Reconciling Eventually-Consistent Data with CRDTs

Starring **Noel Welsh**

A myna Production

In conjunction with

_.underscore

Showing at

Scala eXchange 2013


Why are we here?

CRDTs

(An overview)

Conflict-free Replicated Data Type

Conflict-free Replicated Data Type

Conflict-free
Replicated
Data Type

Conflict-free Replicated Data Type

Conflict-free
Convergent
Commutative

Merge data automatically

#1

Scala

#2

Abstract Algebra

#3

Special Relativity

Why do we care?

You have an **awesome** web site

So you want

sub-Second

page load

Use Scala

Spray: >200K requests/s
Rails: 4640 requests/s

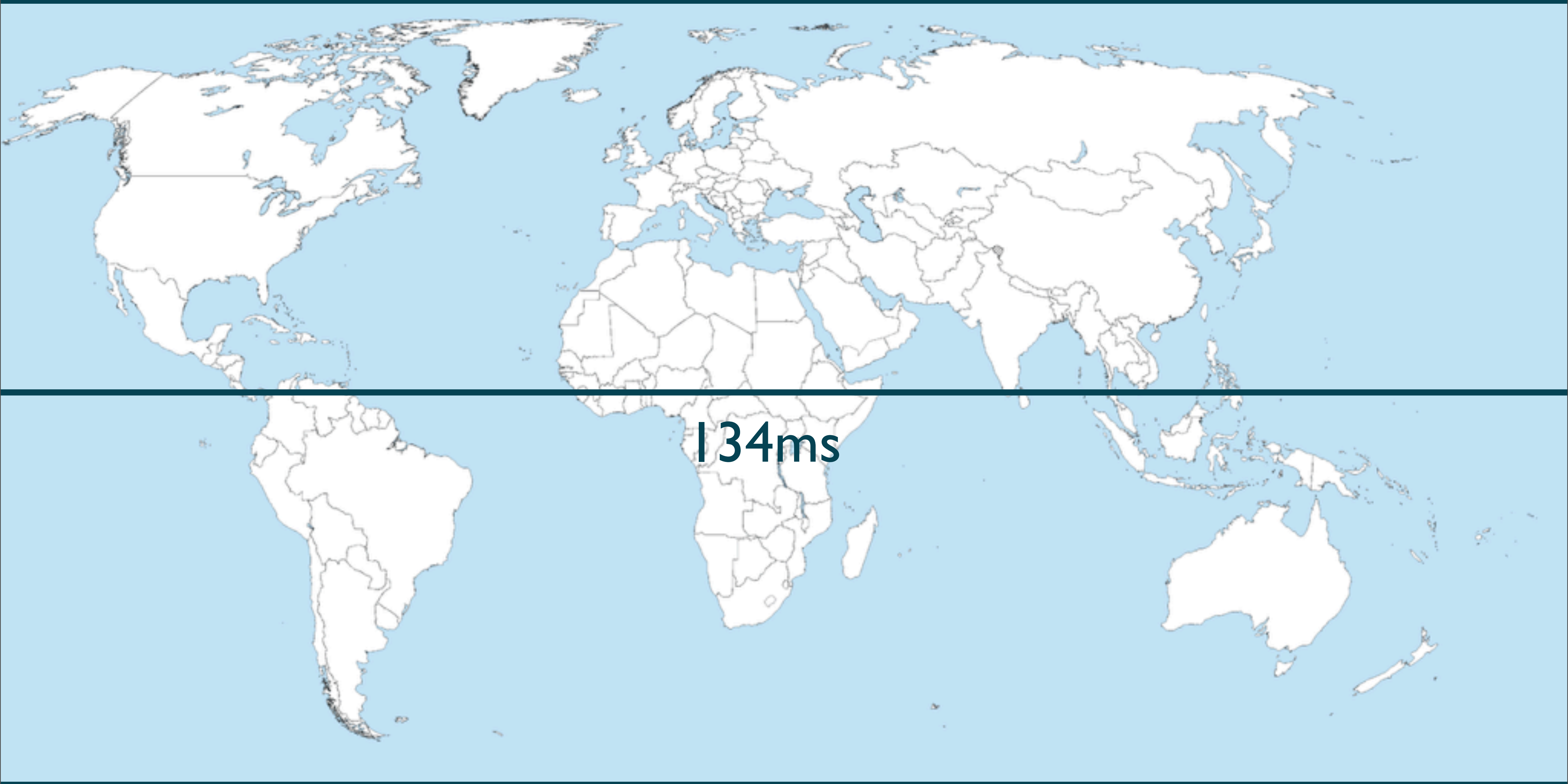
Source: Tech Empower JSON serialization benchmark
http://www.techempower.com/benchmarks/

That's

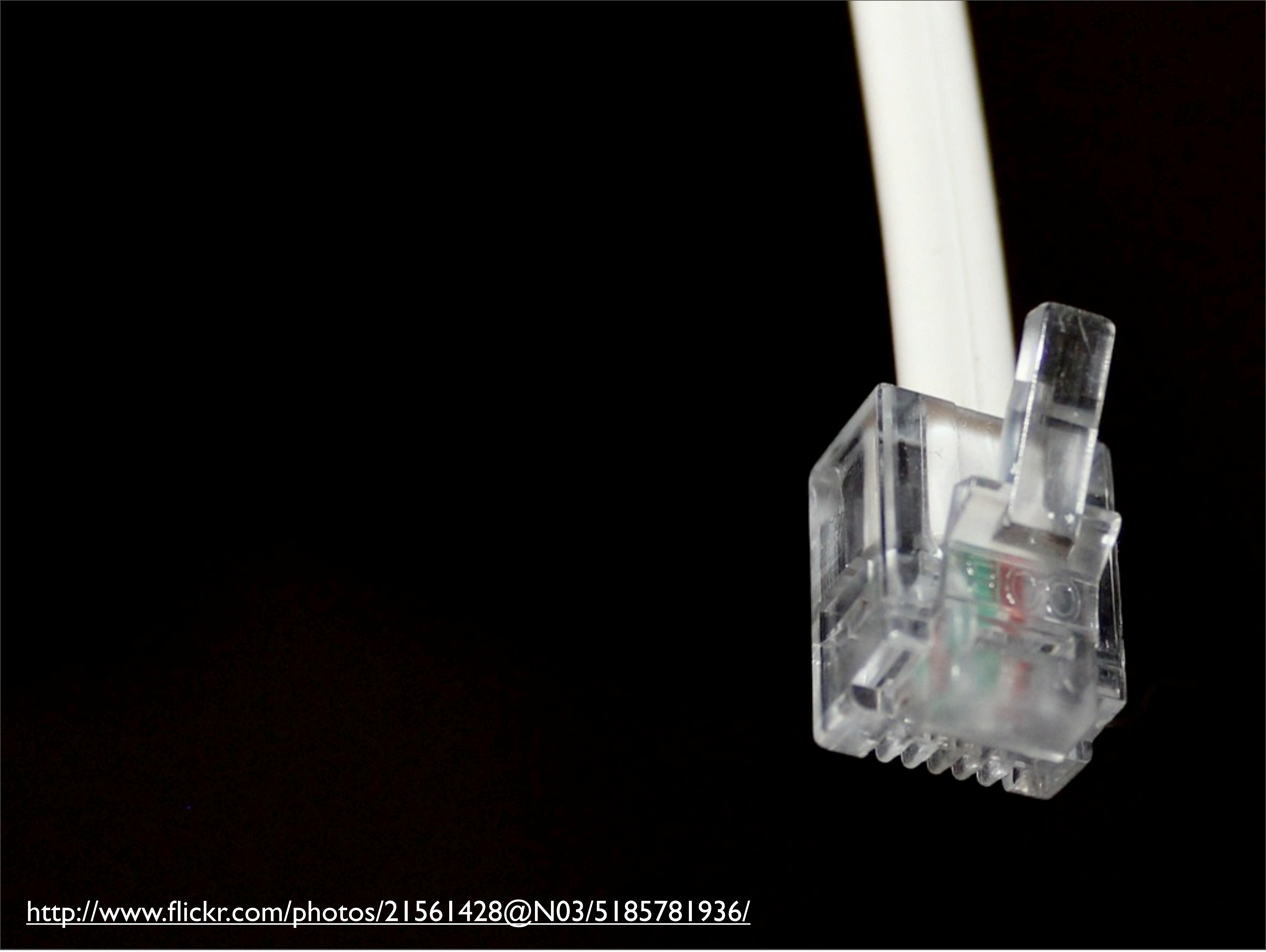
not

enough

$$E = MC^2$$

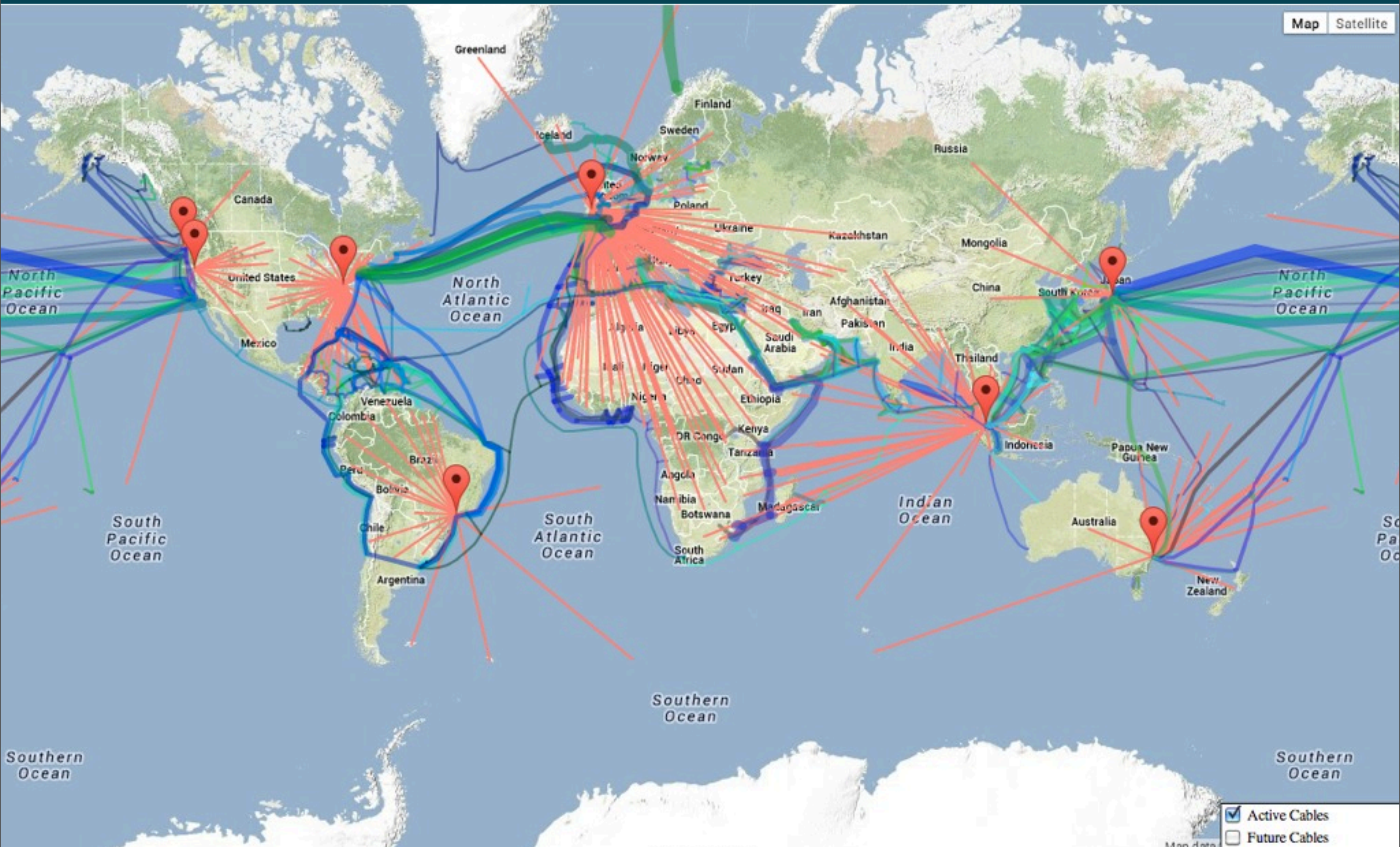


World map from Wikimedia Commons



http://www.flickr.com/photos/21561428@N03/5185781936/

LOCATION
LOCATION
LOCATION

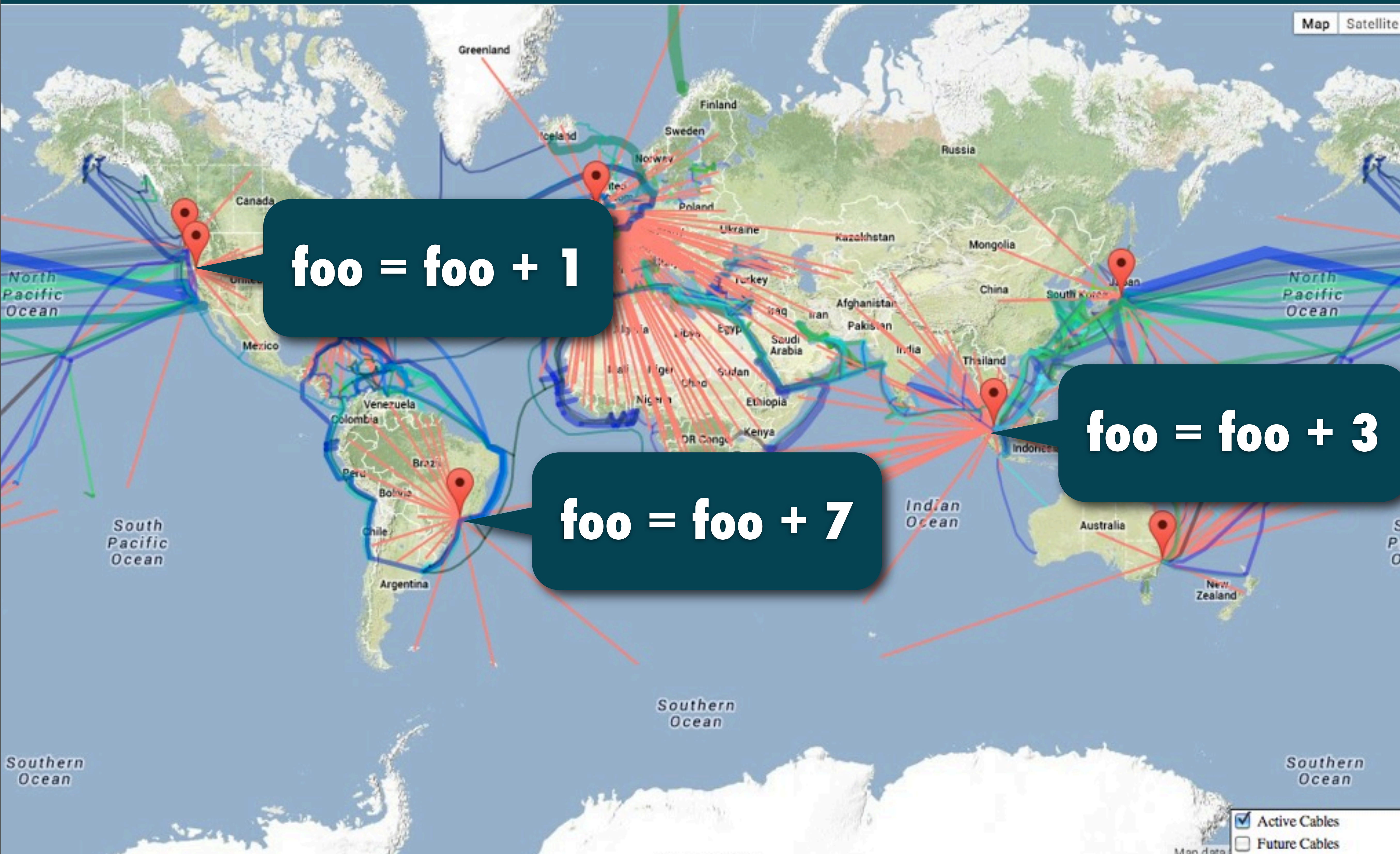


http://turnkeylinux.github.io/aws-datacenters/

Problem

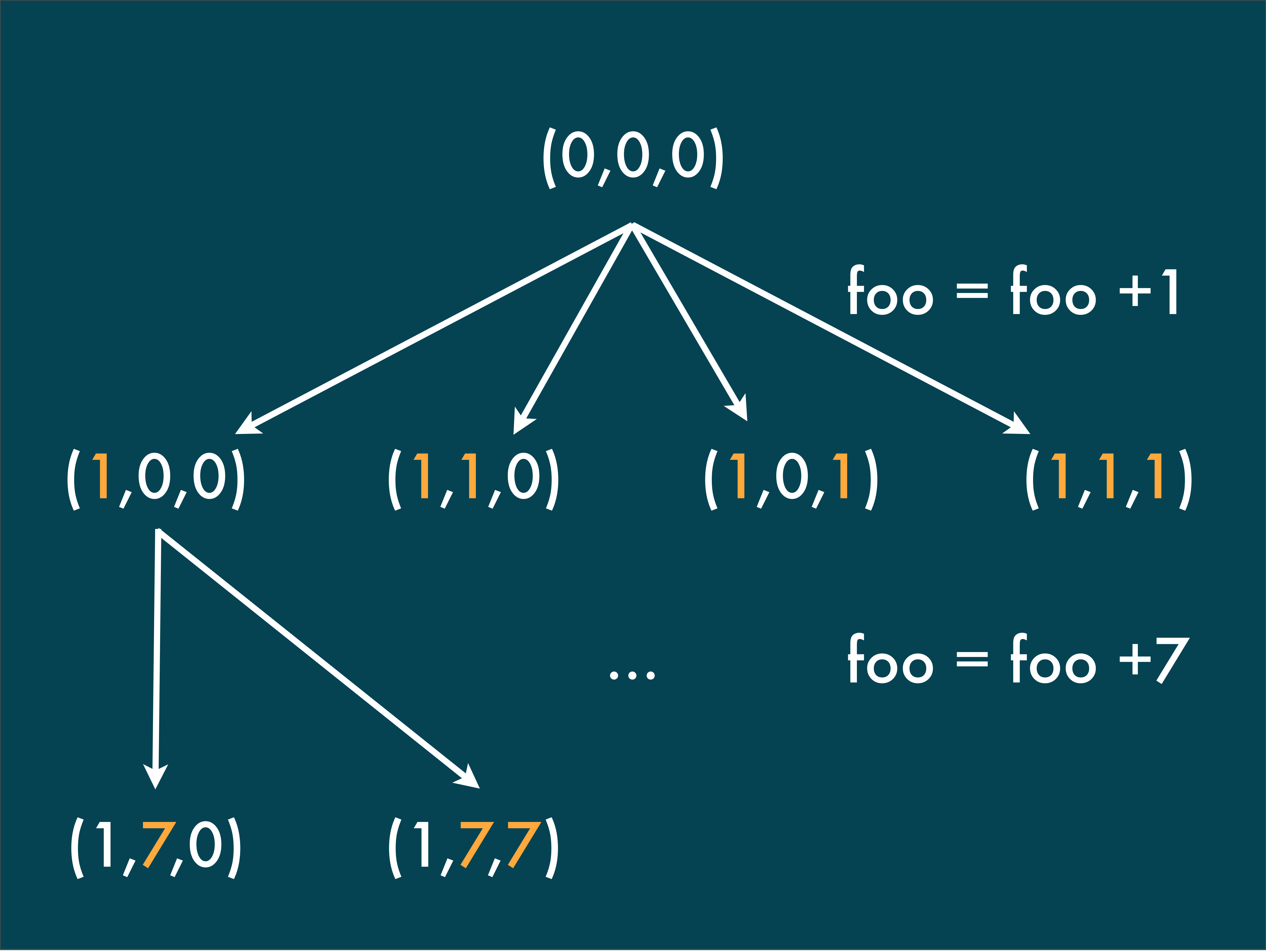
SOLVED!

Problem
SOLVED?



http://turnkeylinux.github.io/aws-datacenters/

What the FOO?


(0,0,0)
foo = foo +1
(1,0,0)
(1,1,0)
(1,0,1)
(1,1,1)
...
foo = foo +7
(1,7,0)
(1,7,7)

We have conquered

Latency

We have lost

Consistency

Solution #1 (Quantum Mechanics)

Simply consider all possible world states and ...

:-)

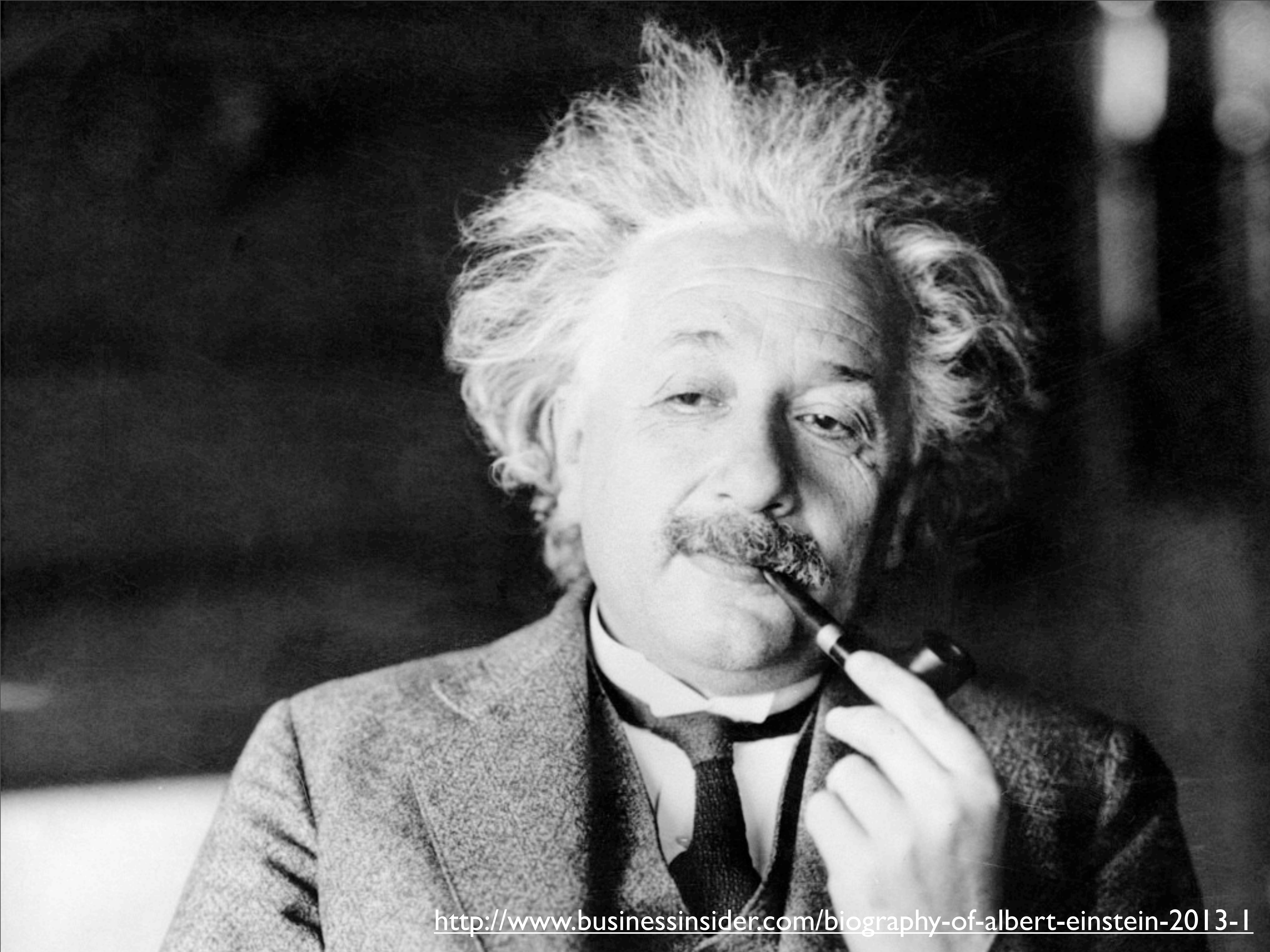
Solution #2 (Google)

Simply use atomic clocks to establish temporal ordering of events and distributed transactions ...

:-)

Solution #3 (Special Relativity)

Simply trade time for space!


http://www.businessinsider.com/biography-of-albert-einstein-2013-1

G-Counter

A counter that can only GROW

G-Counter insight: Store a separate counter for each machine

G-Counter

Machine A

A:0

B:0

Machine B

A:0

B:0

A machine can **only** increment its own counter

G-Counter

Machine A	Machine B
A:5	A:0
B:0	B:7

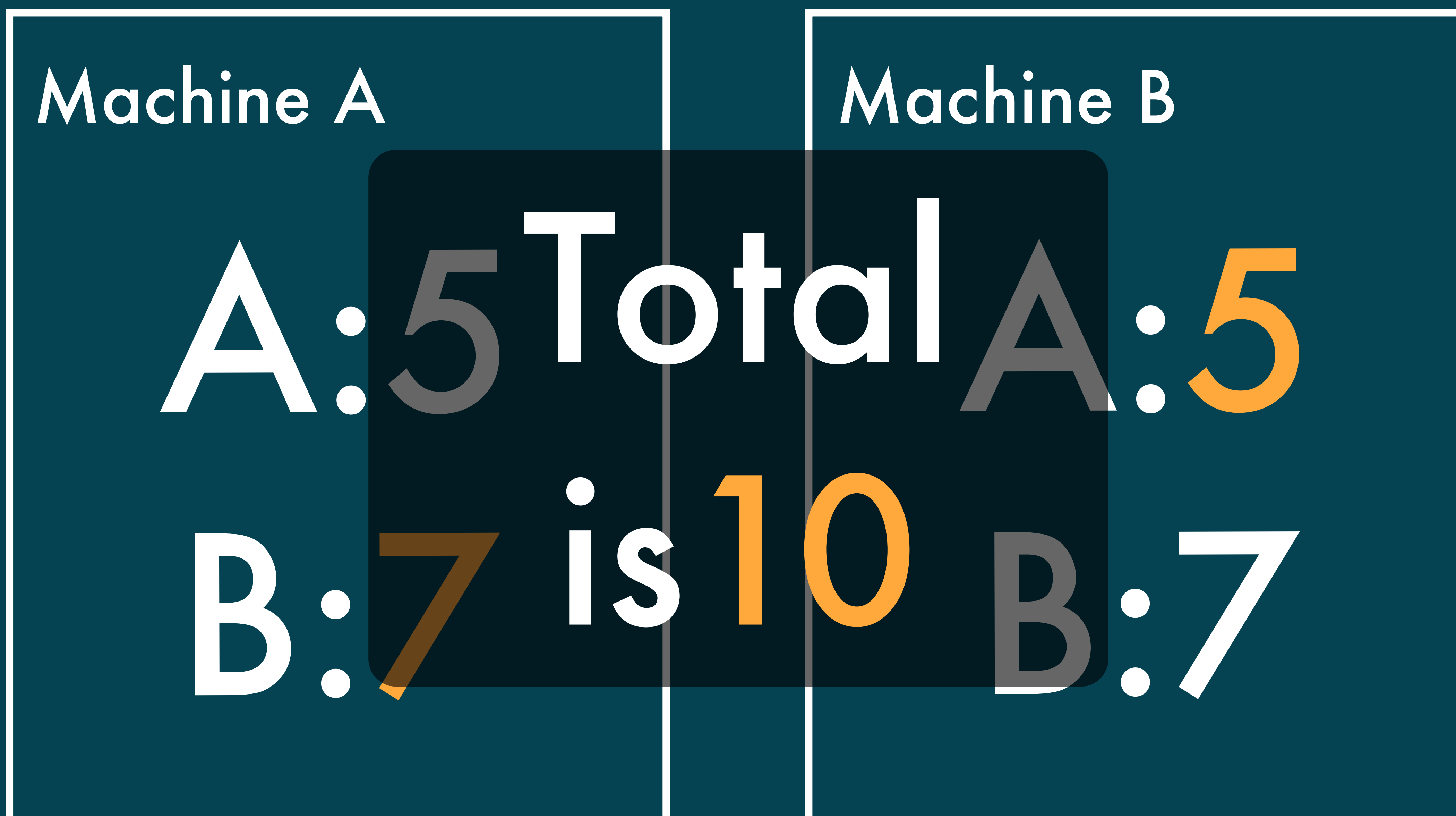
Merge is simply the **max**

G-Counter

Machine A	Machine B
A:5	A:5
B:7	B:7

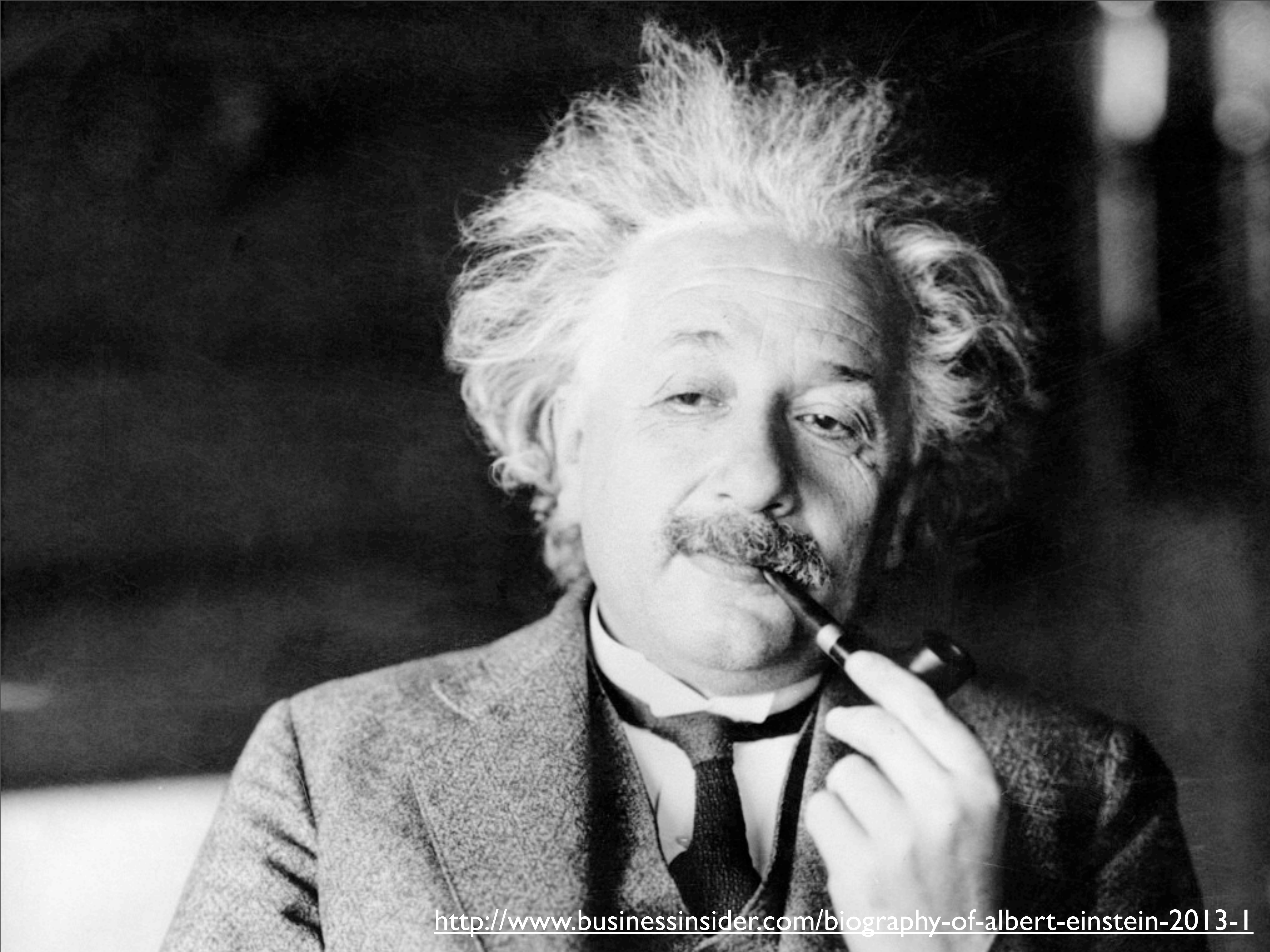
The counter’s value is simply the **total**

G-Counter



We have a distributed eventually-consistent increment-only counter

We have used **space** to become invariant to **time**


http://www.businessinsider.com/biography-of-albert-einstein-2013-1

Can we **abstract** this idea?

```scala
trait GCounter[Id,Elt] {
  def inc(id: Id, amt: Elt)
  def total: Elt
  def merge(c: GCounter[Id,
Elt]): GCounter[Id, Elt]
}
```

total requires **Elt** has **+**

inc requires **Elt** has **+,0**

+ must be
Invariant to order

Formally:

Associative

$$(x \bullet y) \bullet z = x \bullet (y \bullet z)$$

Formally:

Commutative

$$x \bullet y = y \bullet x$$

A Commutative Monoid!

o/*\o

High 5!

```scala
def inc(id: Id, amt: Elt)
(implicit m: Monoid[Elt])

def total(implicit m:
Monoid[Elt]): Elt
```

merge requires

Elt has **max**

max must be

Invariant to order

Formally:

Associative

$$(x \bullet y) \bullet z = x \bullet (y \bullet z)$$

Formally:

Commutative

$$x \bullet y = y \bullet x$$

max must

Converge to the correct value

Formally:

Idempotent

$$x \bullet x = x$$

An Idempotent Commutative Monoid!!!

```
def merge(c: GCounter[Id,
Elt])(implicit m: Monoid[Elt
@@ Max]): GCounter[Id, Elt]
```

Number (+)
Number (*)
Tuple
Map
Option
Average
Moving average
t-digest

Set (intersection)
Set (union)
Hyperloglog
Bloom filter
Count-min
Vector
Q-Tree
SGD

G-Set

Machine A	Machine B
A:{x}	A:{}
B:{}	B:{y,z}

G-Set Merge

Machine A	Machine B
A:{x}	A:{x}
B:{y,z}	B:{y,z}

G-Set Total

Machine A

A:{x}
B:{y,z}

Machine B

A:{x}
B:{y,z}

Set is
{x,y,z}

PN-Counter

A counter that can GROW and SHRINK

Can't use a G-Counter as we can't use max to merge

Use

Two

G-counters!

PN-Counter

Machine A

Additions

A: 4, B: 2

Subtractions

A: 5, B: 3

Machine B

Additions

A: 4, B: 7

Subtractions

A: 3, B: 4

Merge is simply the MAX

PN-Counter

Machine A

Additions

A: 4, B: 7

Subtractions

A: 5, B: 4

Machine B

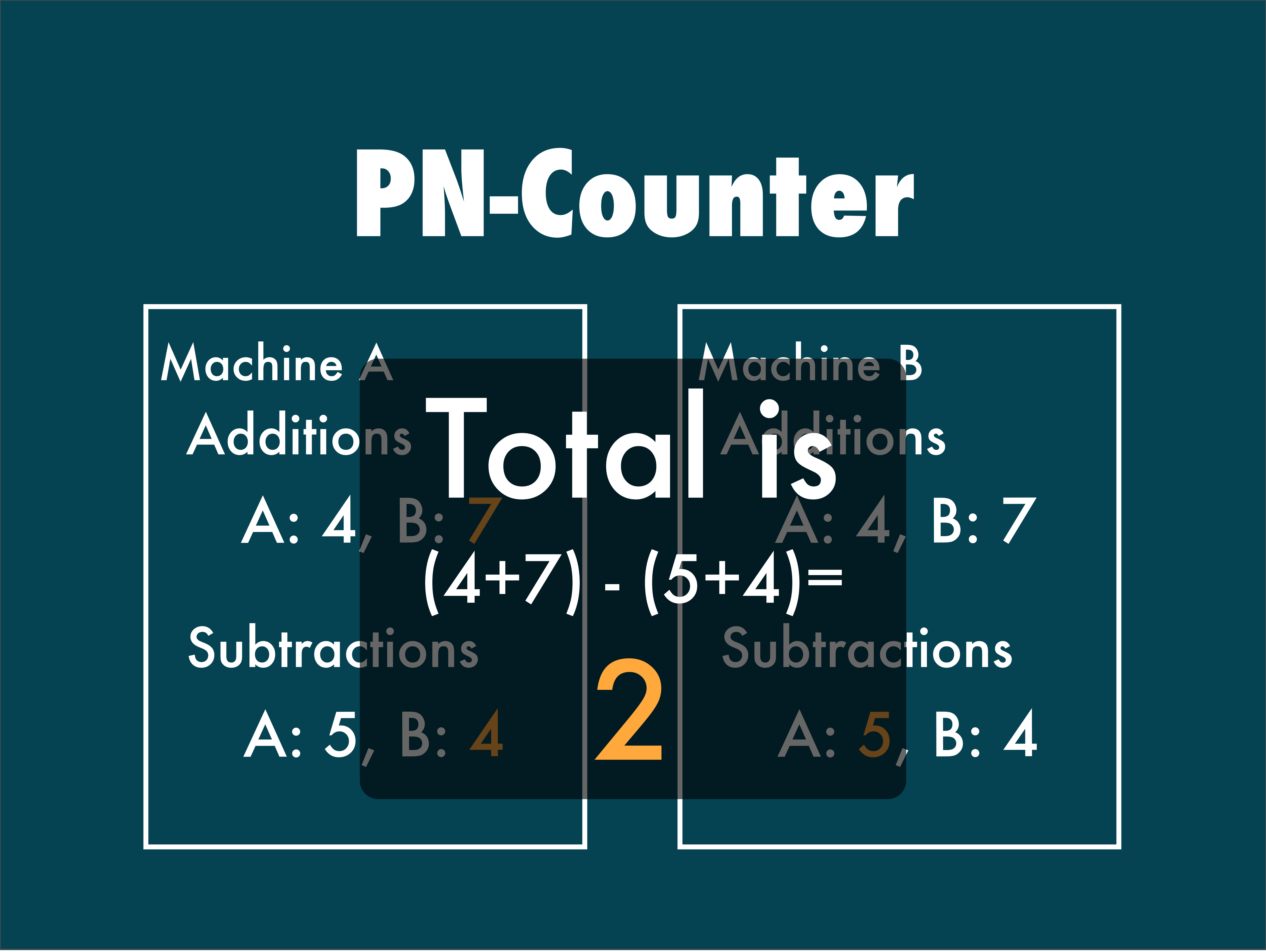
Additions

A: 4, B: 7

Subtractions

A: 5, B: 4

The counter's value is simply the TOTAL


PN-Counter
Machine A
Additions
A: 4, B: 7
Subtractions
A: 5, B: 4
Machine B
Additions
A: 4, B: 7
Subtractions
A: 5, B: 4
Total is
(4+7) - (5+4)=
2

```
trait PNCounter[Id,Elt] {
 def inc(id: Id, amt: Elt)
 def dec(id: Id, amt: Elt)
 def total: Elt
 def merge(c: GCounter[Id, Elt]): GCounter[Id, Elt]
}
```

PN-Counter requires Elt has addition, zero, and subtraction

A Commutative Group!

```
trait PNCounter[Id,Elt] {

  def inc(id: Id, amt: Elt)(implicit m: Monoid
[Elt])

  def dec(id: Id, amt: Elt)(implicit m: Monoid
[Elt])

  def total(implicit m: Group[Elt]): Elt

  def merge(c: GCounter[Id, Elt])(implicit m:
Monoid[Elt @@ Max]): GCounter[Id, Elt]

}
```

Numbers are clearly a commutative group

Sets with union and set difference are a commutative group

PN-Set

2P-Set

Machine A

Additions

A: {x}, B: {y}

Subtractions

A: {x}, B: {}

Machine B

Additions

A: {x}, B: {y, z}

Subtractions

A: {}, B: {y}

2P-Set Merge

Machine A

Additions

A: {x}, B: {y, z}

Subtractions

A: {x}, B: {y}

Machine B

Additions

A: {x}, B: {y, z}

Subtractions

A: {x}, B: {y}

2P-Set Total

Machine A

Additions

A: {x}, B: {y, z}

Subtractions

A: {x}, B: {y}

Machine B

Additions

A: {x}, B: {y, z}

Subtractions

A: {x}, B: {y}

Set is

{z}

Deleted elements stored indefinitely. Called **tombstones**

2P-Set allows elements to be added and removed **once**

C-Set

Store element and count

C-Set

Machine A

Additions
A: {(x, 2)},
B: {(y, 1)}

Subtractions
A: {(x, 1)},
B: {}

Machine B

Additions
A: {(x, 1)},
B: {(y, 1), (z, 2)}

Subtractions
A: {},
B: {(y, 1)}

C-Set Merge

Machine A	Machine B
Additions	**Additions**
A: {(x, 2)},	A: {(x, 2)},
B: {(y, 1), (z, 2)}	B: {(y, 1), (z, 2)}
Subtractions	**Subtractions**
A: {(x, 1)},	A: {(x, 1)},
B: {(y, 1)}	B: {(y, 1)}

C-Set Total

Machine A

Additions

A: {(x, 2)},

B: {(y, 1), (z, 2)}

Subtractions

A: {(x, 1)},

B: {(y, 1)}

Machine B

Additions

A: {(x, 2)},

B: {(y, 1), (z, 2)}

Subtractions

A: {(x, 1)},

B: {(y, 1)}

Set is

{x, z}

C-Set allows elements to be added and removed **many times**

C-Set allows elements to be removed **more times** than they have been added

OR-Set

Store element and unique token

OR-Set

Machine A

Additions

A: {(x, #a), (x, #d)},
B: {(y, #b)}

Subtractions

A: {(x, #a)},
B: {}

Machine B

Additions

A: {(x, #a)},
B: {(y, #b), (z, #c)}

Subtractions

A: {},
B: {(y, #b)}

OR-Set Merge

Machine A

Additions

A: {(x, #a), (x, #d)},
B: {(y, #b), (z, #c)}

Subtractions

A: {(x, #a)},
B: {(y, #b)}

Machine B

Additions

A: {(x, #a), (x, #d)},
B: {(y, #b), (z, #c)}

Subtractions

A: {(x, #a)},
B: {(y, #b)}

OR-Set Total

Machine A

Additions

A: {(x, #a), (x, #d)}

B: {(y, #b), (z, #c)}

Subtractions

A: {(x, #a)},

B: {(y, #b)}

Machine B

Additions

A: {(x, #a), (x, #d)},

B: {(y, #b), (z, #c)}

Subtractions

A: {(x, #a)},

B: {(y, #b)}

Set is

{x, z}

OR-Set **works** the way we expect

From sets, build trees, graphs, etc.

CRDTS vs The Real World

Strong Consistency
Memory Usage
Code

Strong Consistency

Don't build your billing platform on CRDTs

Memory Usage

Tombstones

Machine IDs

Tombstones: Establish **causal order** and delete (Bieniusa et al. 2012)

Tombstones: Prune with **heuristics** (often based on time)

Machine IDs: OR-Sets **don't** need them

Machine IDs: Hierarchical organisation allows pruning (Almeida & Baquero. 2013)

Code

Riak 2.0

Various open source libraries


Thank you!
Now go forth and
DISTRIBUTE!
http://stjost.deviantart.com/art/Stomping-Off-Into-the-Sunset-277086274

More:

noelwelsh.com