Embracing Concurrency at scale

(it's about time!)

Justin Sheehy
justin@basho.com

Concurrency Matters

- Herb Sutter, 2005

Concurrency Matters

- Herb Sutter, 2005

New Problems, Old Solutions

Distributed Systems matter now more than ever,
and we must learn from the past to build the future.

Don't do what I say. (yet)

Working at scale isn't just "more." It is different.

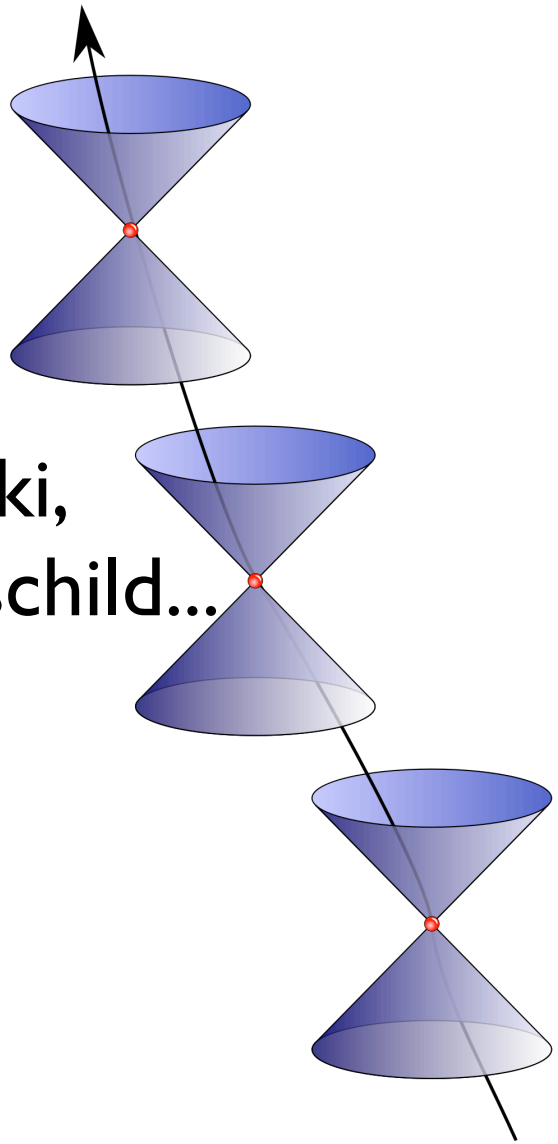
What is Concurrency?

concurrent: occurring at the same time

concurring: agreeing with others

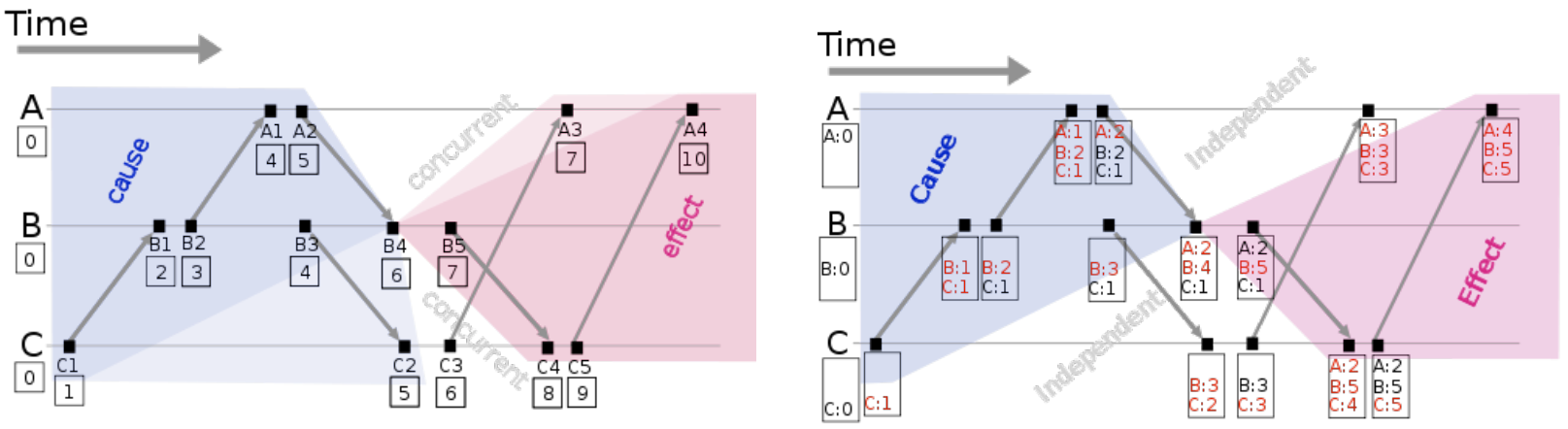
Time is a Hard Problem

Einstein,
Minkowski,
Schwarzschild...



Time in Computing

Lamport, 1978 -- gave us “happened before”



Mattern, 1989 -- closer to Minkowski causality

Time is a Hard Problem

In computing, we like to pretend it's easy.

This is a trap!

Distributed Computing is Asynchronous Computing

Synchrony (distributed transactions) throws away the biggest gains of being distributed!

Three Kinds of Computing

- Pat Helland

memories: **at time `T`, I learned fact `F`**

guesses: **based on my memories, I will try `G`**

apologies: **`G` didn't work out, oops**

There is no “Global State”

You only know about the past -- deal with it!

This sadly often means giving up on ACID.
(globally, not locally)

This is going to hurt!

Atomicity
Consistency
Isolation
Durability

~~Atomicity~~

Consistency

~~Isolation~~

Durability

Basically

Available

~~Atomicity~~

Consistency

~~Isolation~~

~~Durability~~

Basically

Available

Soft State

~~Atomicity~~
~~Consistency~~
~~Isolation~~
~~Durability~~

Basically
Available
Soft State
Eventually-Consistent

This is a real tradeoff -- if you make it, understand it!

(Eric Brewer, 1997)

Basically
Available
Soft State
Eventually-Consistent

CAP tradeoffs

Consistency
Availability
Partition-Tolerance

You want all three, but
you can't have them all at once.

CAP tradeoffs

Consistency
Availability
Partition-Tolerance

Distributed Transactions
(on any real network, this fails)

CAP tradeoffs

Consistency
Availability
Partition-Tolerance

Quorum Protocols &
typical Distributed Databases
(nodes outside the quorum fail)

CAP tradeoffs

Consistency
Availability
Partition-Tolerance

Sometimes allow stale data...
...but everything can keep going.

CAP tradeoffs

Consistency
Availability
Partition-Tolerance

This is where BASE leads us.

This is a real tradeoff -- if you make it, understand it!

Basically
Available
Soft State
Eventually-Consistent

Eventually-Consistent doesn't mean "not consistent"!

It just forces you to remember that
everything is probabilistic.

It also doesn't mean slow.
BASE and DIRT are not in conflict!
Sometimes you go "eventual" in order to go fast.

RPC is a scaling antipattern.

Treating remote communication like local function calls is a fundamentally bad abstraction.

- Network can fail after call "succeeds".
- Data copying cost can be hard to predict.
- Tricks you by working locally.
  (and then failing in a real dist sys)
- Prevents awareness of swimlanes.
  (and thus causes cascading failure)

Protocols vs. APIs

- Explicit understanding of boundaries.
  (trust boundaries, failure boundaries...)

- Better re-use and composition.
  (unintuitive but true in the large)

- Asynchronous reality, described accurately.
  (see Clojure or Erlang/OTP libraries)

Successful Protocols

Kings of the Internet: DNS & HTTP

What do they have in common?

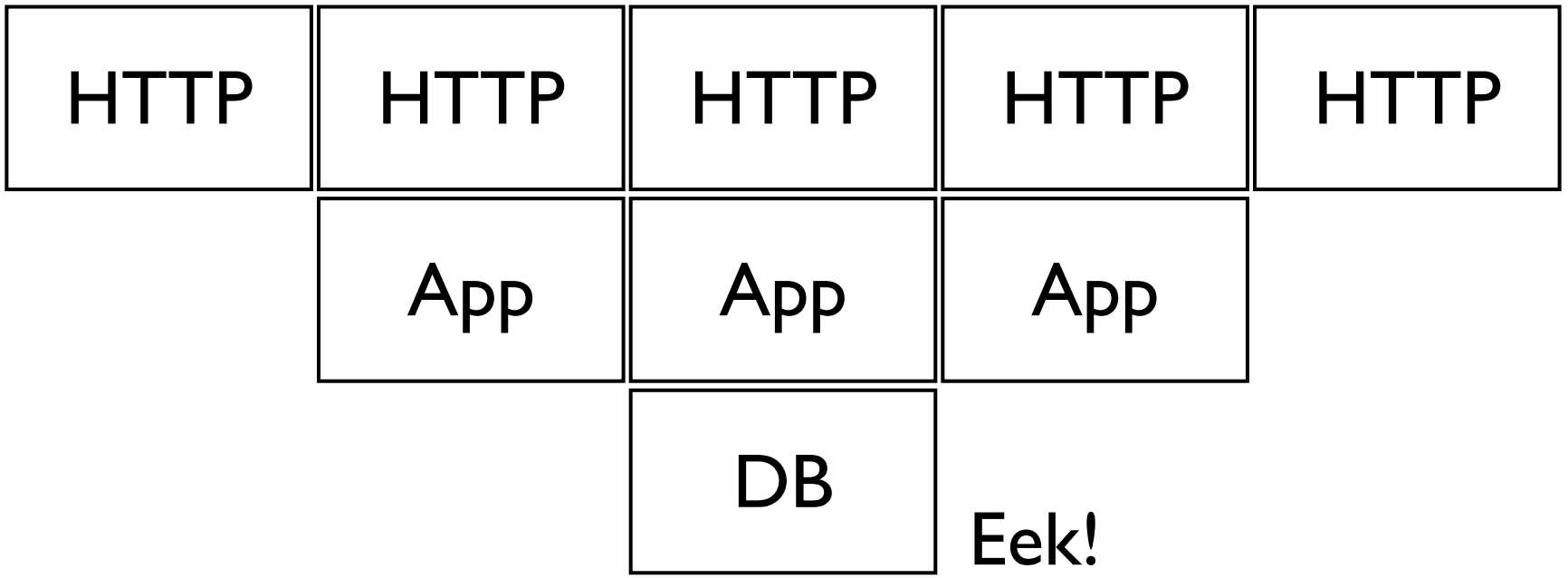
B
A
S
E

The Web

(the second most successful distributed system ever)

- no global state (closest: DNS root & MIME)
- well-defined caching for eventual consistency
  - idempotent operations!
- loose coupling
  - links instead of global relations
  - no must-understands except HTTP

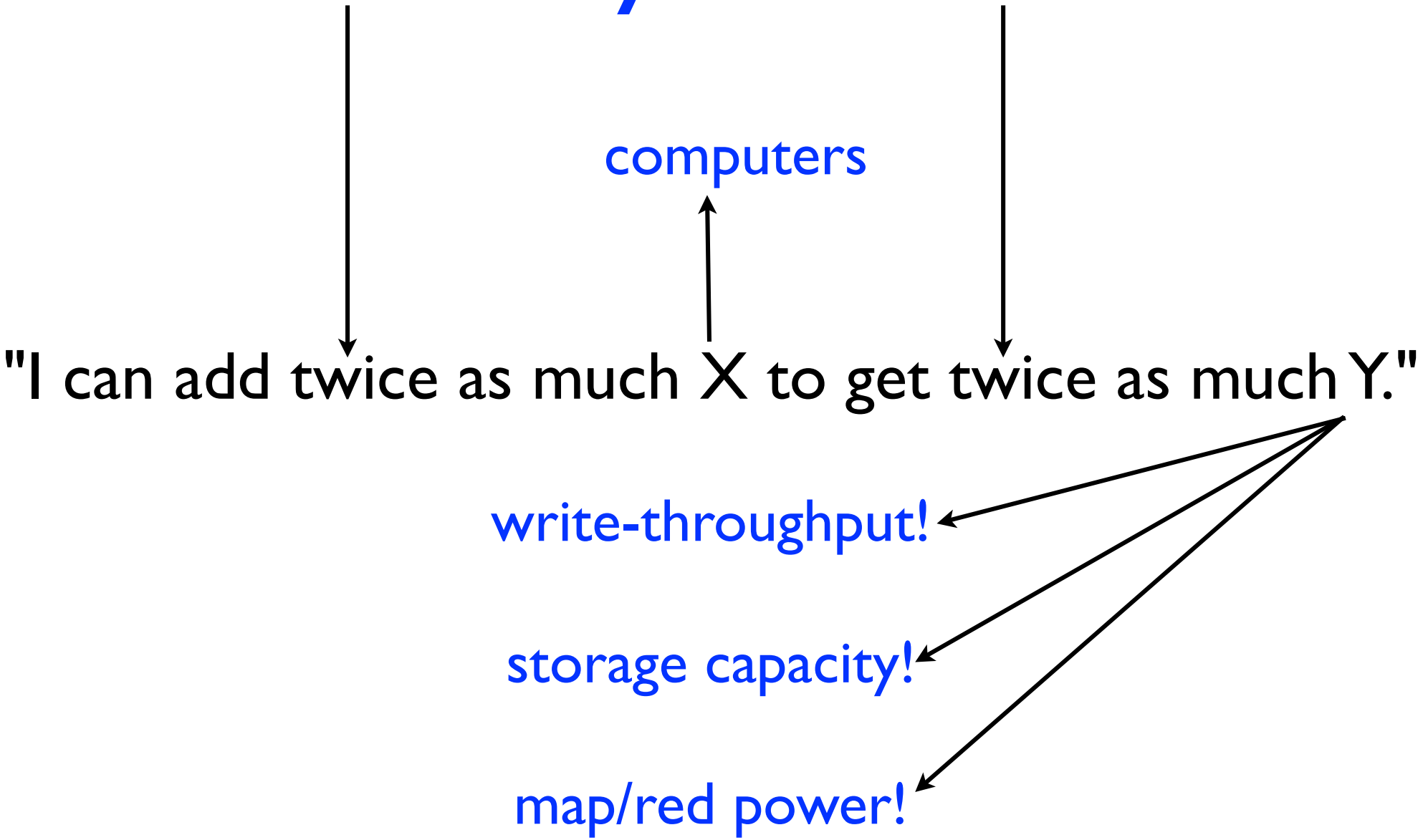
History of Scaling The Web



Eek!

Help from "NoSQL"?

Linearly Scalable



Measurement

Today's networked world is full of
cascading implicit and explicit SLAs

Reason about your behavior,
but also measure it in production.

Measurement

In dist. sys. if you don't measure everything, then you'll pick the wrong bottlenecks.

Measure your systems top to bottom, and correlate information cross-system.

Resilient

Assume that failures will happen.

At scale, they are ALWAYS happening.

Designing whole systems and components
with individual failures in mind
is a plan for predictable success.

Know How You Degrade

Plan it and understand it before your users do.

You might prevent whole system failure if you're lucky and good, but what happens during partial failure?

Know How You Degrade

Plan it and understand it before your users do.

You think you know
which parts will break.

Know How You Degrade

Plan it and understand it before your users do.

You think you know
which parts will break.

You are wrong.

Harvest and Yield

harvest: a fraction

`data available / complete data`

yield: a probability

`queries completed / q's requested`

in tension with each other:
(harvest * yield) ~ constant

goal: failures cause known linear reduction to one of these

Harvest and Yield

traditional ACID demands 100% **harvest**
but success of modern applications is
often measured in **yield**

plan ahead, know when you care!

Sometimes, you will fail.

Plan it and understand it before your users do.

If you think you can prevent failure, then you aren't developing your ability to respond.

- Paul Hammond

Being able to recover quickly from failure is more important than having failures less often.

- John Allspaw

Sometimes, you will fail.

Plan it and understand it before your users do.

Applications built for scale can make recovery either easier or harder. You get to choose.

two things to make it easier:

minimal, async interfaces when possible

locality of computation and reasoning

Embracing Concurrency at scale

(it's about time!)

Justin Sheehy
justin@basho.com