Clustering in Go

May 2016

Wilfried Schobeiri
MediaMath

Who am I?

- Go enthusiast
- These days, mostly codes for fun
- Focused on Infrastruture & Platform @ MediaMath (http://careers.mediamath.com)
- We're hiring!

Why Go?

- Easy to build services
- Great stdlib
- Lot's of community libraries & utilities
- Great built-in tooling (like go fmt, test, vet, -race, etc)
- Compiles as fast as a scripting language
- Just "feels" productive
- (This is not a language pitch talk)

Why "Clustering in Go"?

Why "Clustering in Go"?

- Clustering is not batteries included in Golang
- Lots of newer libraries, none very mature
- More often not, services roll it themselves
- So, here's one way of building a clustered, stateful service in Go.

And now, for a (fake-ish) scenario

- Multiple datacenters
- Separated by thousands of miles each (eg, ORD - HKG - AMS),
- With many events happening concurrently at each one.
- **We want to count them.**

With some constraints:

- Counting should be **fast**, we can't afford to cross the ocean every time
- Counts should be **correct** (please don't lose my events)

Starting to look like an AP system, right?

Let's get started

First, a basic counter service

- One node
- Counter = Atomic Int
- Nothin Fancy

```
$ curl http://localhost:4000/

0

$ curl http://localhost:4000/inc?amount=1

1
```

A basic Counter Service

```
type Counter struct {
    val int32
}

// IncVal increments the counter's value by d
func (c *Counter) IncVal(d int) {
    atomic.AddInt32(&c.val, int32(d))
}

// Count fetches the counter value
func (c *Counter) Count() int {
    return int(atomic.LoadInt32(&c.val))
}
```

Run

Ok, let's geodistribute it.

Ok, let's geodistribute it.

- A node (or several) in each datacenter
- Route increment requests to the closest node

Let's stand one up in each.

Demo

Duh, we're not replicating state!

- We need the counters to talk to each other
- Which means we need the nodes to know about each other
- Which means we need to to solve for cluster membership

Enter, the memberlist (https://github.com/hashicorp/memberlist) package

Memberlist

- A Go library that manages cluster membership
- Based on SWIM (https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf), a gossip-style membership protocol
- Has baked in member failure detection
- Used by Consul, Docker, libnetwork, many more

About SWIM

"Scalable Weakly-consistent Infection-style Process Group Membership Protocol"

Two goals:

- Maintain a local membership list of non-faulty processes
- Detect and eventually notify others of process failures

SWIM mechanics

- Gossip-based
- On join, a new node does a full state sync with an existing member, and begins gossipping its existence to the cluster
- Gossip about memberlist state happens on a regular interval and against number of randomly selected members
- If a node doesn't ack a probe message, it is marked "suspicious"
- If a suspicious node doesn't dispute suspicion after a timeout, it's marked dead
- Every so often, a full state sync is done between random members (expensive!)
- Tradeoffs between bandwidth and convergence time are configurable

More details about SWIM can be found here (https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf) and here (http://prakhar.me/articles/swim/) .

Becoming Cluster-aware via Memberlist

```
members  = flag.String("members", "", "comma seperated list of members")

...

c := memberlist.DefaultWANConfig()

m, err := memberlist.Create(c)

if err != nil {
    return err
}

//Join other members if specified, otherwise start a new cluster
if len(*members) > 0 {
    members_each := strings.Split(*members, ",")
    _, err := m.Join(members_each)
    if err != nil {
        return err
    }
}
```

Demo

CRDTs to the rescue!

CRDTs, simplified

- CRDT = Conflict-Free Replicated Data Types
- Counters, Sets, Maps, Flags, et al
- Operations within the type must be associative, commutative, and idempotent
- Order-free
- Therefore, very easy to handle failure scenarios: just retry the merge!

CRDTs are by nature eventually consistent, because there is no single source of truth.

Some notes can be found here (http://hal.upmc.fr/inria-00555588/document) and here (https://github.com/pfraze/crdt_notes) (among many others!).

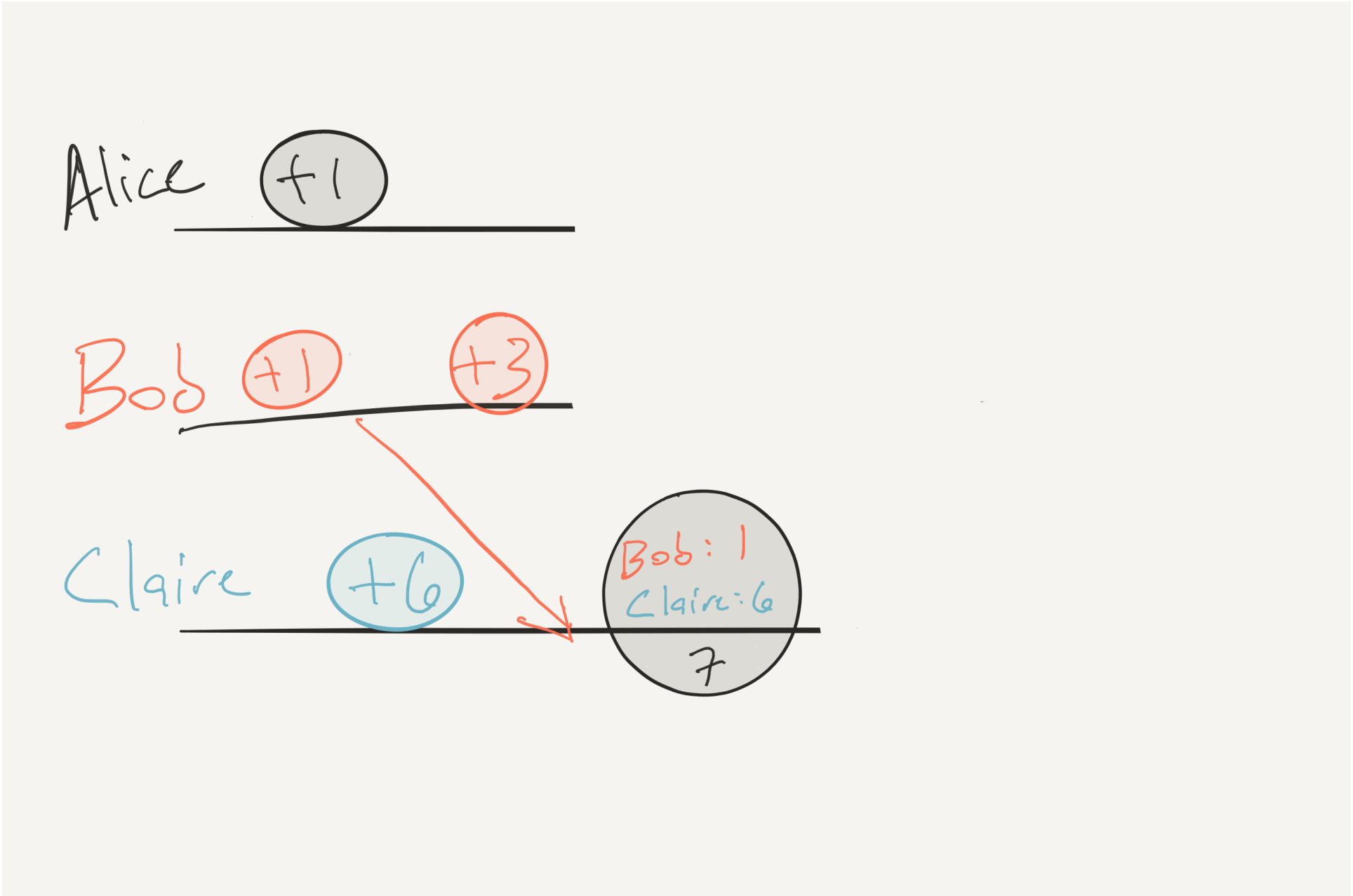
G-Counter

Perhaps one of the most basic CRDTs:

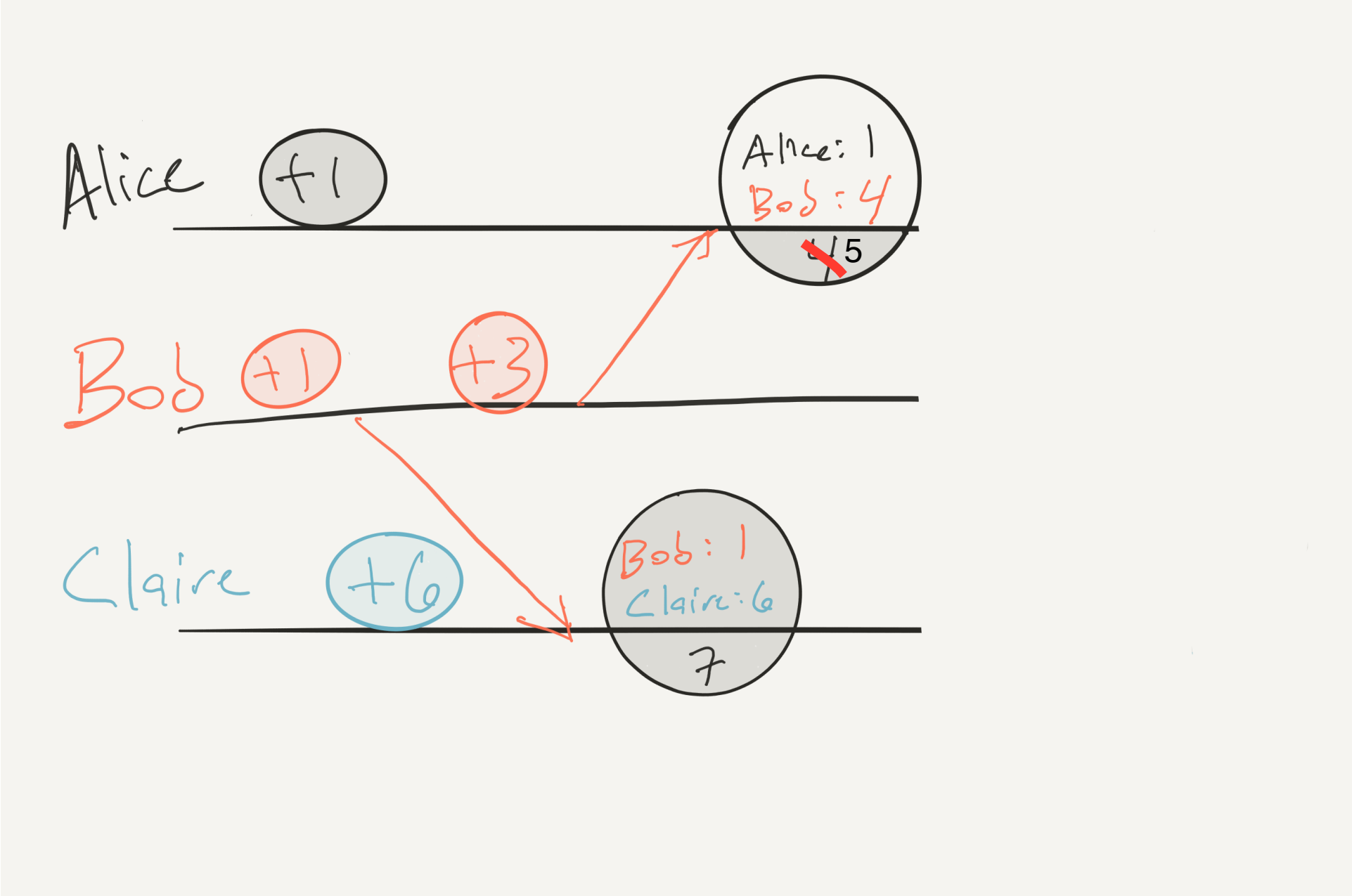
- A counter with only two ops: increment and merge (no decrement!)
- Each node manages its own count
- Nodes communicate their counter state with other nodes
- Merges take the max() count for each node

G-Counter's Value is the sum of all node count values

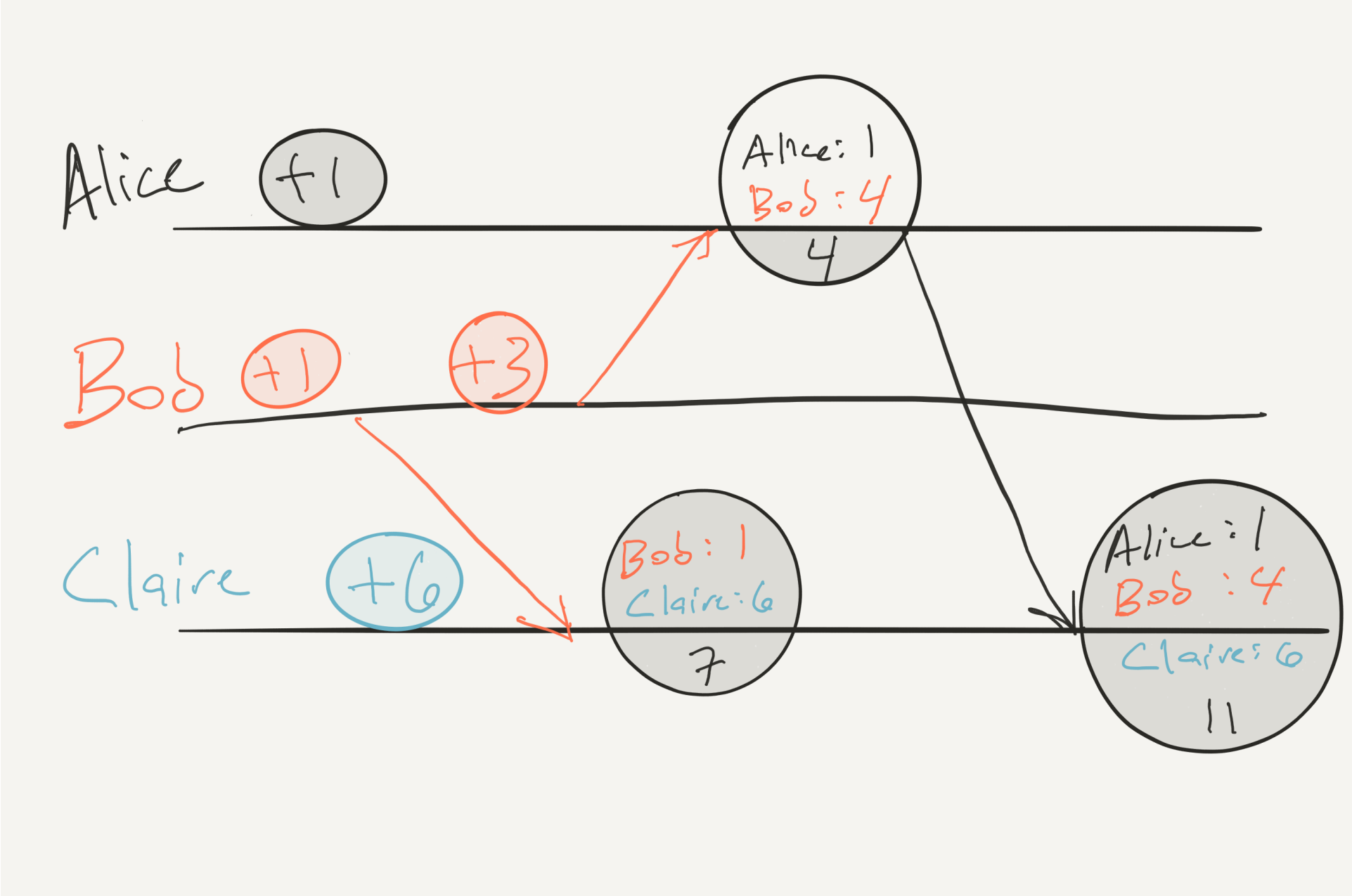
G-counter



G-counter



G-counter


Alice
+1
Alice: 1
Bob: 4
4
Bob
+1
+3
Claire
+6
Bob: 1
Claire: 6
7
Alice: 1
Bob: 4
Claire: 6
11

```
type GCounter struct {
    // ident provides a unique identity to each replica.
    ident string

    // counter maps identity of each replica to their counter values
    counter map[string]int
}

func (g *GCounter) IncVal(incr int) {
    g.counter[g.ident] += incr
}

func (g *GCounter) Count() (total int) {
    for _, val := range g.counter {
        total += val
    }
    return
}

func (g *GCounter) Merge(c *GCounter) {
    for ident, val := range c.counter {
        if v, ok := g.counter[ident]; !ok || v < val {
            g.counter[ident] = val
        }
    }
}
```

Demo

Let's Merge State!

Merging State via Memberlist

```
[DEBUG] memberlist: Initiating push/pull sync with: 127.0.0.1:61300
```

- Memberlist does a "push/pull" to do a complete state exchange with another random member
- We can piggyback this state exchange via the `Delegate` interface: `LocalState()` and `MergeRemoteState()`
- Push/pull interval is configurable
- Happens over TCP

Let's use it to eventually merge state in the background.

Merging State via Memberlist

```
// Share the local counter state via MemberList to another node
func (d *delegate) LocalState(join bool) []byte {

    b, err := counter.MarshalJSON()

    if err != nil {
        panic(err)
    }

    return b
}

// Merge a received counter state
func (d *delegate) MergeRemoteState(buf []byte, join bool) {
    if len(buf) == 0 {
        return
    }

    externalCRDT := crdt.NewGCounterFromJSON(buf)
    counter.Merge(externalCRDT)
}
```

Demo

Success! (Eventually)

Want so sync faster?

It's possible to broadcast to all member nodes, via Memberlist's `QueueBroadcast()` and `NotifyMsg()`.

```
func BroadcastState() {
    ...
    broadcasts.QueueBroadcast(&broadcast{
        msg:    b,
    })
}

// NotifyMsg is invoked upon receipt of message
func (d *delegate) NotifyMsg(b []byte) {
    ...
    switch update.Action {
    case "merge":
        externalCRDT := crdt.NewGCounterFromJSONBytes(update.Data)
        counter.Merge(externalCRDT)
    }
    ...
}
```

Faster sync for more bandwidth. Still eventually consistent.

Demo

Next:

- No tests? For shame!
- Implement persistence and time windowing
- We probably want more than one node per datacenter
- Jepsen all the things
- Implement a real RPC layer instead of Memberlist's `delegate` for finer performance and authn/z control
- Run it as a unikernel within docker running inside a VM in the cloud
- Sprinkle some devops magic dust on it
- Achieve peak microservice

TL;DR:

It's not (that) hard to build a clustered service in go.

Fin!

- Questions?
- Slides and code can be found at github.com/nphase/go-clustering-example (github.com/nphase/go-clustering-example)
- MediaMath is hiring!
- Thanks!

Thank you

Wilfried Schobeiri
MediaMath
@nphase (http://twitter.com/nphase)