Workshop: Implementing Distributed Consensus


Dan Lüdtke

danrl@google.com


Kordian Bruck

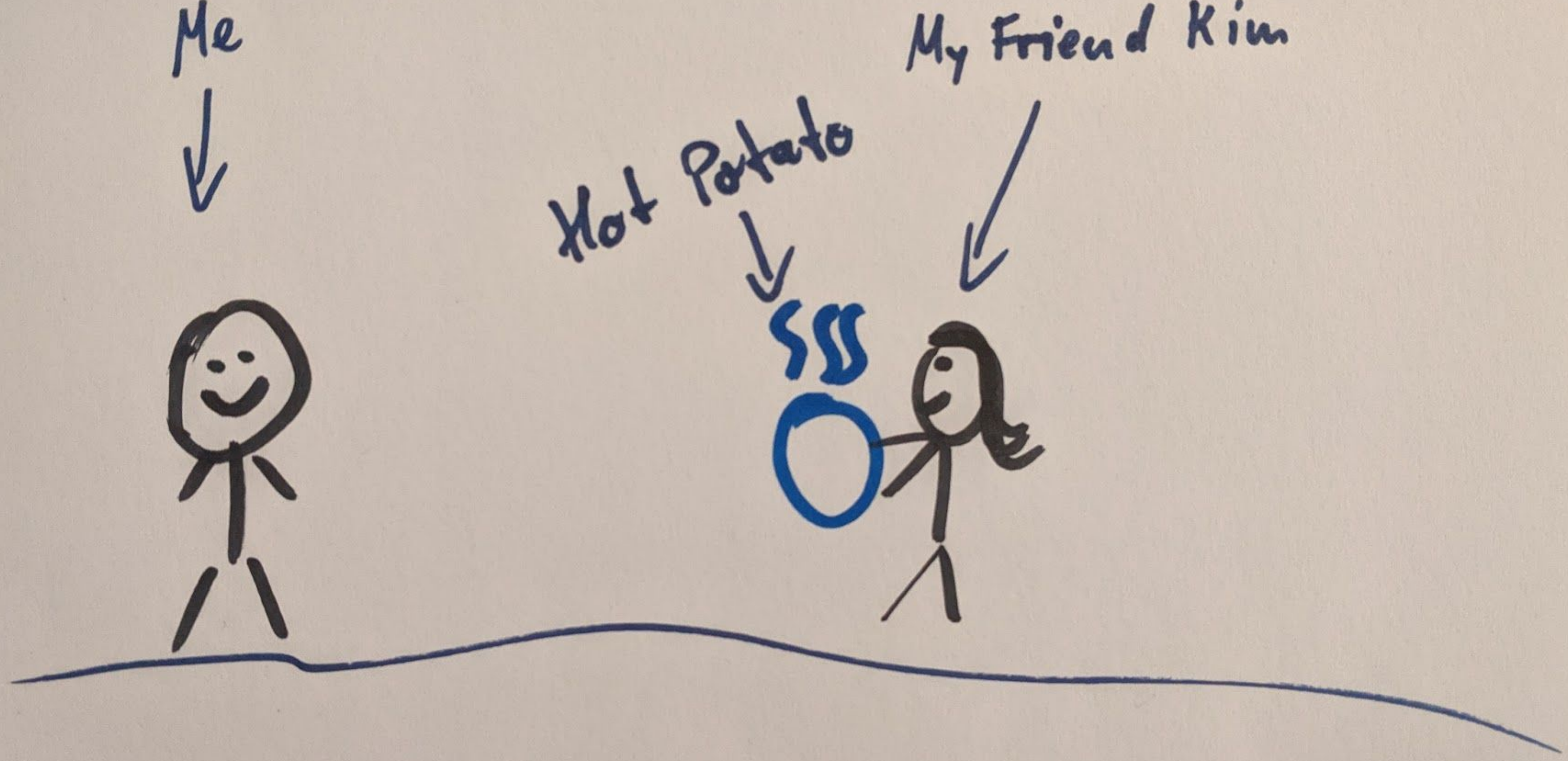
picatso@google.com

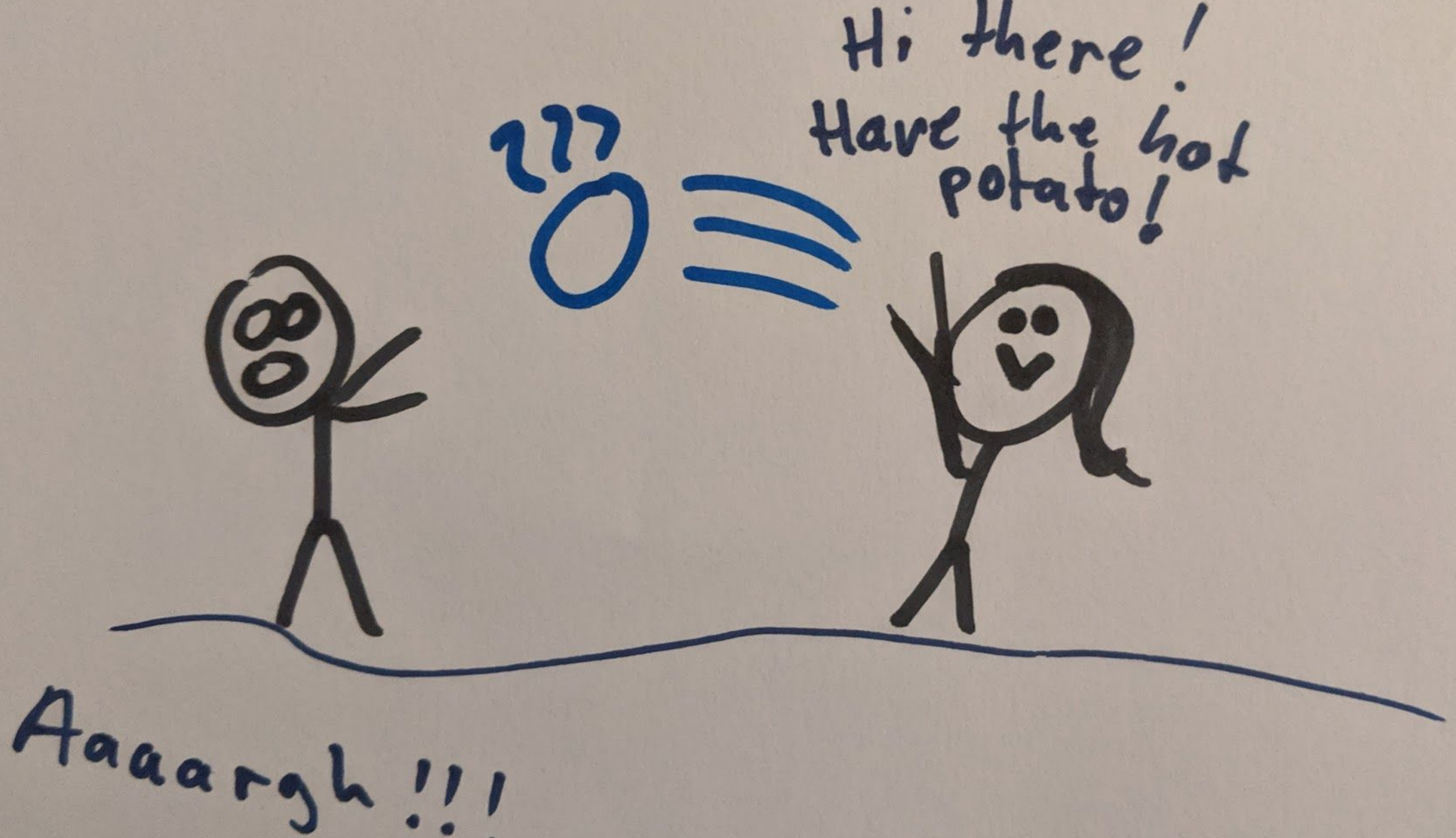
Disclaimer This work is not affiliated with any company (including Google). This talk is the result of a personal education project!

Agenda

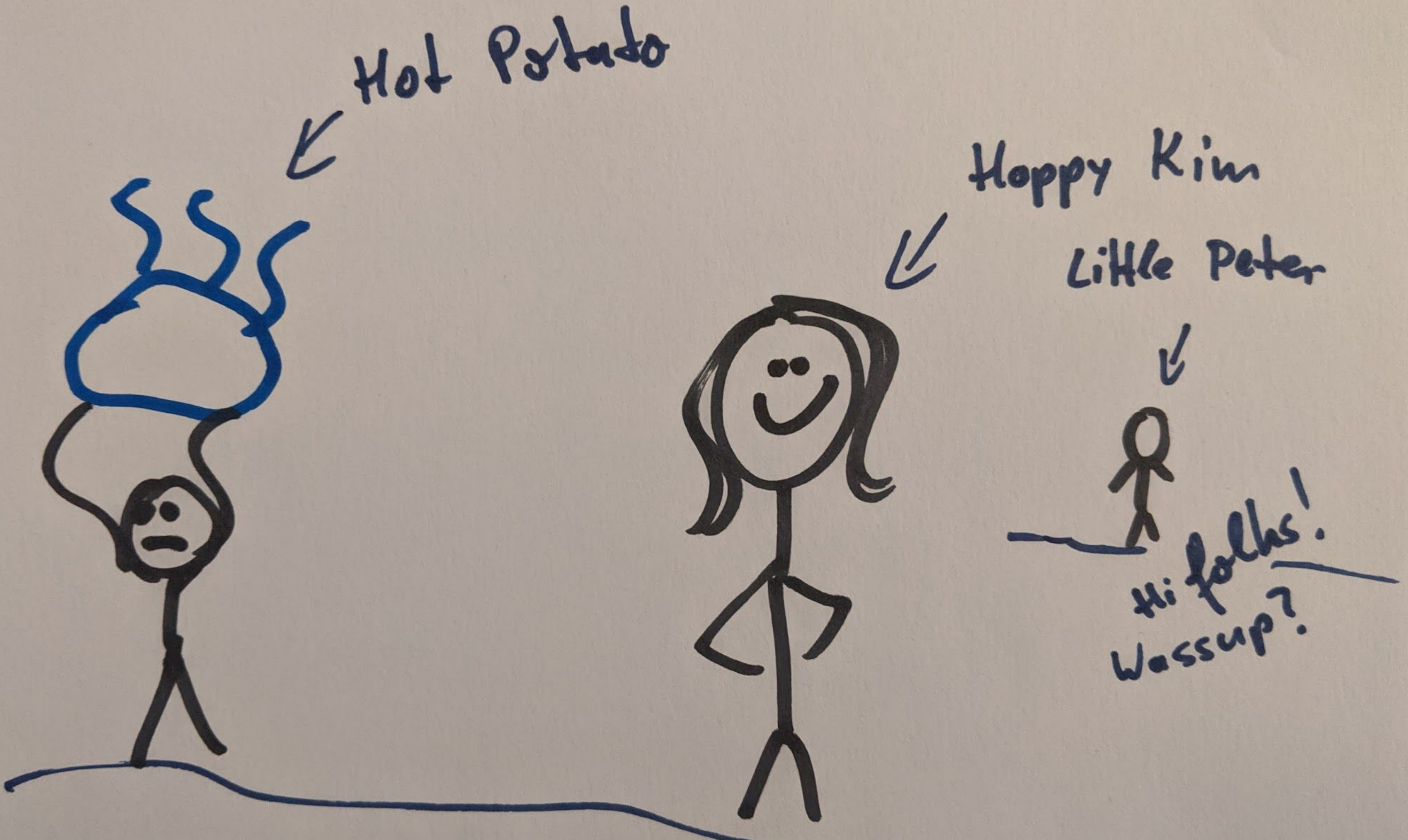
- Part I - Hot Potato at Scale
  - Why we need Distributed Consensus
- Part II - Experiments
  - Introduction to **Skinny**, an educational distributed lock service
  - How Skinny reaches consensus
  - How Skinny deals with instance failure
- Part III - Implementation
  - A simple Paxos-like protocol
  - Making our protocol more reliable

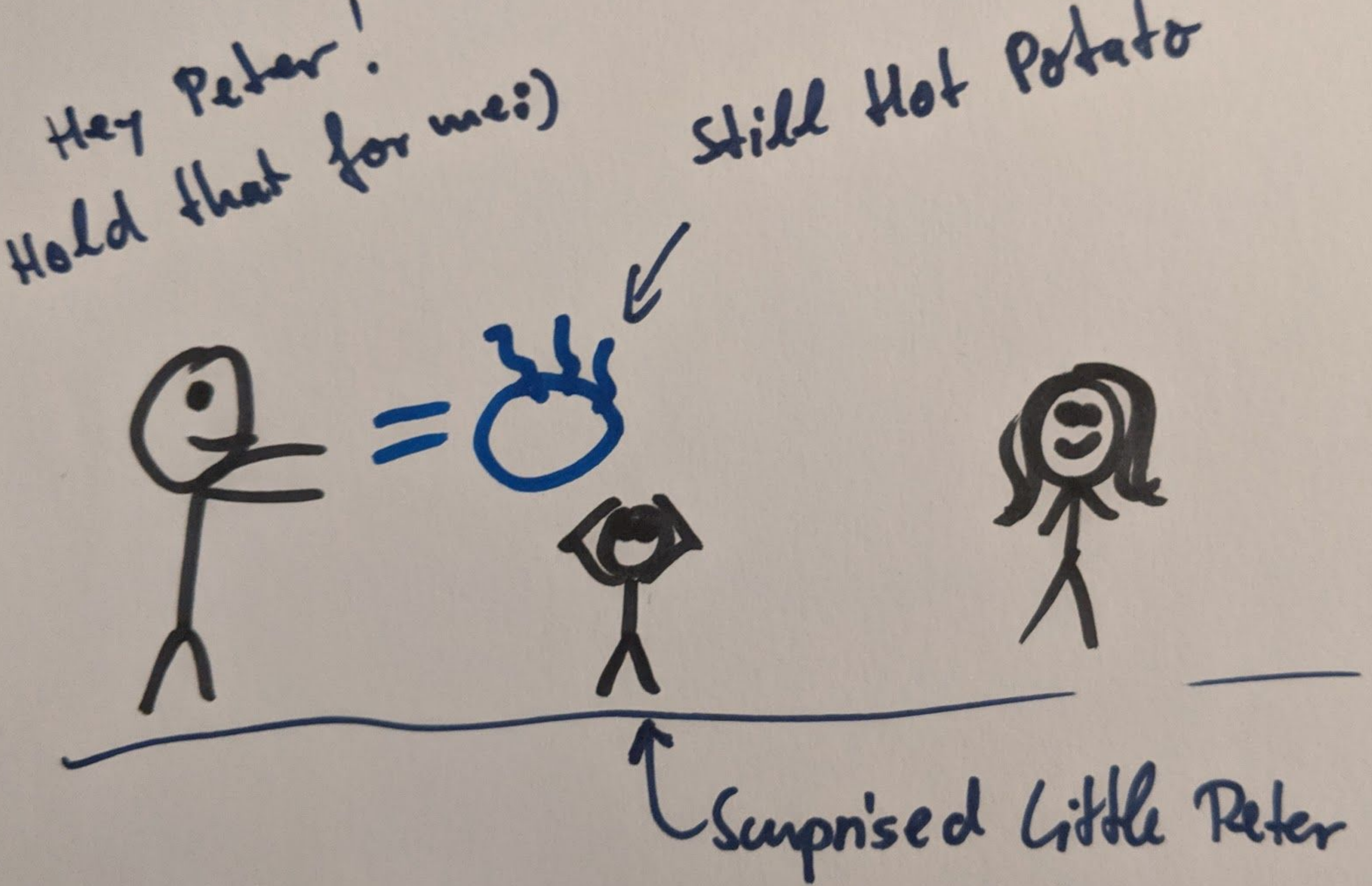
Part I
Hot Potato at Scale

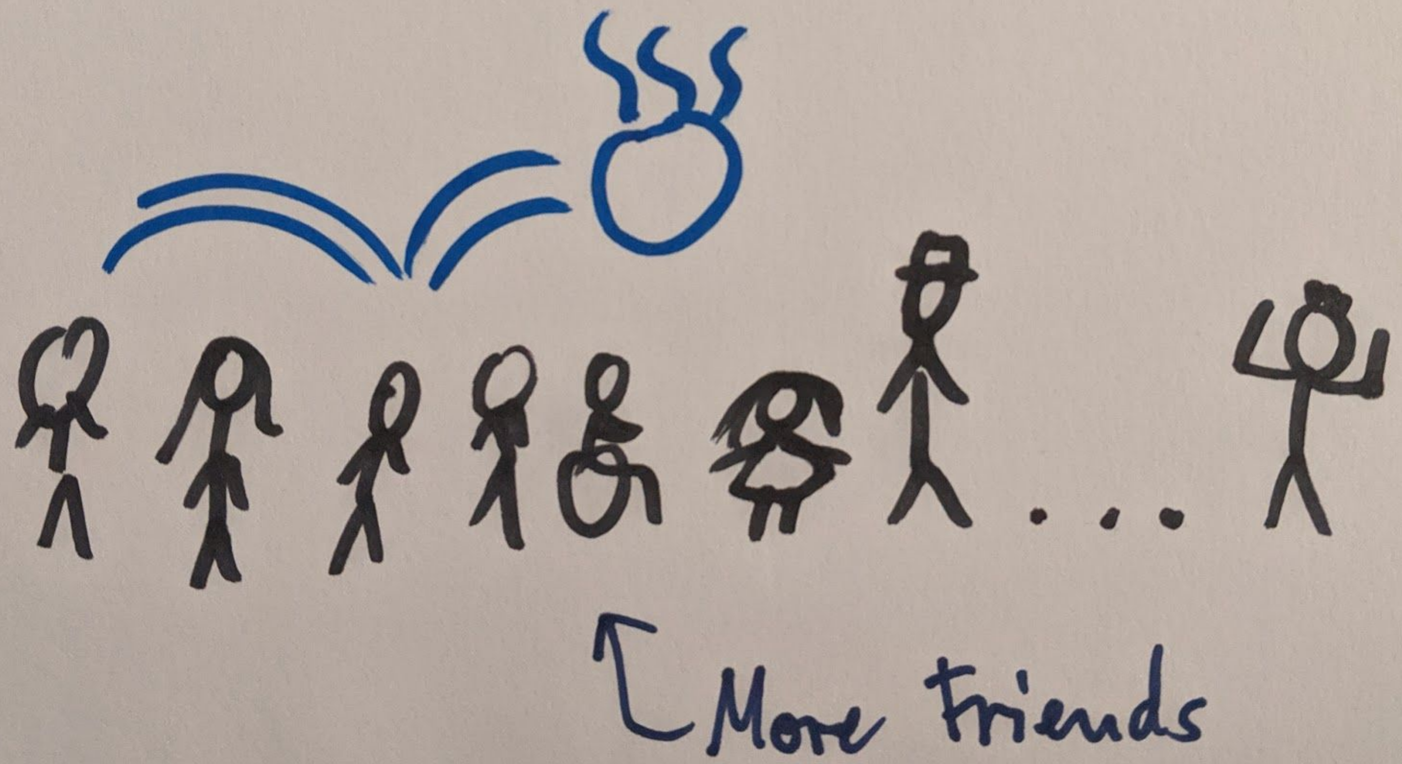

Me
Hot Potato
My Friend Kim

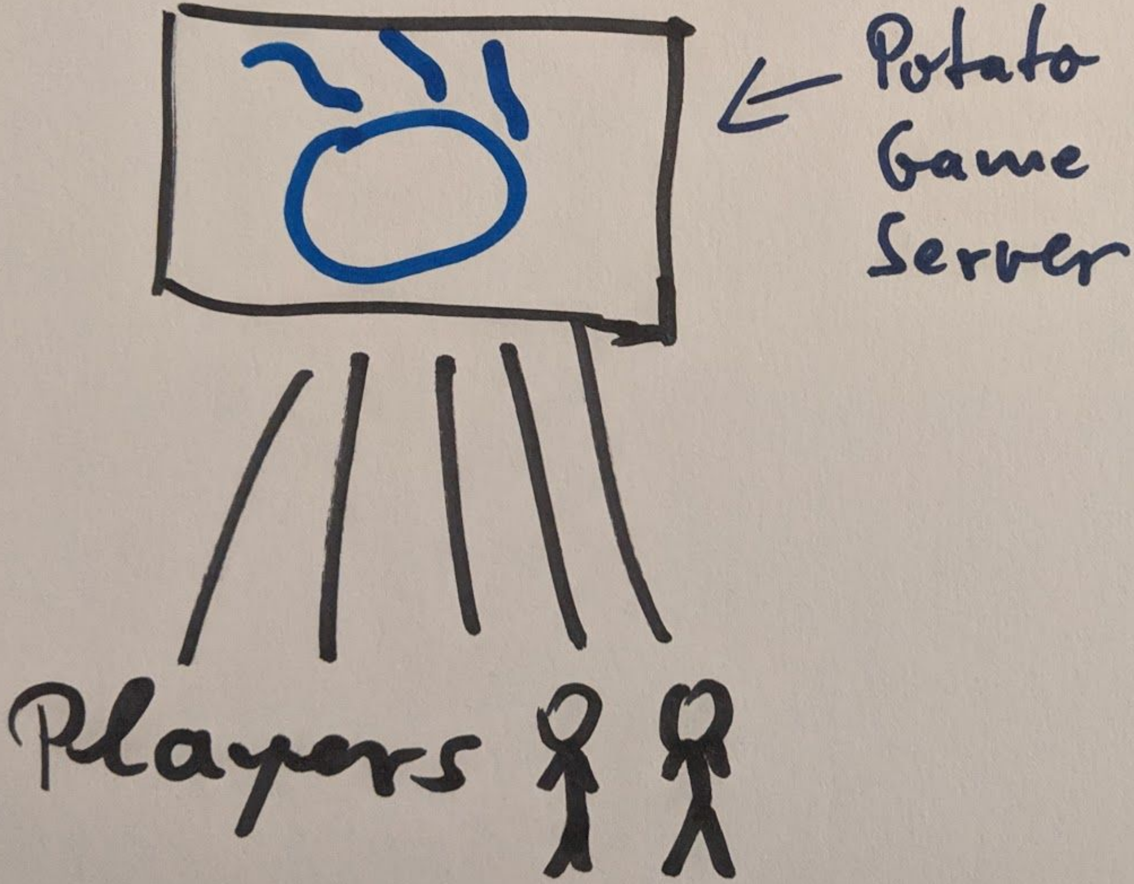


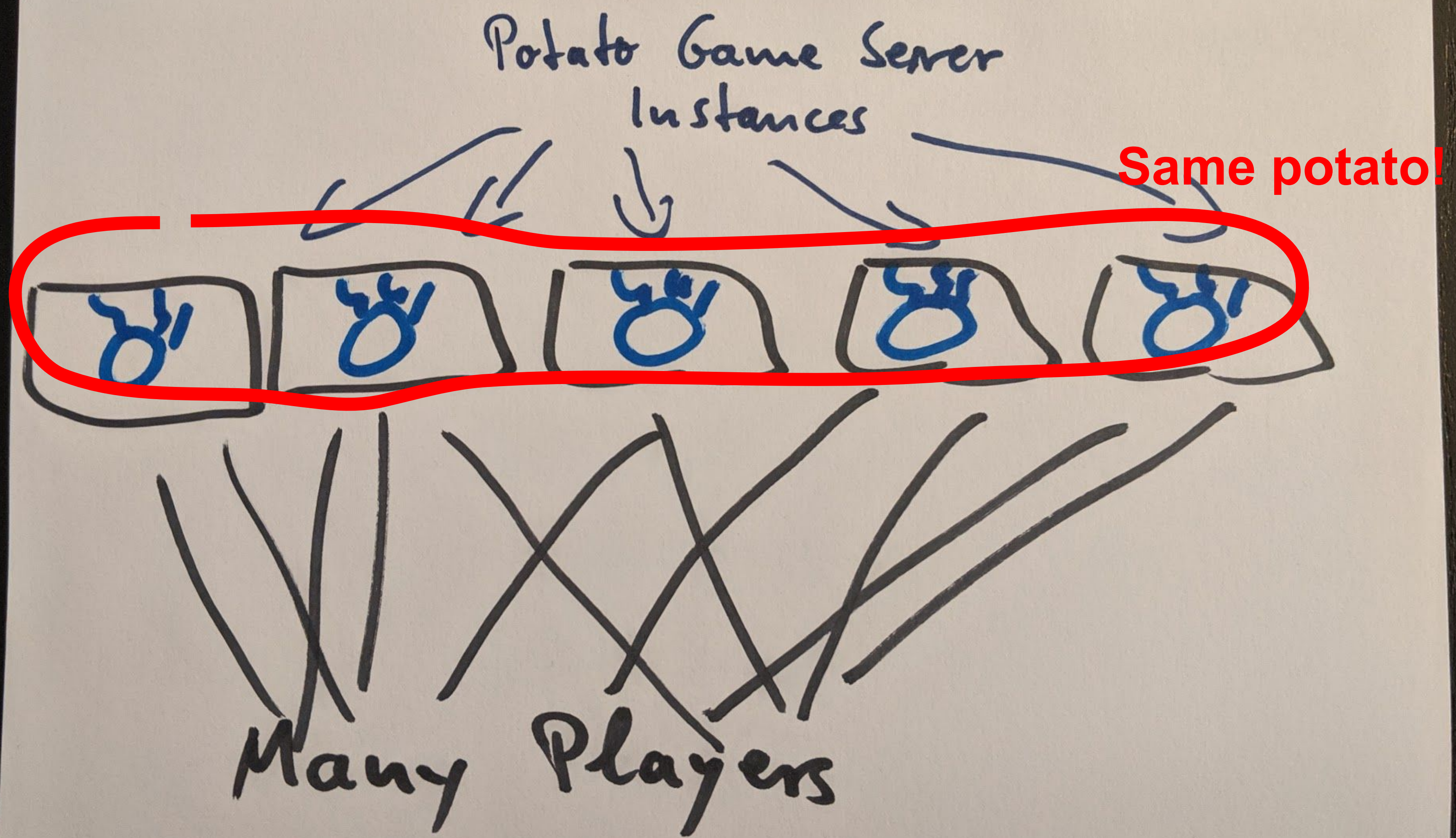
Hi there!
Have the hot potato!
???
Aaaaargh!!!

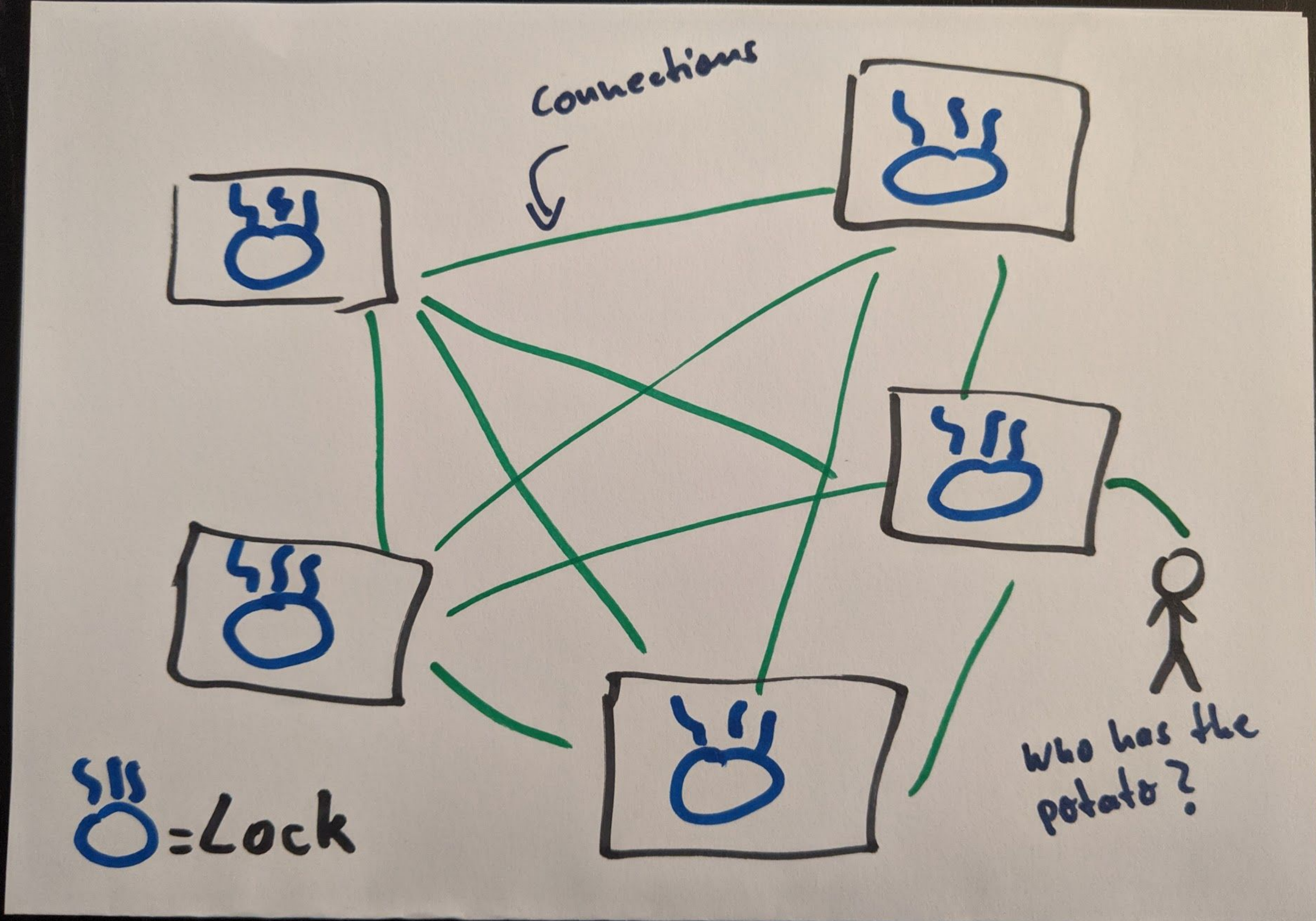

Hot Potato
Happy Kim
Little Peter
Hi folks!
Wassup?


Hey Peter!
Hold that for me:)
Still Hot Potato
Surprised Little Peter


More Friends


Potato
Game
Server
Players


Potato Game Server
Instances
Same potato!
Many Players


Connections
=Lock
who has the potato?

Protocols

- Paxos
  - Multi-Paxos
  - Cheap Paxos
- Raft
- ZooKeeper Atomic Broadcast
- Proof-of-Work Systems
  - Bitcoin
- Lockstep Anti-Cheating
  - Age of Empires

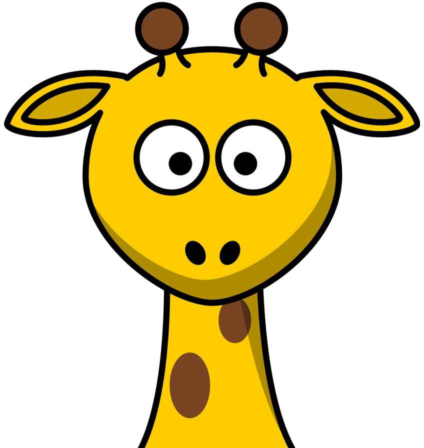
Implementations

- Chubby
  - Coarse grained lock service
- etcd
  - A distributed key value store
- Apache ZooKeeper
  - A centralized service for maintaining configuration information, naming, providing distributed synchronization

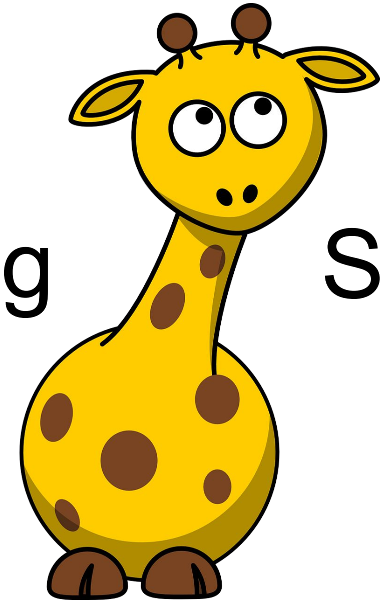
Want more theory?

See `paxos-roles.pdf`

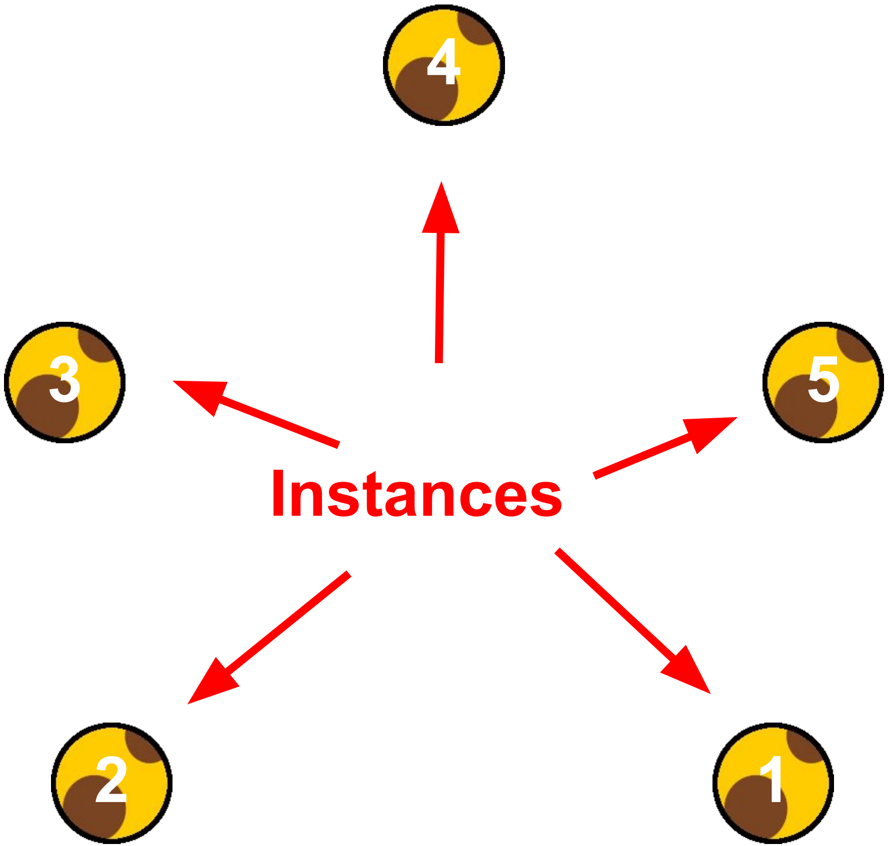
at https://danrl.com/talks/

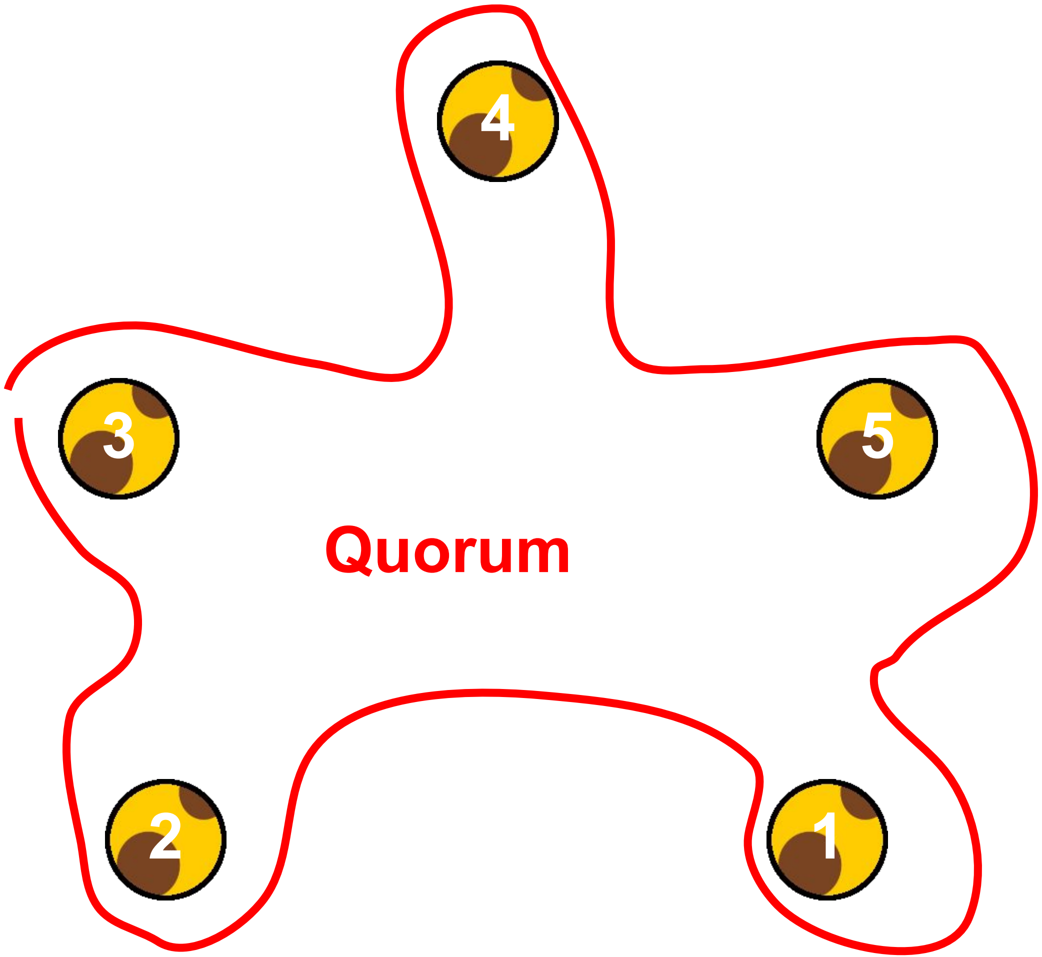


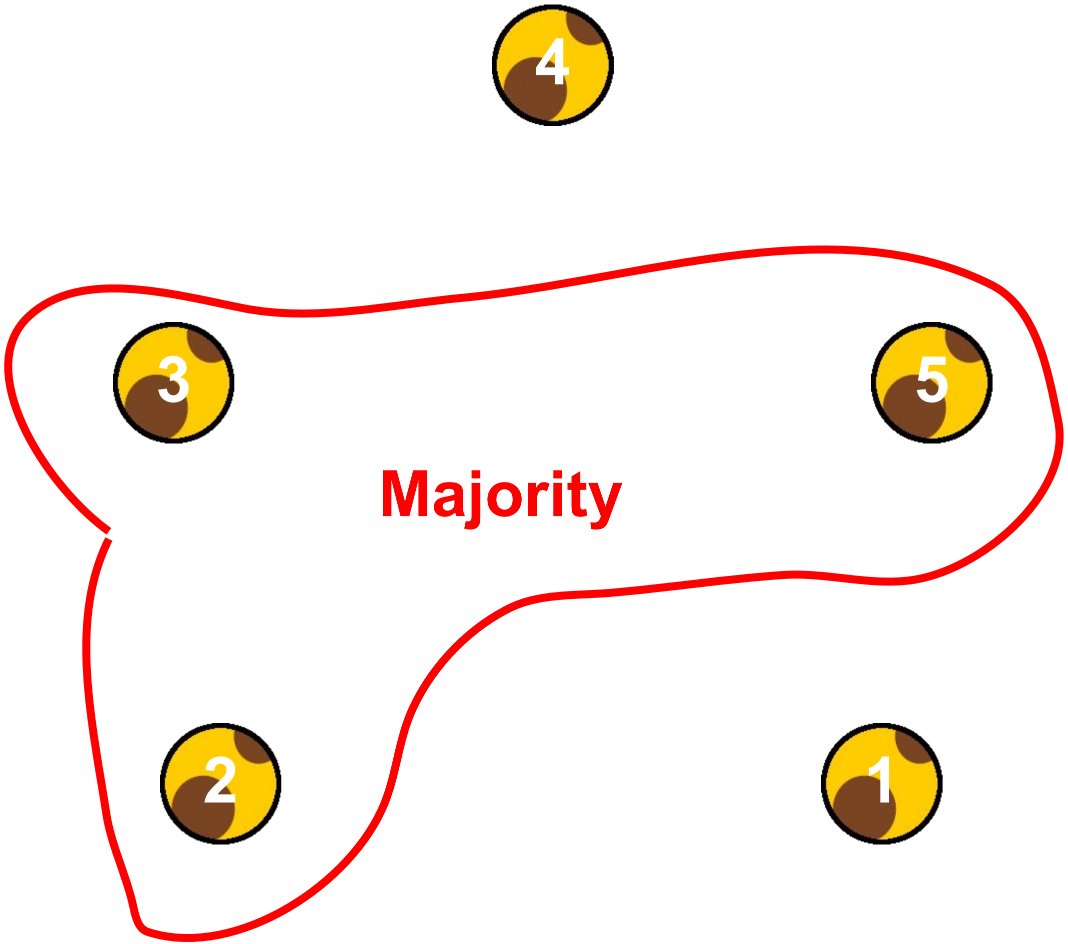
Part II
Distributed Consensus
Hands-on

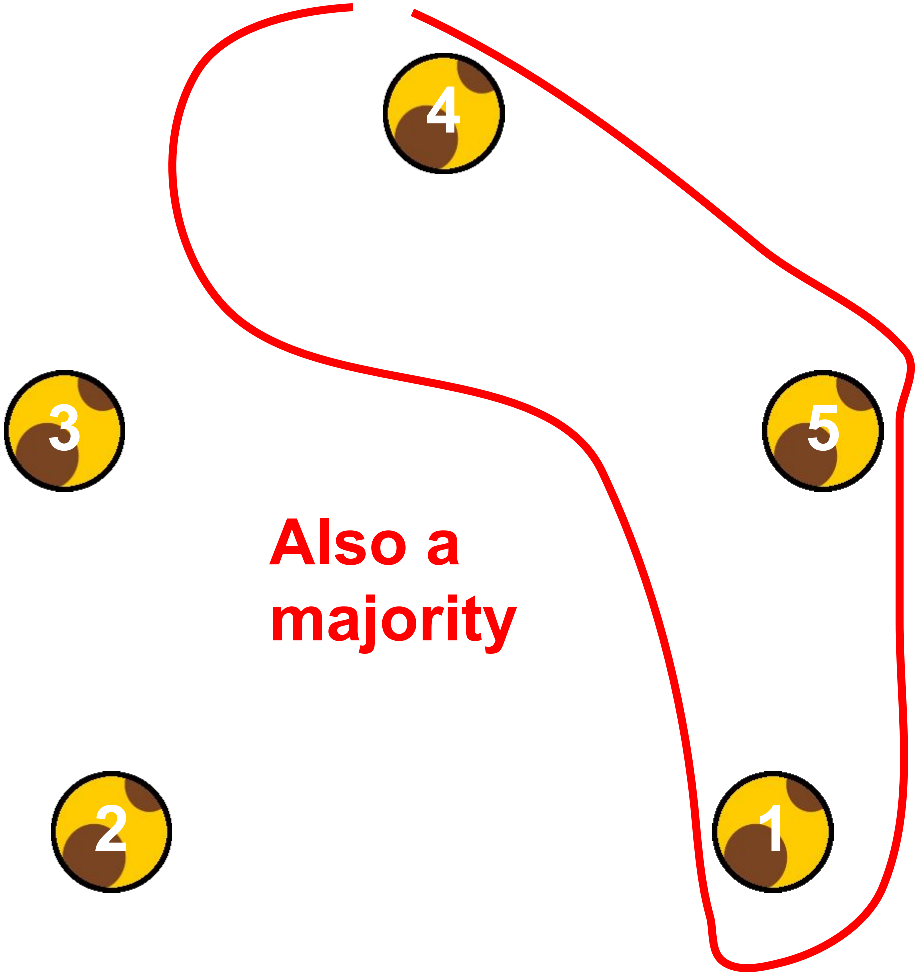
Introducing  Skinny

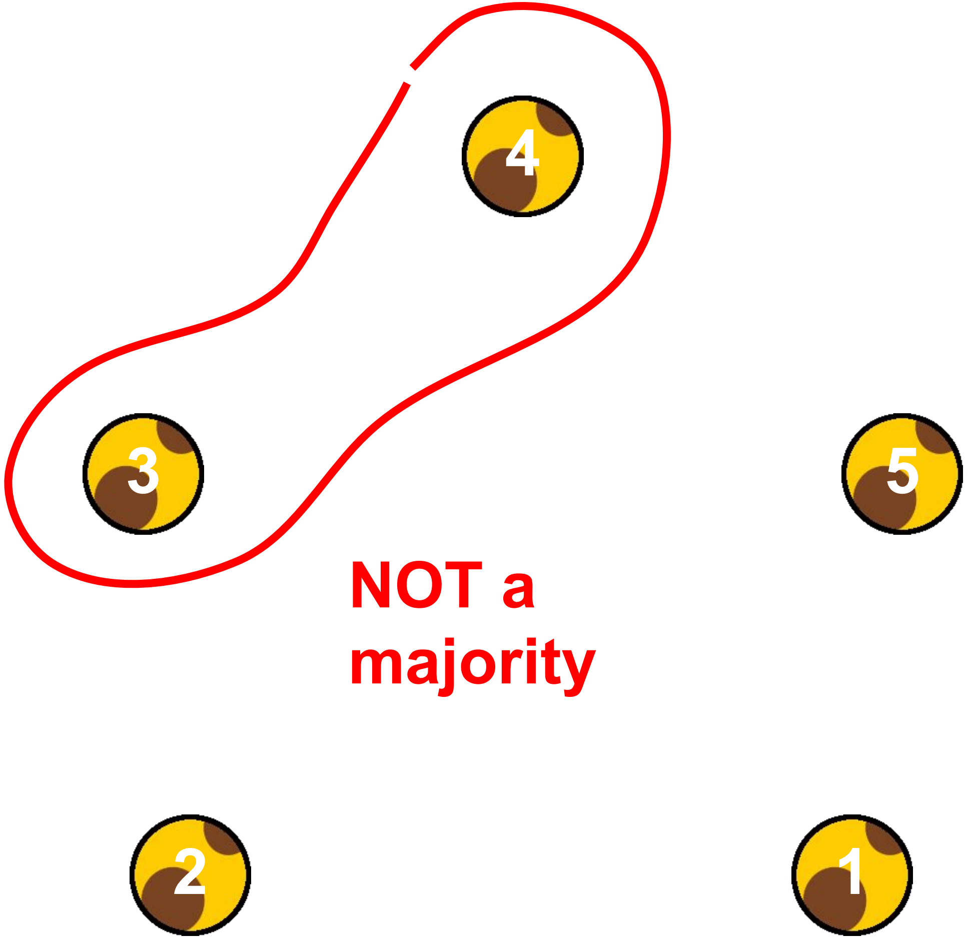
- Paxos-based
- Minimalistic
- Educational
- Lock Service

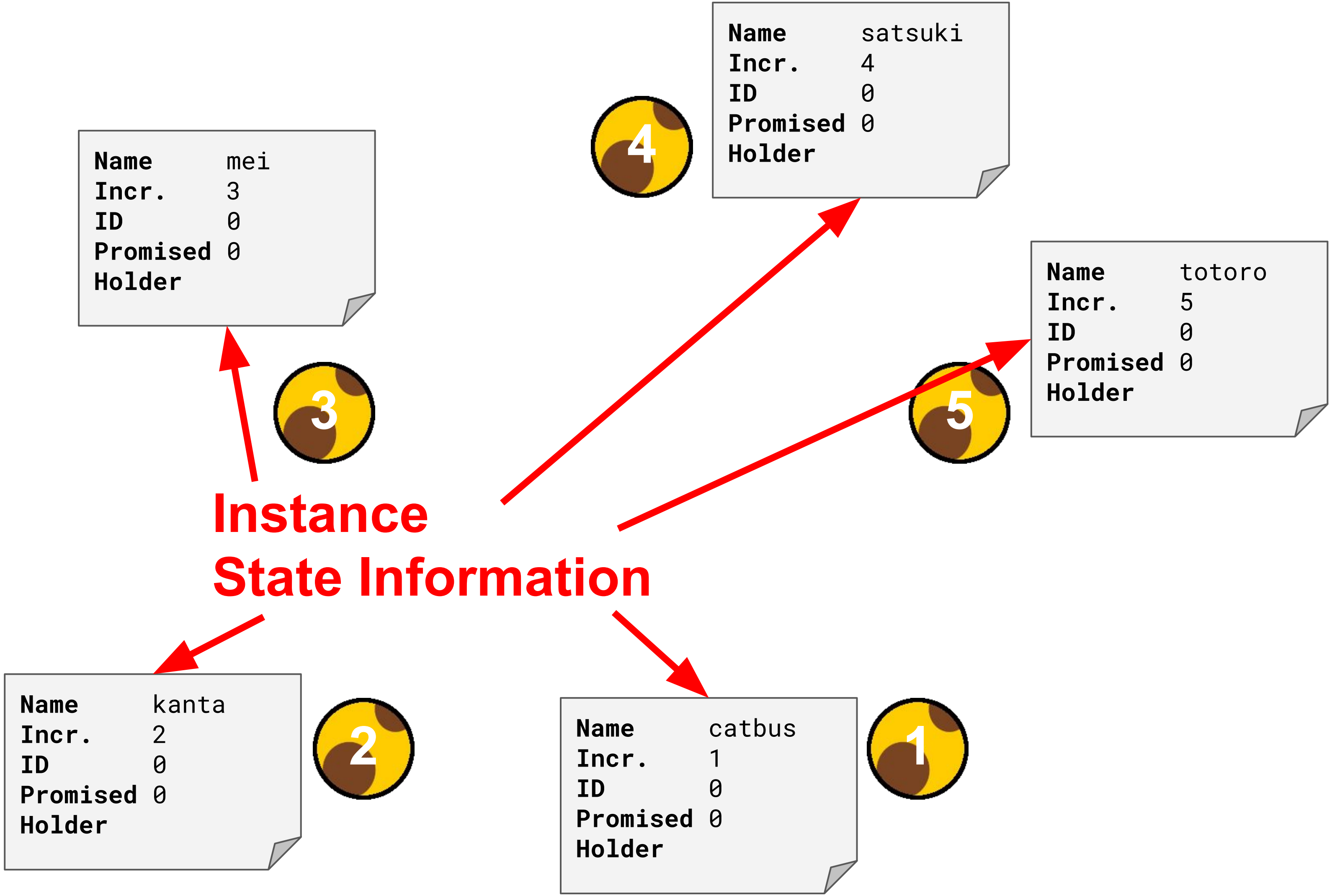

4
3
5
Instances
2
1

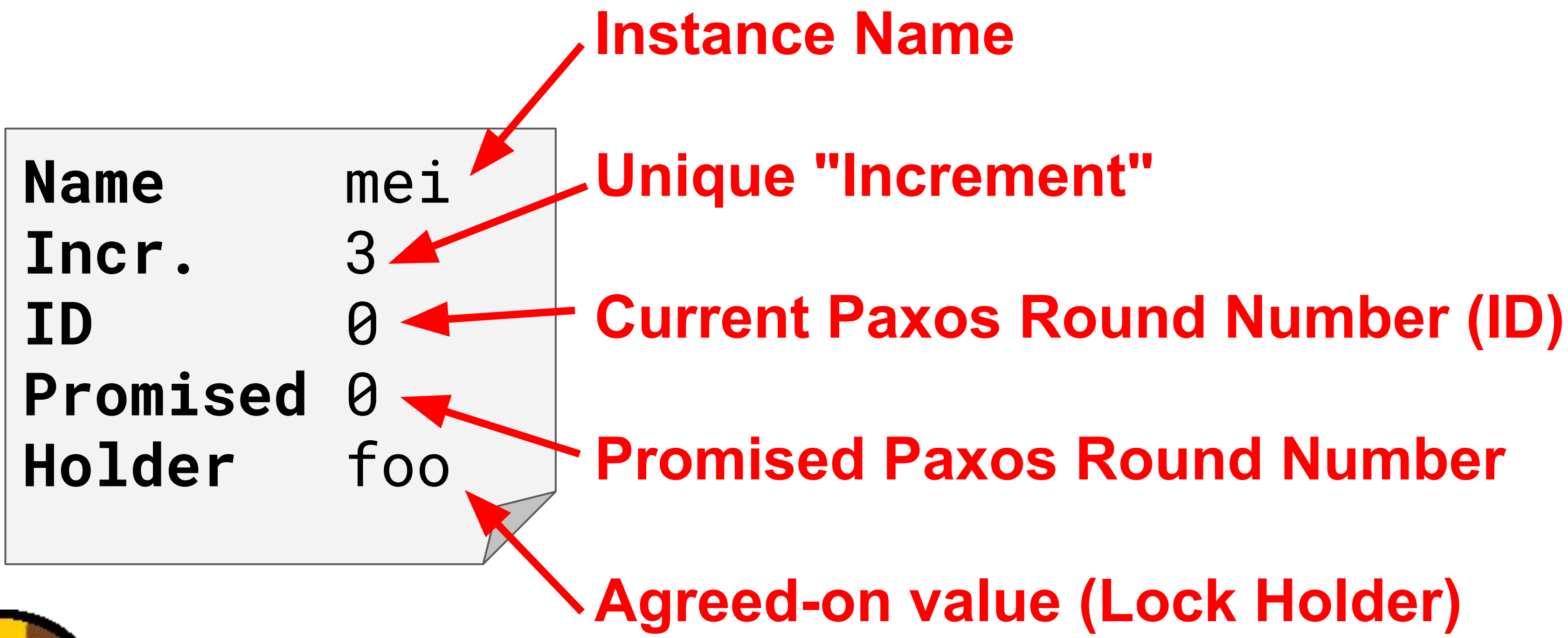

4
3
5
Quorum
2
1

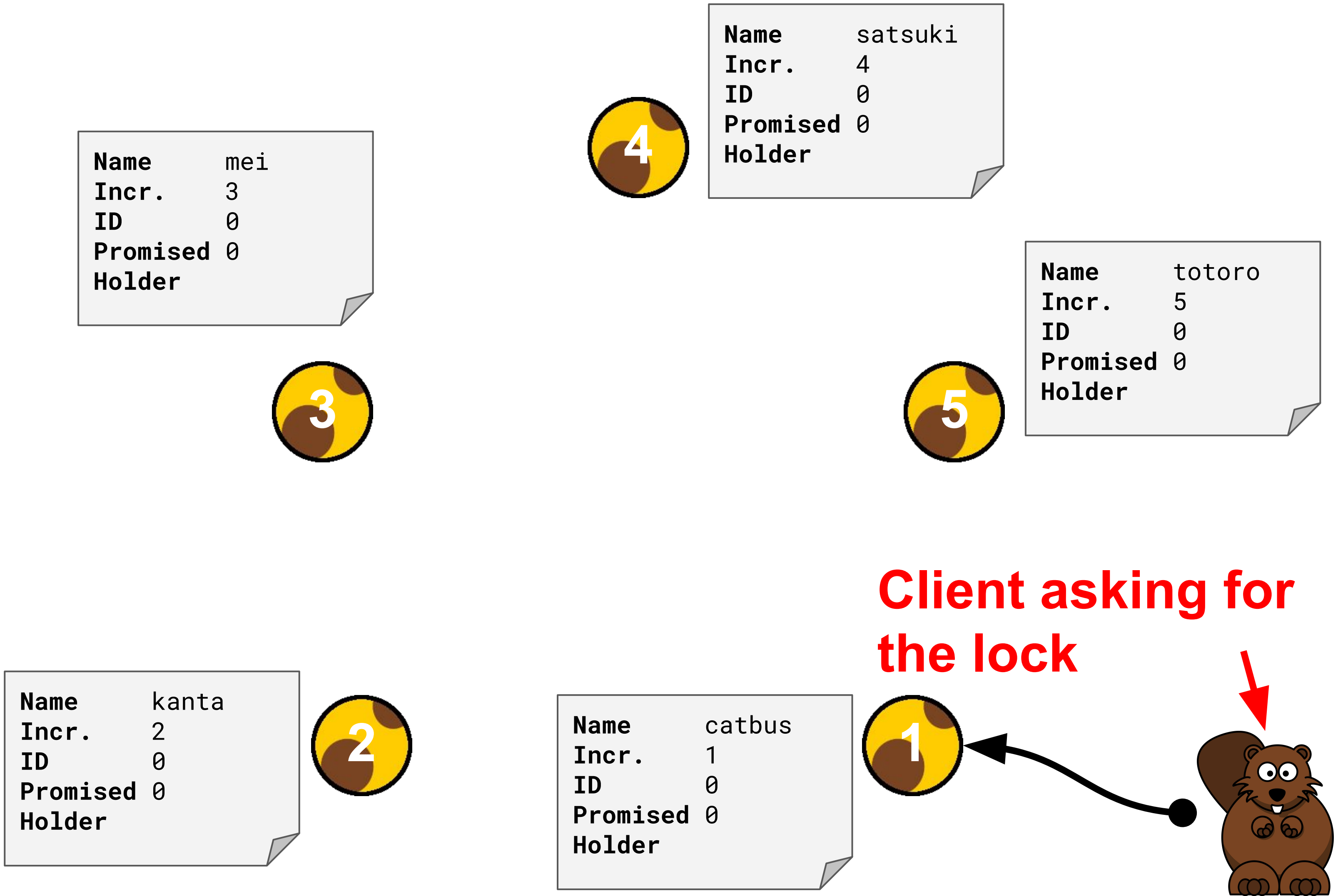

4
3
5
Majority
2
1


4
3
5
Also a majority
2
1


4
3
5
NOT a
majority
2
1


Name satsuki
Incr. 4
ID 0
Promised 0
Holder
4
Name mei
Incr. 3
ID 0
Promised 0
Holder
3
Name totoro
Incr. 5
ID 0
Promised 0
Holder
5
Instance
State Information
Name kanta
Incr. 2
ID 0
Promised 0
Holder
2
Name catbus
Incr. 1
ID 0
Promised 0
Holder
1


Name mei
Incr. 3
ID 0
Promised 0
Holder foo
Instance Name
Unique "Increment"
Current Paxos Round Number (ID)
Promised Paxos Round Number
Agreed-on value (Lock Holder)



Name mei
Incr. 3
ID 0
Promised 0
Holder
Name satsuki
Incr. 4
ID 0
Promised 0
Holder
Name totoro
Incr. 5
ID 0
Promised 0
Holder
Name kanta
Incr. 2
ID 0
Promised 0
Holder
Name catbus
Incr. 1
ID 0
Promised 0
Holder
1
2
3
4
5
Client asking for the lock

Let's get used to the lab...

Lab Machine Folder Structure

```
/home/ubuntu/
      └── skinny
           ├── bin
           ├── cmd
           ├── config
           ├── doc
           │    ├── ansible
           │    ├── examples
           │    ├── img
           │    ├── plots
           │    ├── terraform
           │    └── workshop
           │         ├── ami
           │         ├── code
           │         ├── configs
           │         └── scripts
           ├── proto
           ├── skinny
           └── vendor
```

Binaries

Source of the Skinny CLI tools

Source of the Skinny config parser module

Our working directory

Lab virtual machine disk image

Lab code/config/scripts
for our experiments

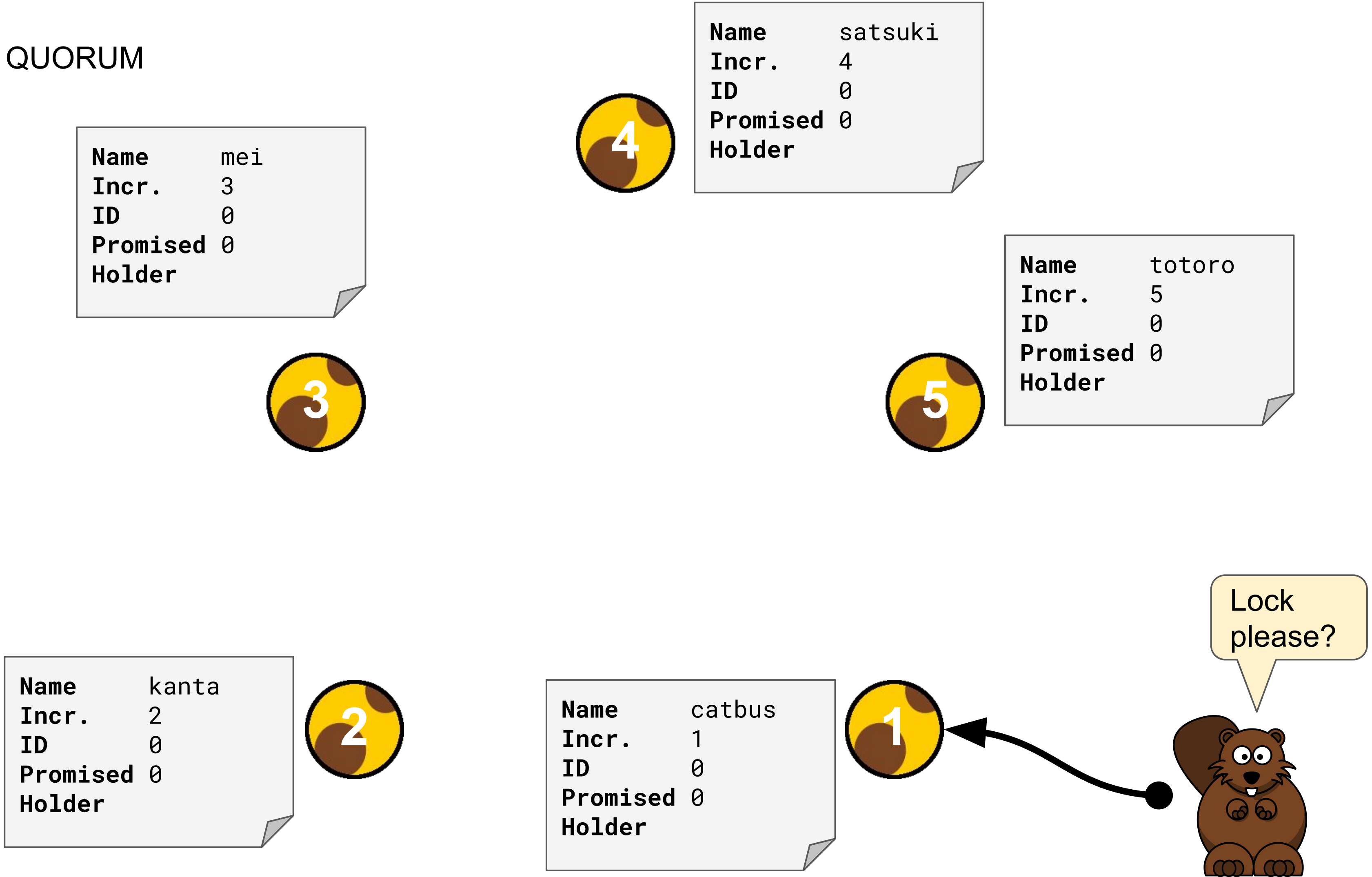
Protocol buffer definitions (API definitions)

Main Skinny source code

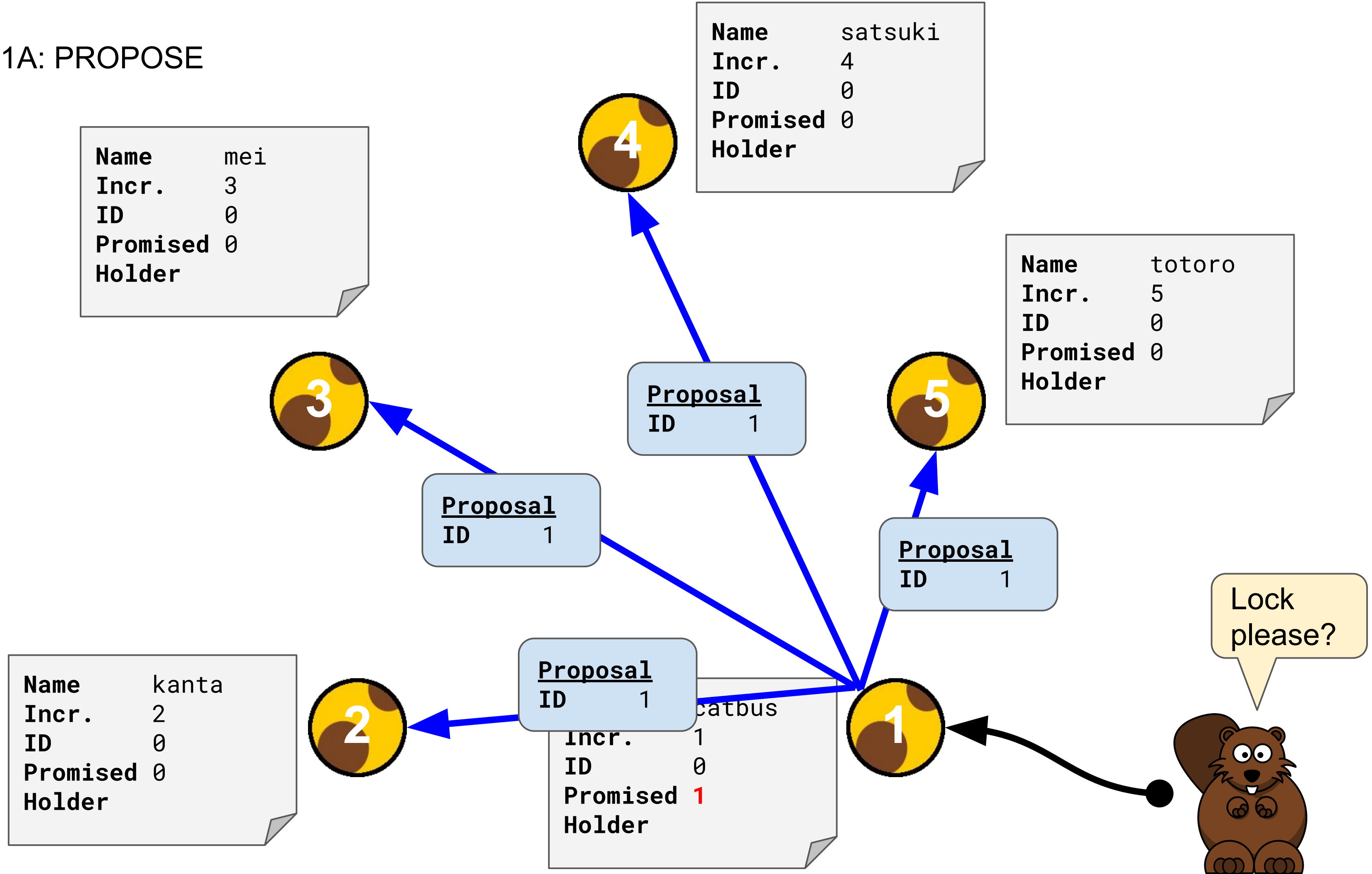
3rd party source code

How Skinny reaches consensus

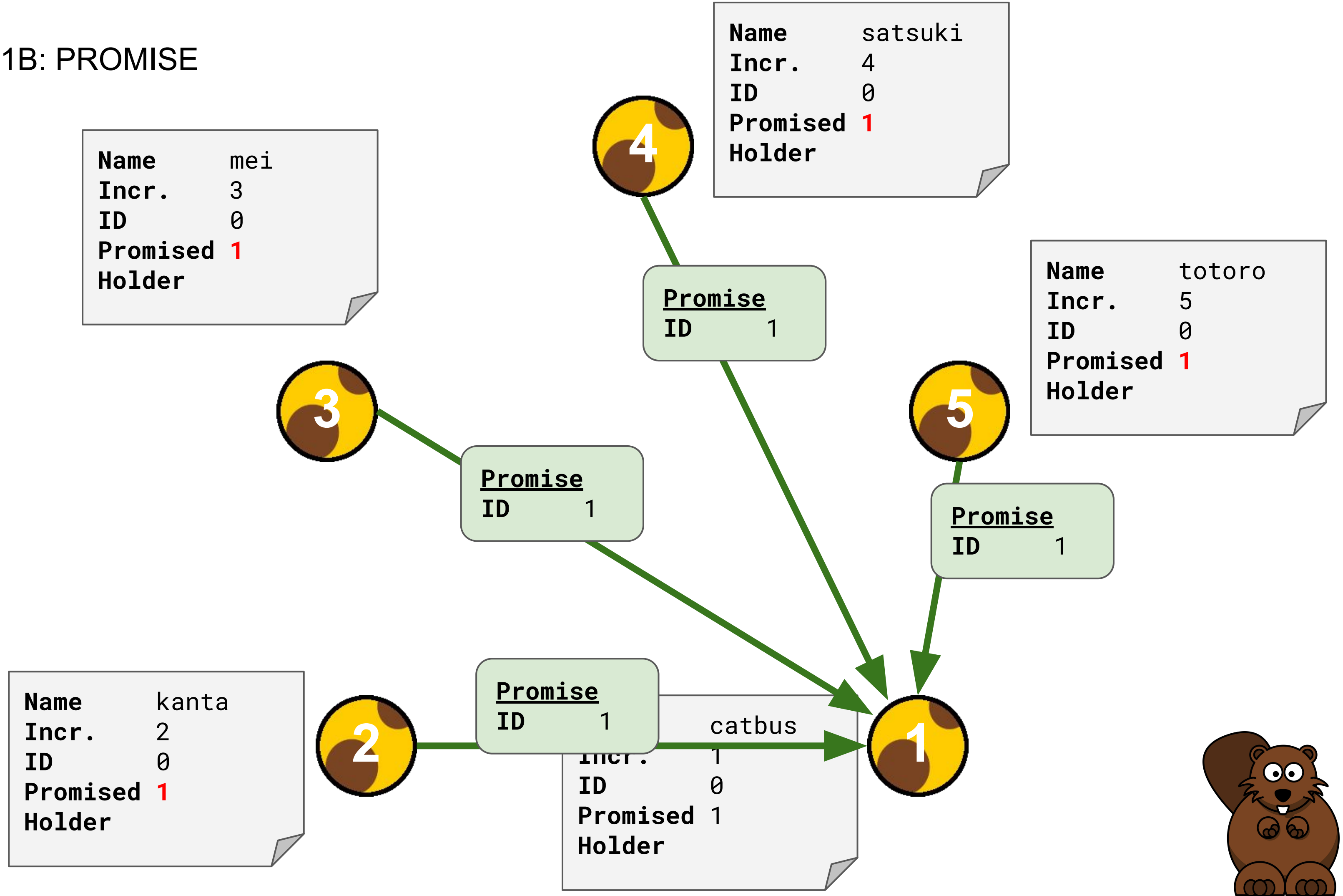
SKINNY QUORUM



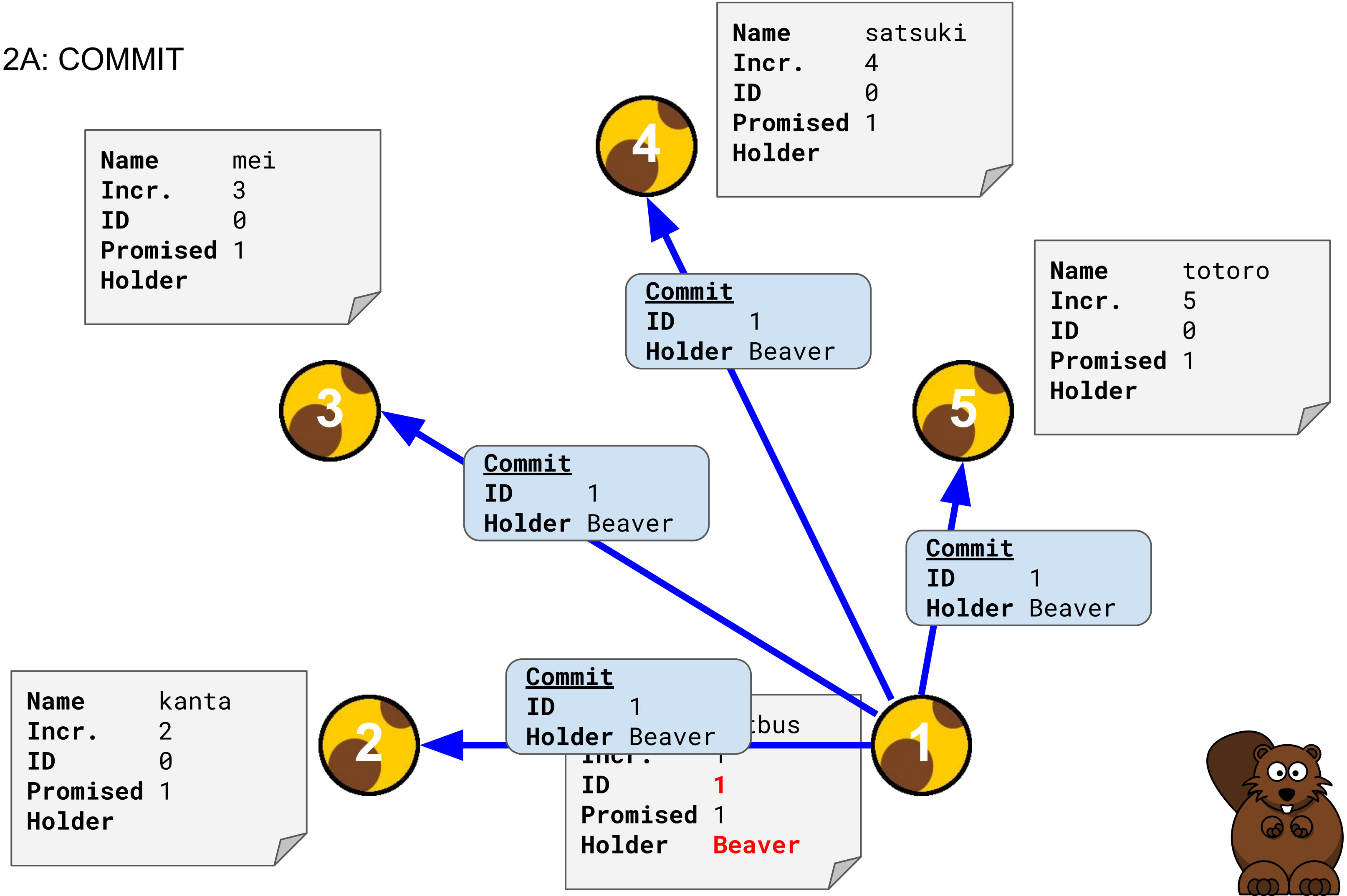
PHASE 1A: PROPOSE



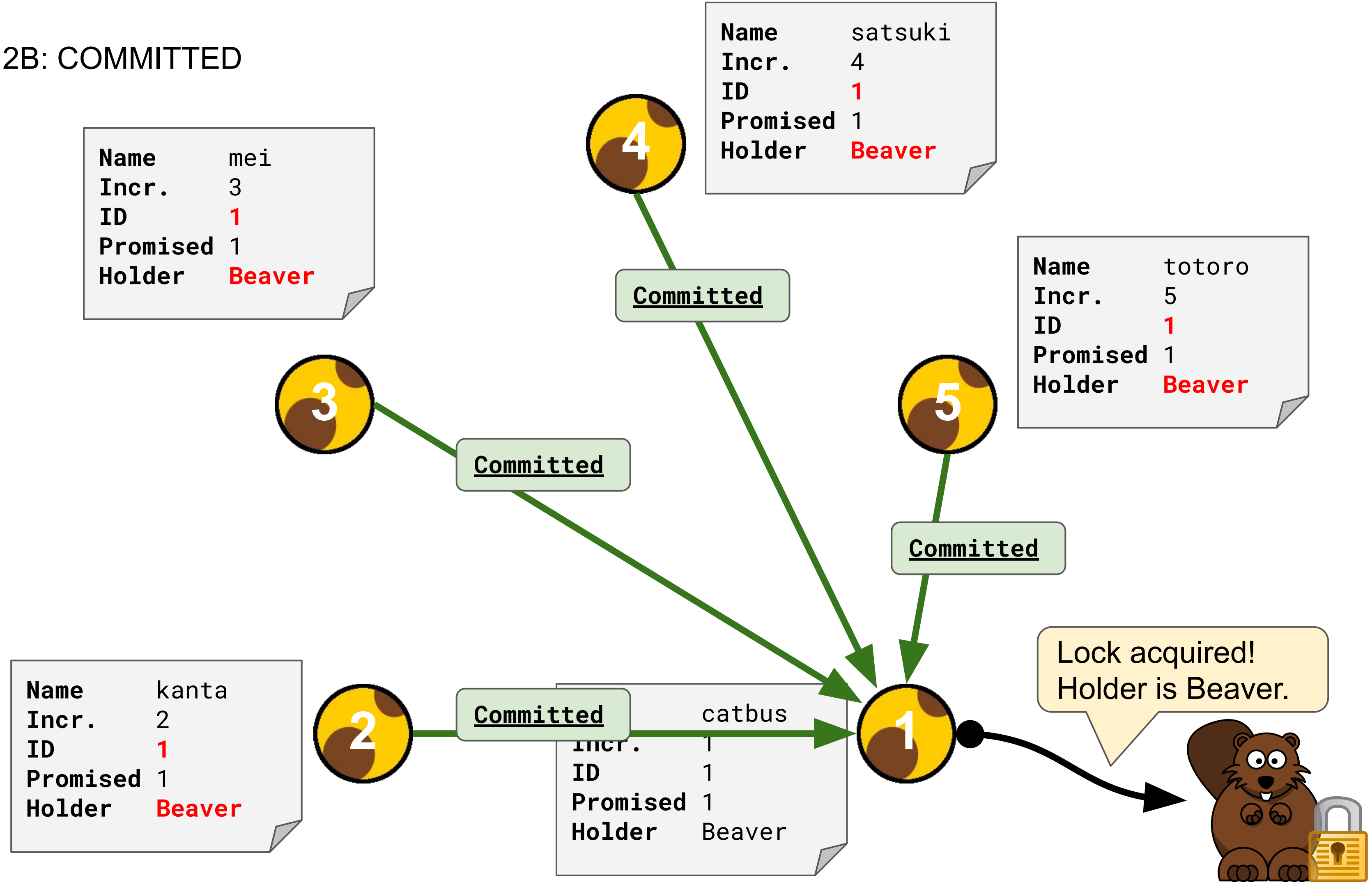
PHASE 1B: PROMISE



PHASE 2A: COMMIT



PHASE 2B: COMMITTED



Experiment One

Important: Run all commands in folder `~/skinny/doc/workshop/`

1.) Inspect Quorum
```
skinnyctl status
```

2.) Acquire Lock for "beaver" (using instance *catbus*)
```
skinnyctl acquire --instance=catbus beaver
```

3.) Inspect Quorum
```
skinnyctl status
```

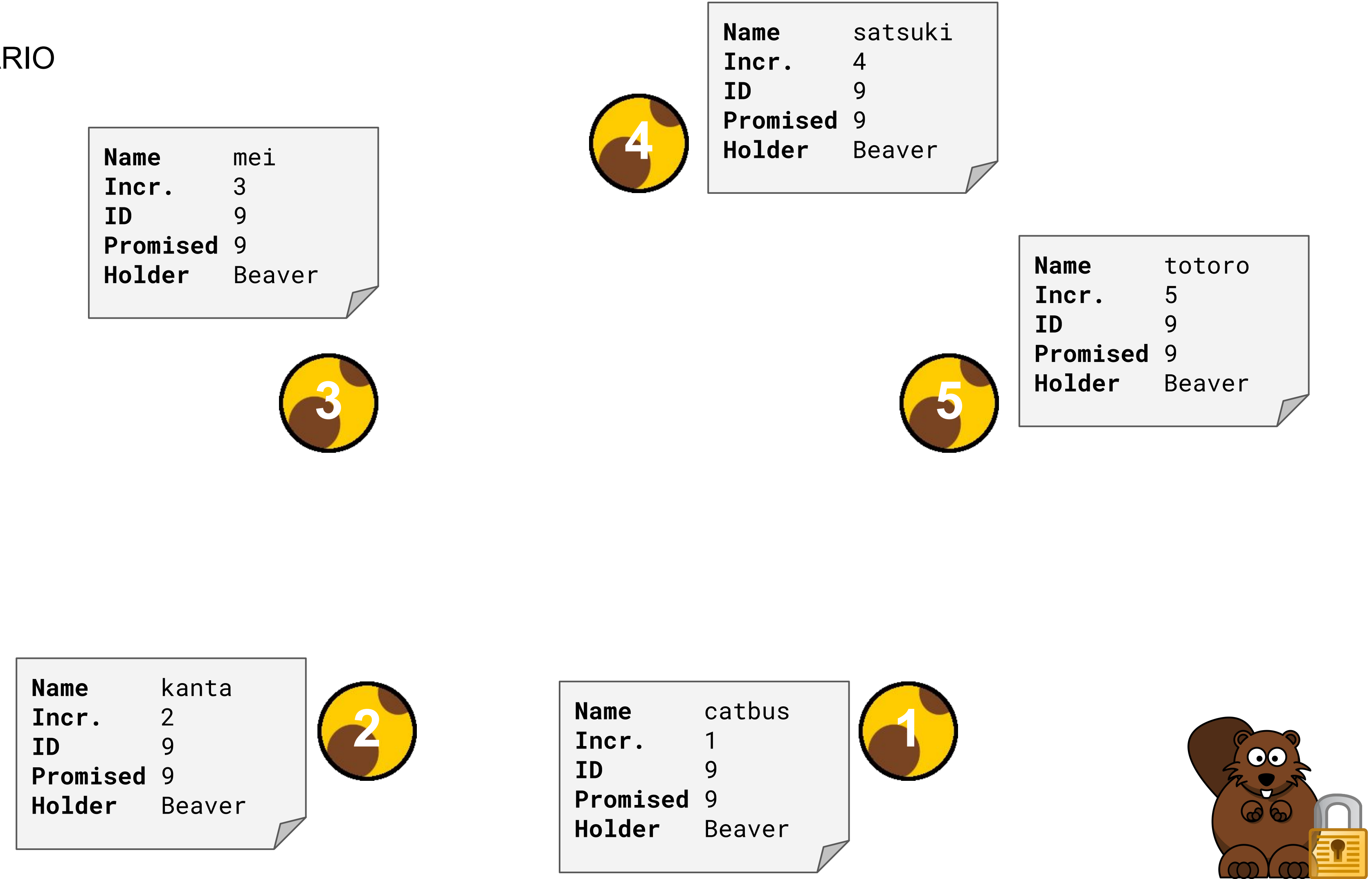
4.) Release Lock (using random instance)
```
skinnyctl release
```

5.) Inspect Quorum
```
skinnyctl status
```

Note: Reset the Quorum to initial state to start over!
```
./scripts/reset-experiment-one.sh
```

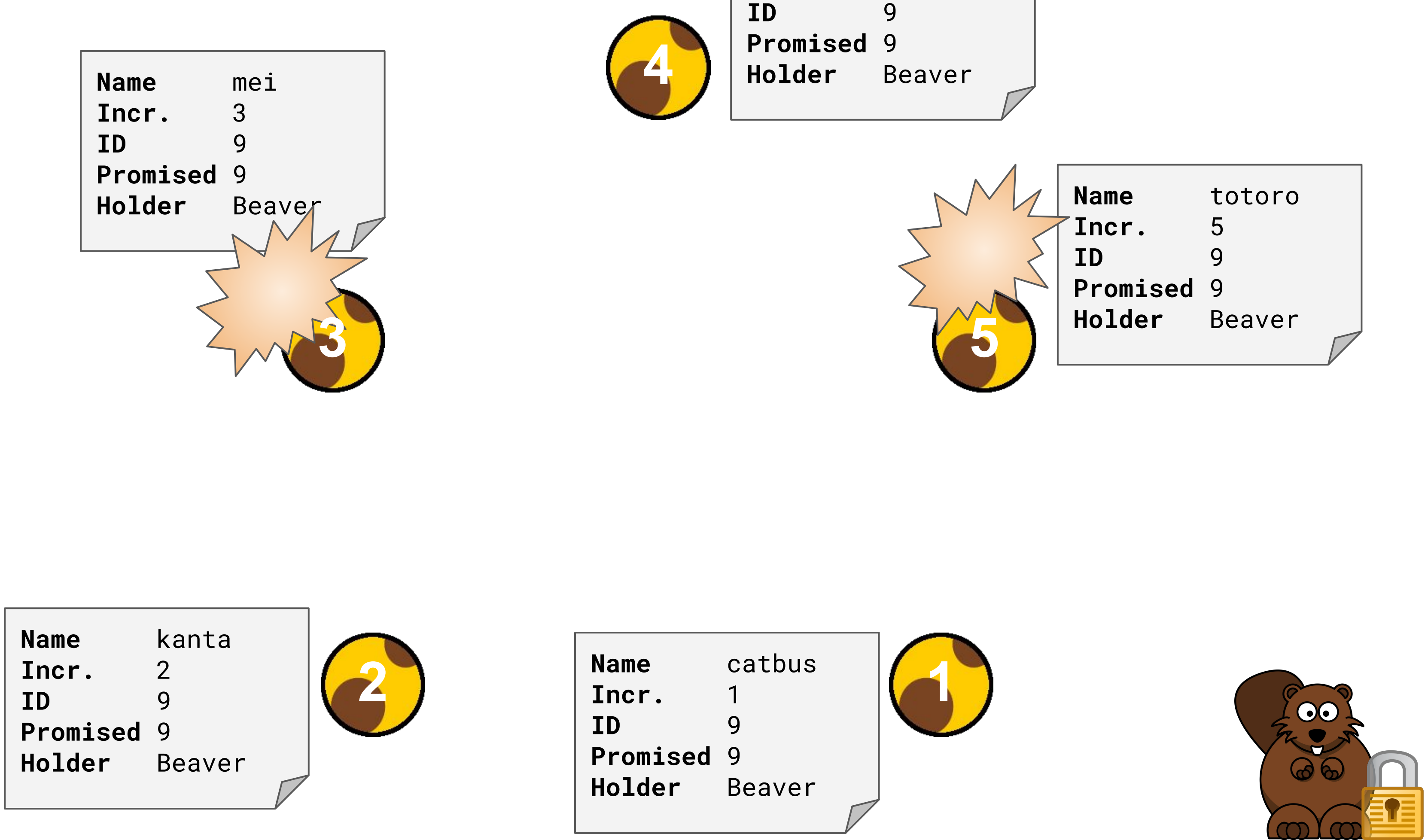
How Skinny deals with Instance Failure

SCENARIO

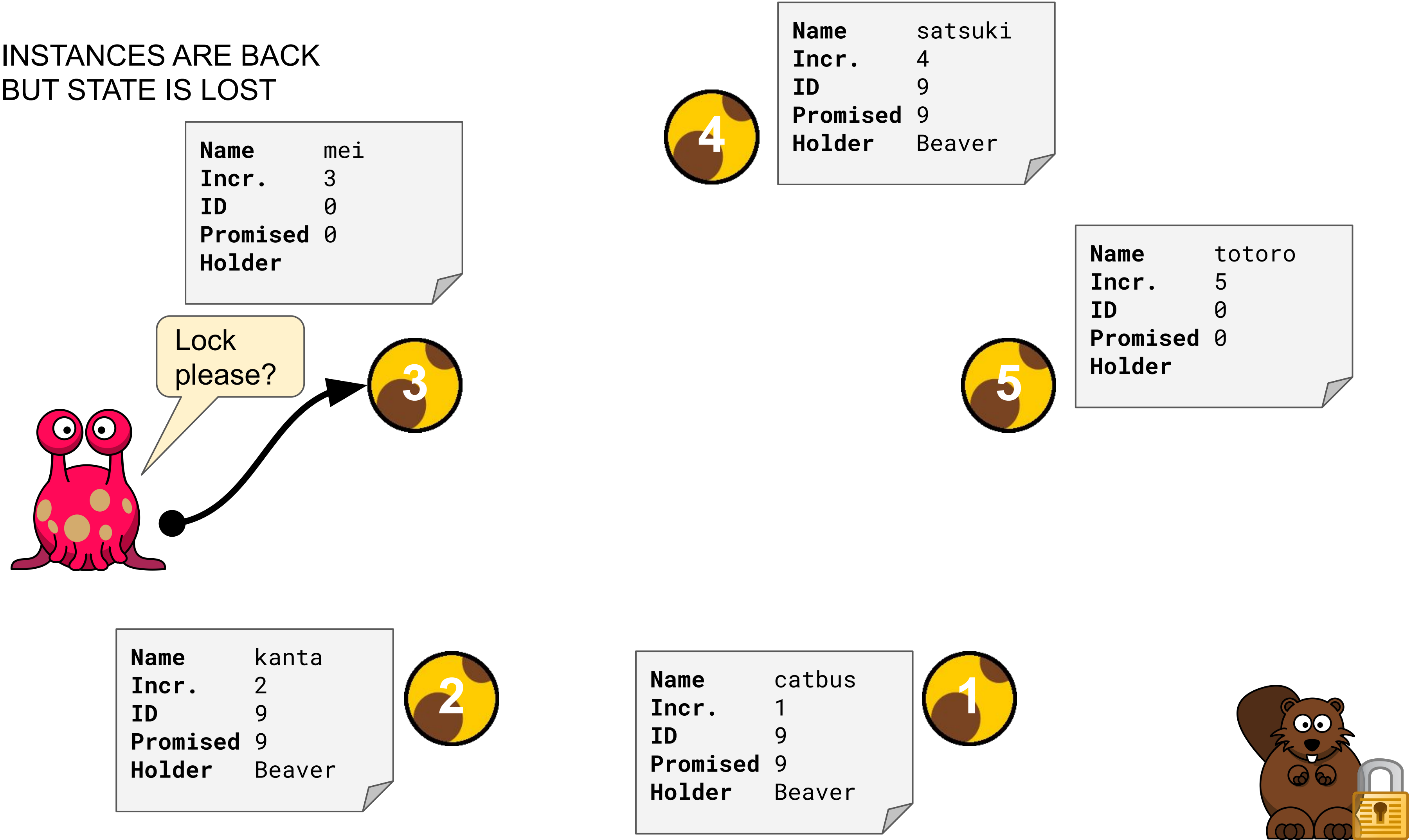


Name satsuki
Incr. 4
ID 9
Promised 9
Holder Beaver
4
Name mei
Incr. 3
ID 9
Promised 9
Holder Beaver
3
Name totoro
Incr. 5
ID 9
Promised 9
Holder Beaver
5
Name kanta
Incr. 2
ID 9
Promised 9
Holder Beaver
2
Name catbus
Incr. 1
ID 9
Promised 9
Holder Beaver
1

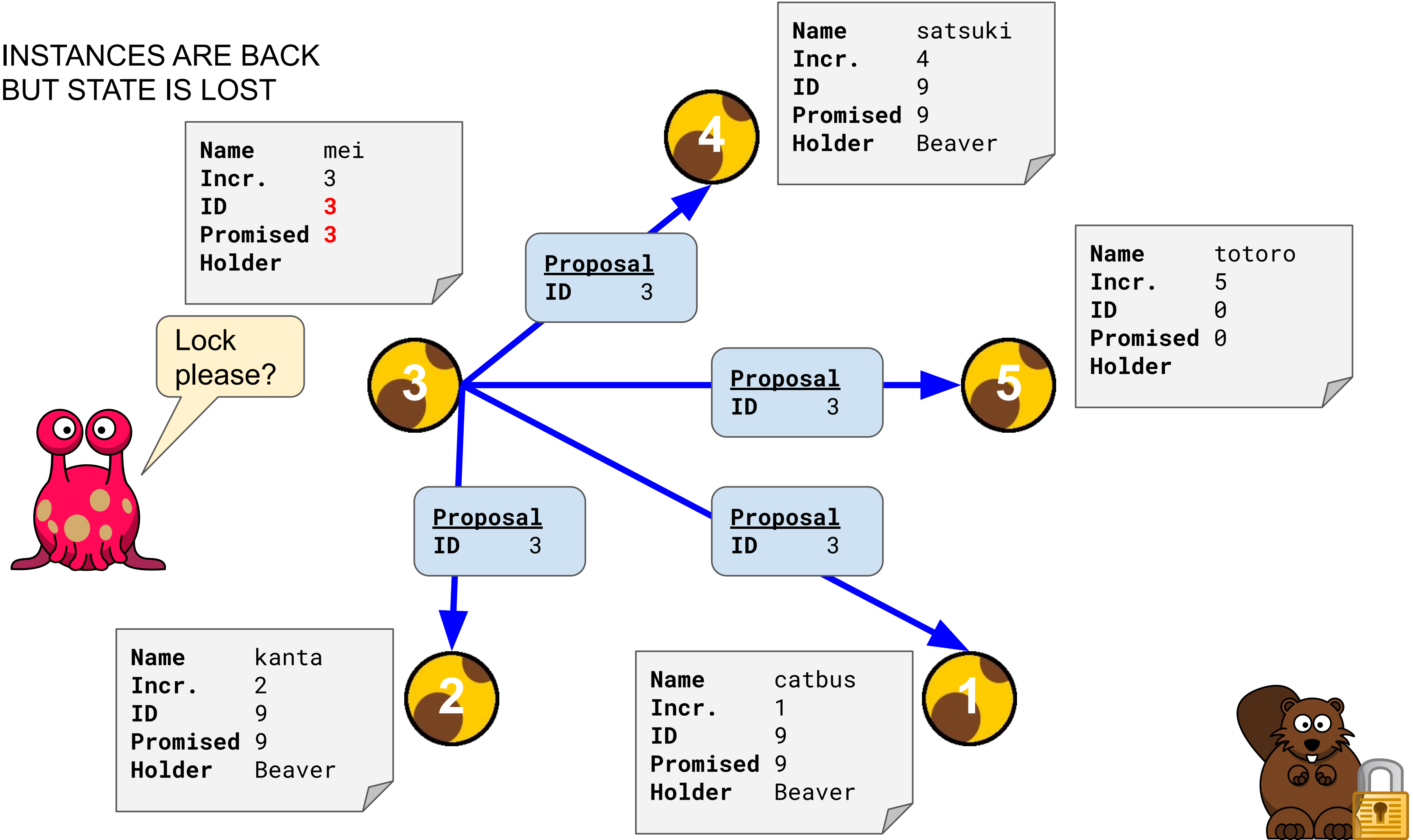
TWO INSTANCES FAIL



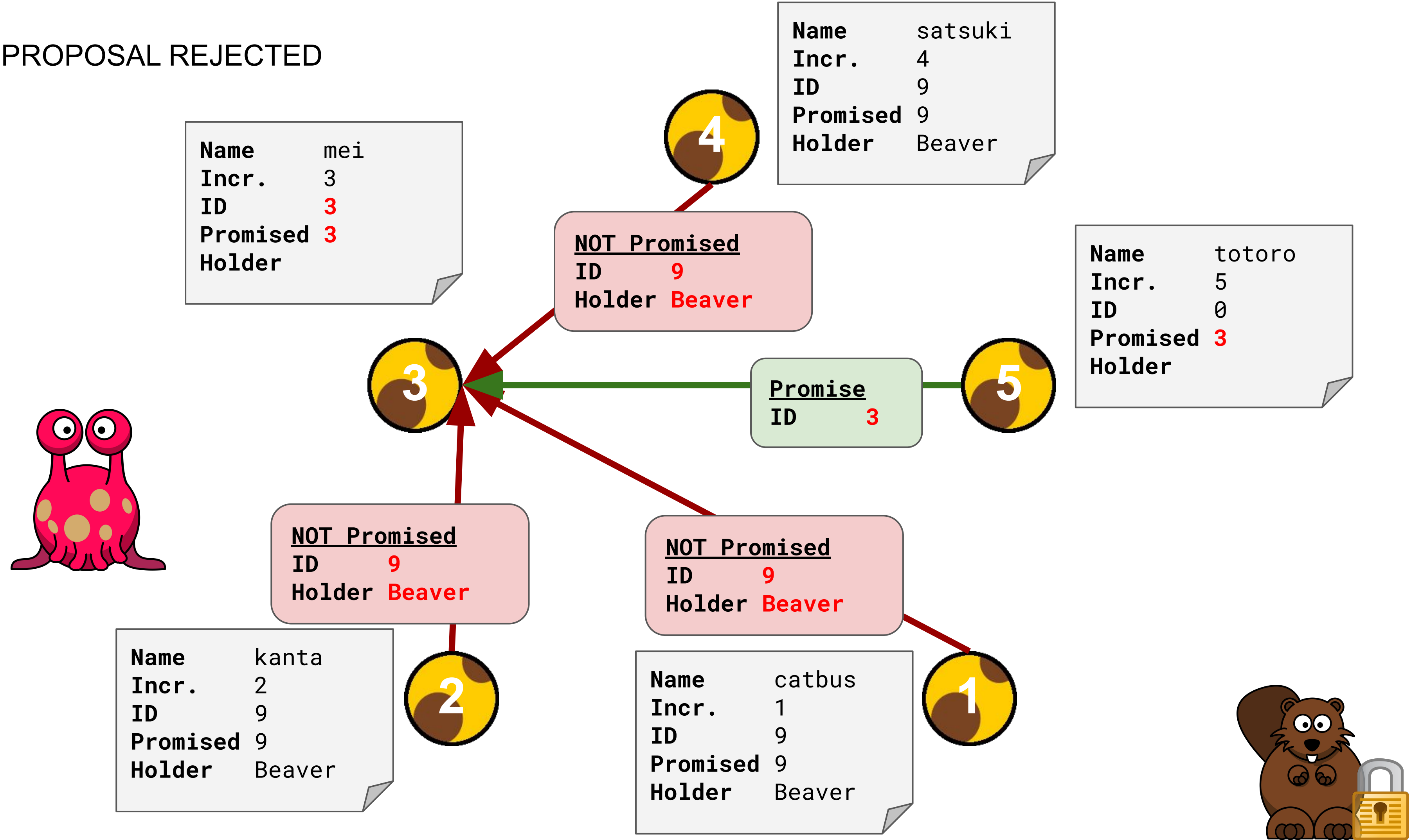
INSTANCES ARE BACK BUT STATE IS LOST



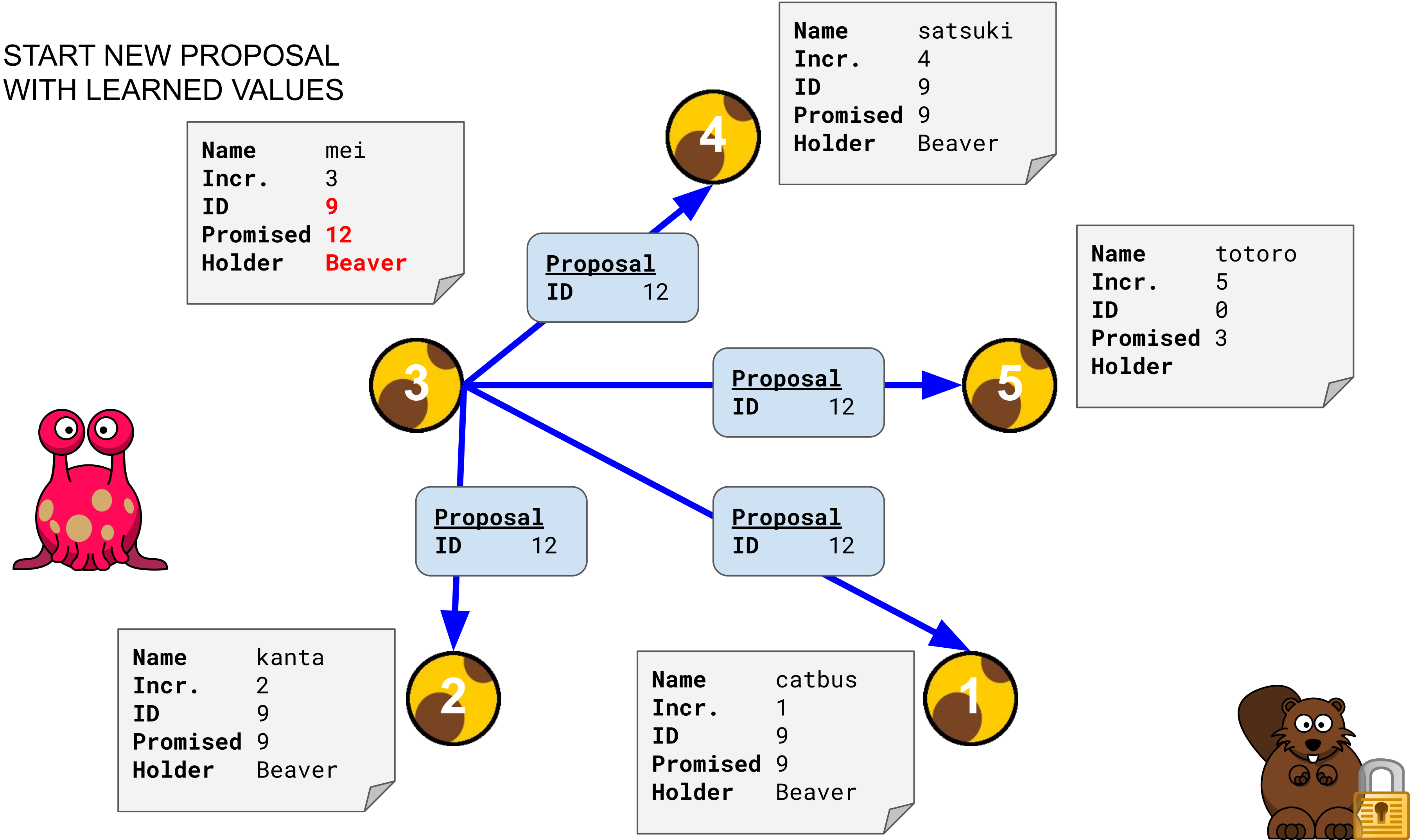
INSTANCES ARE BACK BUT STATE IS LOST



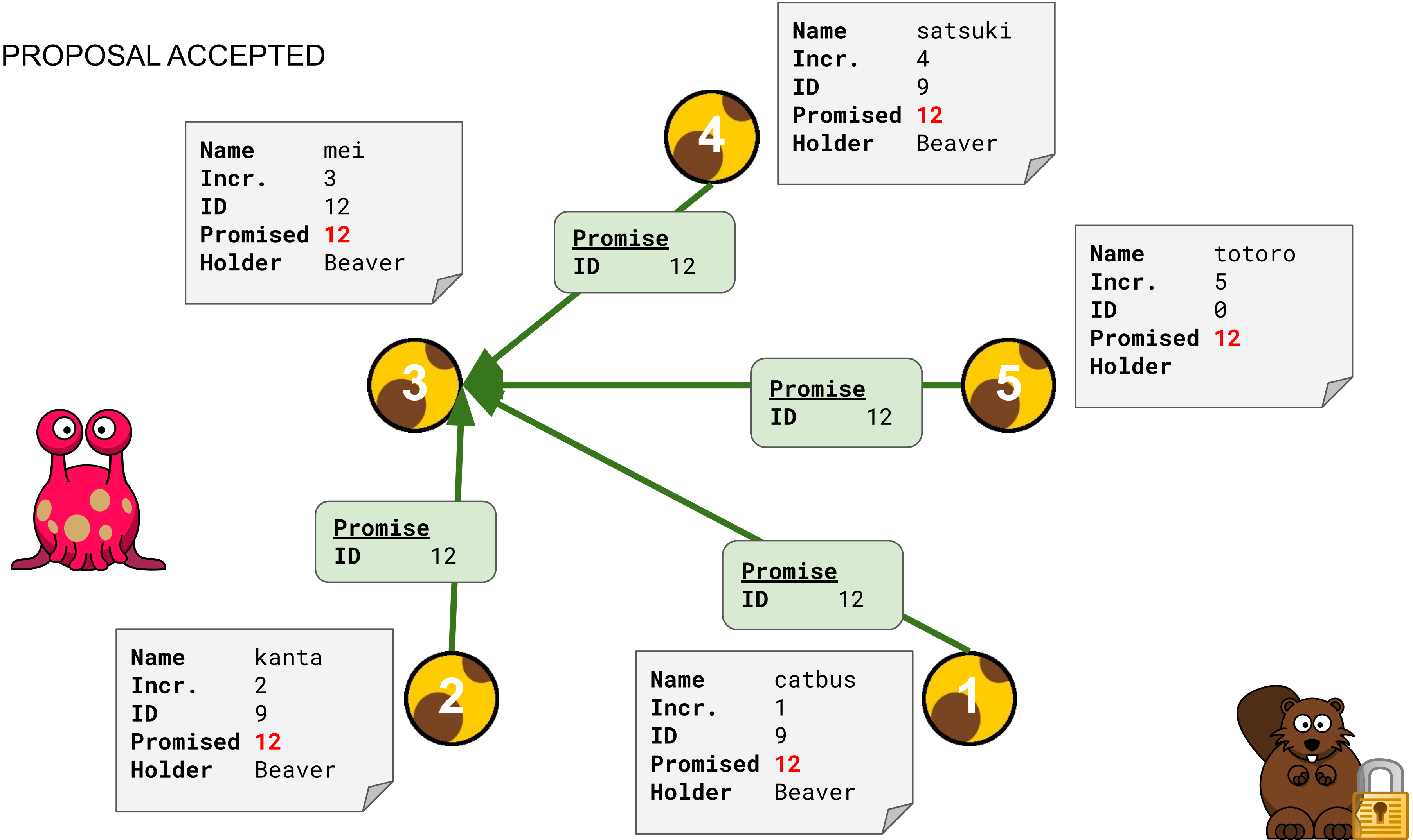
PROPOSAL REJECTED



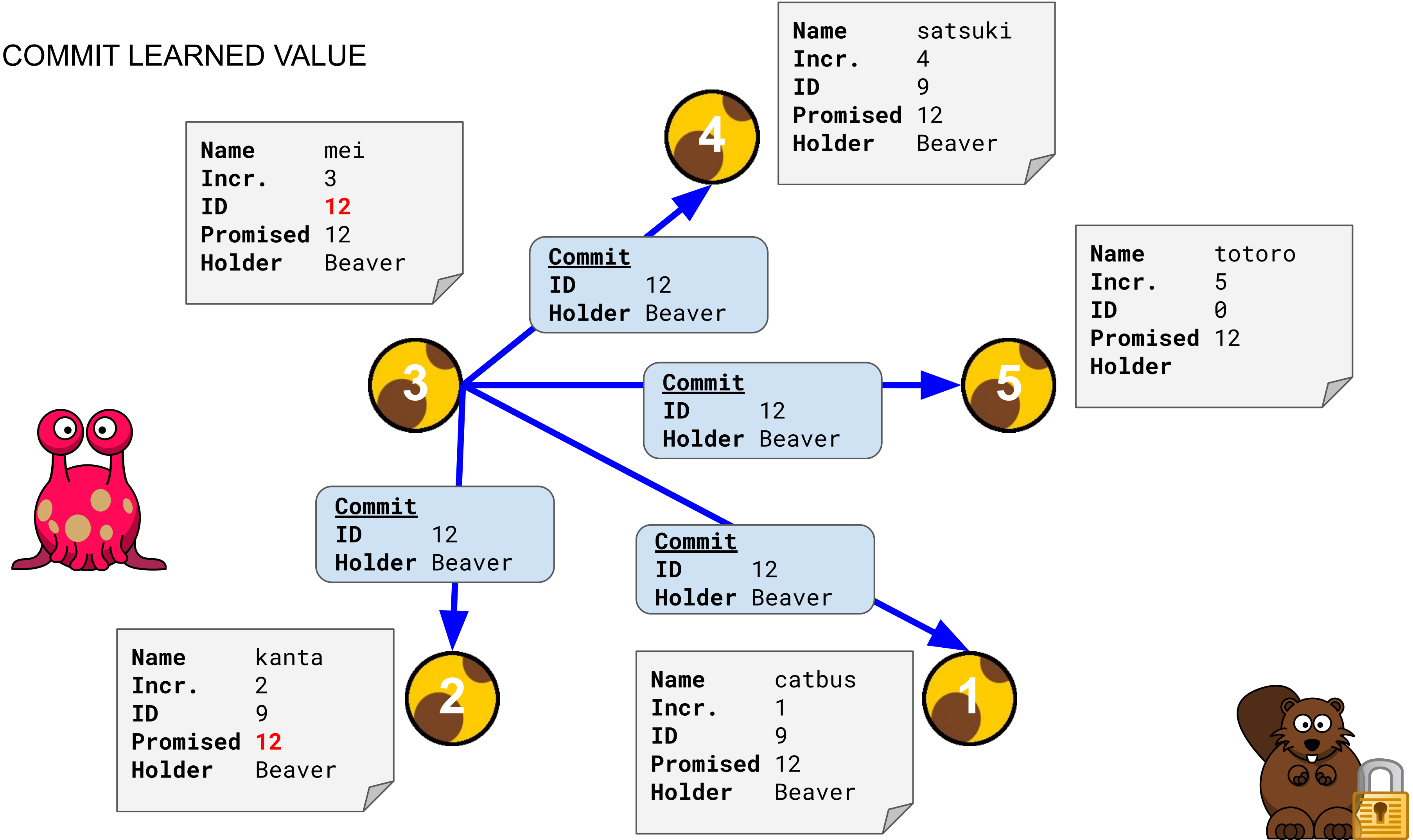
START NEW PROPOSAL WITH LEARNED VALUES



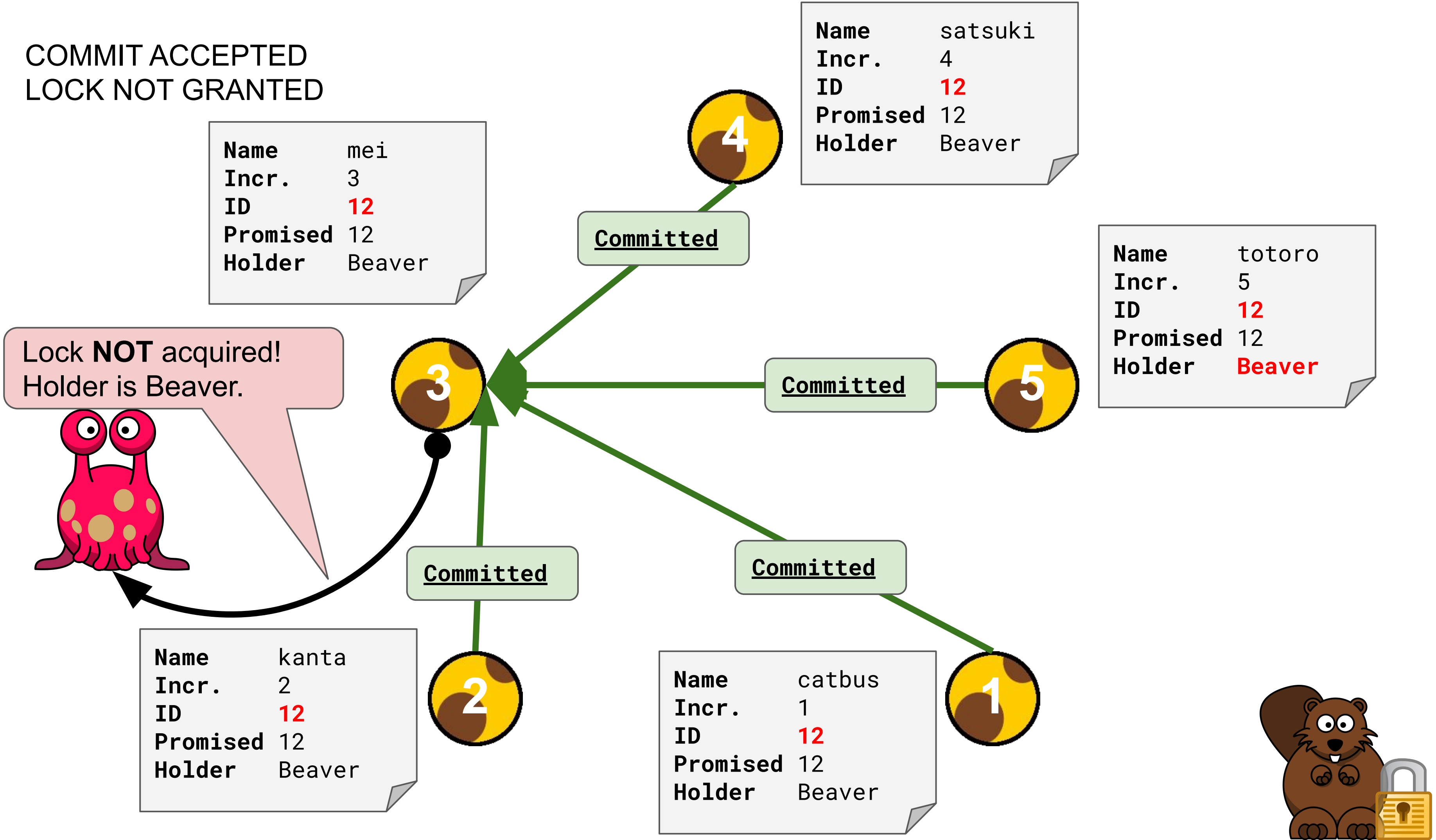
PROPOSAL ACCEPTED



COMMIT LEARNED VALUE



COMMIT ACCEPTED
LOCK NOT GRANTED



Experiment Two

Important: Run all commands in folder `~/skinny/doc/workshop/`

Screwed up? No worries!
Reset the Quorum to initial state via:
`./scripts/reset-experiment-two.sh`

1.) Inspect Quorum
```
skinnyctl status
```

2.) Stop instances *mei* and *totoro*
```
sudo systemctl stop skinny@mei
sudo systemctl stop skinny@totoro
```

3.) Inspect Quorum. Verify that instances *mei* and *totoro* are down!
```
skinnyctl status
```

4.) Start instances *mei* and *totoro* again
```
sudo systemctl start skinny@mei
sudo systemctl start skinny@totoro
```

5.) Inspect Quorum. Verify that instances *mei* and *totoro* are out of sync!
```
skinnyctl status
```

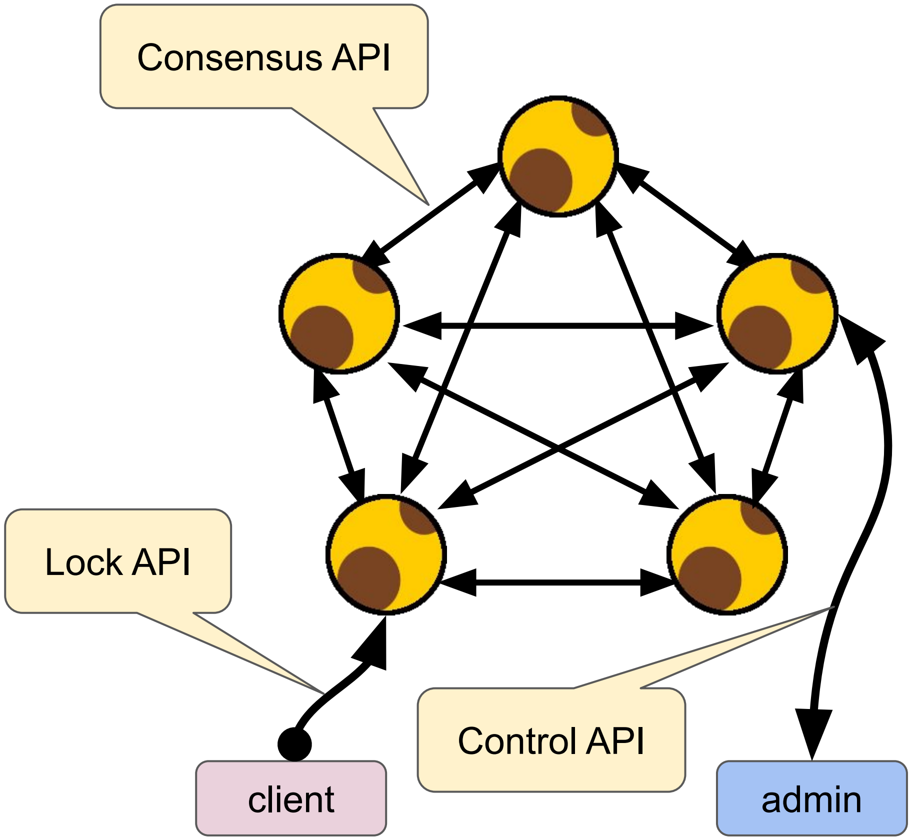
6.) Acquire Lock for "alien" using instance *mei*
```
skinnyctl acquire --instance=mei alien
```

Part III
Implementing Distributed Consensus

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

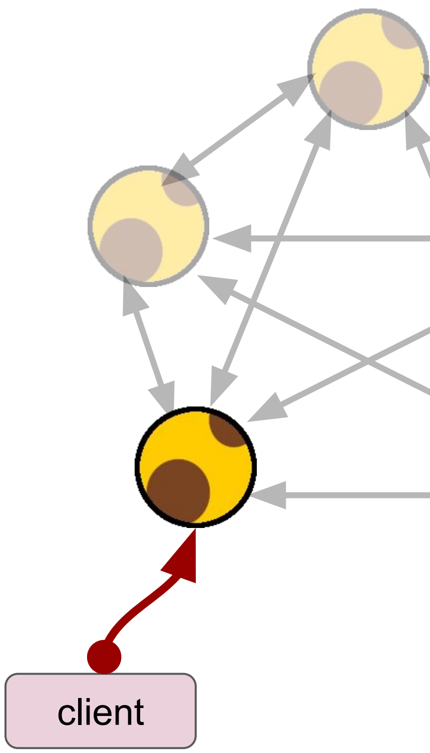


Lock API

```
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);
}
```

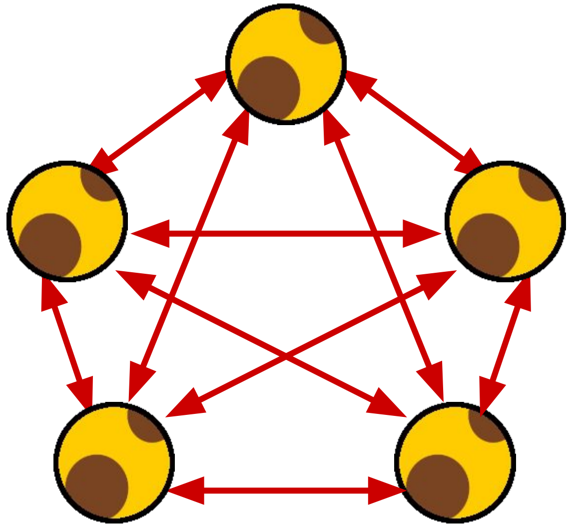


Consensus API

```
// 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;
}
```

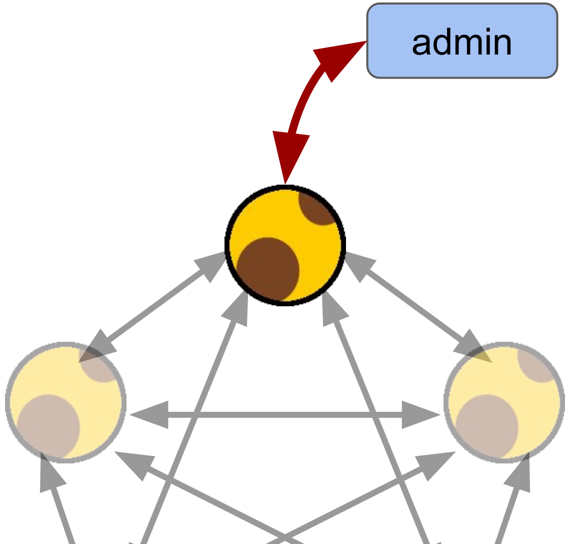
```
service Consensus {
   rpc Promise (PromiseRequest) returns (PromiseResponse);
   rpc Commit (CommitRequest) returns (CommitResponse);
}
```



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);
}
```



Reaching Out...

Skinny Instance

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

```go
// 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
}
```

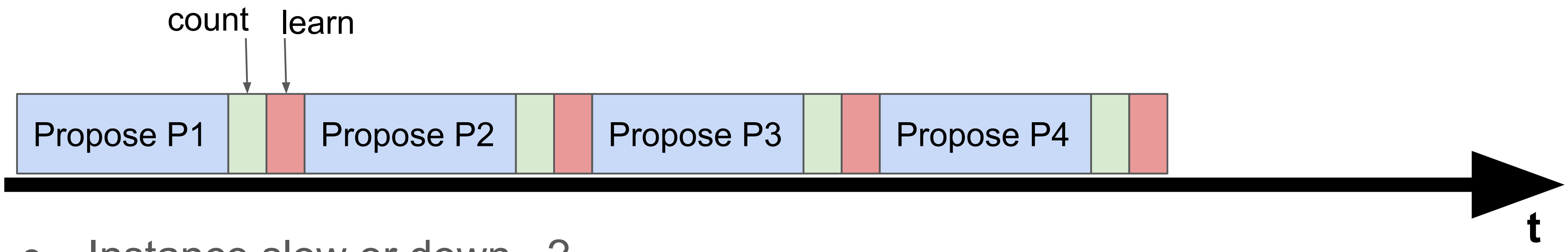
Propose Function

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

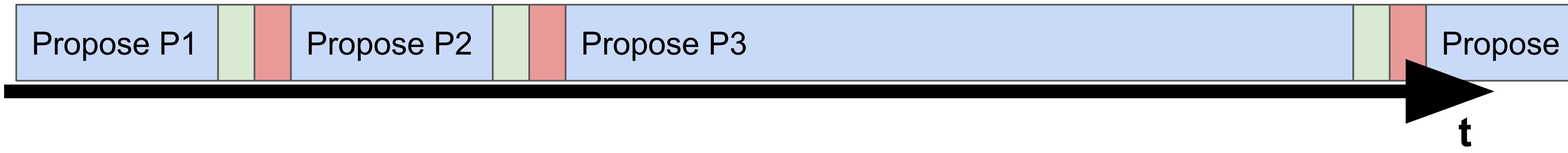
```go
for _, p := range in.peers {
	// send proposal
	resp, err := p.client.Promise(
		context.Background(),
		&pb.PromiseRequest{ID: proposal})
	if err != nil {
		continue
	}
	if resp.Promised {
		yea++
	}
	learn(resp)
}
```

Resulting Behavior

- Sequential Requests
- Waiting for IO

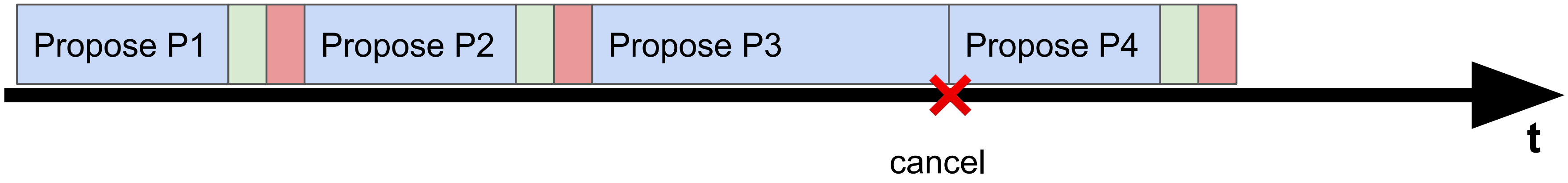


- Instance slow or down...?



Improvement #1

- Limit the Waiting for IO



Timeouts

- WithTimeout()
  - Here: Hardcoded
  - Real world: Configurable
- Cancel() to prevent context leak

```go
for _, p := range in.peers {
    // send proposal
    ctx, cancel := context.WithTimeout(
        context.Background(),
        time.Second*10)
    resp, err := p.client.Promise(ctx,
        &pb.PromiseRequest{ID: proposal})
    cancel()
    if err != nil {
        continue
    }
    if resp.Promised {
        yea++
    }
    learn(resp)
}
```

Improvement #2

- Concurrent Requests
- Synchronized Counting
- Synchronized Learning

Concurrency

- 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*10)
        defer cancel()

        resp, err := p.client.Promise(ctx,
            &pb.PromiseRequest{ID: proposal})
        if err != nil { return }

        // now what?
    }(p)
}
```

Synchronizing

- Define response data structure
- Channels to the rescue!
- Write responses to channel as they come in

```go
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
- yea := 1
    - Because we always vote for ourselves
- Learning

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

What's wrong?

- We did not close the channel
- **`range`** is blocking forever

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

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

Solution: More synchronizing!

- Use **`WaitGroup`**
- Close channel when all requests are done

```go
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

Experiment Three

Important: Run all commands in folder `~/skinny/doc/workshop/`

1.) Copy source code of experiment three
`cp code/consensus.go.experiment-three ../../skinny/consensus.go`

2.) Build Skinny from source
`mage -d ../../ build`

3.) Restart Quorum
`./scripts/restart-quorum.sh`

4.) Inspect Quorum
`skinnyctl status`

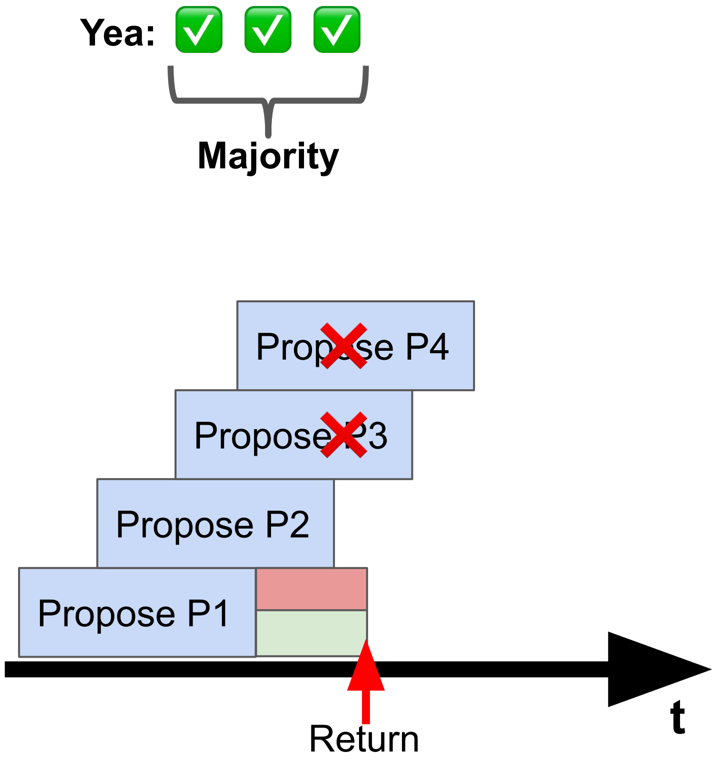
5.) Acquire Lock for "beaver" and stop the time
`skinnyctl acquire beaver`

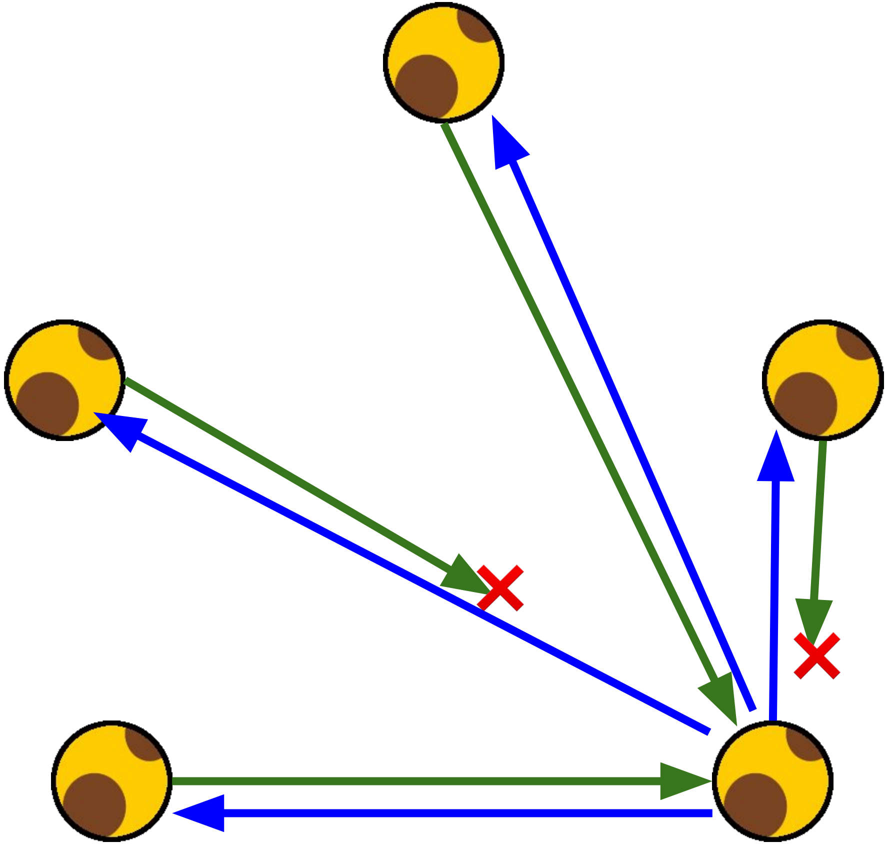
6.) Repeat previous step a couple of times.
How long does it take Beaver to acquire the lock on average (estimated)?
Do you have an idea why it took the amount of time it took?
What could be changed to improve lock acquisition times without violating the majority requirement?

Hint: Specify an instance when acquiring/releasing and Inspect the instance's logs (`cheat-sheet.pdf`)

Ignorance Is Bliss?

Early Stopping





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

```go
type response struct {
    from     string
    promised bool
    id       uint64
    holder   string
}
responses := make(chan *response)

ctx, cancel := context.WithTimeout(
    context.Background(),
    time.Second*10)

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

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

    learn(r)

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

Homework

1.) Copy source code of experiment three (start from there)
`cp code/consensus.go.experiment-three ../../skinny/consensus.go`

2.) Implement "early stopping"
   a.) Use a global context
   b.) Distinguish between context errors and other errors
   Handle them differently
   c.) Make sure to stop as soon as you have a majority
   Note: A majority of negative answers is still a majority!

Hint: You can find a reference implementation in `skinny/consensus.go`

Sources

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.*

…act

W… …ribe our experiences with the Chubby lock ser-
…ice, which is intended to provide coarse-grained lock-
…s well as reliable (though low-volume) storage for
…osely-coupled distributed system. Chubby provides
…in…rface much like a distributed file system with ad-
…cks, but the design emphasis is on availability
…liability 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-

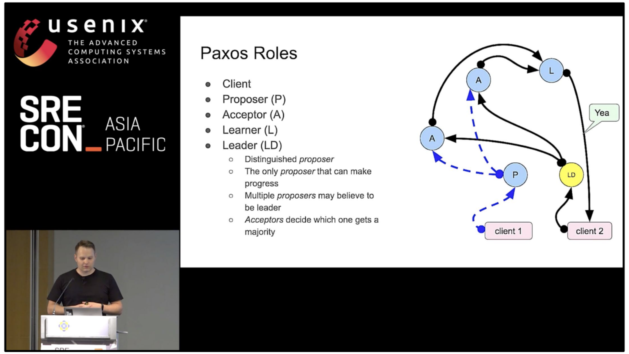
https://research.google.com/archive/chubby-osdi06.pdf

Further Watching

Paxos Agreement - Computerphile
Dr. 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

Further Watching



SREcon19 APAC - Implementing Distributed Consensus
Yours truly
https://youtu.be/nyNCSM4vGF4

Try, Play, Learn!

- The Skinny lock server is open source software!
- Terraform module
- Ansible playbook
- Packer config

github.com/danrl/skinny

NAME	INCREMENT	PROMISED	ID	HOLDER	LAST SEEN
london	1	2	2		now
oregon	2	2	2		now
spaulo	3	2	2		now
sydney	4	2	2		now
taiwan	5	2	2		now

Find me on Twitter **@danrl_com**
I blog about SRE and technology: https://**danrl.com**