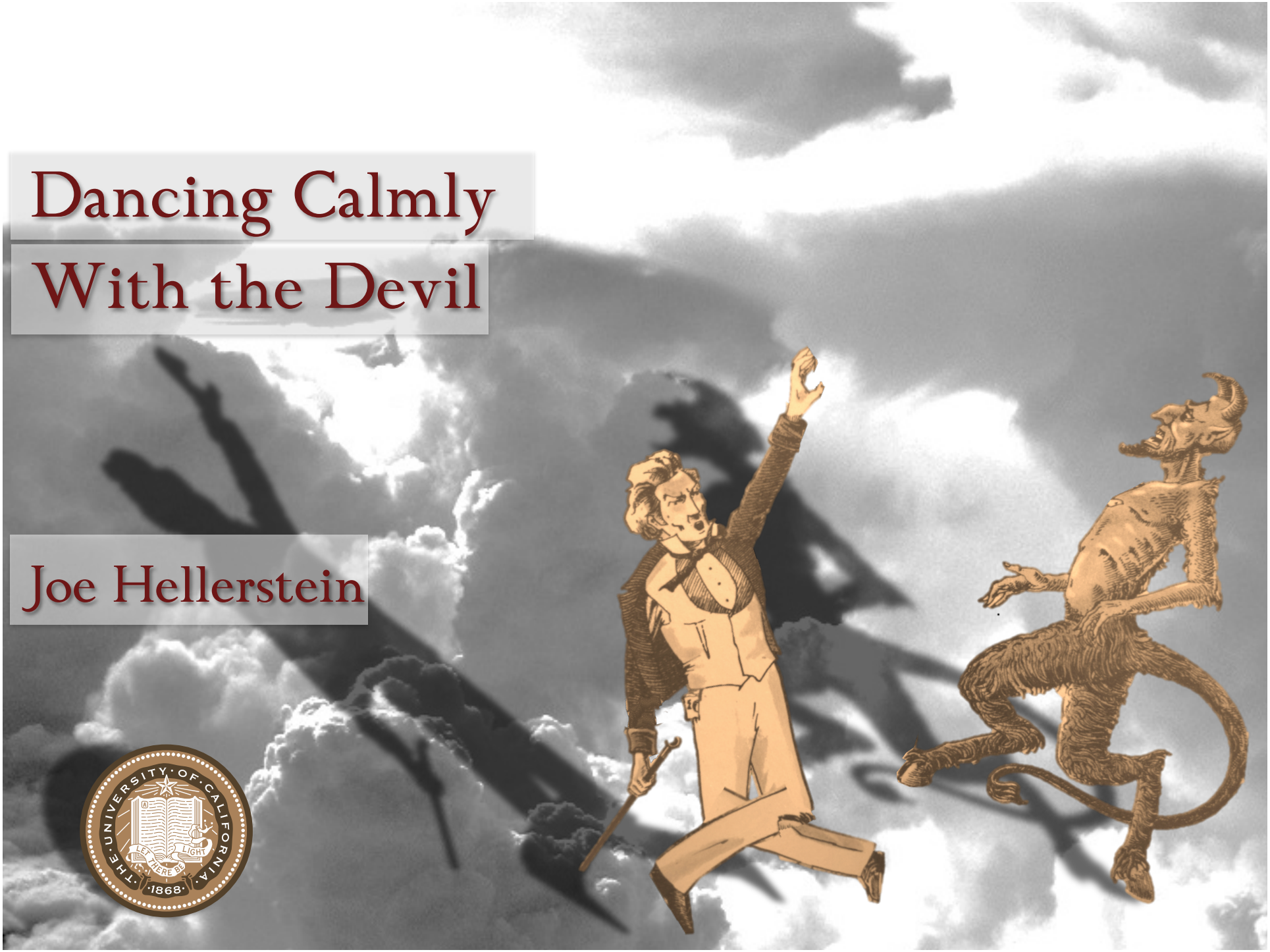
Dancing Calmly With the Devil

Joe Hellerstein



BOOM Team

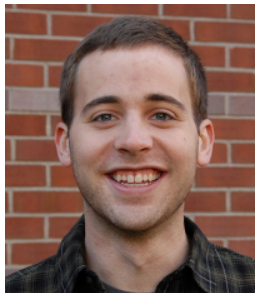
joe hellerstein

david maier

ras bodik

alan fekete

peter alvaro



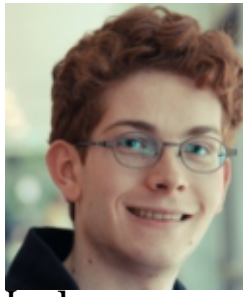
peter bailis

neil conway

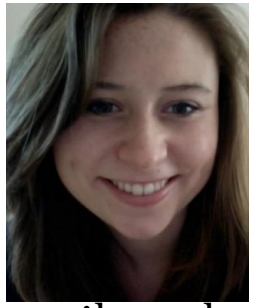
bill marczak

haryadi gunawi

sriram srinivasan



Joshua rosen



emily andrews

andy hutchinsor

I Can Give You Power

All the Compute you desire

All the Storage you desire

All the Data you desire

At What Cost?

The loss of *illusions*

- Sequential computing
- Single-copy state
- Reliable components

Dancing with the Devil

- **Coordination-Free Distributed Computing**
  - Write sequential code for each processor
  - Communicate without waiting
  - Full-bandwidth computation
- Beware the risks:
  - Non-determinism

Dancing with the Devil

- **Coordination-Free Distributed Computing**
  - Write sequential code for each processor
  - Communicate without waiting
  - Full-bandwidth computation
- Beware the risks:
  - Non-determinism
  - Split brain

Paying the Devil His Due

- **Coordination**: the last expensive thing
  - But maybe it's wisest to pay?

Get Away, Satan!

- **Coordination**: the last expensive thing

"The first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them"

—James Hamilton (IBM, MS, Amazon)

[Birman, Chockler: "Toward a Cloud Computing Research Agenda", LADIS09]

Are you blithely asserting that transactions aren't webscale?

Some people just want to see the world burn.
Those same people want to see the world use inconsistent databases.

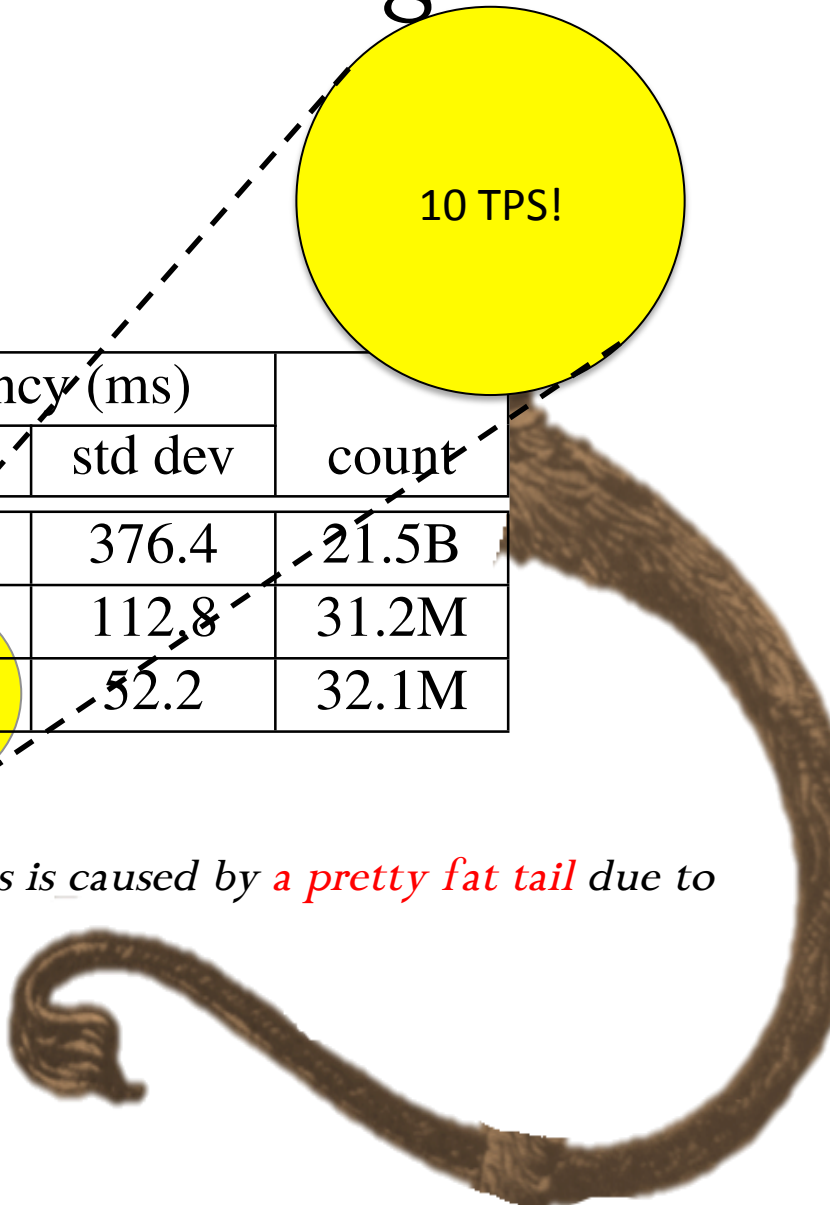
- Emin Gün Sirer

Paying the Devil at Google

- Spanner latency costs

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

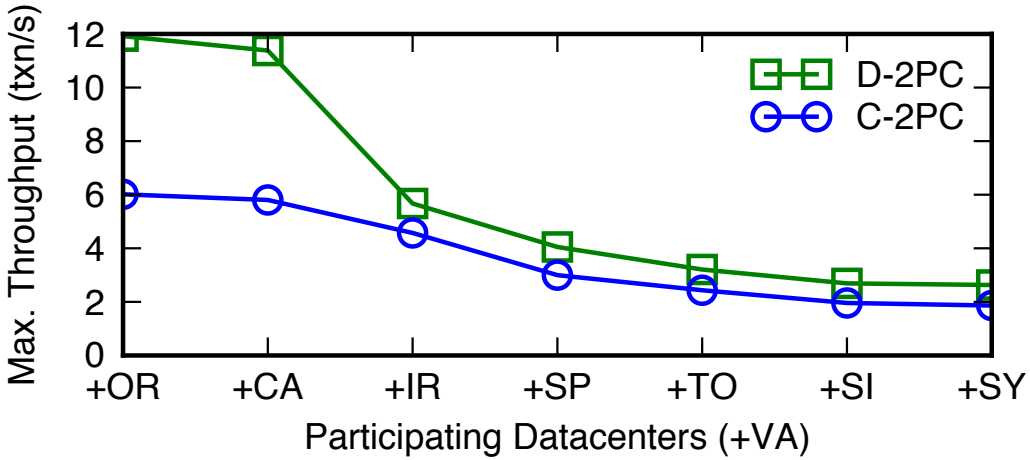
10 TPS!



"The large standard deviation in write latencies is caused by a pretty fat tail due to lock conflicts.

[Corbett, et al. "Spanner:...", OSDI12]

Distributed Throughput Costs



Curse you, speed of light!
Only 7 global round-trips per sec

[Bailis et al., "Coordination Avoidance...", VLDB 2015]

The Big Question: Dance or Pay?

- That is:
  - Run without coordination, and risk inconsistency?
  - Or pay for coordination?
- More subtly: when to coordinate?
  - A case-by-case decision?
  - Can uncoordinated stuff taint your coordinated stuff?

Takeaway… and Foreshadowing

- Coordination is the last expensive thing in computing

- ***When*** can we avoid coordination without inconsistency?
  - CALM Theorem answers this question

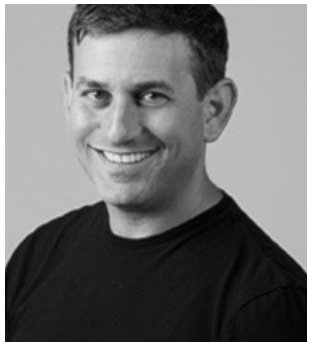
- ***How*** can we avoid coordination?
  - Not via Read/Write consistency games
  - At application-level—preferably with language support

Outline

- Cloud: A Deal with the Devil
- **Bottom-Up and Top-Down systems**
- Creativity from the bottom
- Good news from the top: CALM
- Grounding CALM: Bloom and Blazes
- Lessons and Challenges

CS262 @ Berkeley

- Joint OS/DB intro grad course, 1999 and on
  - Brewer + Hellerstein
  - An early sense of convergence: data-driven services
- Initial lectures
  - UNIX: Bottom-up system elegance
  - System R: Top-down semantic guarantees
- Good system designers fluidly transit worldviews



A Bottom-Up Hazard

- Starting from the wrong bottom...

The Von Neumann Model

Focus on *Mutable State*

Primacy of Ordering

- LIST of Instructions
- ARRAY of Memory
- MUTATION in time (R/W)

The Von Neumann Model

Focus on *Mutable State*

Primacy of Ordering

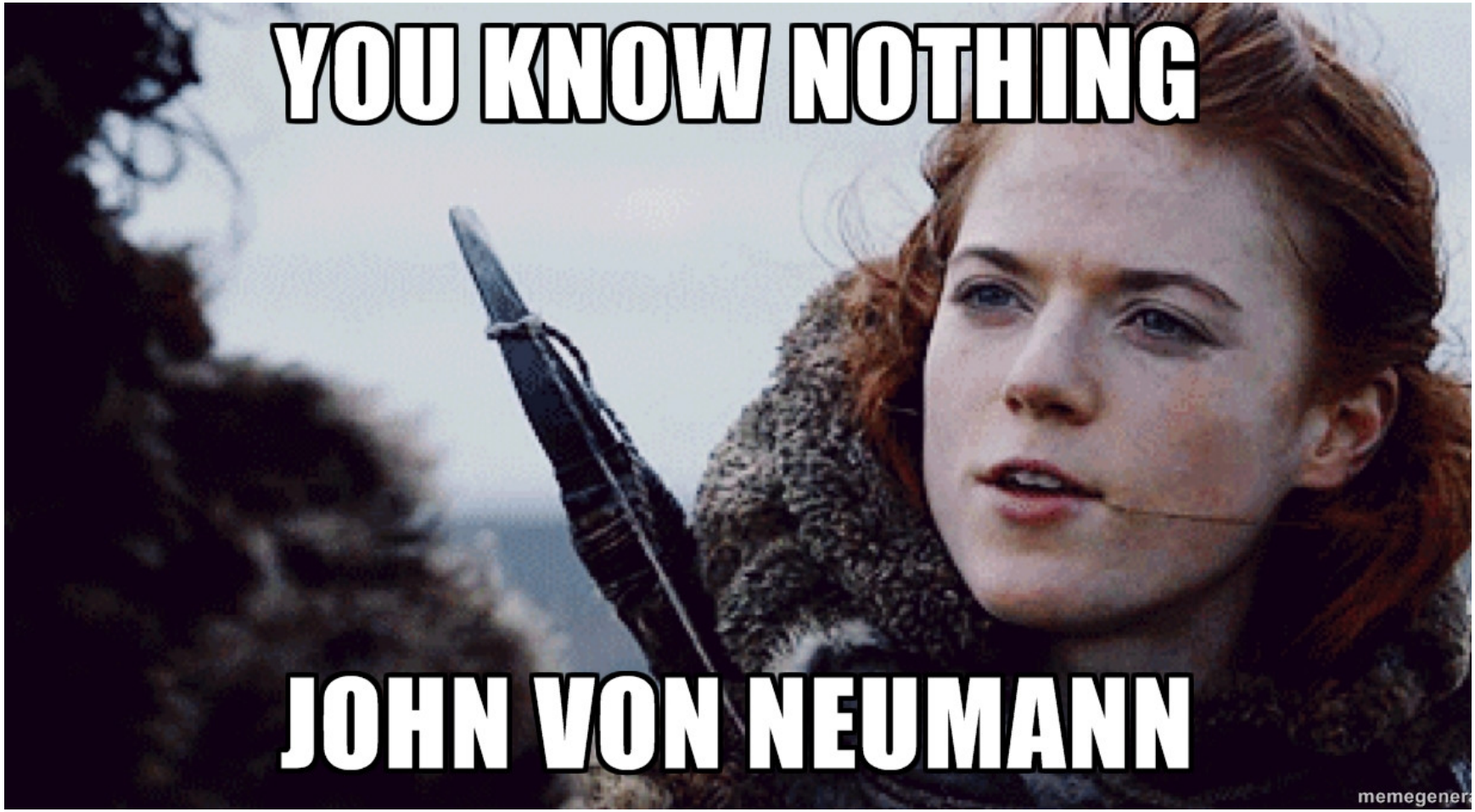
- LIST of Instructions
- ARRAY of Memory
- MUTATION in time (R/W)
- *Remember our lost illusions?*

The Von Neumann Model

Focus on *Mutable State*

Primacy of Ordering

- LIST of Instructions
- ARRAY of Memory
- MUTATION in time (R/W)
- *Remember our lost illusions?*
  - *Sequential computing*
  - *Single-copy state*
  - *Reliable components*


YOU KNOW NOTHING
JOHN VON NEUMANN
memegener

Common Modern Responses

- Bottom-Up
  - Define specific consistency guarantees for R/W interface
    - Causal, weak isolated xactions, session guarantees…
- Top-Down
  - Build consistent apps despite inconsistent storage
    - Dynamo shopping cart
- Know-Nothing
  - Consistency? Why worry?*

- Much dispute, esp. in NoSQL. Each is (often) right.

*[Bailis, et al., "Probabilistically Bounded Staleness…", VLDB12]

Last Week on Twitter

Last Week on Twitter

Puppy Agenda @aphyr · 8h

@damienkatz I'm starting to suspect that not only do you not *have* a consensus algorithm; you don't even know what the problem *means*.

1

damien mutant Qatz @damienkatz · 8h

@aphyr Sigh. WHAT PROBLEM ARE YOU TRYING TO SOLVE? Two systems need to agree upon something. What is that thing?

Dancing on the Wrong Bottom

- Actually, top-down can be made to work
  - Consistent apps on inconsistent storage
  - Much to be learned here from developer patterns!

- But the tools are a poor fit for the patterns
  - Sequential languages
  - Debuggers for ordered R/W of state
  - Test harnesses that can't cover the space

- End results that are hard to test, hard to trust

Takeaway … and Foreshadowing

- Von Neumann model underlies all our bottom-up thinking
  - And it's a terrible match to the cloud

- What lessons can we learn from today's successful developers?

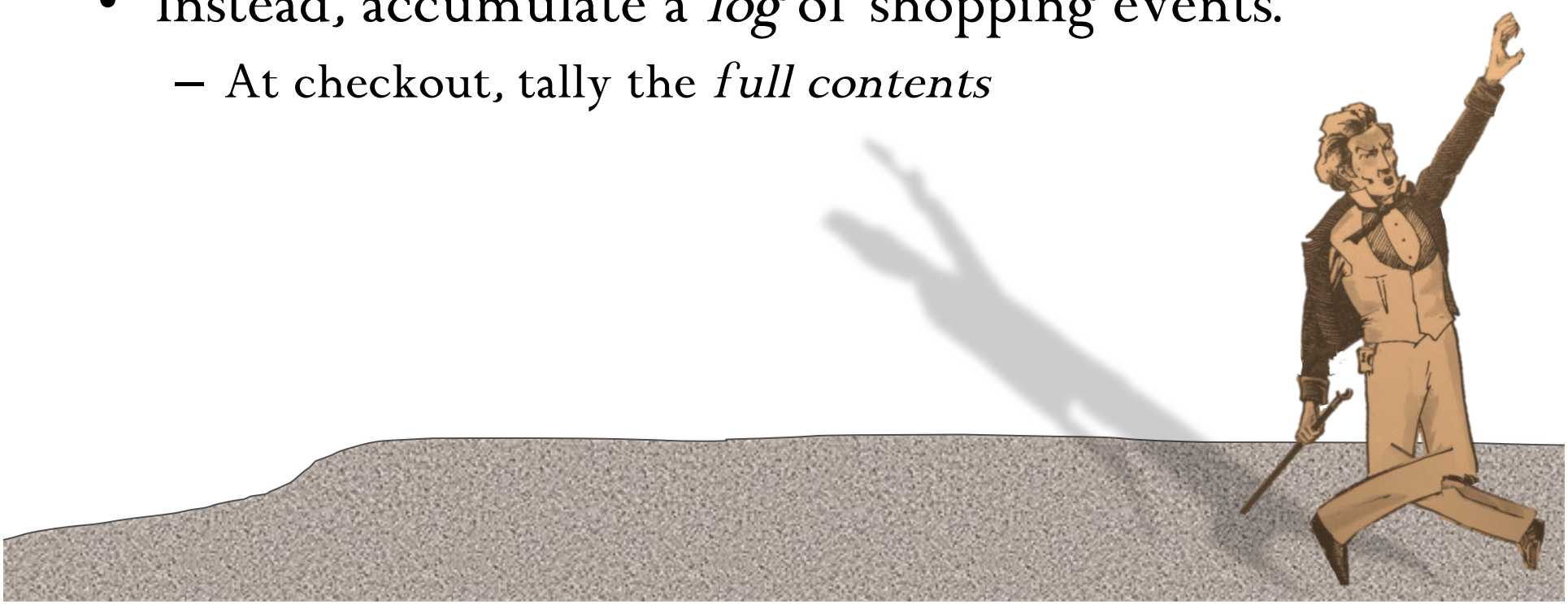
Outline

- Cloud: A Deal with the Devil
- Bottom-Up and Top-Down systems
- **Creativity from the bottom**
- Good news from the top: CALM
- Grounding CALM: Bloom and Blazes
- Lessons and Challenges

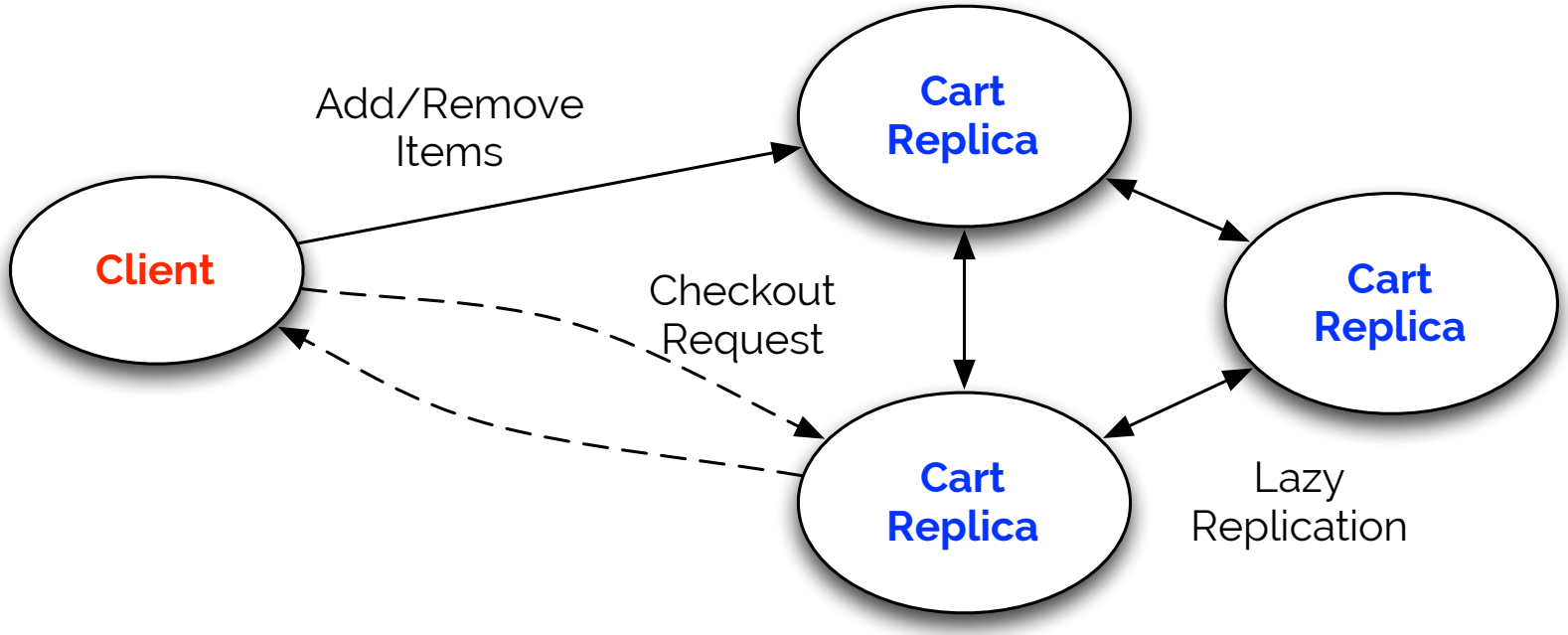
Dynamo: Building on Quicksand

[DeCandia, et al. 2007]
[Campbell and Helland, 2009]

- The roots of NoSQL
- Write a shopping cart on a mutable key/value store?
  - You'll need to coordinate R/W!
- Instead, accumulate a *log* of shopping events.
  - At checkout, tally the *full contents*



The Dynamo Shopping Cart



The KVS Cart

- Built on a replicated key-value store (KVS)
  - put(item, count-so-far)
  - get(item, count-so-far)

Key	Value
	2
	1

Key	Value
	2
	1

The Coordinated KVS Cart

- Build on a replicated KVS
- With a round of Paxos or 2PC per write

Key	Value

Key	Value

Key	Value

Key	Value

Key	Value
	1

Key	Value
	1

Key	Value
	1

Key	Value
	1

Key	Value
	1
	1

Key	Value
	1
	1

Key	Value
	1
	1

Key	Value
	1
	1

Key	Value
	2
	1

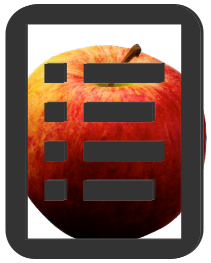
Key	Value
	2
	1

The Disorderly Log Cart

- Using an no-overwrite event log per session
  - append(cart, action)


Proceed to
Checkout


60
5
10
15
20
25
30
35
40
45
50
55
SHOCKRESISTANT



✔ ✔ A “seal” or “manifest”

Takeaways … and Foreshadowing

- Learning from Developers
- Anti-Pattern: R/W mutable shared state
- Pattern: "ACID 2.0"
  - Associative, Commutative, Idempotent, Distributed
  - See also CRDTs
  - See also Event Log Exchange

- Questions:
  1. Can I *always* write code that follows the pattern?
  2. Will I sometimes need to coordinate? When and How?

Patterns —> Theorems —> Software

Again I Ask: Dance or Pay?

A Theory Question! (Patterns —> Theorems)

- Why coordinate? When can I avoid it?
- The CALM theorem

Note well:

- These are *not* questions about reads, writes, and races!
  - Maybe a better programmer can avoid the contention!
  - *Must* think top-down here!

- These are expressivity/complexity questions
  - What can be computed without a coordination construct?

Again I Ask: Dance or Pay?

A Practical Question! (Theorems —> Software)

- Languages/libraries that encourage coordination-freeness
  - E.g. Bloom
- Program analysis that detects the need for coordination
  - E.g. Blazes

Outline

- Cloud: A Deal with the Devil
- Bottom-Up and Top-Down systems
- Creativity from the bottom
- **Good news from the top: CALM**
- Grounding CALM: Bloom and Blazes
- Lessons and Challenges

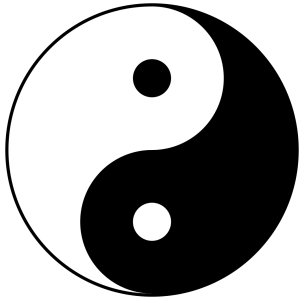
Keep CALM

As it turns out, a *data centric* view helps a lot

- But *not* from the transactions literature
  - The limitations of R/W thinking
- Better: dataflow, queries, data lineage!

There are positive results to be had!

The CALM Theorem



Monotonic => Consistent

- Dance monotonically with the Devil
- Consistent w/o coordination!

¬Monotonic => ¬Consistent

- To achieve consistency, you must use coordination
- "Seal" input to non-monotonic operations.

[Hellerstein: PODS '09 keynote, "The Declarative Imperative"]

Also:

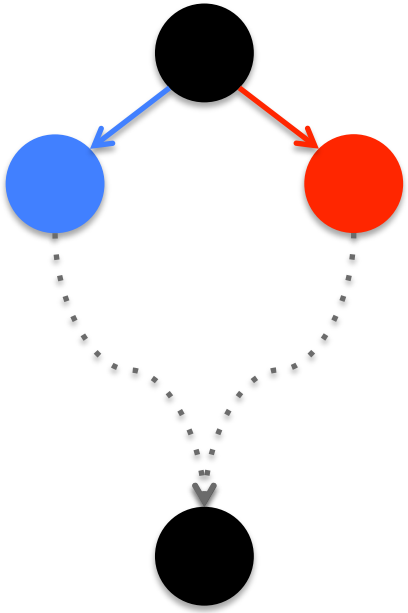
- CRON Conjecture
- Coordination Complexity

Much Depends on Definitions

- Consistency
- Monotonicity
- Coordination

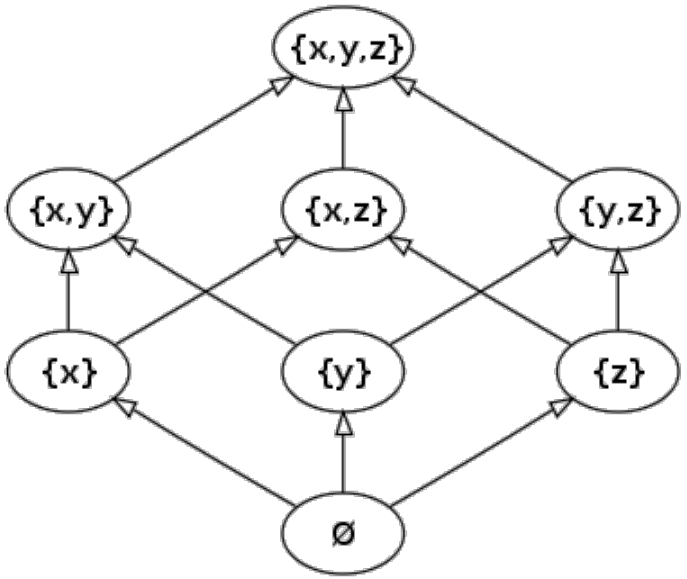
Consistency: Confluence

- Non-Determinism (of Message Ordering)
- Yet deterministic outcomes
  - Upon eventual receipt of same *set* of messages
  - Deterministic outcomes ("state" and "computation")



CALM Intuition: Logic & Sets

- Monotonic logic
  - Sets with accumulation
  - Select/Project/Join
  - *Streaming execution*



- Non-Monotonic logic
  - Negation (Not Exists)
  - Deletion/Mutation
  - Set Difference
  - *No streaming execution. Requires "sealing" a set.*

Intuition from the Integers

VON NEUMANN

```
int ctr;

operator:= (x) {
   // assign
   ctr = x;
}
```

ACID 2.0

```
int ctr;

operator<= (x) {
   // merge
   ctr = MAX(ctr, x);
}
```

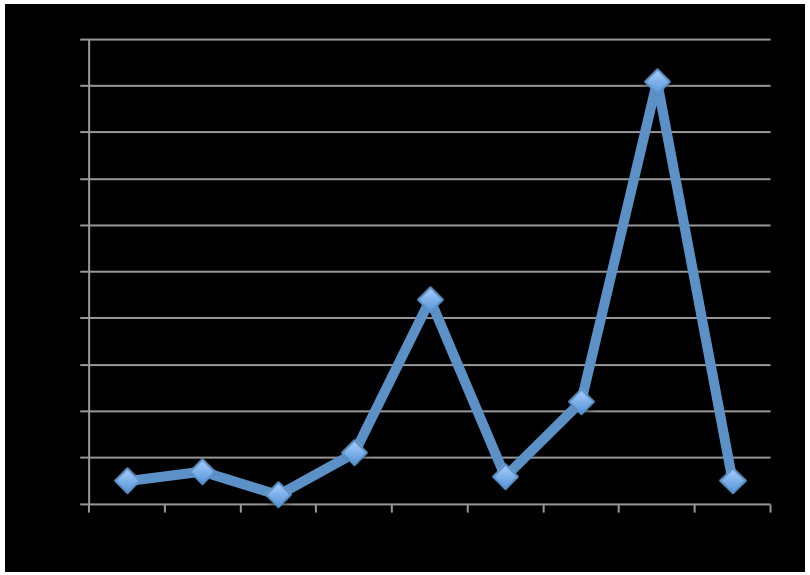
DISORDERLY INPUT STREAMS:

2, 5, 6, 7, 11, 22, 44, 91

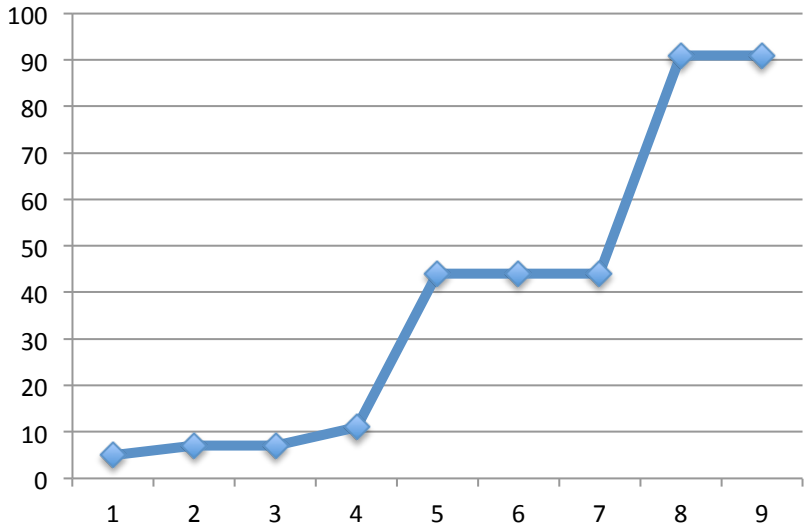
5, 7, 2, 11, 44, 6, 22, 91, 5

Intuition from the Integers

VON NEUMANN



ACID 2.0



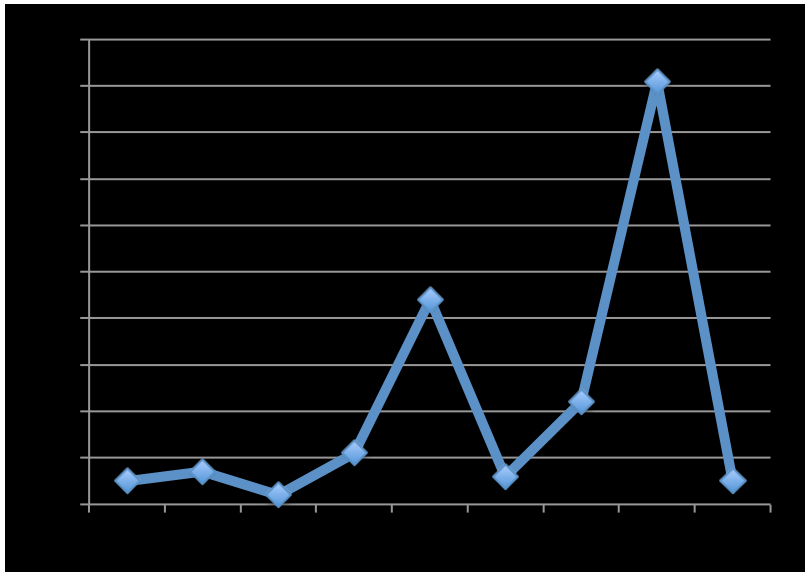
DISORDERLY INPUT STREAMS:

2, 5, 6, 7, 11, 22, 44, 91

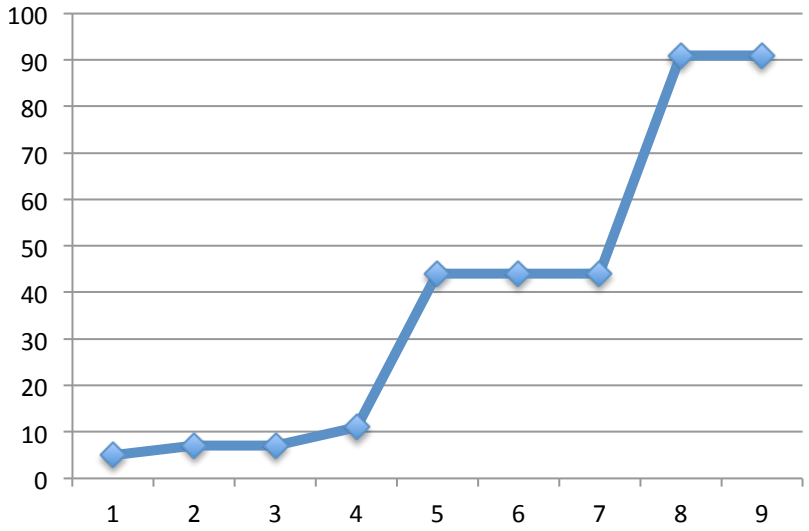
5, 7, 2, 11, 44, 6, 22, 91, 5

Intuition: Storing an Integer

VON NEUMANN



ACID 2.0



DISORDERLY INPUT STREAMS:

2, 5, 6, 7, 11, 22, 44, 91

5, 7, 2, 11, 44, 6, 22, 91, 5

+ monotonic "progress"
+ order-insensitive outcome

So Much for Monotonicity

- What's the problem with non-monotonicity?

Sealing, Time, Space, Coordination

- Non-monotonicity requires *sealing* things

  $\neg \exists$ *item* $\in$ *Cart* (*fragile*(*item*))

  $\Leftrightarrow$ $\forall$ *item* $\in$ *Cart* ($\neg$*fragile* (*item*))

- Time: a mechanism to seal fate.
  - Before and after

"Time is what keeps everything from happening at once."
— Ray Cummings

Sealing, Time, Space, Coordination

- Non-monotonicity requires sealing things

  $\neg \exists\, item \in Cart\ (\ fragile(item)\ )$

  $\Leftrightarrow \quad \forall\, item \in Cart\ (\ fragile(item)\ )$

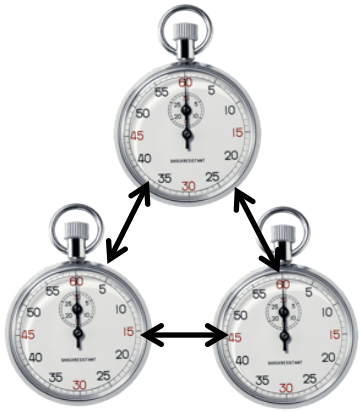
- Time: a mechanism to seal fate.
  - Before and after
- Space: multiple perceptions of time

Sealing, Time, Space, Coordination

- Non-monotonicity requires sealing things

  $\neg \exists\, item \in Cart\ (\ fragile(item)\ )$

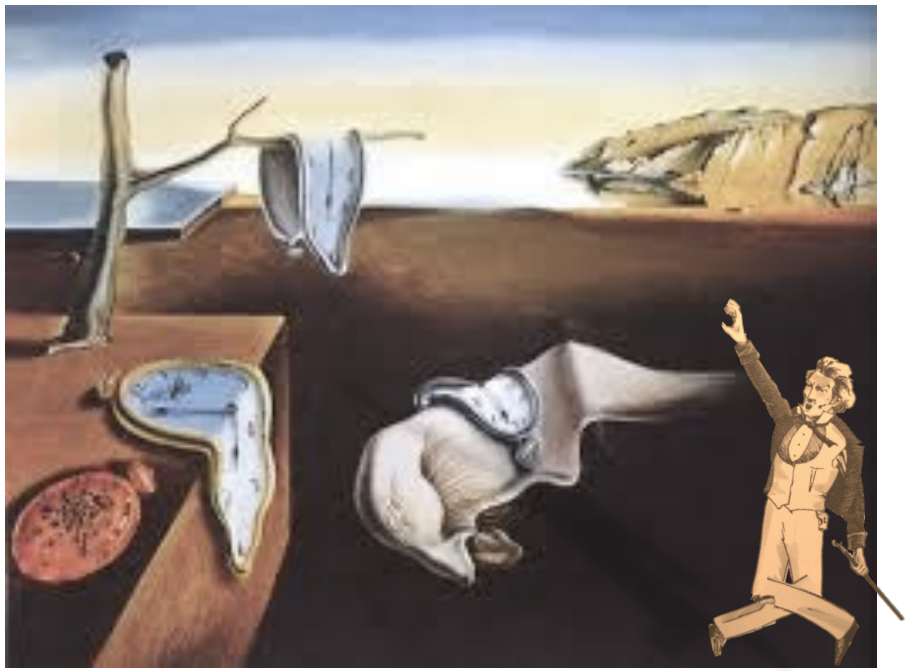
  $\Leftrightarrow \quad \forall\, item \in Cart\ (\ fragile(item)\ )$



- Time: a mechanism to seal fate.
  - Before and after
- Space: multiple perceptions of time
- Coordination: sealing across space/time.
  - Global Consensus on the "final" contents of a piece of state
  - 2-Phase Commit & Paxos are the classic protocols

Is Monotonicity Restrictive?

- Actually, it's all of PTIME!
- Maybe time doesn't matter so much
  - Remember: Time is the thing that prevents everything from happening all at once.
    - Anti-parallelism!
  - Avoid it



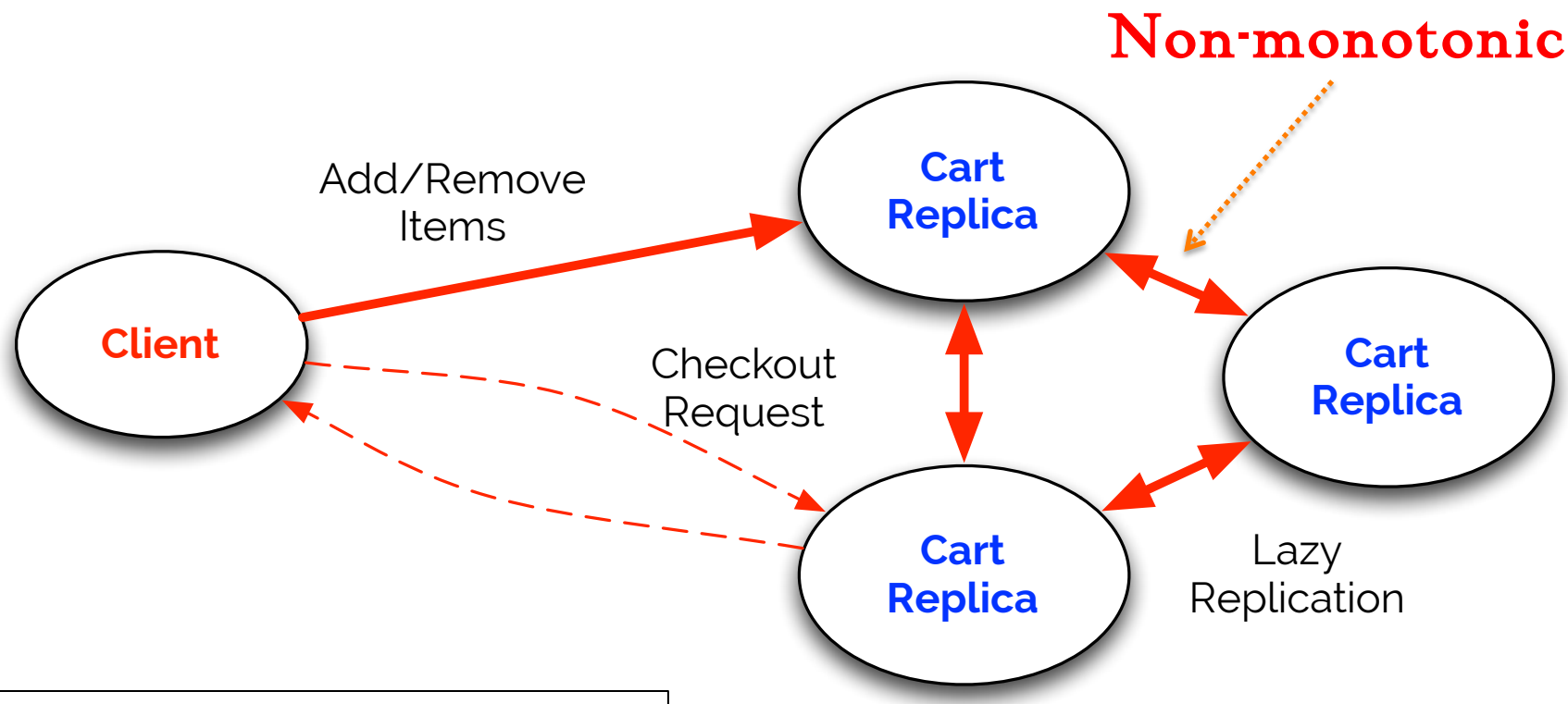
Theoretical Results

- CALM Proofs
  - Abiteboul, et al.: M=>C [PODS '11]
  - Ameloot, et al.: CALM [PODS '11, JACM '13]
  - Marczak, et al.: Model-Theory treatment [Datalog 2.0 '12]
  - Ameloot, et al.: More permissive M [PODS '14 best paper]
- CRON (Proofs & Refutations)
  - Ameloot, et al.: [JCSS '15]
- Coordination Complexity: MP Model
  - Koutris & Suciu (min-coordination & LB): [PODS '11]
  - Beame et al. (minimizing replication): [PODS '13]
- More! See survey by Ameloot [SIGMOD Record 6/14]

Thinking CALMly

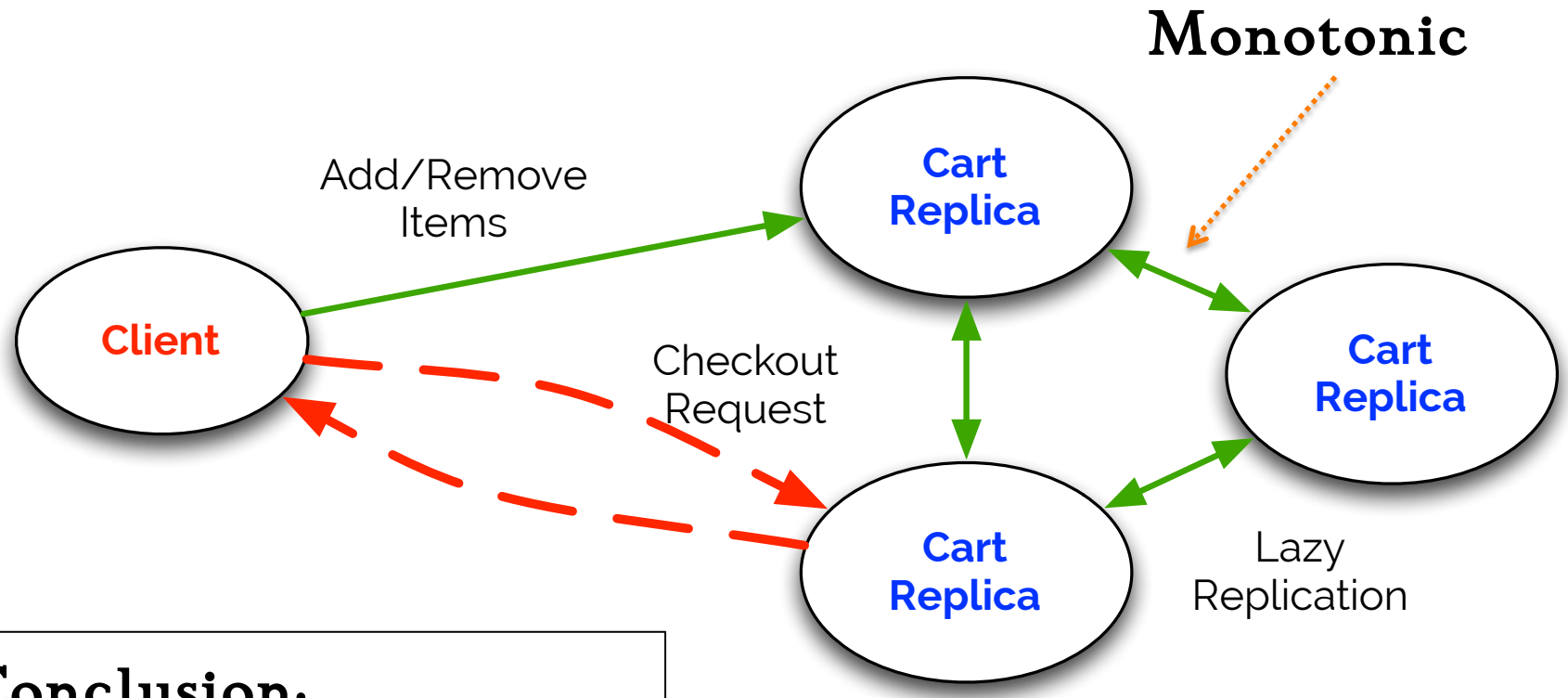
- Using CALM as a guide to analyze designs...

CALM Analysis: KVS Cart



Conclusion:
Every operation might require coordination!

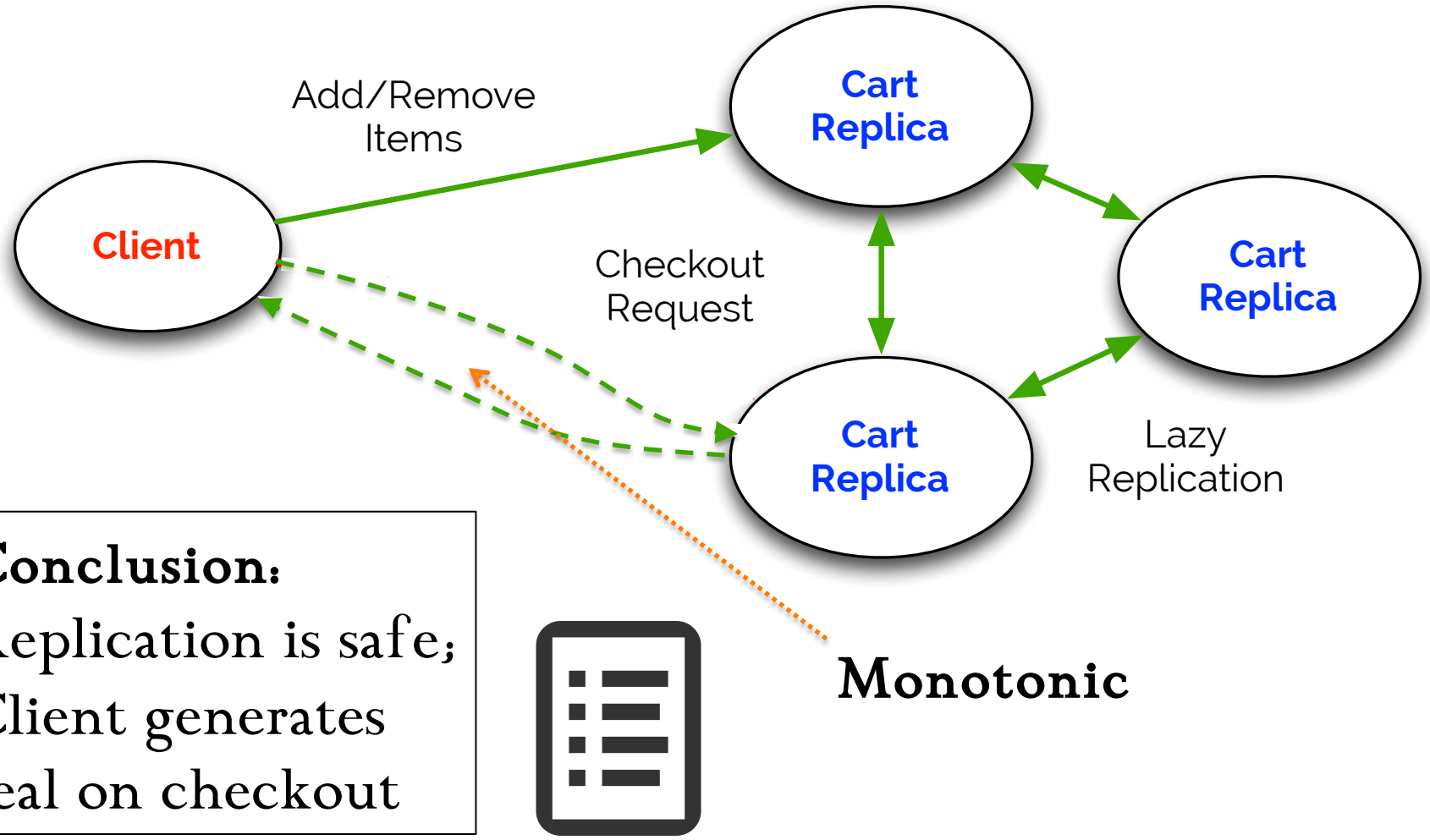
CALM Analysis: Disorderly Log Cart



Conclusion:
Replication is safe;
coordinate on checkout

Proceed to Checkout

CALM Analysis: Disorderly Logs with Seals



Conclusion:
Replication is safe;
Client generates seal on checkout

Takeaways … and Foreshadowing

- Dance monotonically
- Pay for non-monotonicity
- Try to find ways to be monotonic
  - Or not to care! E.g. confluence only of invariants*

- How do we get back to bottom-up?
  - Can software worry about coordination for us?
  - How can we test our code for monotonicity?
  - How can we write monotonic code?

* [Bailis, et al. "Invariant Confluence…", VLDB 2015]

Outline

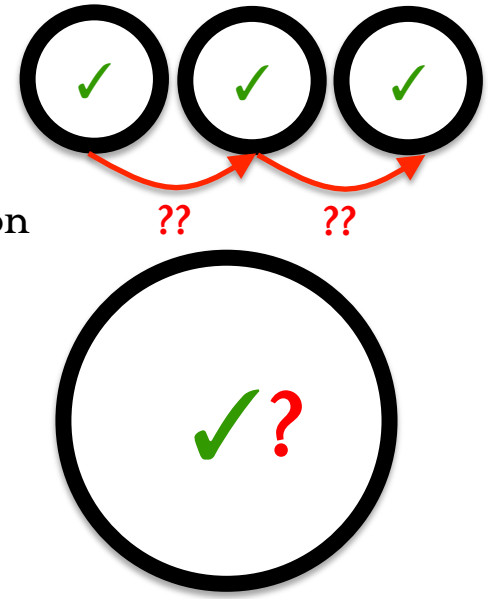
- Cloud: A Deal with the Devil
- Bottom-Up and Top-Down systems
- Creativity from the bottom
- Good news from the top: CALM
- **Grounding CALM: Bloom** and Blazes
- Lessons and Challenges

Getting Practical

- How can new PLs/libraries help?
  1. Encourage monotonicity
  2. Guard non-monotonicity cheaply
- Can they address hard debugging problems?
  1. Consistency and Coordination
  2. Fault tolerance
  3. Garbage collection
- Can we define a nice PL that people can use?

One Direction: ACID 2.0 as a Datatype

- CRDTs: ACID 2.0 object classes (lattices)
- Natural library of lattices
  - Sets with Union
  - Booleans with OR
  - Integers with Max
  - Multisets with Union
- Fancier custom CRDTs
  - E.g. concurrent editors
- Problem: Scope Dilemma
  - Guarantees are per-object
  - What happens across objects?
  - How do you test complex CRDTs?
- Bottom upbut little help building up



[Shapiro, et al. "A Comprehensive Study of Convergent and Commutative Replicated Data Types, 2011]

Another direction: Datalog-based DSLs

- Practical success in declarative networks* and SDNs
- Rich theory, monotonicity is easy to analyze
- Can we write everything this way?
  - You can go far: BOOM Analytics**
  - But gotchas with mutable state
- **Dedalus**: Datalog in space and time
  - Minimally captures state evolution and messaging
  - Lovely basis for the theory work
- Problem: Not always natural
  - E.g. vector clocks in Dedalus
- Still too top-downish?

*[Loo, et al. "Declarative Networking", CACM09]
**[Alvaro, et al. "BOOM Analytics...", Eurosys10]

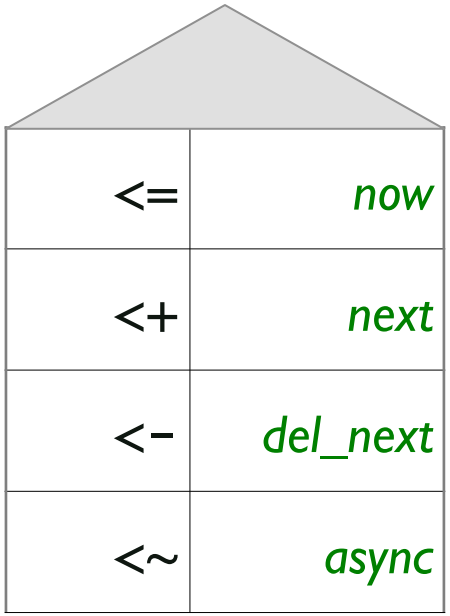
<~ bloom

- A disorderly language of data, space and time
- Based on Alvaro's Dedalus logic
- Extended with lattices and lattice composition

[Alvaro, et al. "Dedalus: Datalog in Time and Space", 2009]
[Hellerstein, et al. "Consistency Analysis in Bloom:...", CIDR '11]
[Conway, et al. "Logic and Lattices for Distributed Progamming", SOCC '12]
http://bloom-lang.org

Syntax: Temporal Merge Rules

state <= expression(events/state)



} time: for mutation

} for communication

DSLs

Domain Specific Languages*

Paul Hudak
Department of Computer Science
Yale University

December 15, 1997

1 Introduction

When most people think of a programming language they think of a *general purpose* language: one capable of programming any application with relatively the same degree of expressiveness and efficiency. For many applications, however, there are more natural ways to express the solution to a problem than those afforded by general purpose programming languages. As a result, researchers and practitioners in recent years have developed many different *domain specific* languages, or DSL's, which are tailored to particular application domains. With an appropriate DSL, one can develop complete application programs for a domain more quickly and more effectively than with a general purpose language. Ideally, a well-designed DSL captures precisely the semantics of an application domain, no more and no less.

Table 1 is a partial list of domains for which DSL's have been created. As you can see, the list covers quite a lot of ground. For a list of some popular DSL's that you may have heard of, look at Table 2.[1] The first example is a set of tools known as Lex and Yacc which are used to build lexers and parsers, respectively. Thus, ironically, they are good tools for *building* DSL's (more on this later). Note that there are several document preparation languages listed; for example, LaTeX was used to create the original draft of this article. Also on the list are examples of "scripting languages," such as PERL, Tcl, and Tk, whose general domain is that of scripting text and file manipulation, GUI widgets, and other software components. When used for scripting, Visual Basic can also be viewed as a DSL, even though it is usually thought of as general-purpose. I have included one other general-purpose language, Prolog, because it

*Appeared as Chapter 3 in *Handbook of Programming Languages, Vol. III: Little Languages and Tools*, Peter H. Salus, ed., MacMillan, Indianapolis, pp. 39–60, 1998.
[1]Both of these tables are incomplete; feel free to add your favorite examples to them.

1

> "A user immersed in a domain already knows the domain semantics. All the DSL designer needs to do is provide a notation to express that semantics."
>
> —Paul Hudak

Vector Clocks: Bloom v. Wikipedia

```
bootstrap do
  my_vc <=
     {ip_port => Bud::MaxLattice.new(0)}
end

bloom do
  next_vc <= out_msg
    { {ip_port => my_vc.at(ip_port) + 1} }
  out_msg_vc <= out_msg
    {|m| [m.addr, m.payload, next_vc]}
  next_vc <= in_msg
    { {ip_port => my_vc.at(ip_port) + 1} }
  next_vc <= my_vc
  next_vc <= in_msg  {|m| m.clock}
  my_vc <+ next_vc
end
```

Initially all clocks are zero.

- Each time a process experiences an internal event, it increments its own logical clock in the vector by one.
- Each time a process prepares to send a message, it increments its own logical clock in the vector by one and then sends its entire vector along with the message being sent.
- Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).

Further Evidence of Fit

- BOOM Analytics & follow-ons
  - BFS
  - KVS variants
    - MV, Causal, Session Guarantees, Transactional, …
- Wide variety of classical protocols
- Concurrent editing
- Programming the Cloud Course
  http://programthecloud.github.io

[Alvaro, et al. "BOOM Analytics: …", Eurosys10]

Takeaways … and Foreshadowing

- Building from a better bottom
  - Lattices are nice disorderly building blocks
- Restarting from the top
  - Dedalus is a formal declarative framework for specifying and computing data lineage across space and time
- Bloom: an approachable compromise of the two

- How can a good DSL help with distributed SW engineering?
  - Coordination minimization (Blazes)
  - Fault tolerance (Molly)
  - Event log garbage collection (Edelweiss)

Outline

- Cloud: A Deal with the Devil
- Bottom-Up and Top-Down systems
- Creativity from the bottom
- Good news from the top: CALM
- Grounding CALM: Bloom and **Blazes**
- Lessons and Challenges

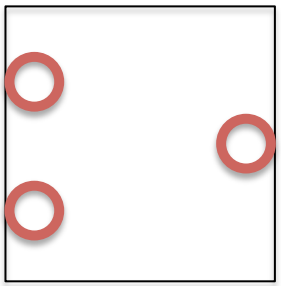
Below Declarative: Dataflow

- A popular semi-imperative model
- Components, dataflow and *composition*
  - Async:
    - Service Oriented Architectures
    - Functional Reactive Programming
  - Bulk/Streaming:
    - Relational Algebra
    - MapReduce
- If you squint, it's all surprisingly the same

Ensuring Confluent Dataflow

- Key flow concern
  - Order-sensitive operator downstream of communication
- Cheap coordination
  - Sometimes we can handle this without global consensus
  - Basic idea: "Sealing" (as in the cart example)
  - Question: how to choose/propagate seals across SW components?

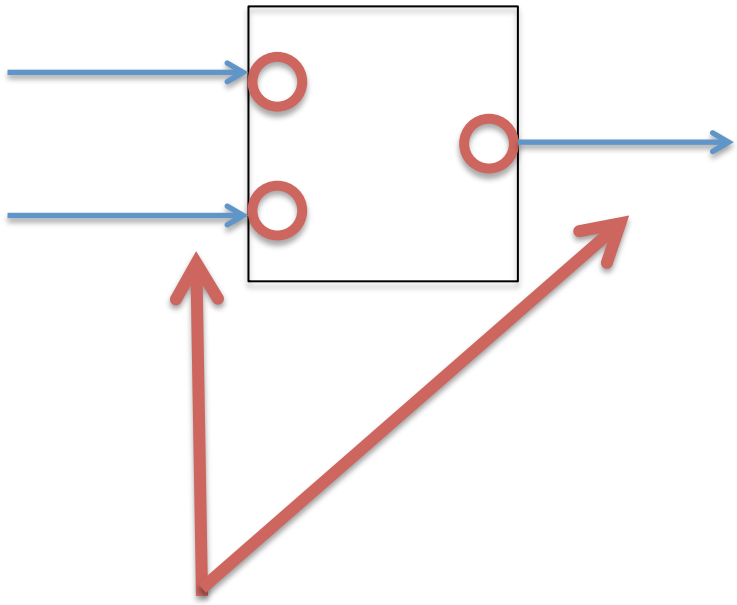
Components



Input interfaces

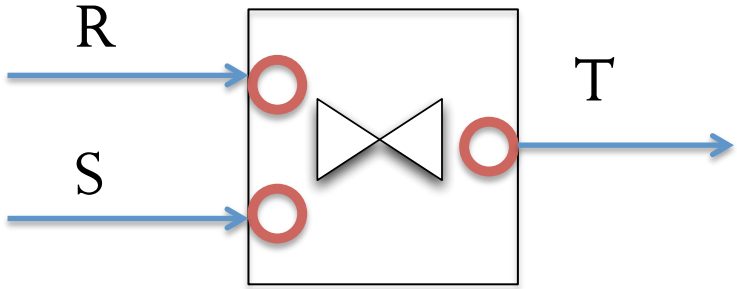
Output interface

Streams

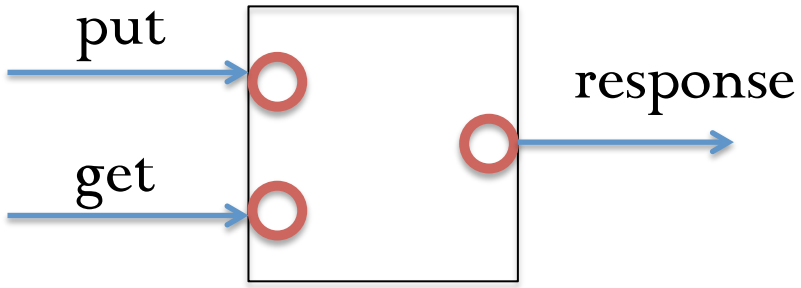


Nondeterministic order

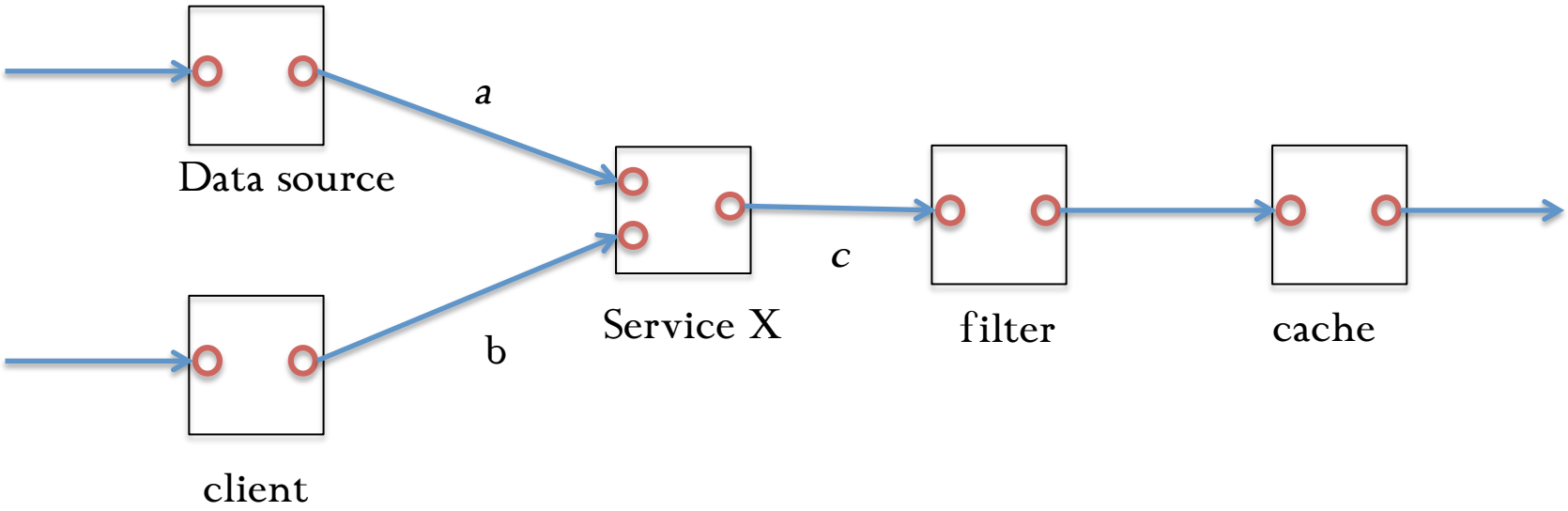
Example: a **join** operator



Example: a **key/value store**

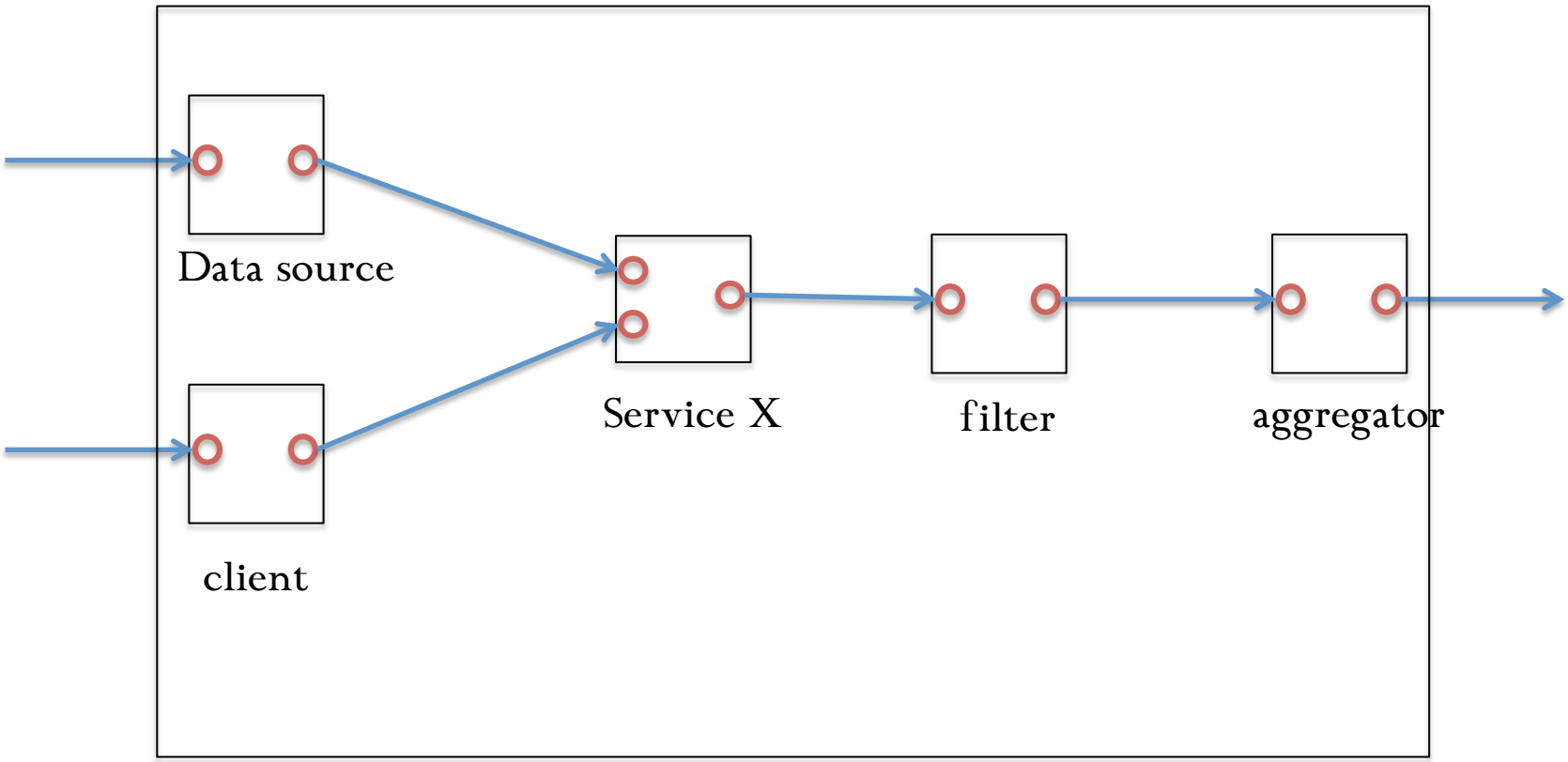


Logical dataflow



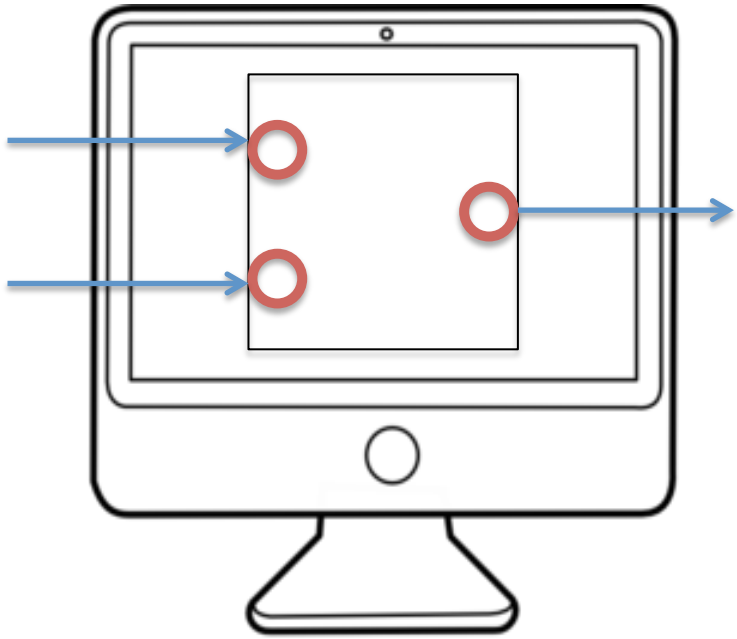
"Software architecture"

Dataflow is compositional

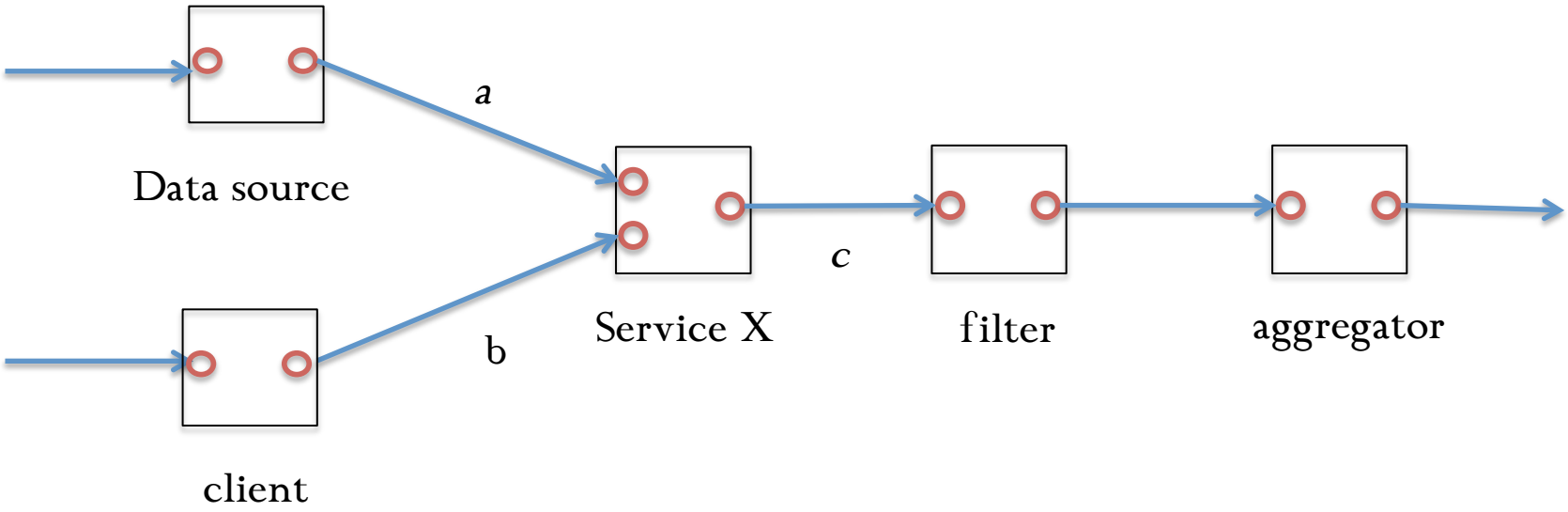


Components are recursively defined

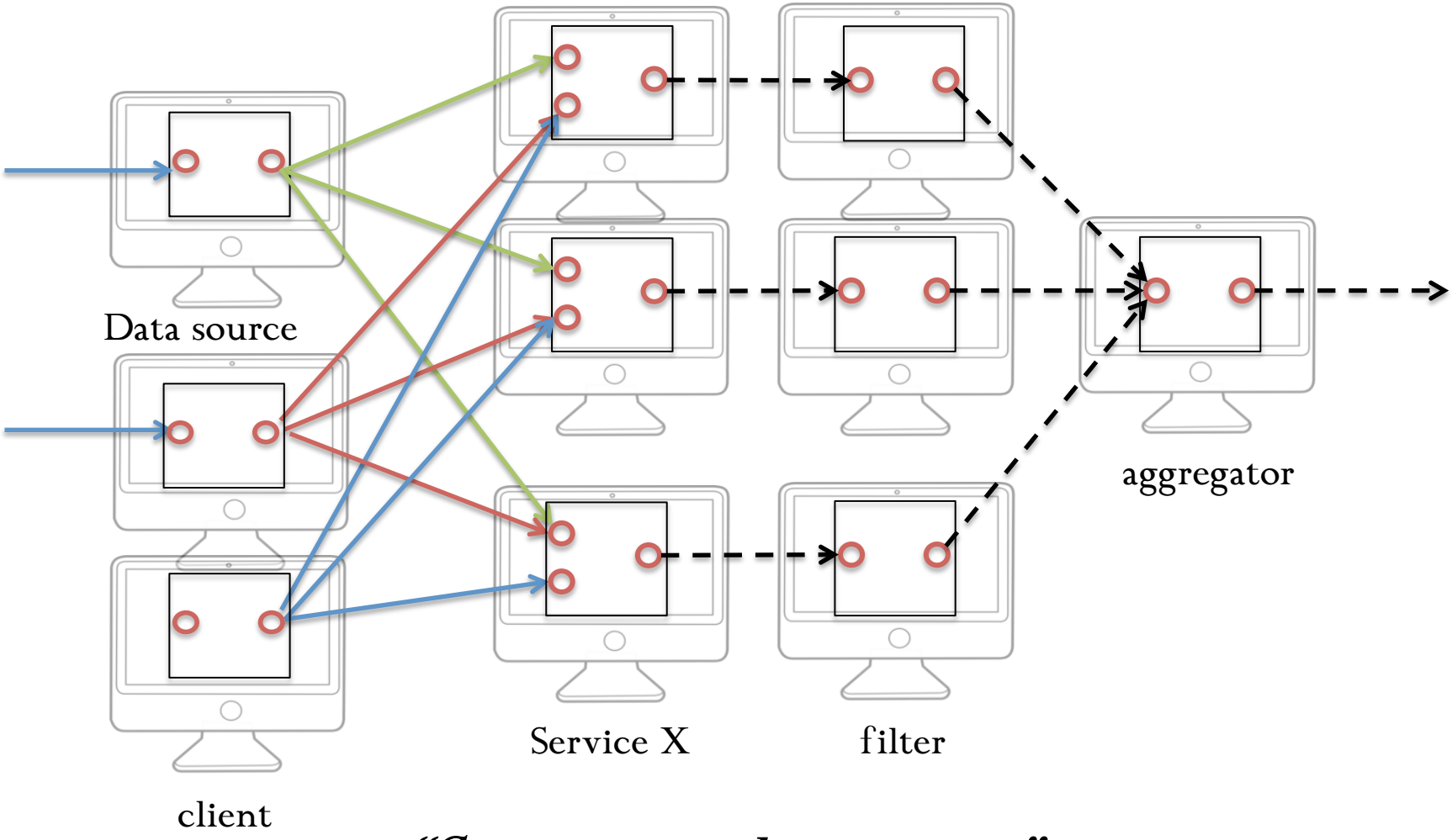
Physical dataflow



Physical dataflow



Physical dataflow

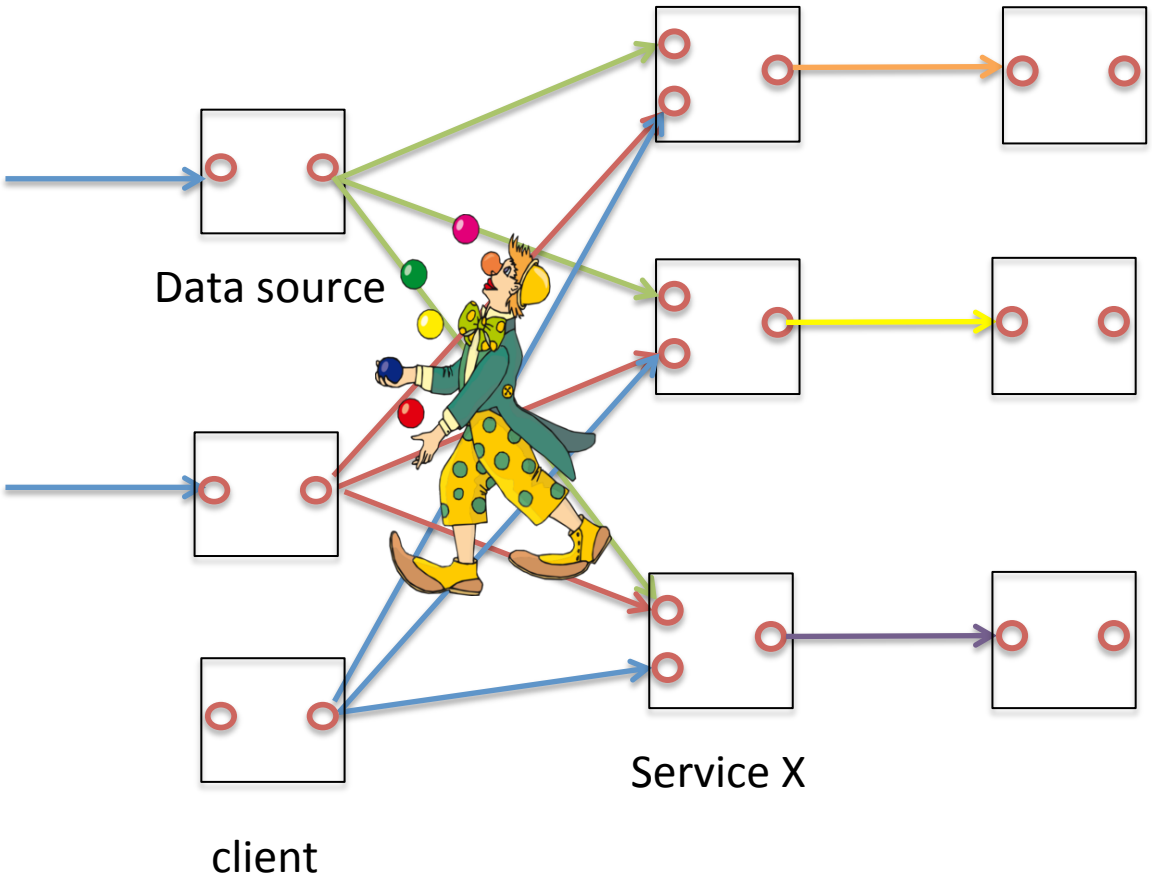


"System architecture"

What Could Go Wrong

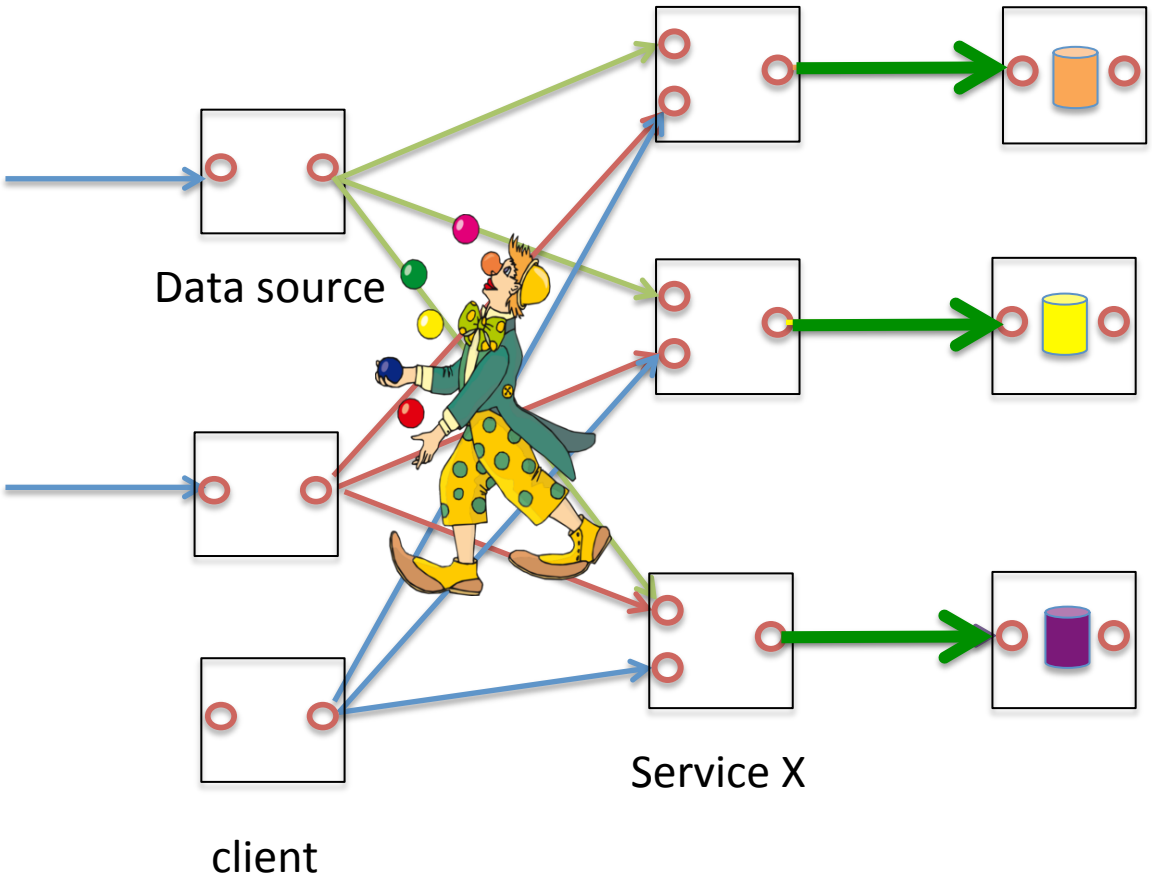
- Transitive Non-Determinism: Order-sensitive component *downstream* from (disorderly) communication
  - Unordered streams, or
  - Multiple interleaved streams

Cross-instance nondeterminism



Transient replica disagreement

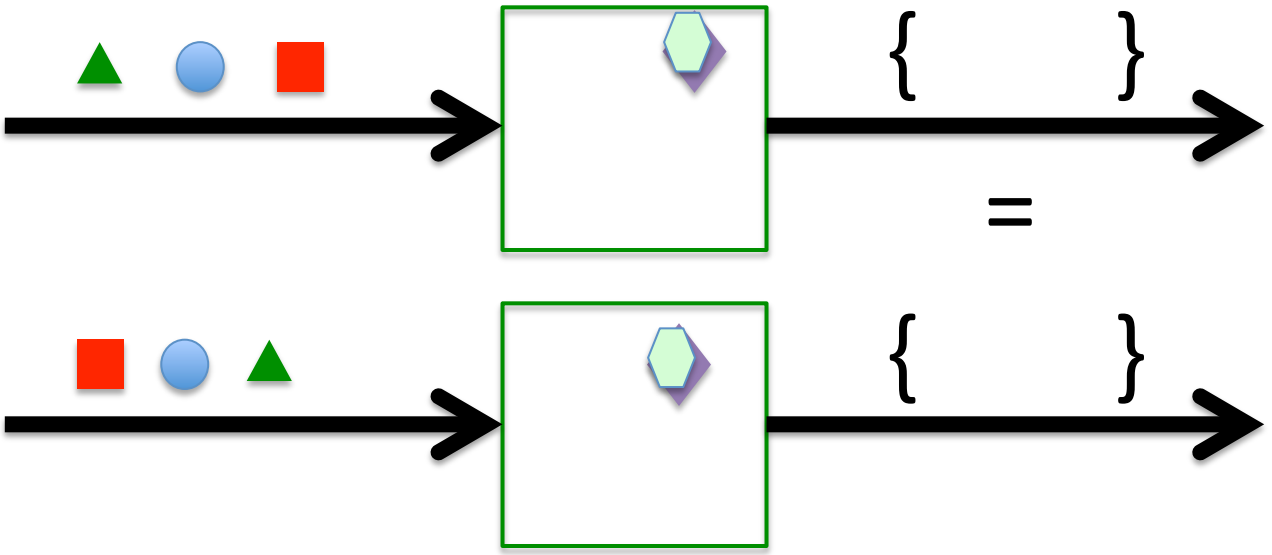
Divergence



Permanent replica disagreement

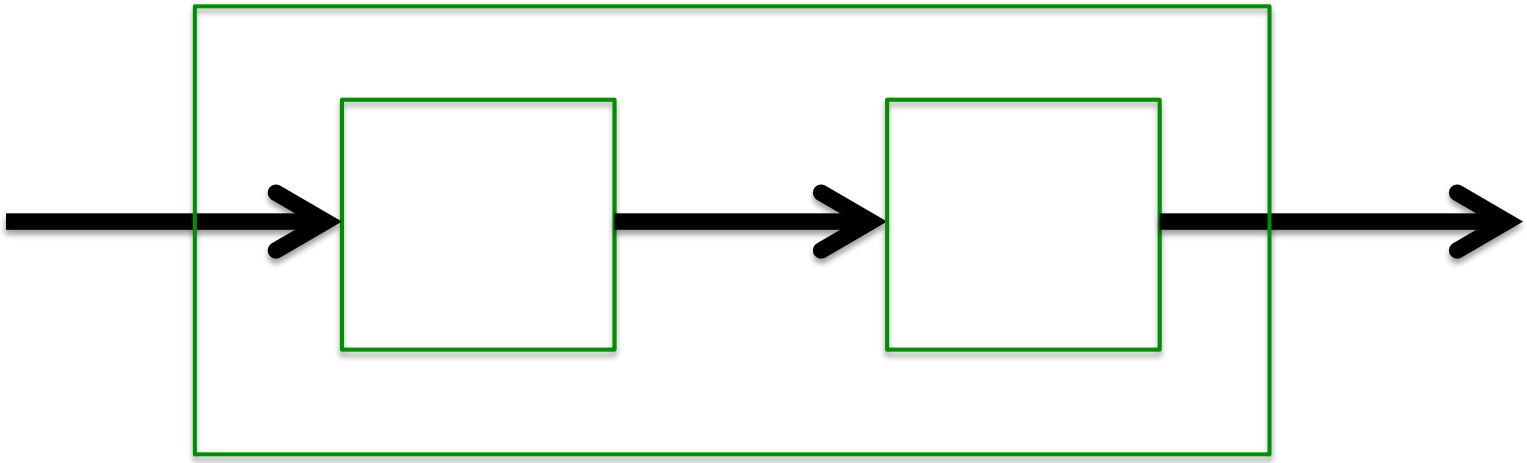
Confluence

output set = f(input set)



Confluence is Compositional

output set = f · g(input set)



Blazes

- Given an annotated dataflow
  - Some operators marked as **order-insensitive**
  - Some keys marked as determining value of other keys
    - E.g. "sessionID" determines value of "cart_contents"
    - A.k.a. functional dependencies

- Add minimalist logic to ensure confluence
  - Win: seal a (sub)set of data without global coordination
  - Very much like we did with shopping cart seals
    - But synthesized automatically!

[Alvaro, et al. "Blazes: Coordination Analysis...", ICDE14]

Annotated dataflow?

- Who adds the annotations
  - Order-insensitivity? Dependencies?
- We can ask a dataflow programmer: "gray boxes"
  - E.g. a Storm programmer, CRDTs
  - Blazes guarantees correct *composition* of these gray boxes
- We can ask a compiler
  - About composition *and* components
  - Starting from a higher-level language
  - Blazes guarantees *entire* Bloom programs, unassisted

Back to Shopping

- Remember the typical KVS cart implementation
  - Bottom-up reusable component
  - But expensive coordination on every write
- The sealed, replicated log as a design pattern
  - A bit more top-down, custom
- What Would Blazes Do
  - If we give it the KVS cart?

KVS

```
module KVS
  state do
    interface input, :put, [:key, :val]
    interface input, :get, [:ident, :key]
    interface output, :response,
              [:response_id, :key, :val]
    table :log, [:key, :val]
  end
  bloom do
    log <+ put
    log <- (put * log).rights(:key => :key)
    response <= (log * get).pairs(:key=>:key) do |s,l|
       [l.ident, s.key, s.val]
    end
  end
end
```

KVS

```
module KVS
  state do
    interface input, :put, [:key, :val]
    interface input, :get, [:ident, :key]
    interface output, :response,
              [:response_id, :key, :val]
    table :log, [:key, :val]
  end
  bloom do
    log <+ put
    log <- (put * log).rights(:key => :key)
    response <= (log * get).pairs(:key=>:key) do |s,l|
       [l.ident, s.key, s.val]
    end
  end
end
```

Negation (→ order sensitive)

KVS

```
module KVS
  state do
    interface input, :put, [:key, :val]
    interface input, :get, [:ident, :key]
    interface output, :response,
              [:response_id, :key, :val]
    table :log, [:key, :val]
  end
  bloom do
    log <+ put
    log <- (put * log).rights(:key => :key)
    response <= (log * get).pairs(:key=>:key) do |s,l|
       [l.ident, s.key, s.val]
    end
  end
end
```

Negation (→ order sensitive)

Partitioned by **:key**

KVS

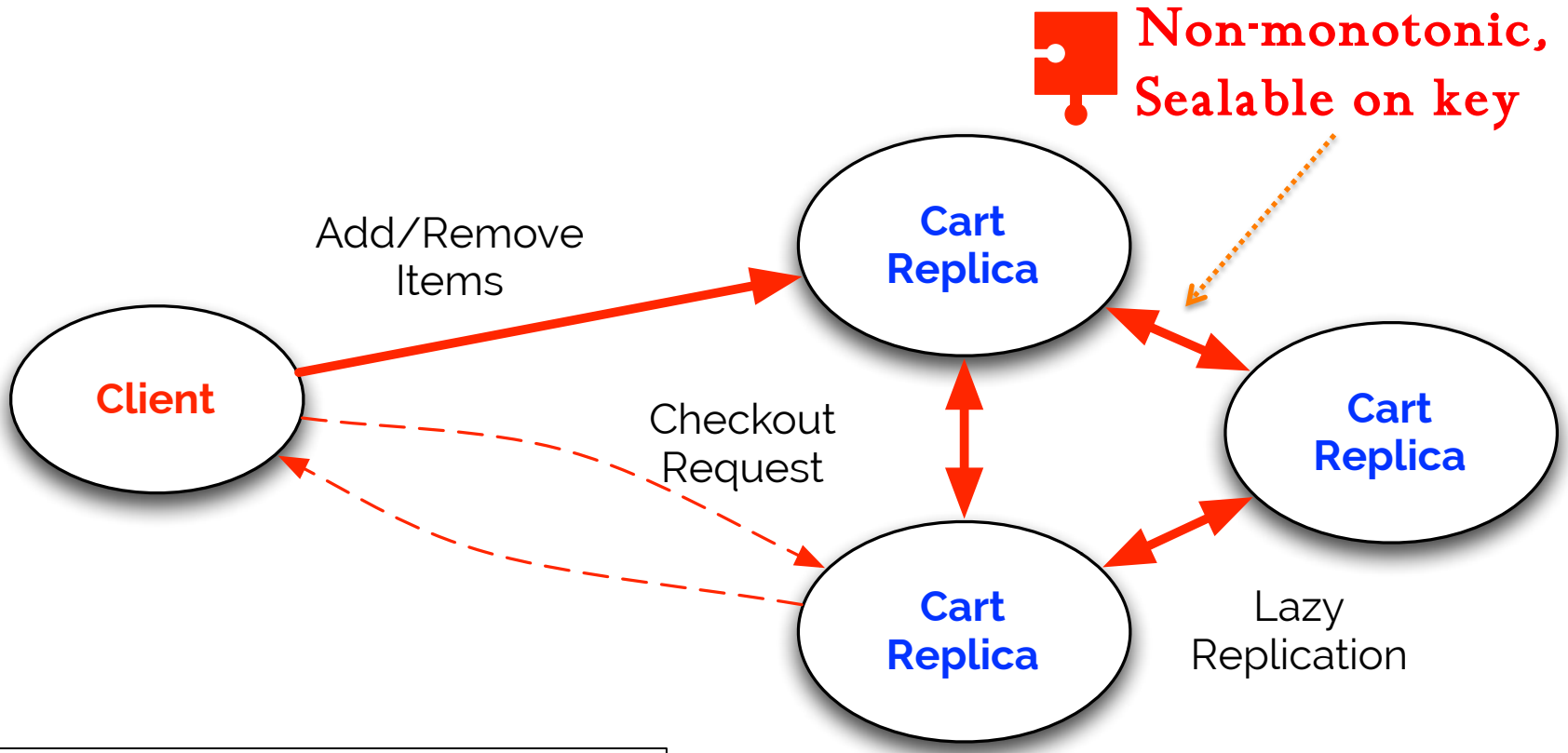
put → response: OW_{key}
get → response: OR_{key}

```
module KVS
  state do
    interface input, :put, [:key, :val]
    interface input, :get, [:ident, :key]
    interface output, :response,
              [:response_id, :key, :val]
    table :log, [:key, :val]
  end
  bloom do
    log <+ put
    log <- (put * log).rights(:key => :key)
    response <= (log * get).pairs(:key=>:key) do |s,l|
       [l.ident, s.key, s.val]
    end
  end
end
```

Negation (→ order sensitive)

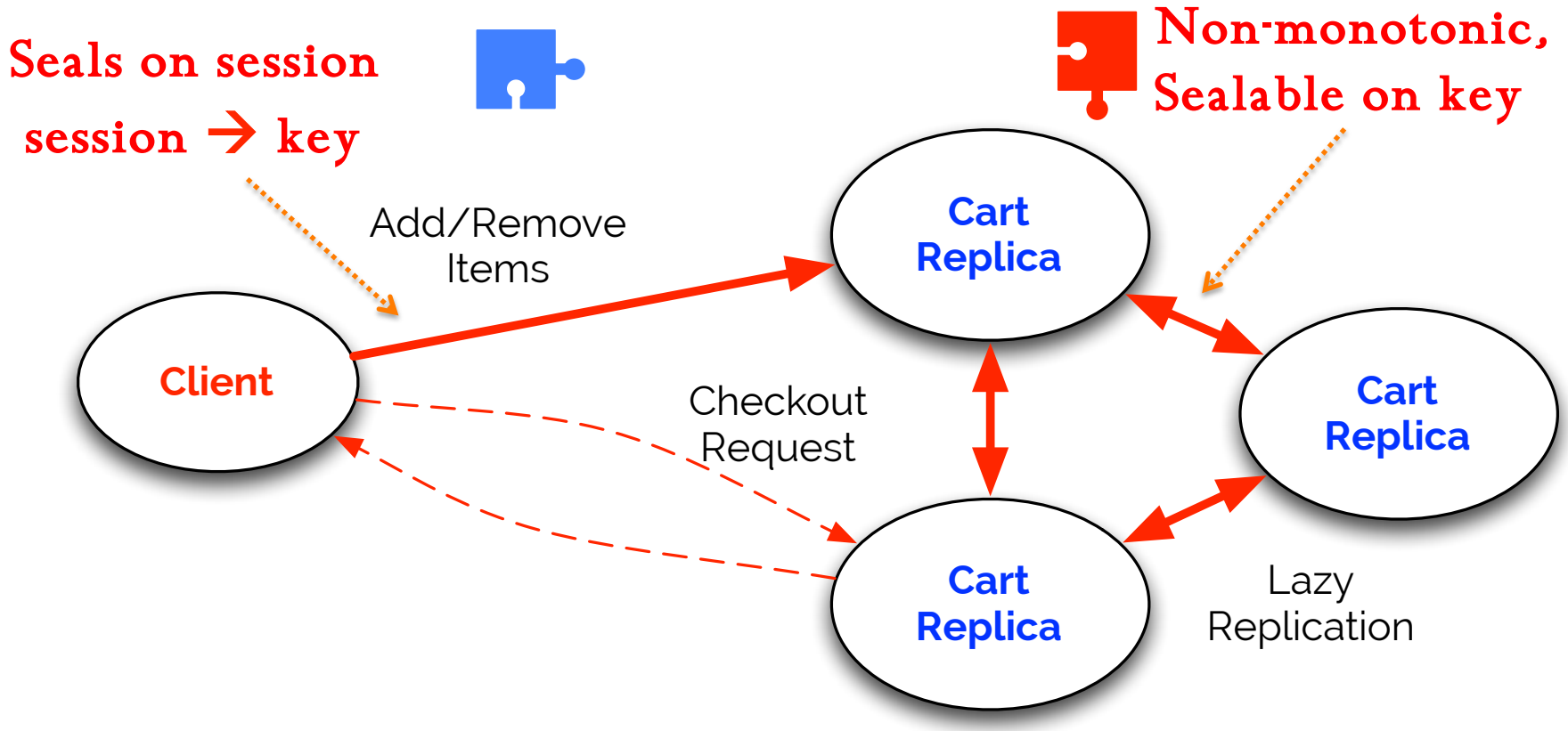
Partitioned by **:key**

Blazes

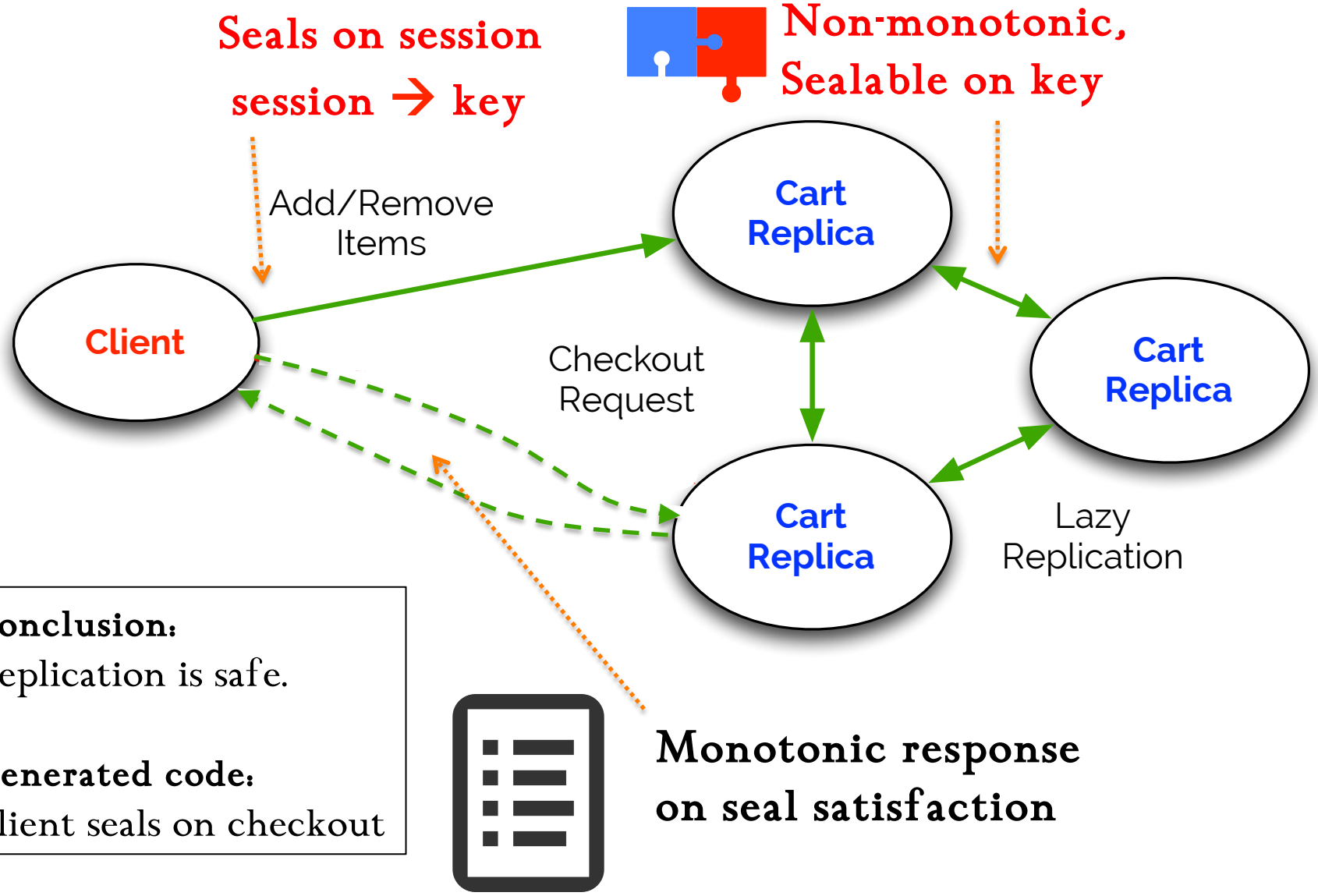


Conclusion:
Every operation might require coordination!

Blazes


Seals on session
session → key
Non-monotonic,
Sealable on key
Add/Remove
Items
Client
Cart
Replica
Cart
Replica
Cart
Replica
Checkout
Request
Lazy
Replication

Blazes



Conclusion:
Replication is safe.

Generated code:
Client seals on checkout

Blazes Takeaways

- CALM intuition exported to dataflow
  - E.g. Apache Storm, via "gray-box" annotations
- Bloom is easy to check in "white-box" mode
  - Dataflow + *annotations* easily pulled from syntax
- Sealing as a cheap source of coordination
  - Data that's partitioned so a single site generate seals

Two more analysis results

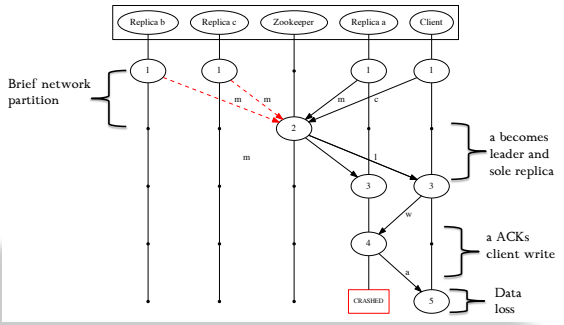
- Failures and Fault Tolerance
- Application-aware Garbage Collection

LDFI … tl;dr

- How to test end-to-end Fault Tolerance?
- Lineage-Driven Fault Injection (LDFI)
  - Start with a known good outcome
  - Ask: what could have gone wrong?
    - A lineage question
  - Synthesize failures to ensure things go wrong
  - Execute and look for a new good outcome to explore

[Alvaro, et al. 2015]

Kafka durability bug



Edelweiss … tl;dr

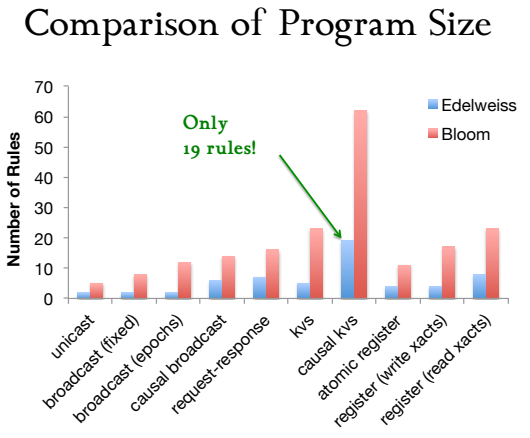
Bloom … and grow?

- If we keep exchanging monotonic logs
- Can we ever throw anything away?

Edelweiss

- A restricted subset of Bloom
- Removes constructs for deletion and mutation

Automatically generate safe, application-specific GC protocols



[Conway, et al. "Edelweiss:…", VLDB 2014]

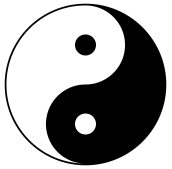
Outline

- Cloud: A Deal with the Devil
- Bottom-Up and Top-Down systems
- Creativity from the bottom
- Good news from the top: CALM
- Grounding CALM: Bloom and Blazes
- **Lessons and Challenges**

Summary

- **Coordination** is the key remaining cost in cloud computing
  - Paxos, 2PC, etc.

- Q: **When** can coordination be avoided w/o inconsistency?
  A: **CALM:** exactly for monotonic programs



- Q: **How** can coordination be avoided practically?
  A: **Application-level reasoning** is the engine of innovation

- **DSLs** are reliable vehicles for that innovation
  - **Patterns:** Reinforce healthy design patterns
  - **Theorems:** Formal approaches supporting analysis and code synthesis
  - **Software:** Data-centric DSLs like Bloom are well-suited to the domain

<~ bloom

Opportunities 1: Rethinking Coordination

- Polyglot Consistency
  - Some of my data needs consistency. Some doesn't.
  - How to avoid leaking inconsistency *taint*?
- Coordination locality
  - E.g. Calvin, Hstore do coordination at job ingress
  - E.g. seal generation in Blazes
  - Optimize programs to "push" coordination to local spots?
- Programming with Apologies
  - Pattern: allow inconsistency, fix things up later (coupons)
  - Can we do Pattern—>Theorem—>Software here?

Opportunities 2: DSLs

- DSLs for orchestration, service composition
  - Deployment is programming! Bugs ensue through incorrect composition.
  - Kubernetes/Chef/Puppet are declarative DSLs; extend to richer SW composition
- Performance optimization
  - Bloom was an exercise in the possible. What about the optimal?
  - High-performance concurrent DSL? Interesting for multicore, NewSQL internals, etc.
- Bottom-up alternatives to Bloom
  - Take a cue from CRDTs, Erlang, Akka, etc.
  - Consider design patterns like Event Log Exchange & Edelweiss
  - Fix the scope dilemma as Bloom did with monotone functions
- Convergence in Big Data
  - World 1: async programming and NoSQL (Bloom, Erlang, Akka, node.js)
  - World 2: parallel analytics, batch processing, streaming (SQL, Hadoop, Spark, Storm)
  - Convergence: Design Opportunity? Benefits?