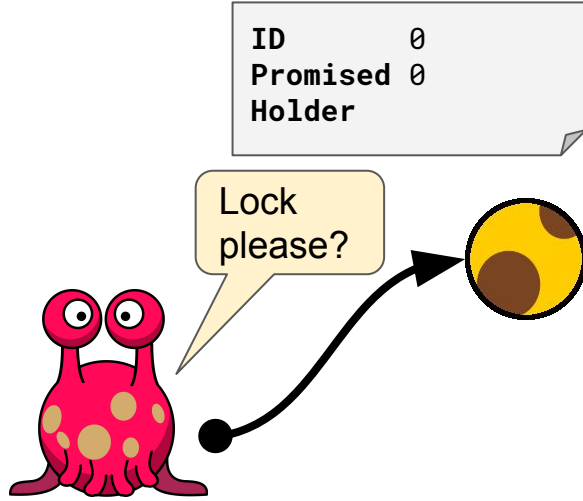
| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |



TWO INSTANCES FAIL



INSTANCES ARE BACK
BUT STATE IS LOST




| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |




| | |
|----------|---|
| ID | 0 |
| Promised | 0 |
| Holder | |

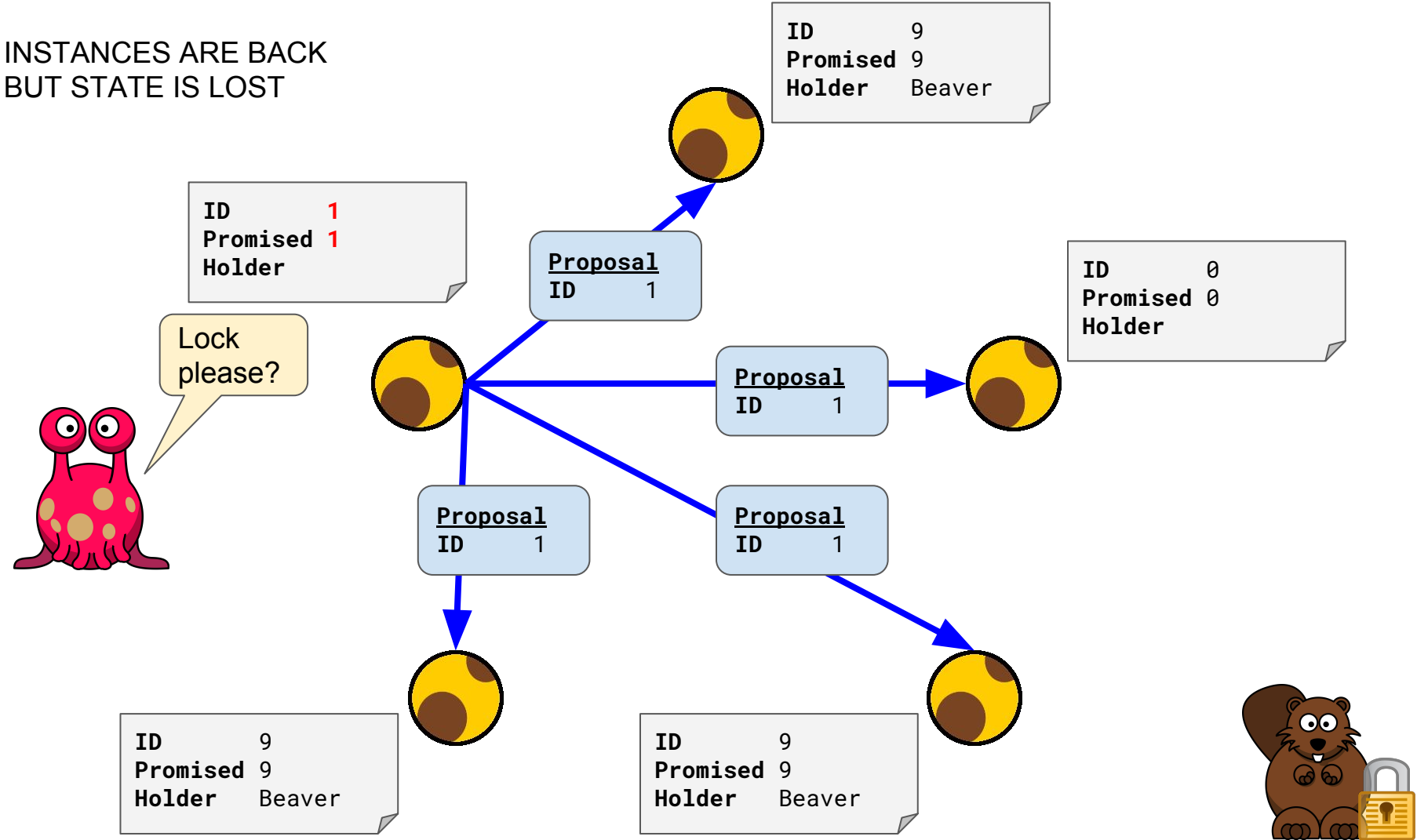
| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |



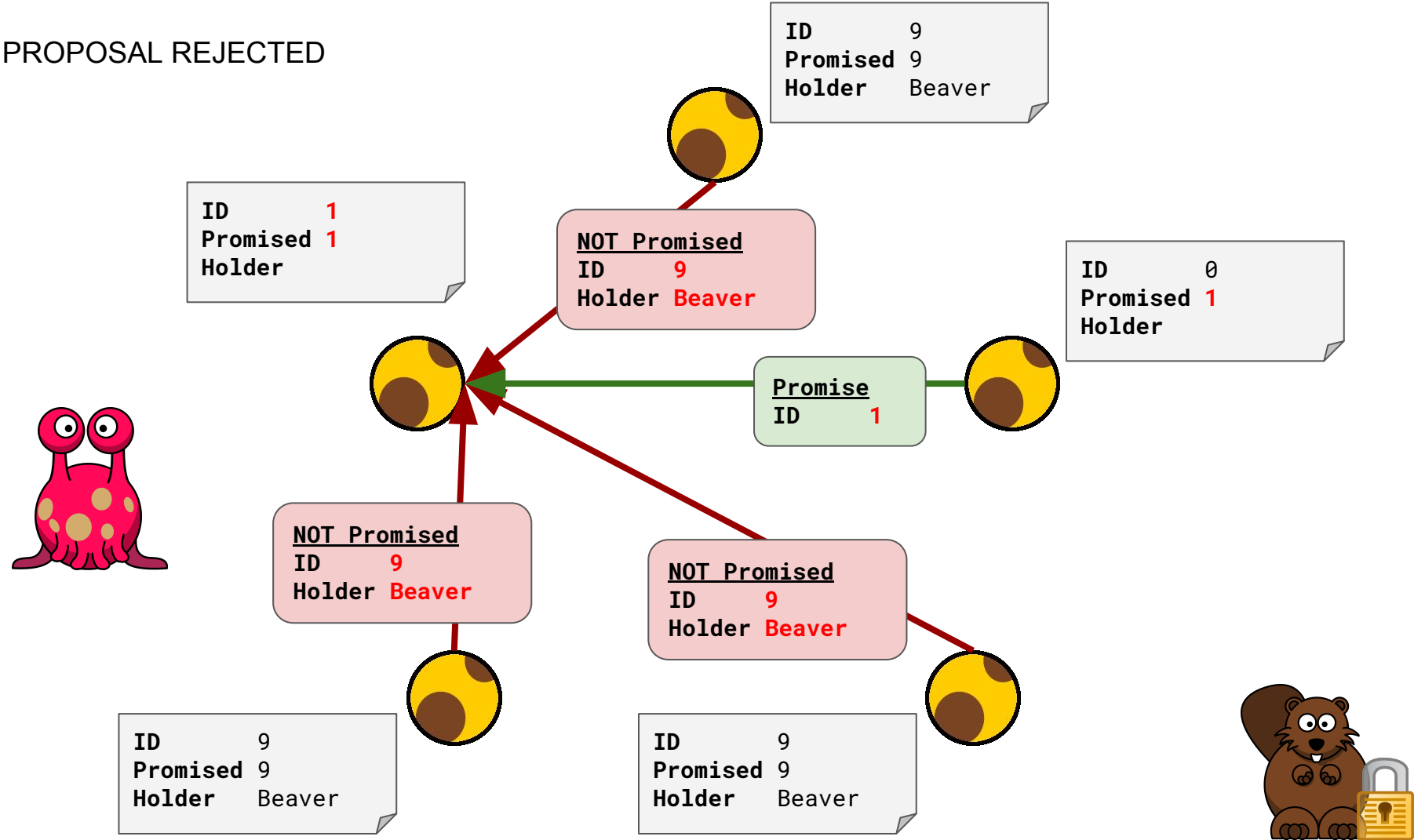
| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |



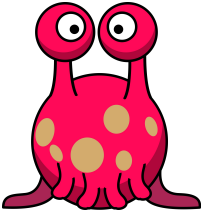
INSTANCES ARE BACK
BUT STATE IS LOST



PROPOSAL REJECTED

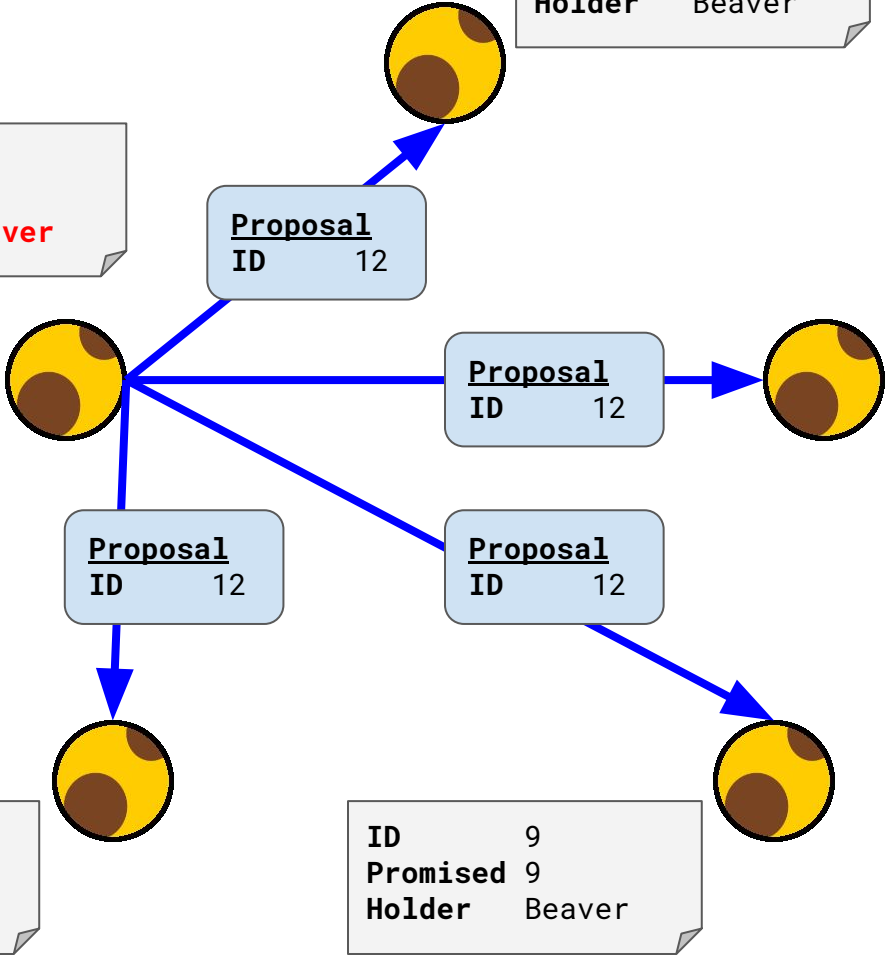


START NEW PROPOSAL
WITH LEARNED VALUES



| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |

| | |
|----------|--------|
| ID | 9 |
| Promised | 12 |
| Holder | Beaver |



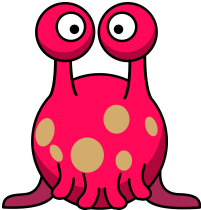
| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |

| | |
|----------|--------|
| ID | 9 |
| Promised | 9 |
| Holder | Beaver |

| | |
|----------|---|
| ID | 0 |
| Promised | 1 |
| Holder | |



PROPOSAL ACCEPTED



ID 9
Promised 12
Holder Beaver

ID 9
Promised 12
Holder Beaver

Promise
ID 12

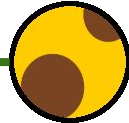


Promise
ID 12



ID 9
Promised 12
Holder Beaver

Promise
ID 12



ID 0
Promised 12
Holder

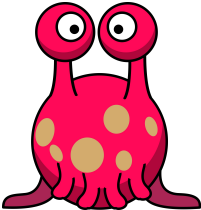
Promise
ID 12



ID 9
Promised 12
Holder Beaver



COMMIT LEARNED VALUE



| | |
|----------|--------|
| ID | 9 |
| Promised | 12 |
| Holder | Beaver |

| | |
|----------|--------|
| ID | 12 |
| Promised | 12 |
| Holder | Beaver |

| | |
|---------------|--------|
| <u>Commit</u> | |
| ID | 12 |
| Holder | Beaver |

| | |
|---------------|--------|
| <u>Commit</u> | |
| ID | 12 |
| Holder | Beaver |

| | |
|---------------|--------|
| <u>Commit</u> | |
| ID | 12 |
| Holder | Beaver |

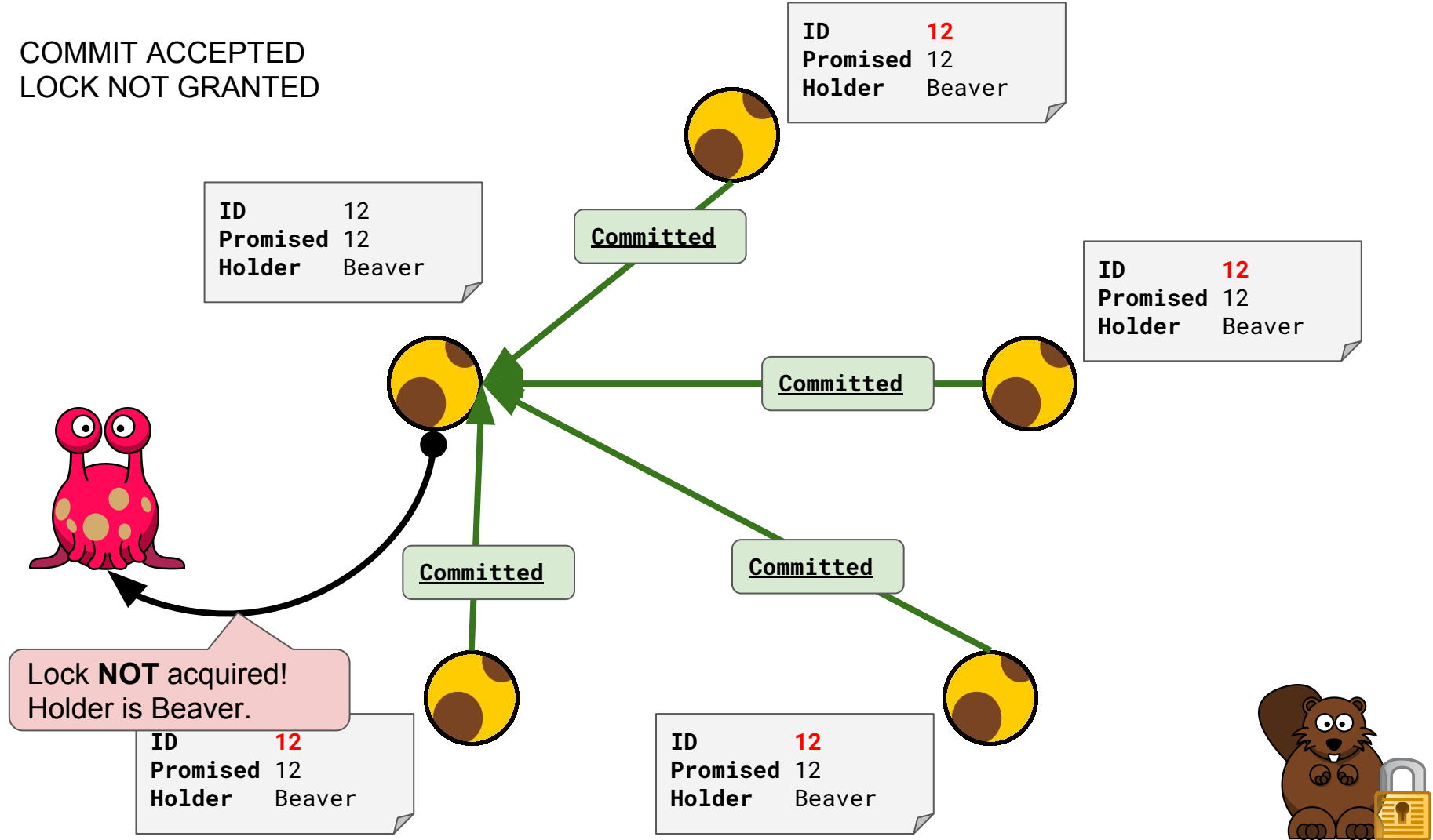
| | |
|----------|--------|
| ID | 9 |
| Promised | 12 |
| Holder | Beaver |

| | |
|----------|--------|
| ID | 9 |
| Promised | 12 |
| Holder | Beaver |

| | |
|----------|--------|
| ID | 9 |
| Promised | 12 |
| Holder | Beaver |



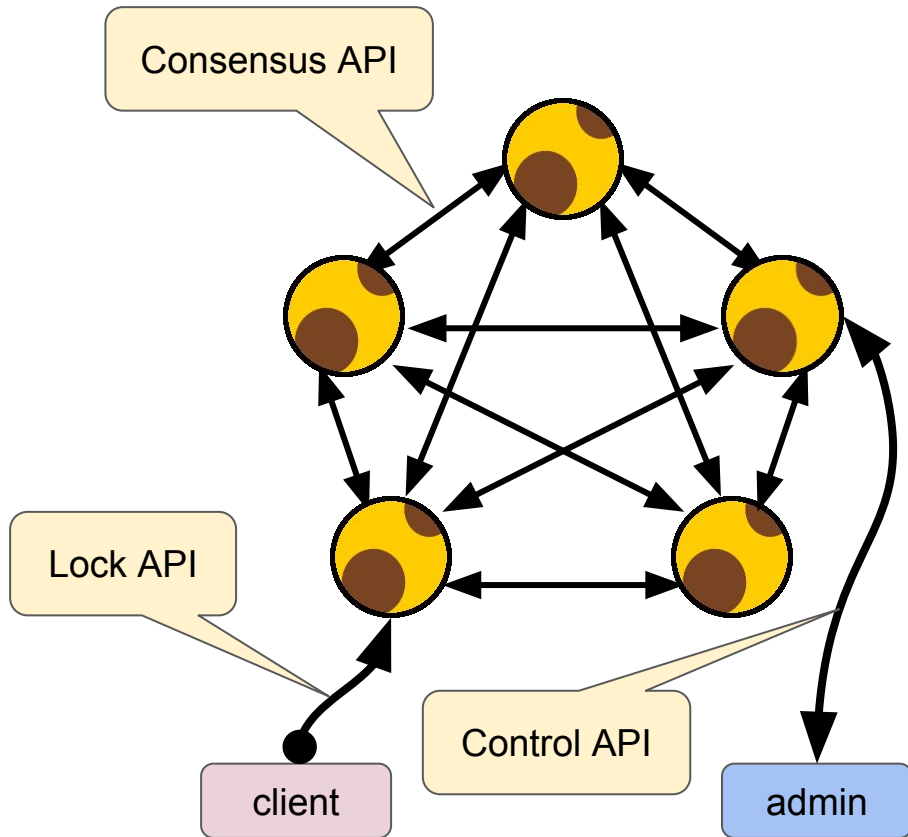
COMMIT ACCEPTED
LOCK NOT GRANTED



Skinny APIs

Skinny APIs

- Lock API
 - Used by clients to acquire or release a lock
- Consensus API
 - Used by Skinny instances to reach consensus
- Control API
 - Used by us to observe what's happening

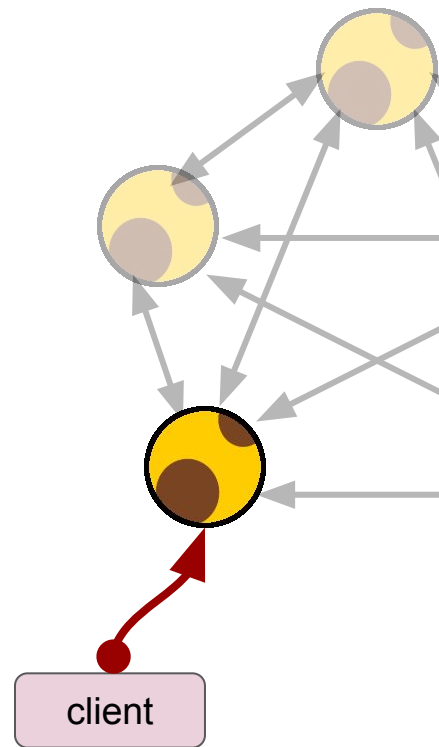


Lock API

```
message AcquireRequest {  
    string Holder = 1;  
}  
message AcquireResponse {  
    bool Acquired = 1;  
    string Holder = 2;  
}
```

```
message ReleaseRequest {}  
message ReleaseResponse {  
    bool Released = 1;  
}
```

```
service Lock {  
    rpc Acquire(AcquireRequest) returns (AcquireResponse);  
    rpc Release(ReleaseRequest) returns (ReleaseResponse);  
}
```

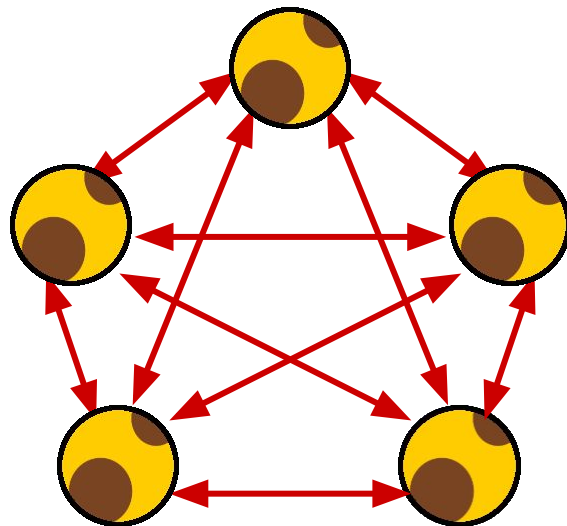


Consensus API

```
// Phase 1: Promise
message PromiseRequest {
    uint64 ID = 1;
}
message PromiseResponse {
    bool Promised = 1;
    uint64 ID = 2;
    string Holder = 3;
}
```

```
// Phase 2: Commit
message CommitRequest {
    uint64 ID = 1;
    string Holder = 2;
}
message CommitResponse {
    bool Committed = 1;
}
```

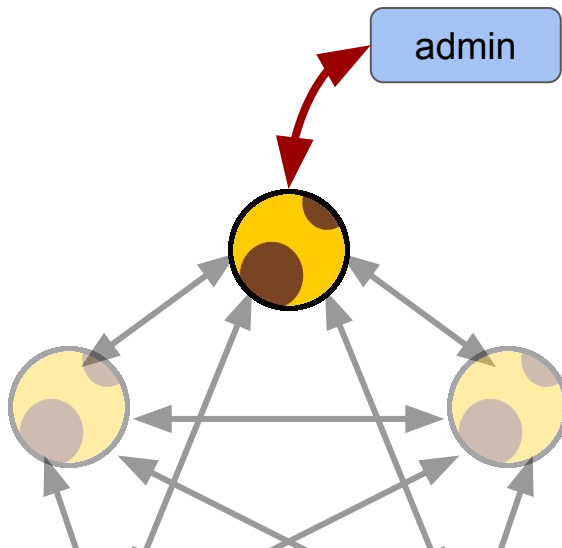
```
service Consensus {
    rpc Promise (PromiseRequest) returns (PromiseResponse);
    rpc Commit (CommitRequest) returns (CommitResponse);
}
```



Control API

```
message StatusRequest {}  
message StatusResponse {  
    string Name = 1;  
    uint64 Increment = 2;  
    string Timeout = 3;  
    uint64 Promised = 4;  
    uint64 ID = 5;  
    string Holder = 6;  
    message Peer {  
        string Name = 1;  
        string Address = 2;  
    }  
    repeated Peer Peers = 7;  
}
```

```
service Control {  
    rpc Status(StatusRequest) returns (StatusResponse);  
}
```



~~My Stupid Mistakes~~

My Awesome Learning Opportunities

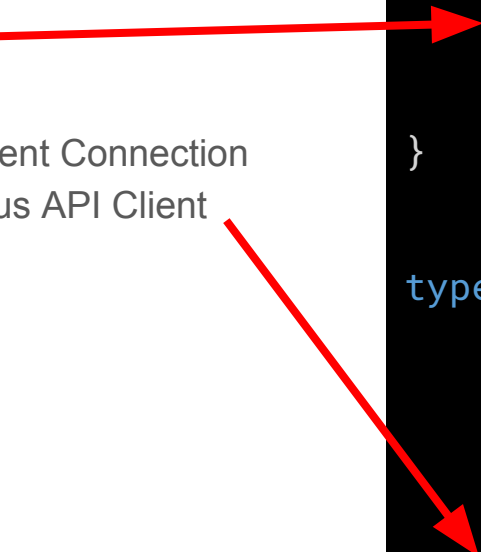
Reaching Out...

Skinny Instance

- List of peers
 - All other instances in the quorum
- Peer
 - gRPC Client Connection
 - Consensus API Client

```
// Instance represents a skinny instance
type Instance struct {
    mu sync.RWMutex
    // begin protected fields
    ...
    peers    []*peer
    // end protected fields
}

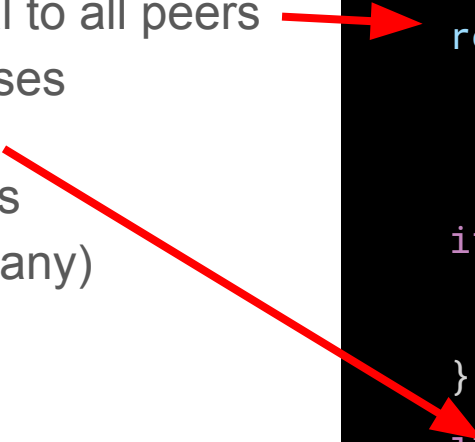
type peer struct {
    name      string
    address   string
    conn      *grpc.ClientConn
    client    pb.ConsensusClient
}
```

Two red arrows originate from the list on the left. One arrow points from the text 'All other instances in the quorum' to the 'peers []*peer' field in the 'Instance' struct. The other arrow points from the text 'Consensus API Client' to the 'client pb.ConsensusClient' field in the 'peer' struct.

Propose Function

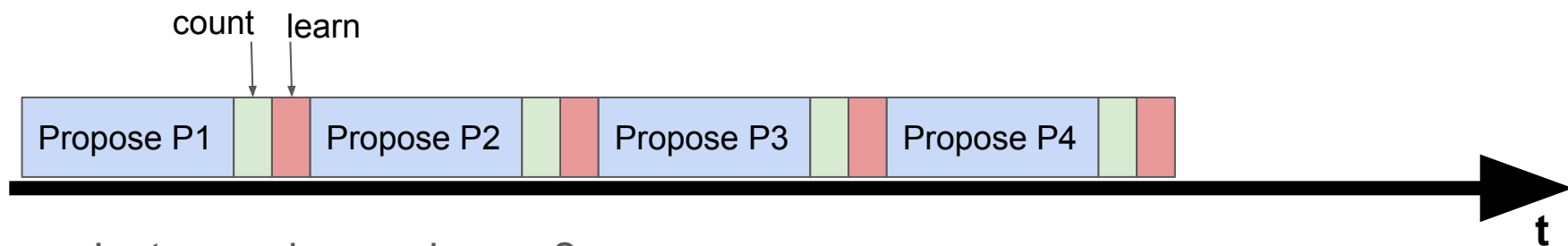
1. Send proposal to all peers
2. Count responses
 - Promises
3. Learn previous consensus (if any)

```
for _, p := range in.peers {  
    // send proposal  
    resp, err := p.client.Promise(  
        context.Background(),  
        &pb.PromiseRequest{ID: proposal})  
    if err != nil {  
        continue  
    }  
    if resp.Promised {  
        yay++  
    }  
    learn(resp)  
}
```

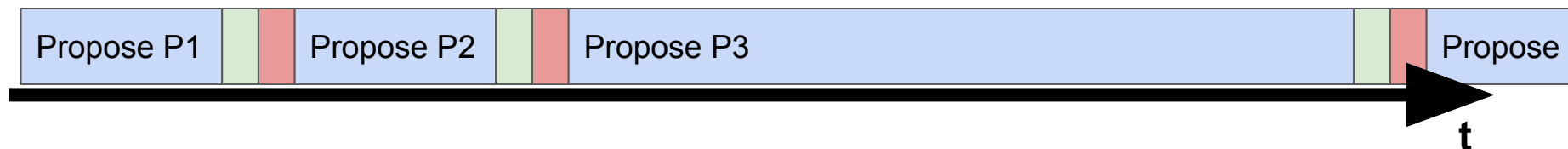


Resulting Behavior

- Sequentiell Requests
- Waiting for IO

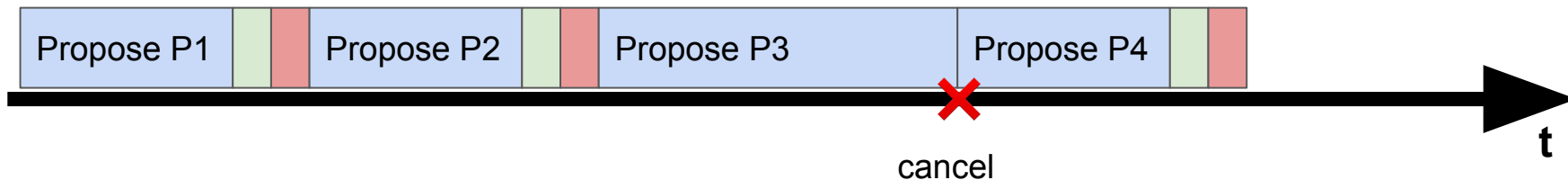


- Instance slow or down...?



Improvement #1

- Limit the Waiting for IO



Timeouts

- This example
 - Hardcoded
- Actual
 - From Configuration

```
for _, p := range in.peers {  
    // send proposal  
    → ctx, cancel := context.WithTimeout(  
    →     context.Background(),  
    →     time.Second*3)  
    resp, err := p.client.Promise(ctx,  
        &pb.PromiseRequest{ID: proposal})  
    → cancel() // prevent context leak  
    if err != nil {  
        continue  
    }  
    if resp.Promised {  
        yay++  
    }  
    learn(resp)  
}
```

Improvement #2 (Idea)

- Parallel Requests



- What's wrong?

Improvement #2 (Corrected)

- Parallel Requests
- Synchronized Counting
- Synchronized Learning



Parallelism

- Goroutine!
- Context with timeout
- But how to handle success?

```
for _, p := range in.peers {  
    // send proposal  
    → go func(p *peer) {  
        → ctx, cancel := context.WithTimeout(  
            context.Background(),  
            time.Second*3)  
            → defer cancel()  
  
            resp, err := p.client.Promise(ctx,  
                &pb.PromiseRequest{ID: proposal})  
            if err != nil { return }  
  
            → // now what?  
        }(p)  
    }  
}
```


Synchronizing

- Channels to the rescue!

```
→ type response struct {  
    from      string  
    promised  bool  
    id        uint64  
    holder    string  
}  
→ responses := make(chan *response)  
for _, p := range in.peers {  
    go func(p *peer) {  
        ...  
        → responses <- &response{  
            from:      p.name,  
            promised:  resp.Promised,  
            id:        resp.ID,  
            holder:    resp.Holder,  
        }  
    }(p)  
}
```

Synchronizing

- Counting
- `yay := 1`
 - Because we always vote for ourselves
- Learning



```
// count the votes
yay, nay := 1, 0
for r := range responses {
    // count the promises
    if r.promised {
        yay++
    } else {
        nay++
    }
    in.learn(r)
}
```

What's wrong?

- We never close the channel
- **range** is blocking forever

```
→ responses := make(chan *response)
for _, p := range in.peers {
    go func(p *peer) {
        ...
        responses <- &response{...}
    }(p)
}

// count the votes
yay, nay := 1, 0
→ for r := range responses {
    // count the promises
    ...
    in.learn(r)
}
```

More synchronizing

- Use Waitgroup
- Close Channel when all requests are done




```
responses := make(chan *response)
wg := sync.WaitGroup{}
for _, p := range in.peers {
    wg.Add(1)
    go func(p *peer) {
        defer wg.Done()
        ...
        responses <- &response{...}
    }(p)
}
// close responses channel
go func() {
    wg.Wait()
    close(responses)
}()
// count the promises
for r := range responses {...}
```

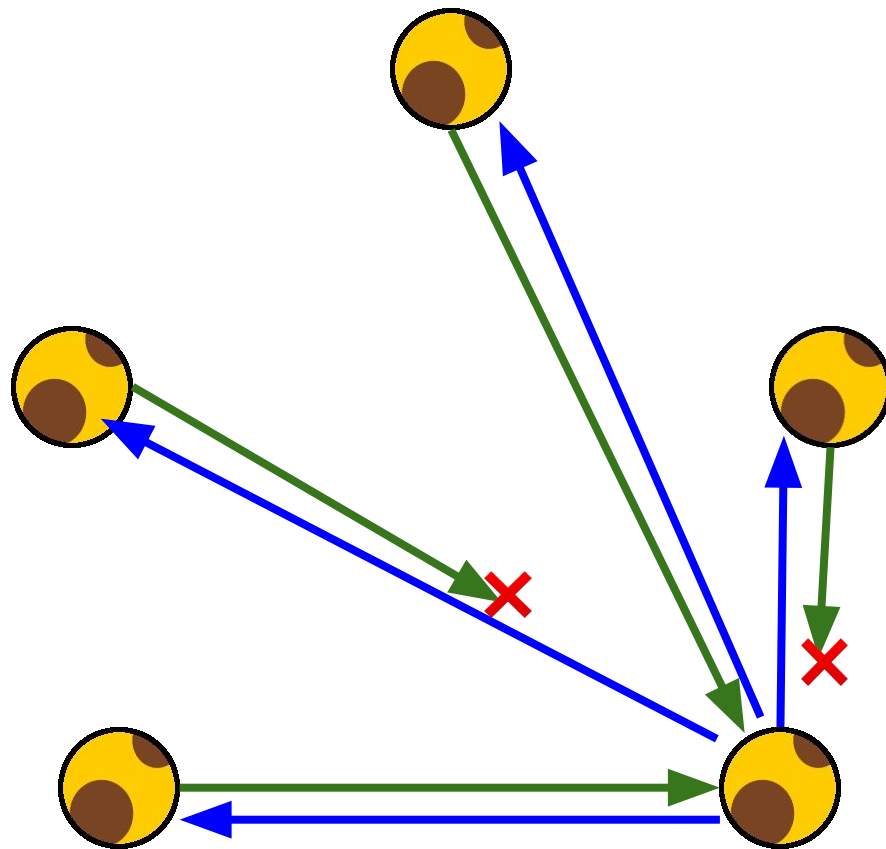
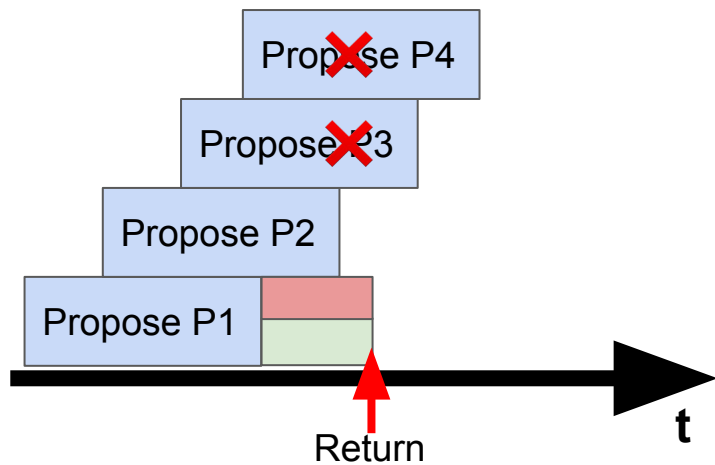
Result



Ignorance Is Bliss?

Early Stopping

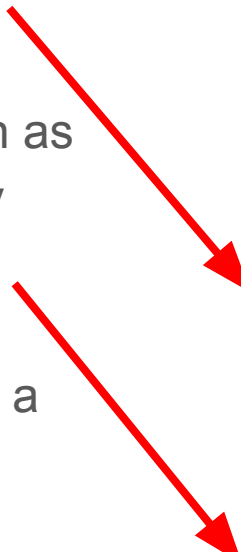
Yay:   
Majority



Early Stopping (1)

- One context for all outgoing promises
- We cancel as soon as we have a majority
- We always cancel before leaving the function to prevent a context leak

```
type response struct {  
    from      string  
    promised  bool  
    id        uint64  
    holder    string  
}  
  
responses := make(chan *response)  
  
ctx, cancel := context.WithTimeout(  
    context.Background(),  
    time.Second*3)  
  
defer cancel()
```



Early Stopping (2)

- Nothing new here

```
wg := sync.WaitGroup{}  
for _, p := range in.peers {  
    wg.Add(1)  
    go func(p *peer) {  
        defer wg.Done()  
  
        resp, err := p.client.Promise(ctx,  
            &pb.PromiseRequest{ID: proposal})  
        → ... // ERROR HANDLING. SEE NEXT SLIDE!  
  
        responses <- &response{  
            from:      p.name,  
            promised:  resp.Promised,  
            id:        resp.ID,  
            holder:    resp.Holder,  
        }  
    }(p)  
}
```

Early Stopping (3)

- We don't care about cancelled requests
- We want errors which are **not** the result of a canceled proposal to be counted as a **negative answer** (nay) later.
- For that we emit an **empty response** into the channel in those cases.

```
resp, err := p.client.Promise(ctx,  
    &pb.PromiseRequest{ID: proposal})
```

```
if err != nil {  
    if ctx.Err() == context.Canceled {  
        return  
    }  
  
    responses <- &response{from: p.name}  
    return  
}
```

```
responses <- &response{...}  
...
```

Early Stopping (4)

- Close responses channel once all responses have been received, failed, or canceled

```
go func() {  
    wg.Wait()  
    close(responses)  
}()
```

Early Stopping (5)

- Count the votes
- Learn previous consensus (if any)
- Cancel all in-flight proposal if we have reached a majority

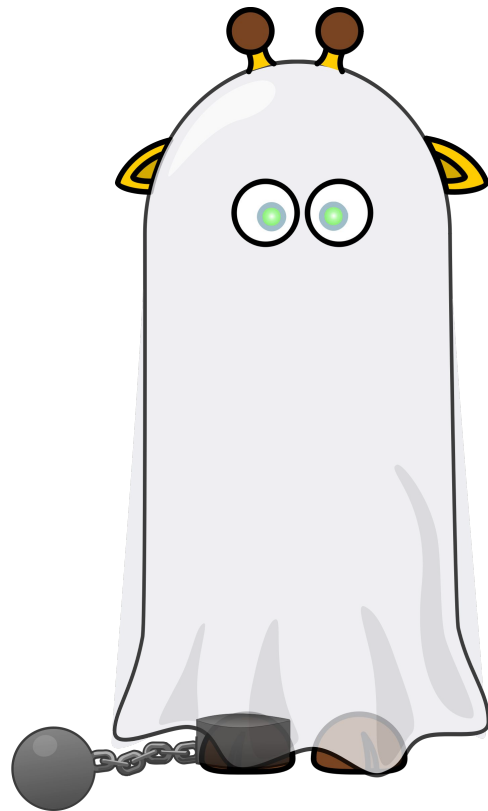
```
yay, nay := 1, 0
canceled := false
for r := range responses {
    if r.promised { yay++ } else { nay++ }

    in.learn(r)

    if !canceled {
        if in.isMajority(yay) || in.isMajority(nay) {
            cancel()
            canceled = true
        }
    }
}
```

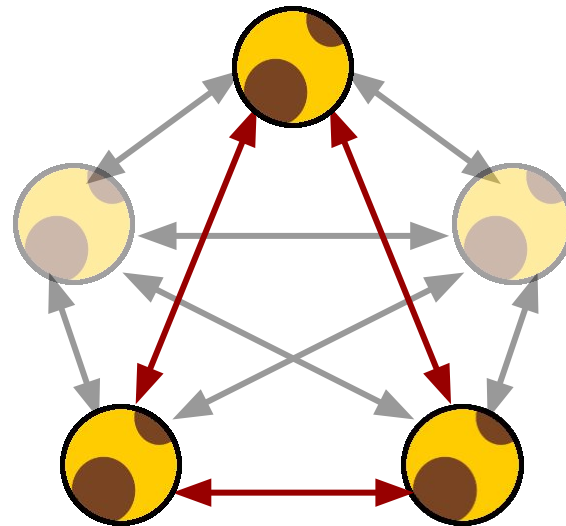
Is this fine?

- Timeouts are now even more critical!
- "Ghost Quorum" Effect
- *Let's all agree to disagree!*
- Idea: Use different timeouts in `propose()` and `commit()`?



Ghost Quorum

- Reason: Too tight timeout
- Some instances always time out
 - Effectively: Quorum of remaining instances
- Hidden reliability risk!
 - If one of the remaining instances fails, the distributed lock service is down!
 - No majority
 - No consensus

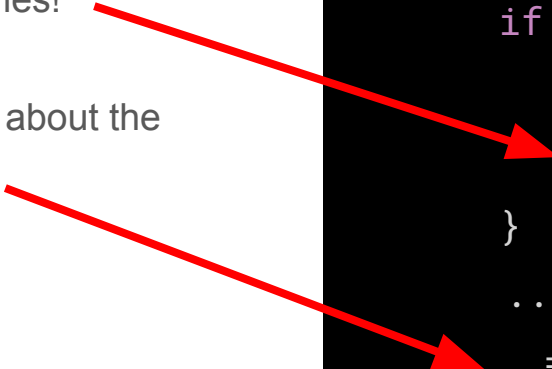


The Duell

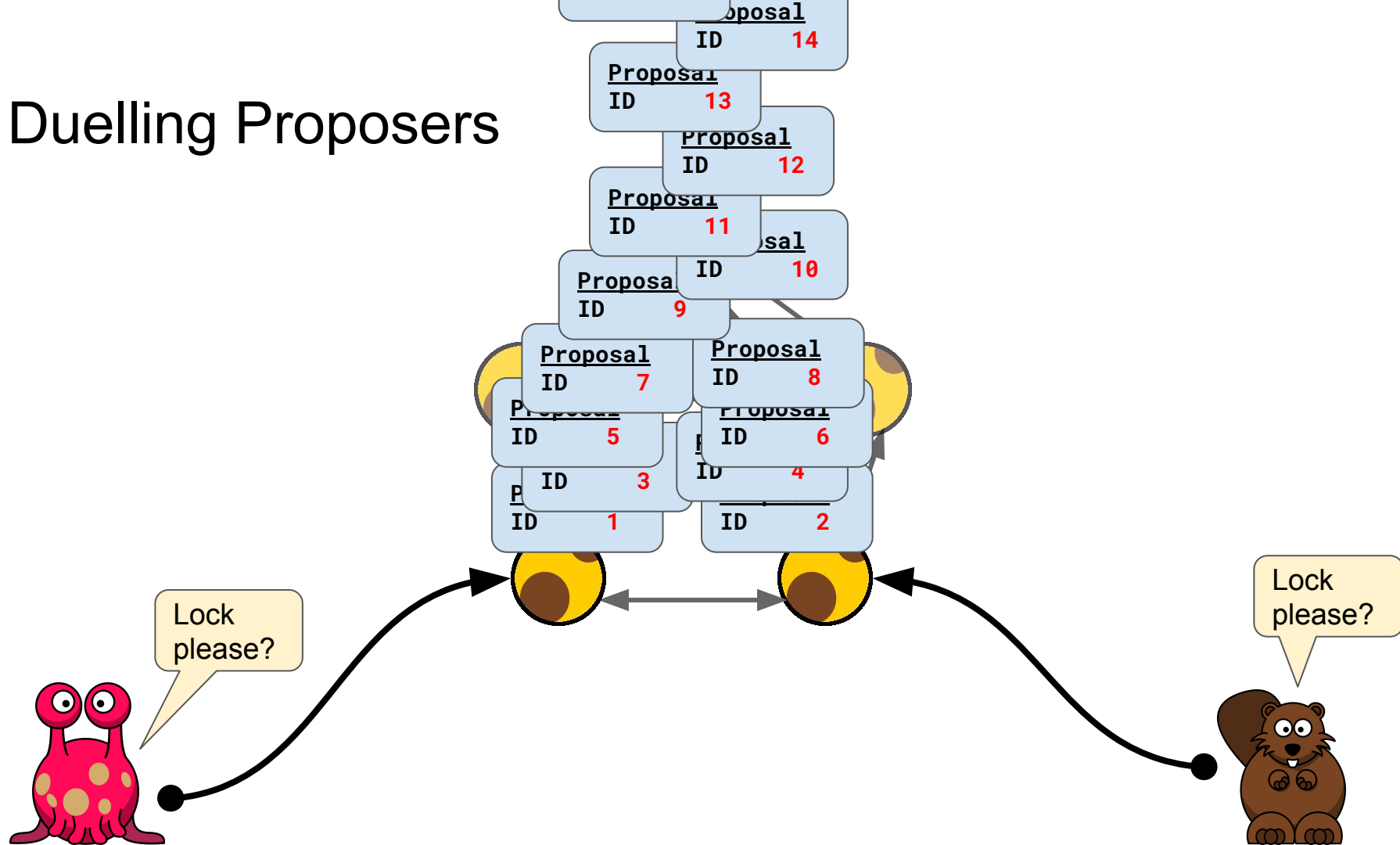
What's wrong?

- Retry Logic
 - Unlimited retries!
- Coding Style
 - I should care about the return value.

```
func foo(...) error {  
    ...  
    retry:  
        promised := in.propose(newID)  
        if !promised {  
            in.log.Printf("retry (%v)", id)  
            goto retry  
        }  
        ...  
        _ = in.commit(newID, newHolder)  
        ...  
        return nil  
    }  
}
```



Duelling Proposers



Soon...

| NAME | INCREMENT | PROMISED | ID | HOLDER | LAST SEEN |
|--------|-----------|----------|---------|--------|------------------|
| london | 3 | 1062520 | 1062520 | — | now |
| oregon | | | | | connection error |
| spaulo | | | | | connection error |
| sydney | 5 | 1062520 | 1062520 | — | 2 seconds ago |
| taiwan | 4 | 1062520 | 1062520 | — | 1 second ago |

Instances **oregon** and **spaulo** were intentionally offline for a different experiment

The Fix

✓ Retry Counter

✗ Backoff

✗ Jitter

```
func foo(...) error {  
    → retries := 0  
    retry:  
        promised := in.propose(newID)  
        if !promised {  
            → if retries < 3 {  
                → retries++  
                in.log.Printf("retry (%v)", id)  
                goto retry  
            }  
        }  
        ...  
        return nil  
    }  
}
```

Demo Time!

Further Reading

Reaching Agreement in the Presence of Faults

M. PEASE, R. SHOSTAK, AND L. LAMPORT

SRI International, Menlo Park, California

ABSTRACT. The problem addressed here concerns a set of isolated processors, some unknown subset of which may be faulty, that communicate only by means of two-party messages. Each nonfaulty processor has a private value of information that must be communicated to each other nonfaulty processor. Nonfaulty processors always communicate honestly, whereas faulty processors may lie. The problem is to devise an algorithm in which processors communicate their own values and relay values received from others that allows each nonfaulty processor to infer a value for each other processor. The value inferred for a nonfaulty processor must be that processor's private value, and the value inferred for a faulty one must be consistent with the corresponding value

<https://lamport.azurewebsites.net/pubs/reaching.pdf>

Further Reading

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

Naming of "Skinny"
absolutely not inspired
by "Chubby" ;)



We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locks as well as reliable (though low-volume) storage for loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with additional locks, but the design emphasis is on availability and reliability as opposed to high performance. Many

example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data struc-

<https://research.google.com/archive/chubby-osdi06.pdf>

Further Watching

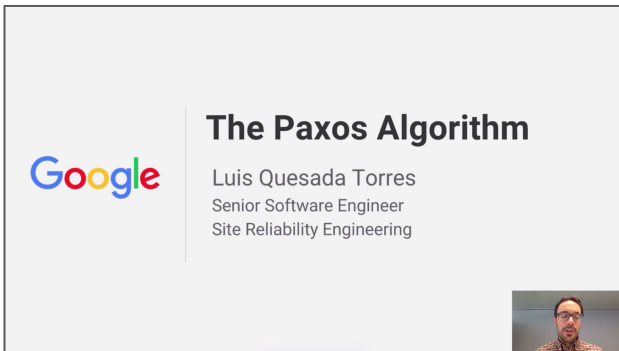


Paxos Agreement - Computerphile

Heidi Howard

University of Cambridge Computer Laboratory

<https://youtu.be/s8JqcZtvnsM>



The Paxos Algorithm



Luis Quesada Torres

Google Site Reliability Engineering

https://youtu.be/d7nAGI_NZPk

Try, Play, Learn!

github.com/danrl/skinny

- The Skinny Lock Server is open source software!
 - **skinnyd** lock server
 - **skinnyctl** control utility
- Terraform modules  Terraform
 - To get you started quickly with the infrastructure
- Ansible playbooks 
 - To help you install and configure your **skinnyd** instances



Find me on Twitter [@danrl_com](https://twitter.com/danrl_com)
I blog about SRE and technology: <https://danrl.com>