BOOM: EXPERIENCES IN LANGUAGE AND TOOL DESIGN FOR DISTRIBUTED COMPUTING

JOSEPH M HELLERSTEIN

http://www.flickr.com/photos/44124348109@N01/1468055021/

Berkeley

TRIFACTA

BOOM TEAM

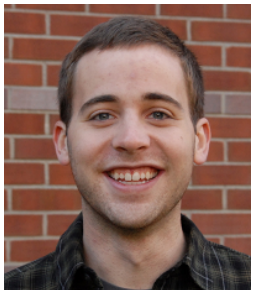

joe hellerstein


david maier


ras bodik


alan fekete


peter alvaro

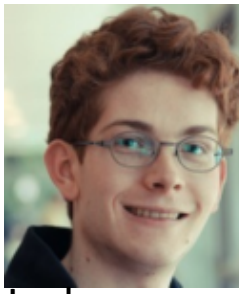

peter bailis

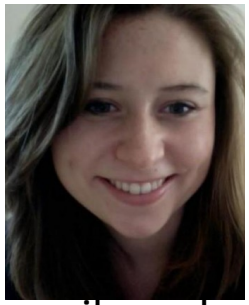

neil conway


bill marczak


haryadi gunawi


sriram srinivasan


Joshua rosen


emily andrews

andy hutchinsor

OUTLINE

- Motivation
- CALM: Positive Theory
- Bloom: Disorderly Programming

PROGRAMMING TODAY

- Non-trivial software is distributed
- Distributed programming is hard[2]
  - (software engineering)
    × (parallelism + asynchrony + failure)
- A SW Engineering imperative


http://en.wikipedia.org/wiki/File:JohnvonNeumann-LosAlamos.gif

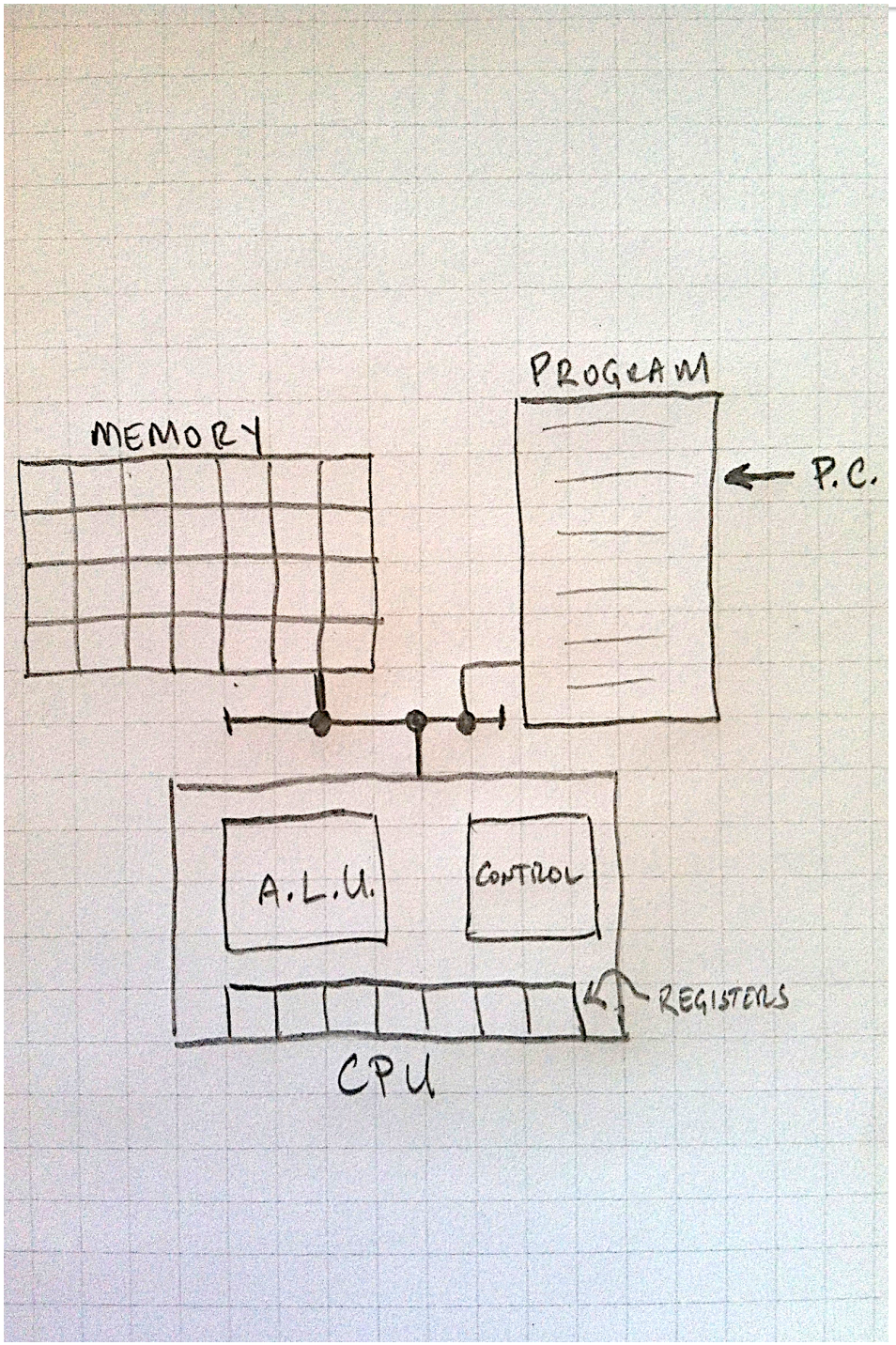
ORDERLY COMPUTING

ORDER

- LIST of Instructions
- ARRAY of Memory

STATE

- Mutation in time



ORDERLY COMPUTING

ORDER

- LIST of Instructions
- ARRAY of Memory

STATE

- Mutation in time

http://www.flickr.com/photos/31608675@N00/103014980/

ORDERLY COMPUTING

ORDER

- LIST of Instructions
- ARRAY of Memory

STATE

- Mutation in time

ORDERLY COMPUTING

ORDER

- LIST of Instructions
- ARRAY of Memory

STATE

- Mutation in time

CLOUD PROGRAMMING

HOSTED for availability

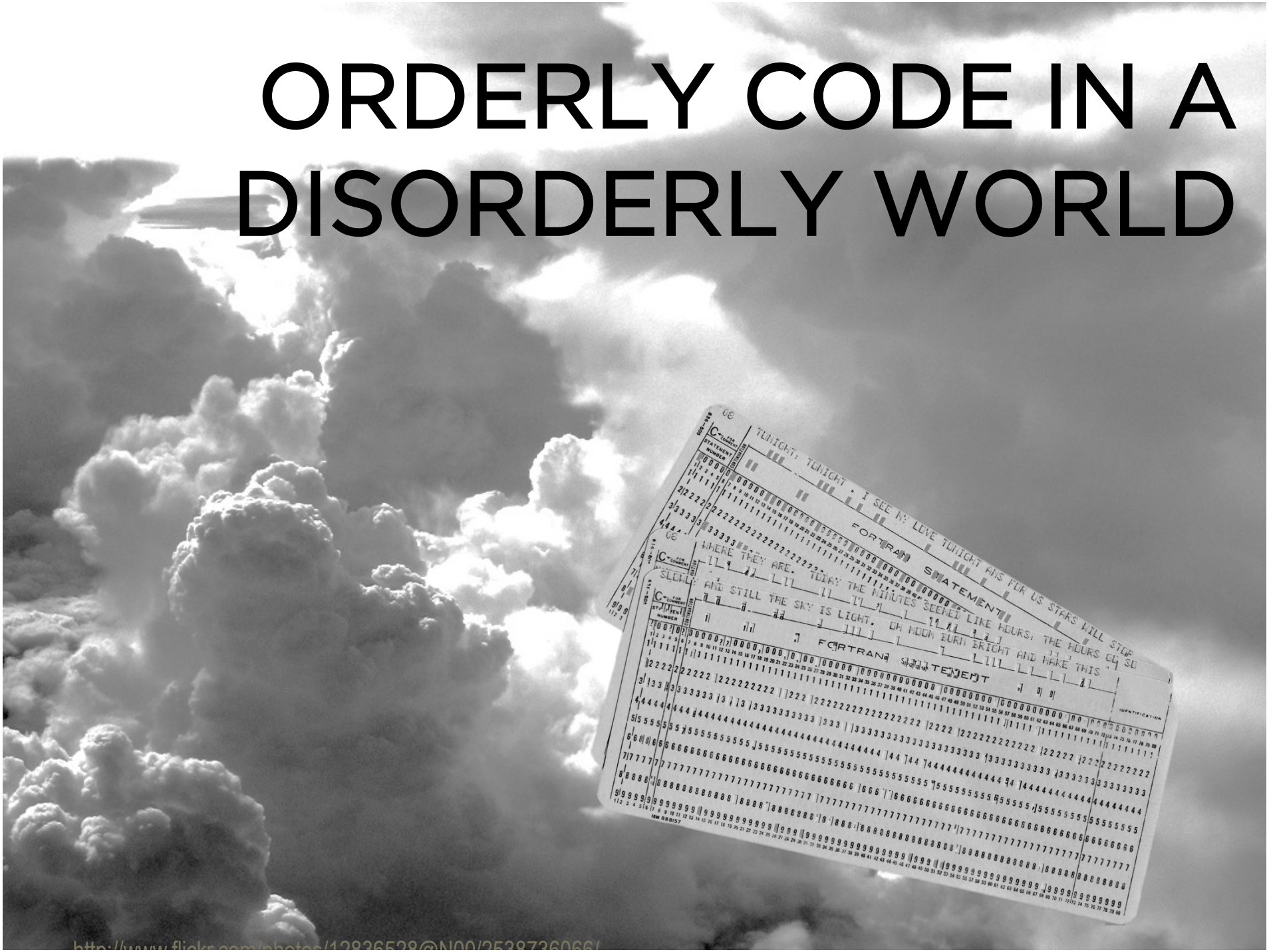
REPLICATED for redundancy

PARTITIONED to scale out

ASYNCHRONOUS for performance

All this ... in Java.

ORDERLY CODE IN A DISORDERLY WORLD



ORDERLY ASSUMPTIONS

Item	Count
	1
	1

Item	Count
	1
	1

WHAT COULD GO WRONG?

Item	Count
	0
	1

Item	Count
	2
	1

CLASSICAL TREATMENT

- Model: Distributed State (R/W)
- Desire: Eventual Consistency
- Mechanism: Linearization (SSI)
  - E.g. Paxos distributed log

Item	Count

Item	Count

Item	Count

Item	Count

Item	Count
	1

Item	Count
	1

Item	Count
	1

Item	Count
	1

Item	Count
	0

Item	Count
	0

Item	Count
	0

Item	Count
	0

Item	Count
	1

Item	Count
	1

Item	Count
	1

Item	Count
	1

Item	Count
	1
	1

Item	Count
	1
	1

ASK THE DEVELOPERS

- Questions
  - Do multiple agents need to coordinate?
  - On which lines of code?
- Variations
  - Concurrent. Replicated. Partitioned parallel.
  - Unreliable network, agents
- Software testing and maintenance

A NEGATIVE RESULT FOR CLASSICAL TREATMENTS

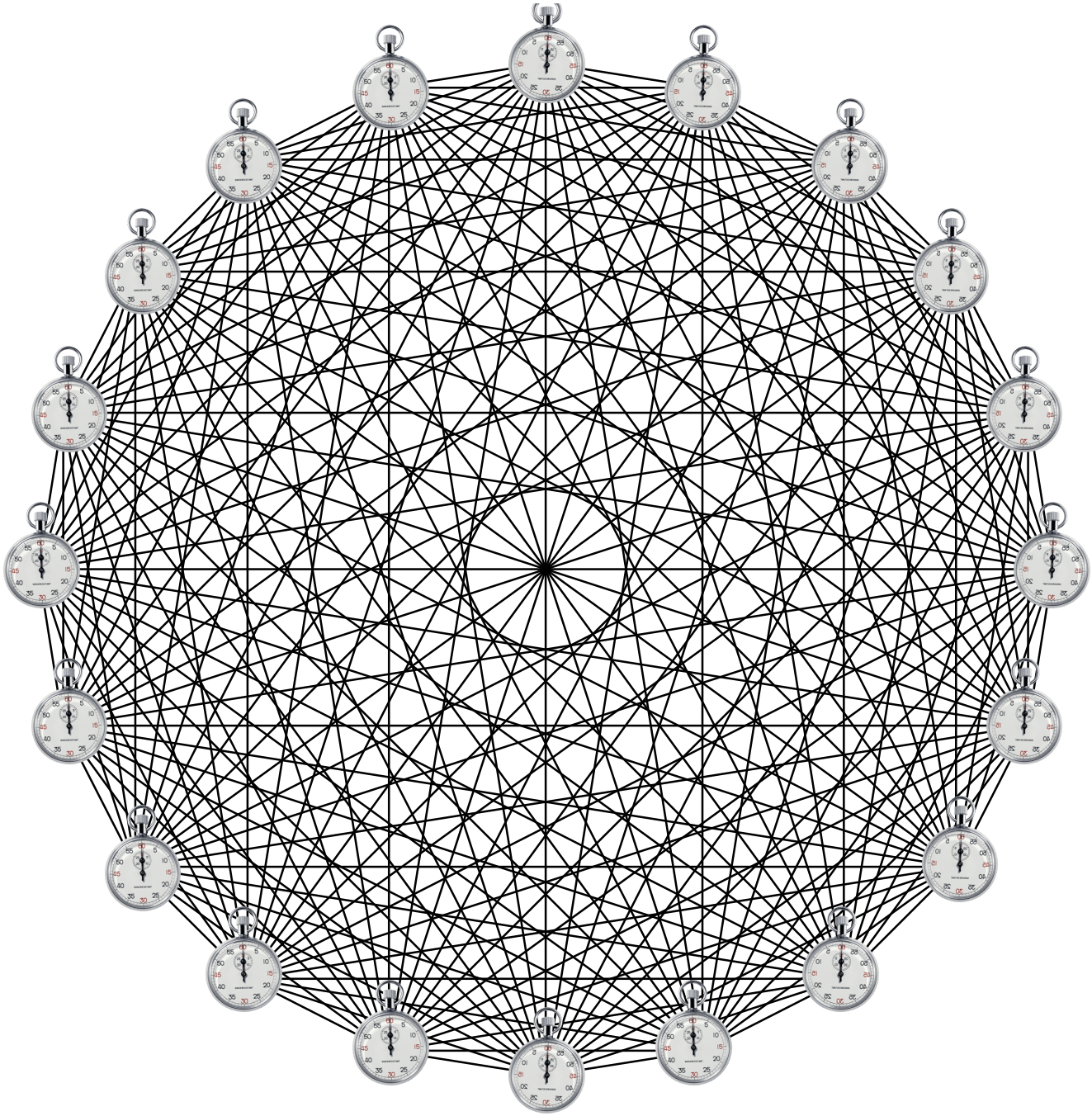
Brewer's CAP Theorem: *It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:*

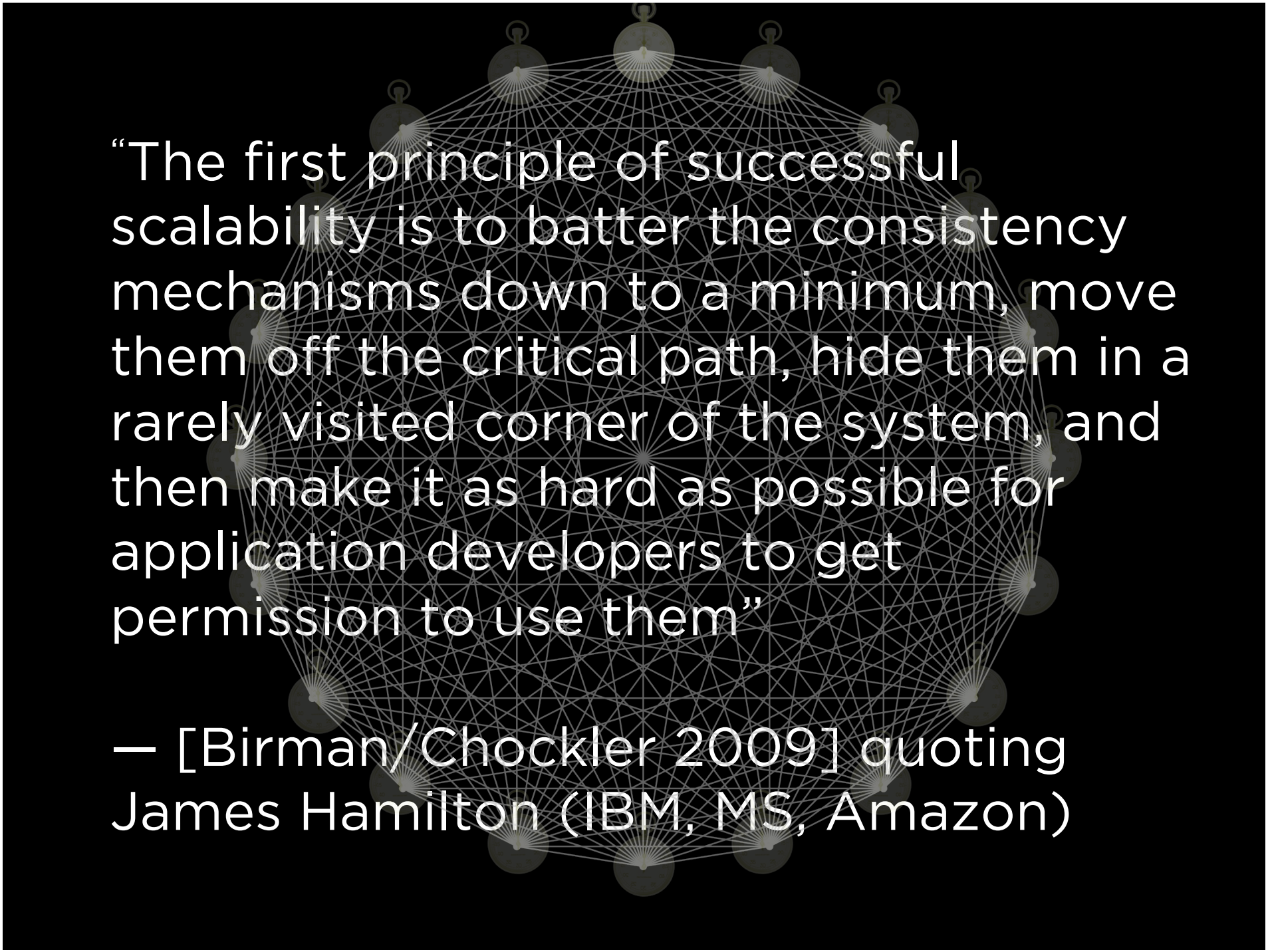
- *Consistency*
- *Availability*
- *Partition-tolerance*

[Gilbert and Lynch 2002]

IN PRACTICE, THERE IS ROOM FOR POSITIVITY

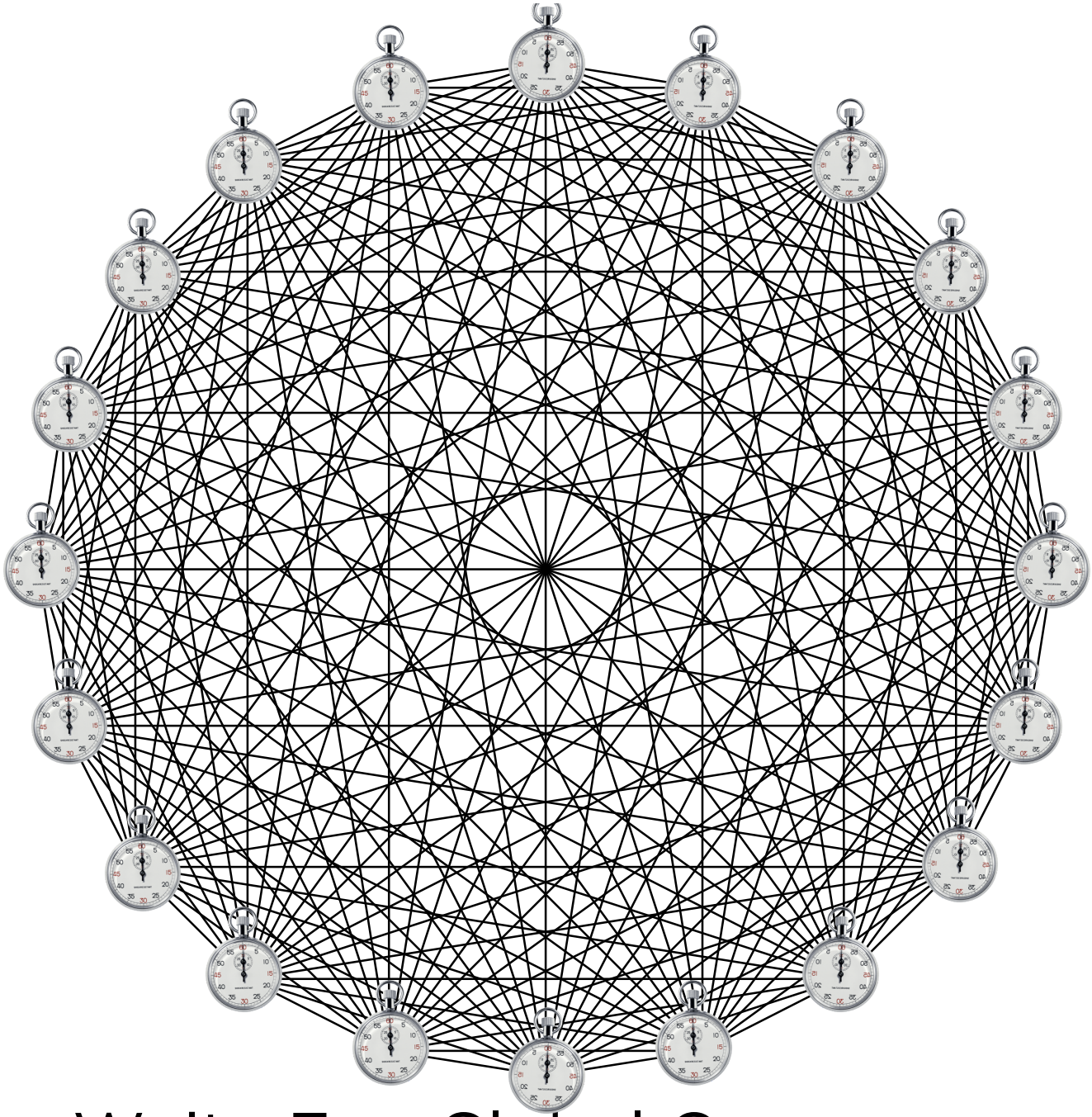
- Partition is rare.
- Hence consistency is possible
- But at what cost?





"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"

— [Birman/Chockler 2009] quoting James Hamilton (IBM, MS, Amazon)

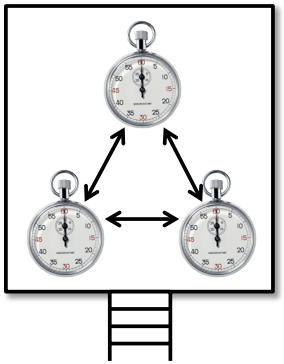


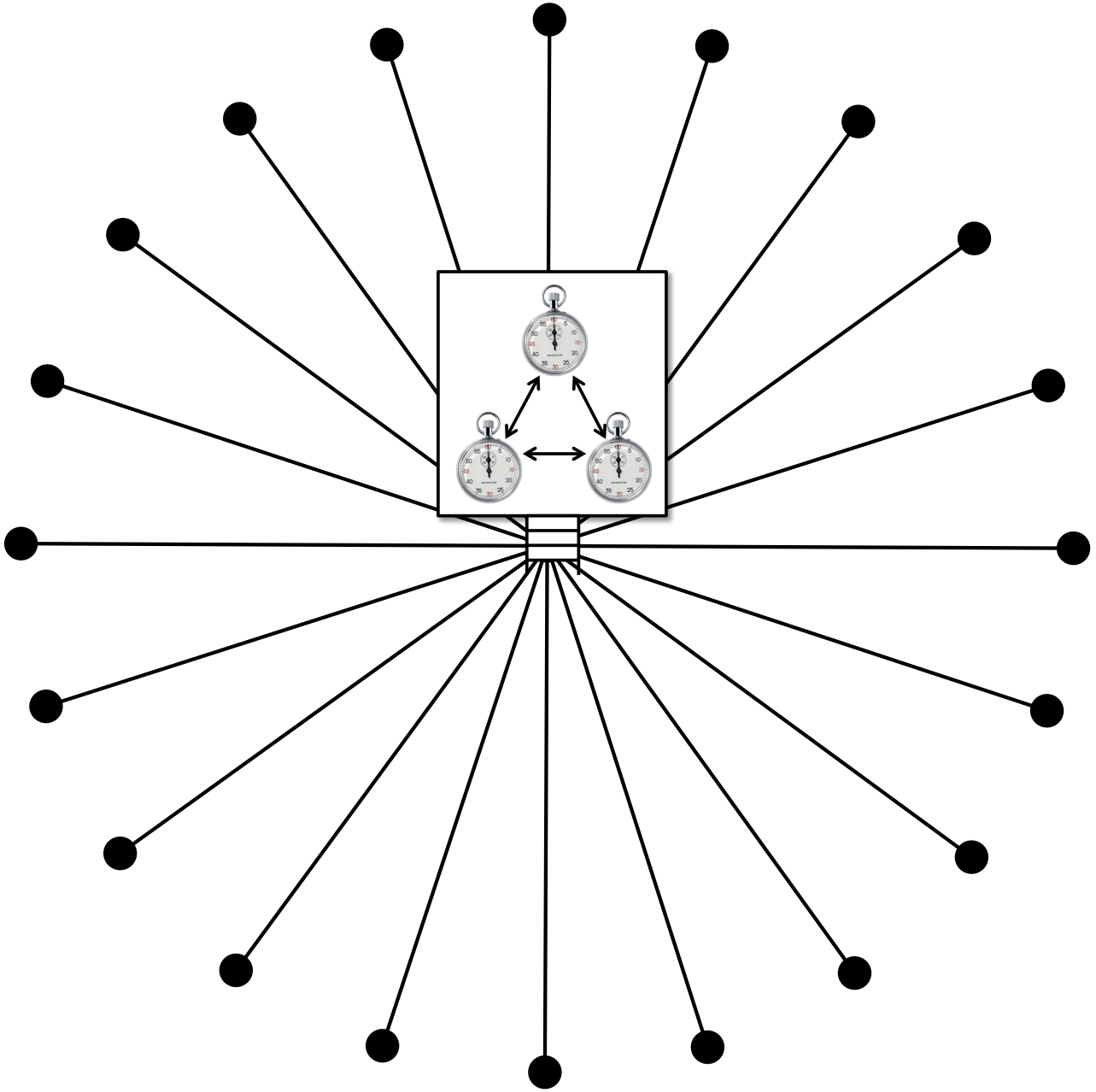
Waits-For: Global Consensus

THE WRONG SIDE OF PROBABILITY

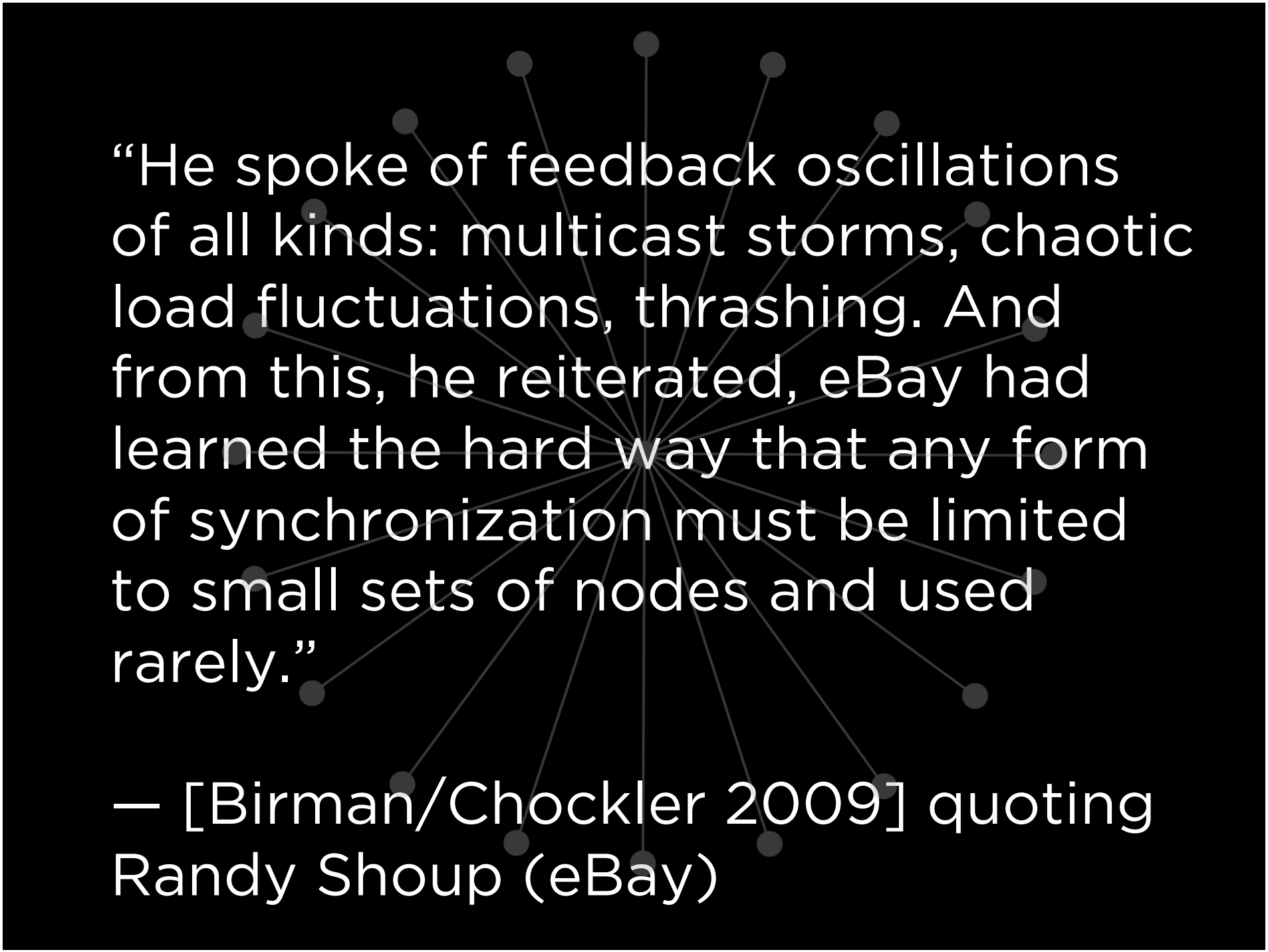
- Where parallelism confounds performance
- Shrink the coordination group?

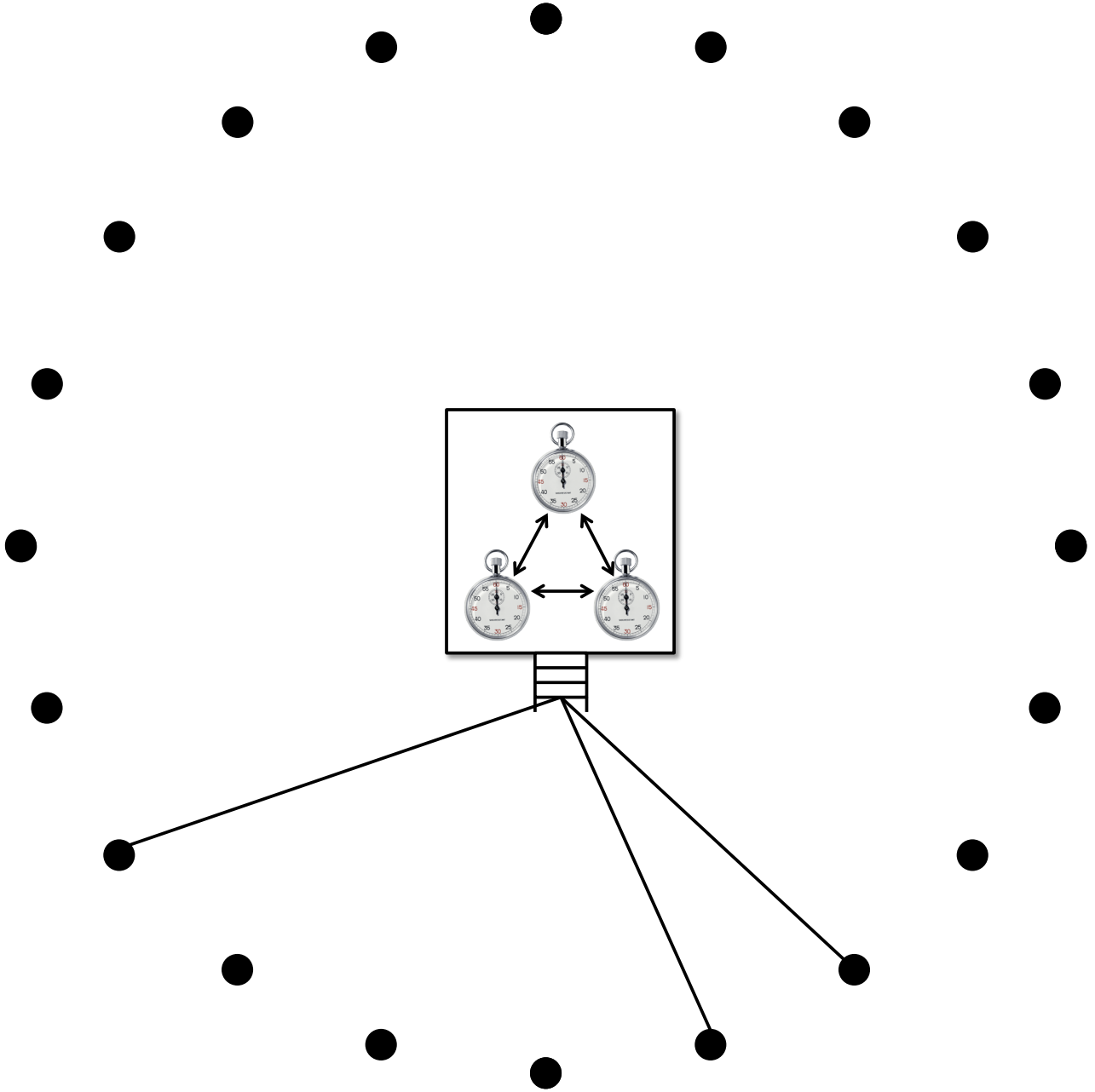

60
5
10
15
20
25
30
35
40
45
50
55
SHOCKRESISTANT





Waits-For: Global Consensus Service


“He spoke of feedback oscillations of all kinds: multicast storms, chaotic load fluctuations, thrashing. And from this, he reiterated, eBay had learned the hard way that any form of synchronization must be limited to small sets of nodes and used rarely.”
— [Birman/Chockler 2009] quoting Randy Shoup (eBay)



Waits-For: Consensus Service

TOWARD POSITIVE RESULTS

What do people do?

- Mutable State is an “anti-pattern”
- Pattern: Log Shipping


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

Item	Count
	1
	1

Item	Count
	1
	1

TOWARD A NEW POSITIVE APPROACH

- Theory Questions
  - When is this pattern possible (and correct)?
  - What to do when impossible?
- Practical Approach
  - “Disorderly” language design
  - Enforce/check good patterns
- Goal: Design → Theory → Practice

DATA
~~CLOUD~~ PROGRAMMING

HOSTED for availability

REPLICATED for redundancy

PARTITIONED to scale out

ASYNCHRONOUS for performance

All this ... in ~~Java~~.

a new disorderly language

AN ONGOING DATA-CENTRIC AGENDA

- 9 years of language and systems experimentation:
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]
  - network caching/proxying [Chu09]
  - relational query optimizers (System R, Cascades, Magic Sets) [Con08]
  - distributed Bayesian inference (e.g. junction trees) [Atul09]
  - distributed consensus and commit (Paxos, 2PC) [Alv09]
  - distributed file system (HDFS) and map-reduce job scheduler [Alv10]
  - KVS variants: causal, atomic, transactional [Alv11]
  - communication protocols: unicast, broadcast, causal, reliable [Con13]
- 2011/2013: “Programming the Cloud” undergrad course
  - http://programthecloud.github.com

Declarative Networking
[Loo et al., CACM ’09]

OUTLINE

- Motivation
- CALM: Positive Theory
- Bloom: Disorderly Programming

From Patterns to Languages

3 LITTLE LANGUAGES

I/O Traces

R(apple);W(apple)

R(apple);W(apple)

R(orange); W(orange)

R(apple);W(apple)

end

Counter Expressions

apple := apple + 1

apple := apple - 1

orange := orange + 1

apple := apple + 1

end

Sets

S <= {apple, 1, a}

S <= {apple, -1, b}

S <= {orange, 1, c}

S <= {apple, 1, d}

O <= S.group(1, SUM(2))

WHY DO WE NEED TIME?

- Conjecture: Non-Monotonicity
  - Sets: group is non-monotonic

    $\neg\exists x \in X\,(p(x))$
  - Counters: -1
  - I/O: Write is destructive
- Let's focus on sets and logic…
  - (We'll return to counters later)

http://www.flickr.com/photos/21649179@N00/9695799592/

MONOTONICITY

Monotonic Code

- Information accumulation
  - The more you know, the more you know
  - E.g. map, filter, join

Non-Monotonic Code

- Belief revision
  - New inputs can change your mind; need to "seal" input
  - E.g. reduce, aggregation, negation, state update

SEALING, TIME, SPACE

- Non-monotonicity: sealing a world

$$\neg\exists x \in X\,(\,p(x)\,) \Leftrightarrow \forall x \in X(\neg p(x)\,)$$

- Time: a mechanism to seal fate

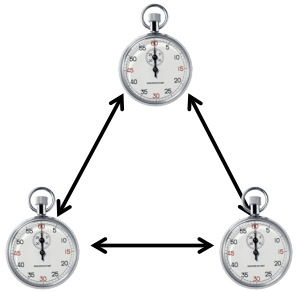
> "Time is what keeps everything from happening at once."
>
> — Ray Cummings

SEALING, TIME, SPACE

- Non-monotonicity: sealing a world

$$\neg\exists x \in X\,(\,p(x)\,) \Leftrightarrow \forall x \in X(\neg p(x)\,)$$

- Time: a mechanism to seal fate
- Space: multiple perceptions of time
- Coordination: sealing in time and space



SEALING, TIME, SPACE

INTUITION: SETS AGAIN

State change in the land of sets

MUTABLE SETS

- Introduce time into each relation

  shirt('Joe', 'black', 1)

- Persistence is induction

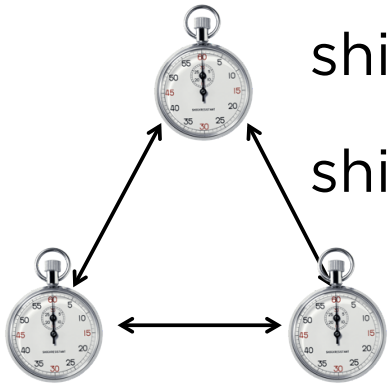
  shirt(x, y, t+1) <= shirt(x, y, t)

- Mutation via negation

  shirt(x, y, t

  shirt(x, z, t+1) <= new_shirt(x, z, t), del_shirt(x, y, t)

"Time is what keeps everything from happening at once."



[Statelog: Ludäscher 95, Dedalus: Alvaro '11]

MUTABLE SETS

- What have we achieved?
  - Time seals fate, prevents paradox
- When can we collapse time?
  - In a language of sets?
  - What about in other languages?

THE CALM THEOREM

- Monotonic => Consistent
  - Accumulative, disorderly computing.
  - Confluence.
  - The log-shipping pattern
- ¬Monotonic => ¬Consistent
  - *Inherent* non-monotonicity requires sealing
  - The reason for coordination

[The Declarative Imperative: Hellerstein '09]

VARIATIONS ON A THEOREM

- Transducers
  - Abiteboul: M => C [PODS '11]
  - Ameloot: CALM [PODS '11]
  - Zinn: subtleties with 3-valued logic [ICDT '12]
- Model Theory
  - Marczak: M=>C, NM+Coord=>C [Datalog 2.0 '12]

COROLLARY: CRON

- Recall Lamport's "causality"
  - Transitive "happens-before" relation on messages and events
  - Causal order: "Sensible" partial order
- CRON
  - Causality Required Only for Non-Monotonicity

[The Declarative Imperative: Hellerstein '09]

THE GRANDFATHER PARADOX

LOG REPLAY

COMPLEXITY

- What can we say with Monotonic logic?
  - [Immerman ’82], [Vardi ’82]: PTIME!!!
- Coordination Complexity
  - Characterize algorithms by coordination rounds
  - MP Model [Koutris, Suciu PODS ’11], and queries with a single round of coordination

OUTLINE

- Motivation
- CALM: Positive Theory
- Bloom: Disorderly Programming
  - Base Language
  - Lattices
  - Tools and Extensions

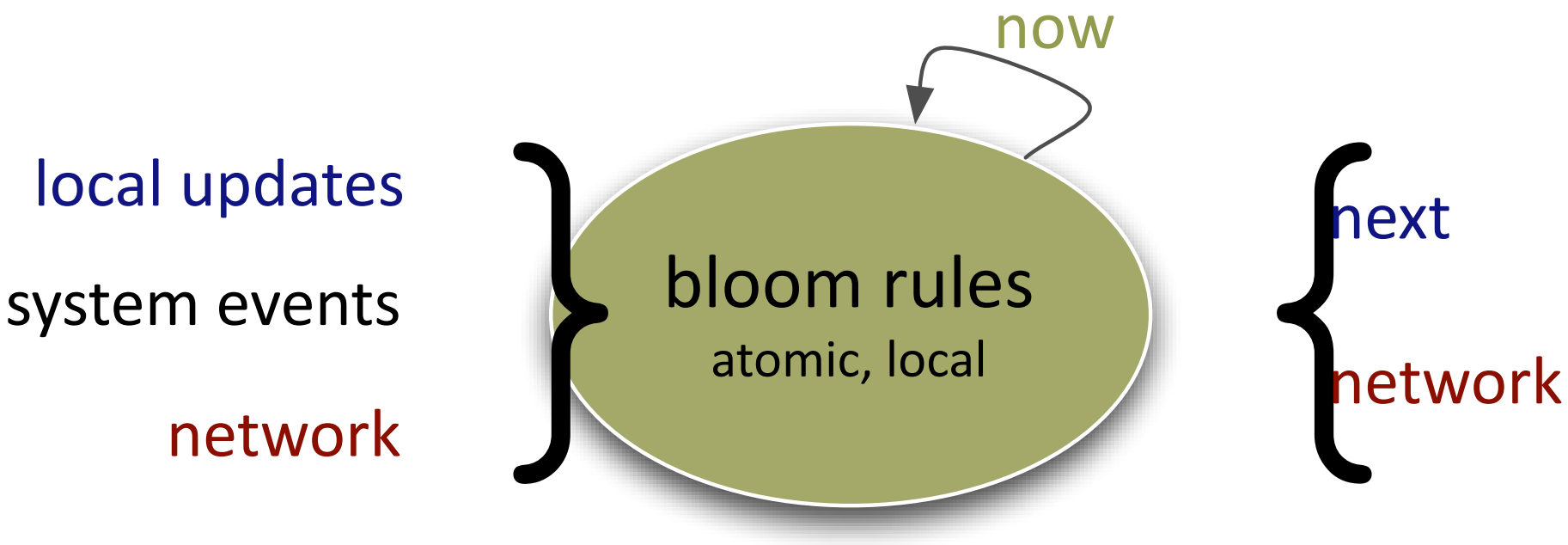
<~ bloom

- A disorderly language of data, space and distributed time
- Based on Alvaro's Dedalus logic

[Hellerstein, CIDR '11]

OPERATIONAL MODEL

- Nodes with local clocks, state
- Timestep at each node:

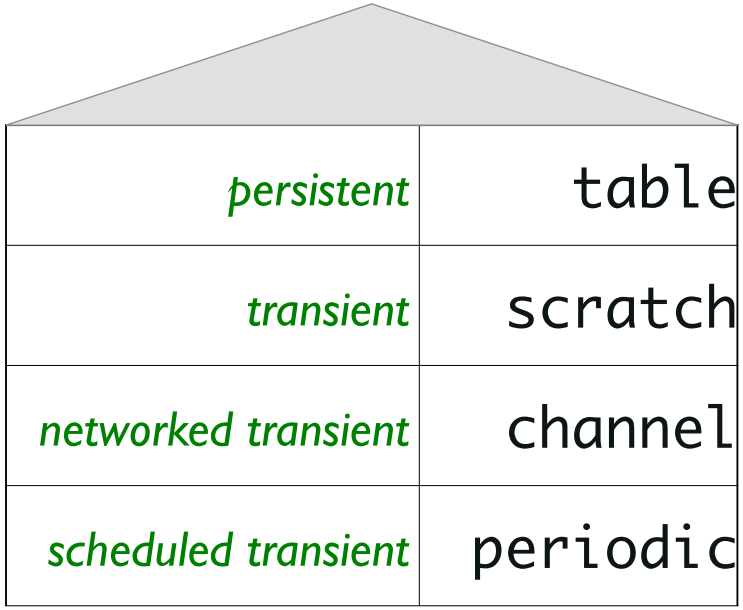


SYNTAX

<object> *<merge>* *<expression>*

SYNTAX

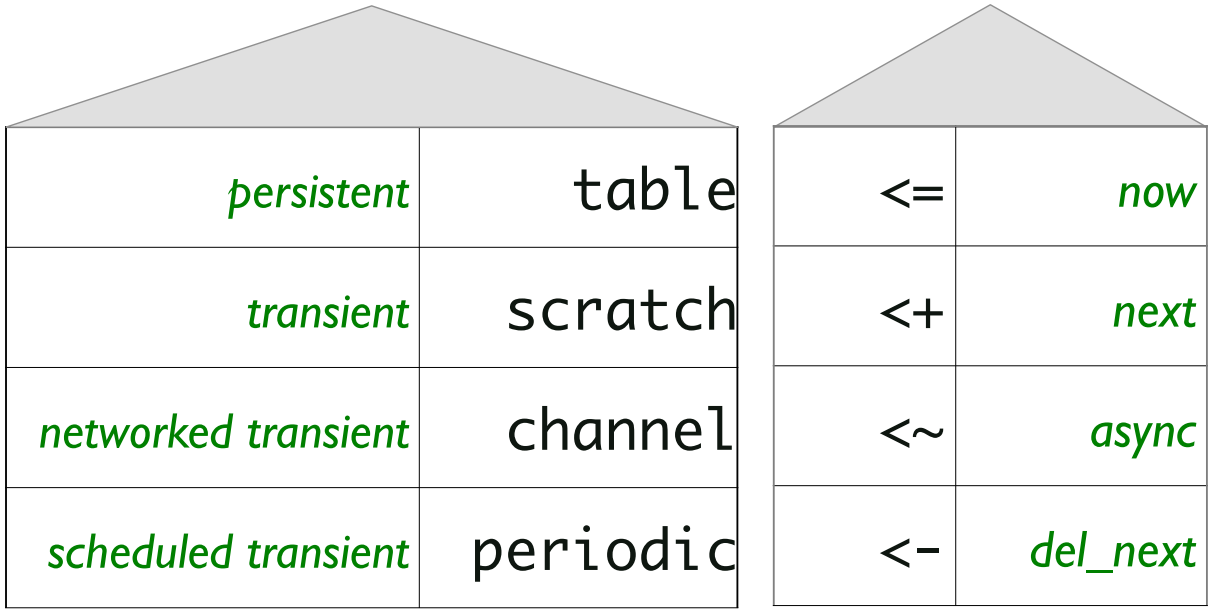
<object> *<merge>* *<expression>*



persistent	`table`
transient	`scratch`
networked transient	`channel`
scheduled transient	`periodic`

SYNTAX

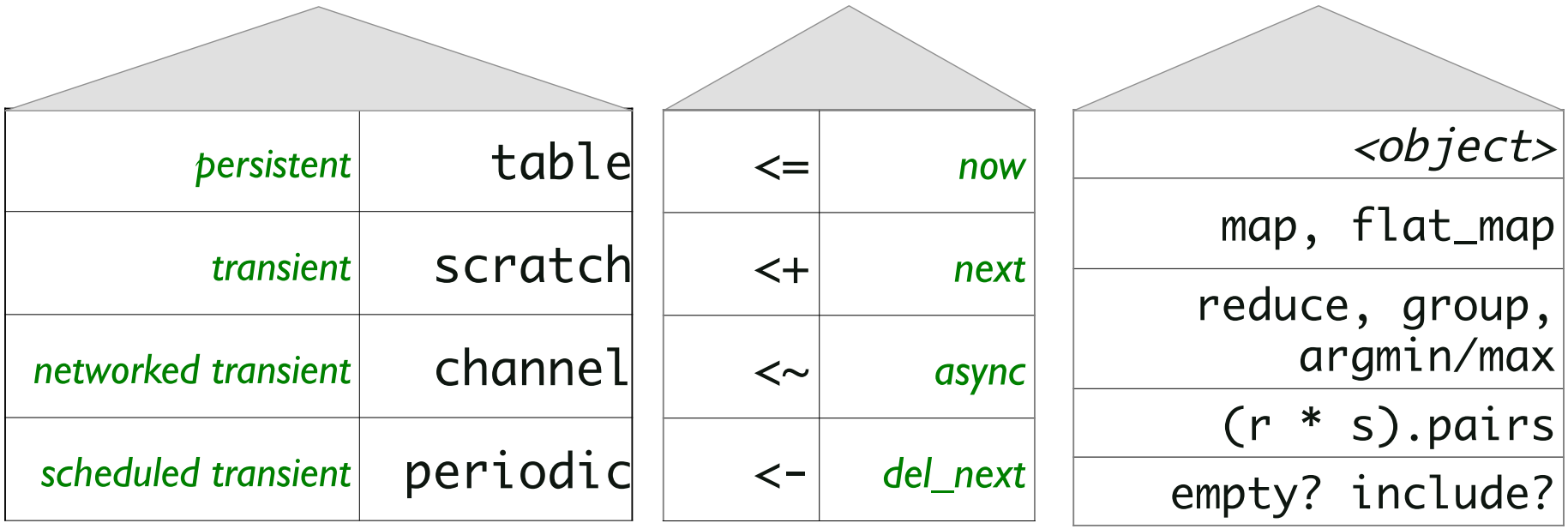
<object> *<merge>* *<expression>*



persistent	`table`
transient	`scratch`
networked transient	`channel`
scheduled transient	`periodic`

`<=`	*now*
`<+`	*next*
`<~`	*async*
`<-`	*del_next*

SYNTAX

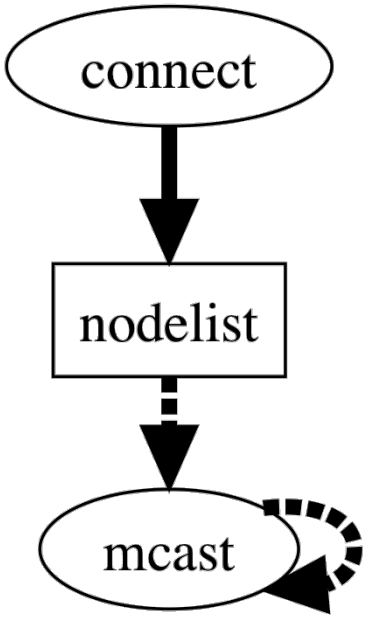

<object>
<merge>
<expression>
persistent
table
transient
scratch
networked transient
channel
scheduled transient
periodic
<=
now
<+
next
<~
async
<-
del_next
<object>
map, flat_map
reduce, group, argmin/max
(r * s).pairs
empty? include?

a chat server

```
module ChatServer
  state do
    table   :nodelist
    channel :mcast; channel :connect
  end

  bloom do
    nodelist <= connect.payloads
    mcast <~ (mcast*nodelist).pairs do |m,n|
      [n.key, m.val]
    end
  end
end
```

a chat server

```
module ChatServer
  state do
    table   :nodelist
    channel :mcast; channel :connect
  end

  bloom do
    nodelist <= connect.payloads
    mcast <~ (mcast*nodelist).pairs do |m,n|
      [n.key, m.val]
    end
  end
end
```



SHOPPING AT AMAZON

[DeCandia et al. 2007]

“Destructive” Cart

- Mutable cart triply-replicated
- Each update coordinated
- Checkout coordinated

Disorderly Cart

- Cart log triply replicated
- Log updates lazily propagated
- Checkout tally coordinated

CALM ANALYSIS

- Dataflow analysis
- Syntax checks for non-monotonic flows
- Asynchrony → non-monotonicity
  - Danger! Races.
- Alvaro diagrams highlight problems

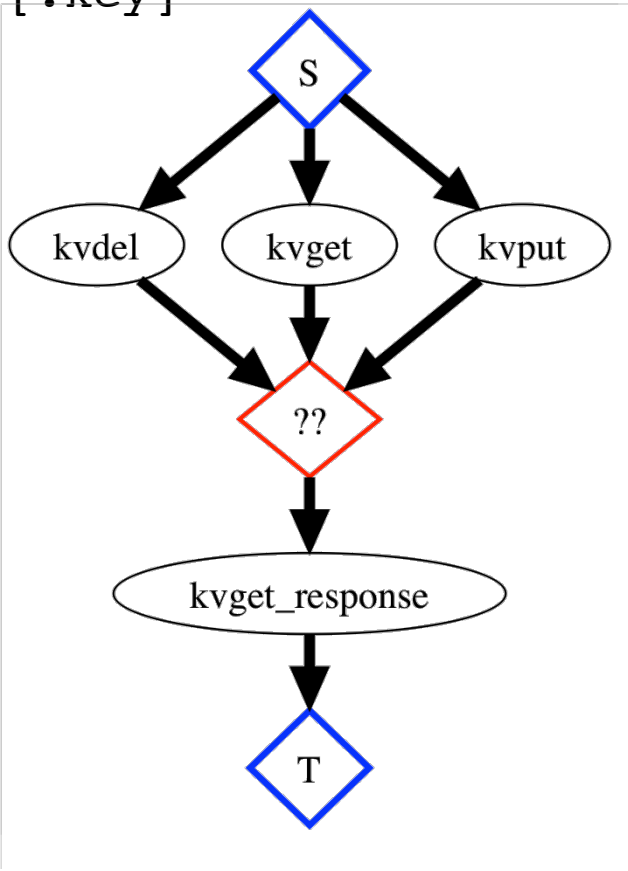
[Hellerstein, CIDR ‘11]

a simple key/value store

```
module KVSProtocol
  state do
    interface input, :kvput, [:key] => [:reqid, :value]
    interface input, :kvdel, [:key] => [:reqid]
    interface input, :kvget, [:reqid] => [:key]
    interface output, :kvget_response,
        [:reqid] => [:key, :value]
  end
end
```

a simple key/value store

```
module KVSProtocol
  state do
    interface input, :kvput, [:key] => [:reqid, :value]
    interface input, :kvdel, [:key] => [:reqid]
    interface input, :kvget, [:reqid] => [:key]
    interface output, :kvget_response,
        [:reqid] => [:key, :value]
  end
end
```

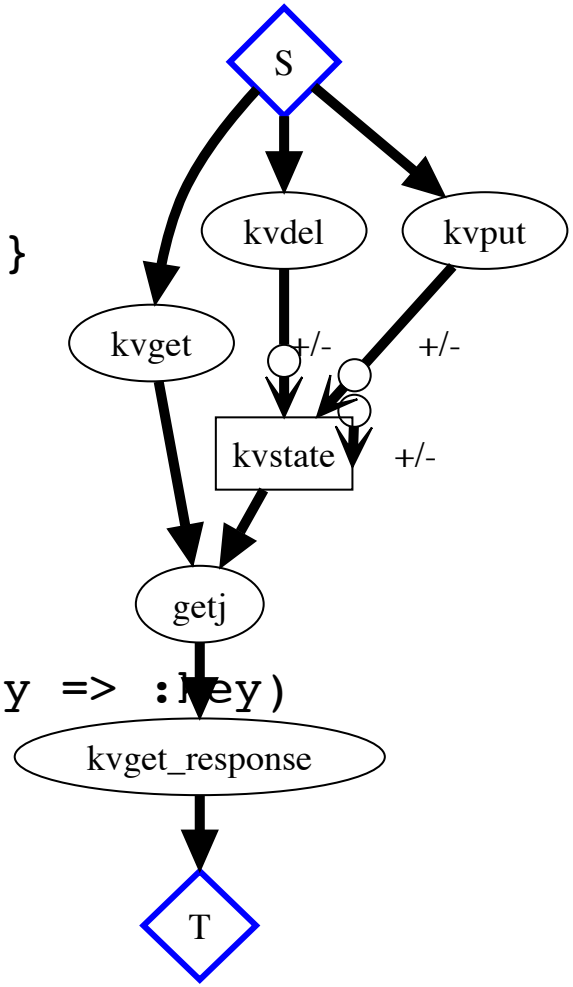


a simple key/value store

```
module BasicKVS
  include KVSProtocol

  state { table :kvstate, [:key] => [:value] }

  bloom do
    # mutate
    kvstate <+- kvput {|s| [s.key, s.value]}

    # get
    temp :getj <= (kvget * kvstate).pairs(:key => :key)
    kvget_response <= getj do |g, t|
      [g.reqid, t.key, t.value]
    end

    # delete
    kvstate <- (kvstate * kvdel).lefts(:key => :key)
  end
end
```

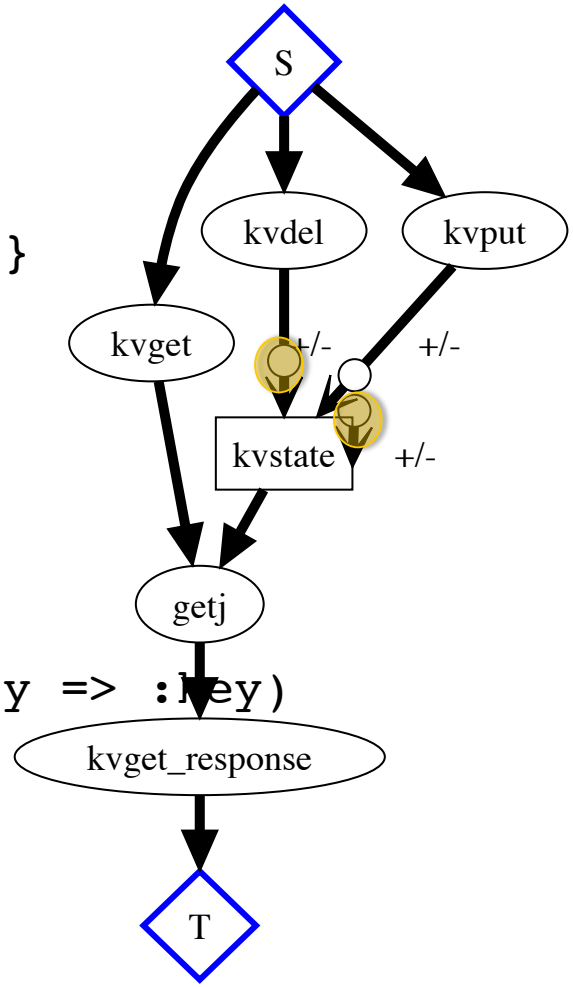
a simple key/value store

```
module BasicKVS
  include KVSProtocol

  state { table :kvstate, [:key] => [:value] }

  bloom do
    # mutate
    kvstate <+- kvput {|s| [s.key, s.value]}

    # get
    temp :getj <= (kvget * kvstate).pairs(:key => :key)
    kvget_response <= getj do |g, t|
      [g.reqid, t.key, t.value]
    end

    # delete
    kvstate <- (kvstate * kvdel).lefts(:key => :key)
  end
end
```



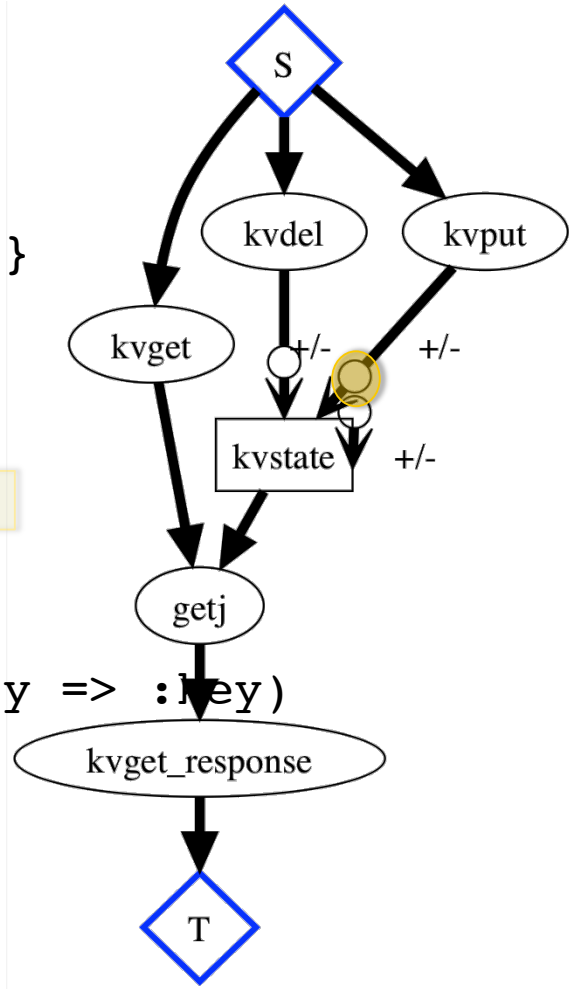
a simple key/value store

```
module BasicKVS
  include KVSProtocol

  state { table :kvstate, [:key] => [:value] }

  bloom do
    # mutate
    kvstate <+- kvput {|s| [s.key, s.value]}

    # get
    temp :getj <= (kvget * kvstate).pairs(:key => :key)
    kvget_response <= getj do |g, t|
      [g.reqid, t.key, t.value]
    end

    # delete
    kvstate <- (kvstate * kvdel).lefts(:key => :key)
  end
end
```



a simple key/value store

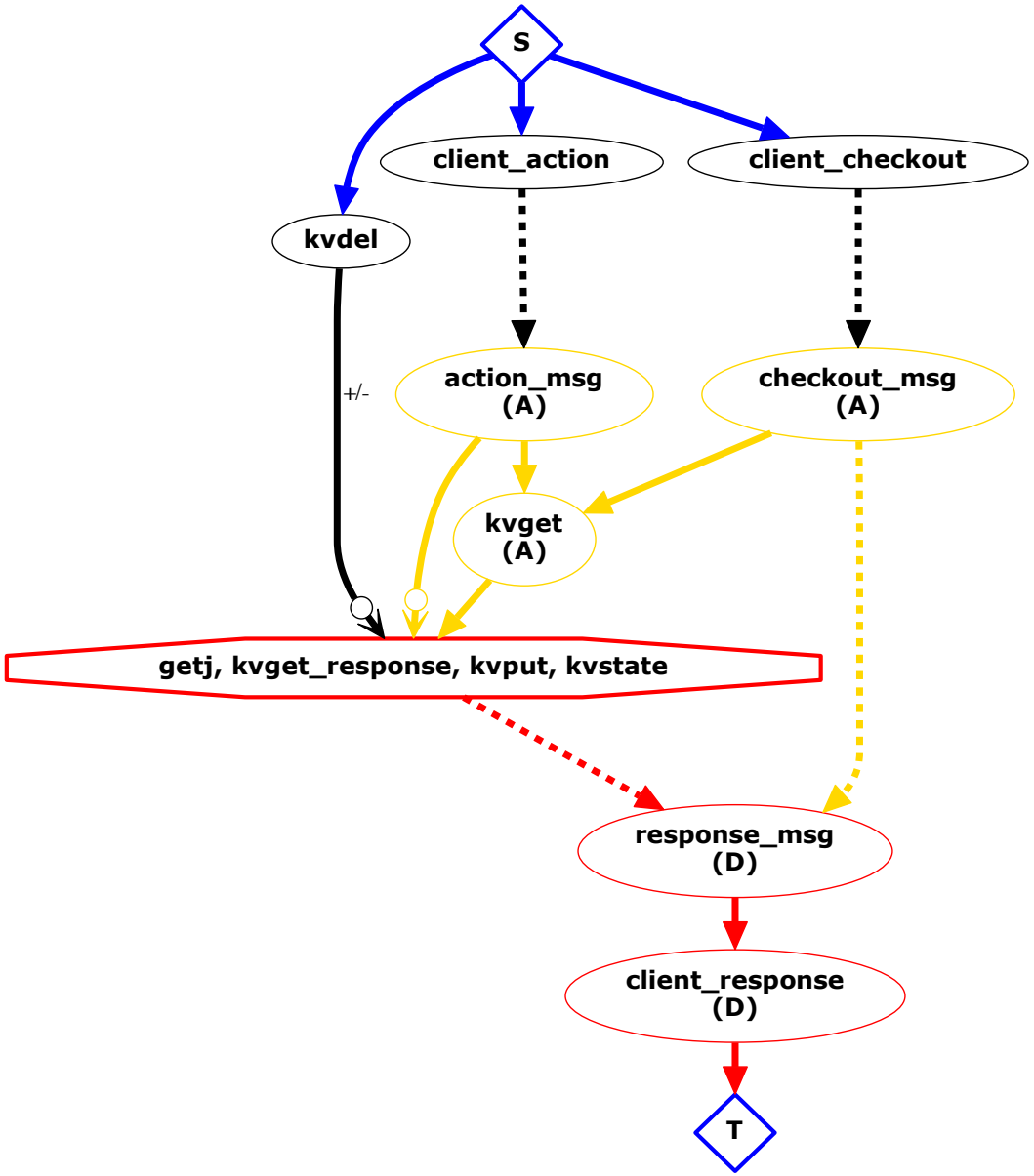
```
module BasicKVS
  include KVSProtocol

  state { table :kvstate, [:key] => [:value] }

  bloom do
    # mutate
    kvstate <+- kvput {|s| [s.key, s.value]}

    # get
    temp :getj <= (kvget * kvstate).pairs(:key => :key)
    kvget_response <= getj do |g, t|
      [g.reqid, t.key, t.value]
    end

    # delete
    kvstate <- (kvstate * kvdel).lefts(:key => :key)
  end
end
```



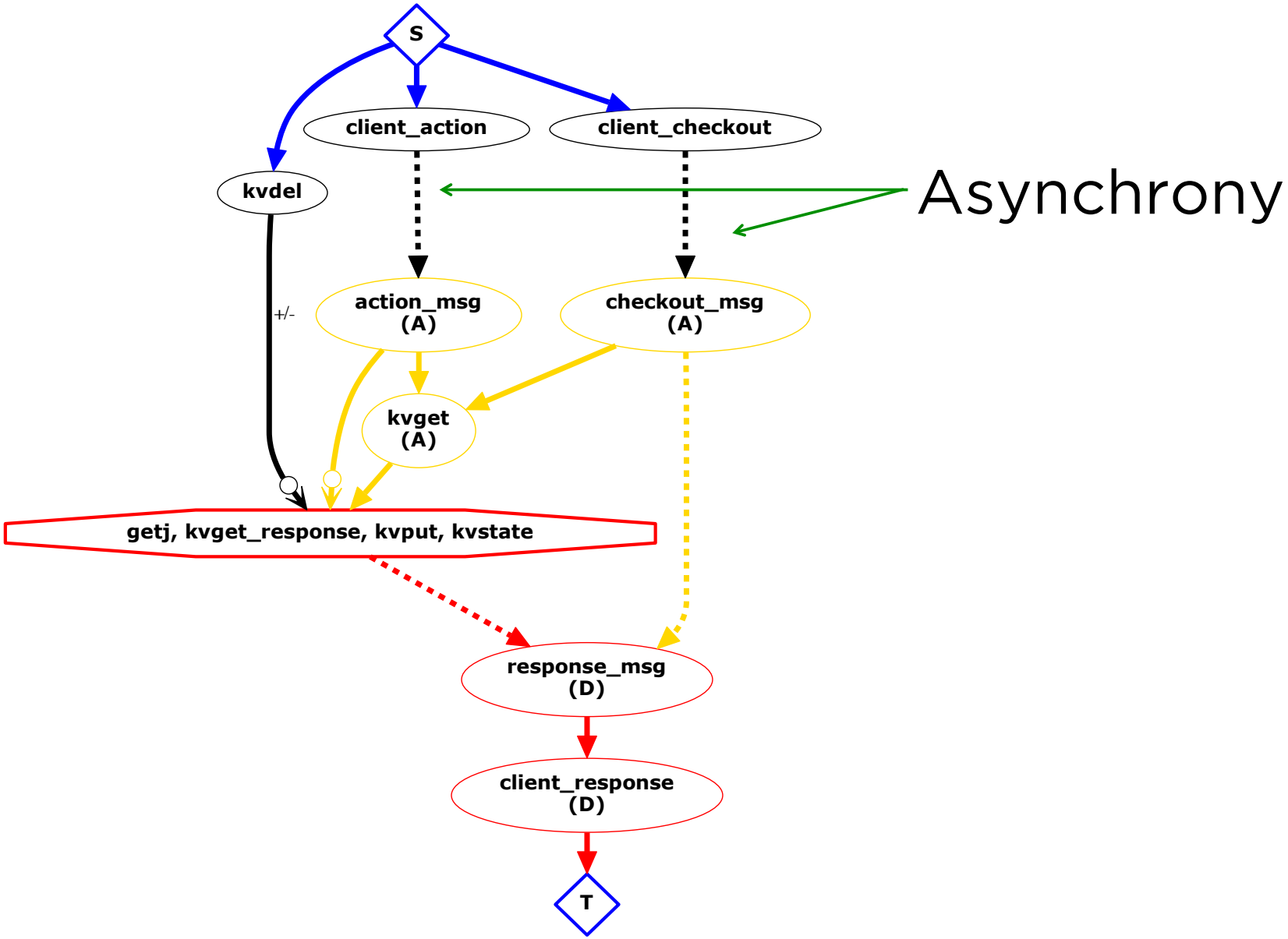
``destructive" cart

```
module DestructiveCart
  include CartProtocol
  include KVSProtocol

  bloom :on_action do
    kvget <= action_msg {|a| [a.reqid, a.session] }
    kvput <= (action_msg * kvget_response).outer(:reqid => :reqid) do |a,r|
      val = r.value || {}
      [a.client, a.session, a.reqid,
       val.merge({a.item => a.cnt}) {|k,old,new| old + new}]
    end
  end

  bloom :on_checkout do
    kvget <= checkout_msg {|c| [c.reqid, c.session] }
    response_msg <~ (kvget_response * checkout_msg).pairs(:reqid => :reqid) do |
r,c|
      [c.client, c.server, r.key, r.value.select {|k,v| v > 0}.sort]
    end
  end
end
```

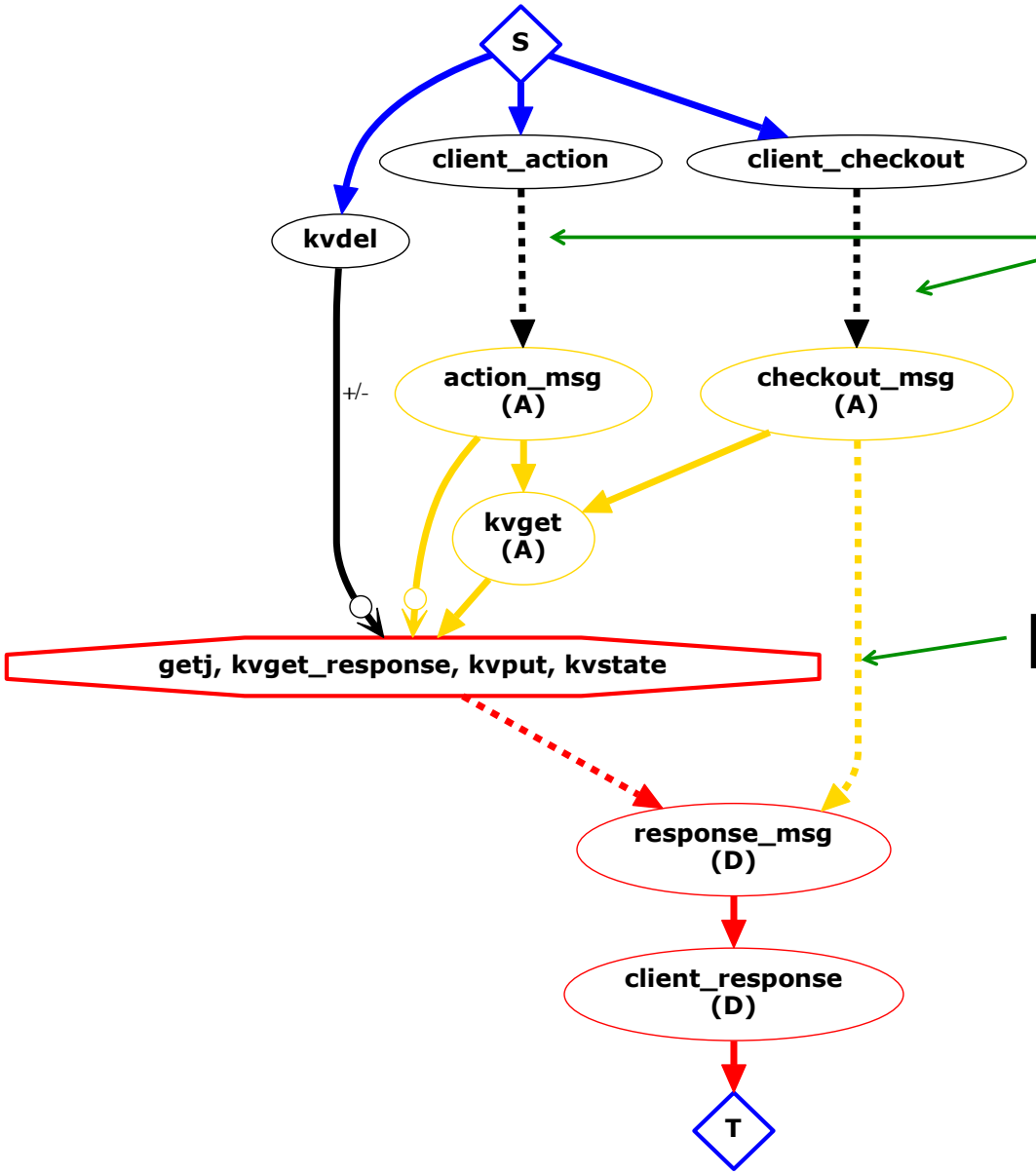
``destructive'' cart



``destructive'' cart



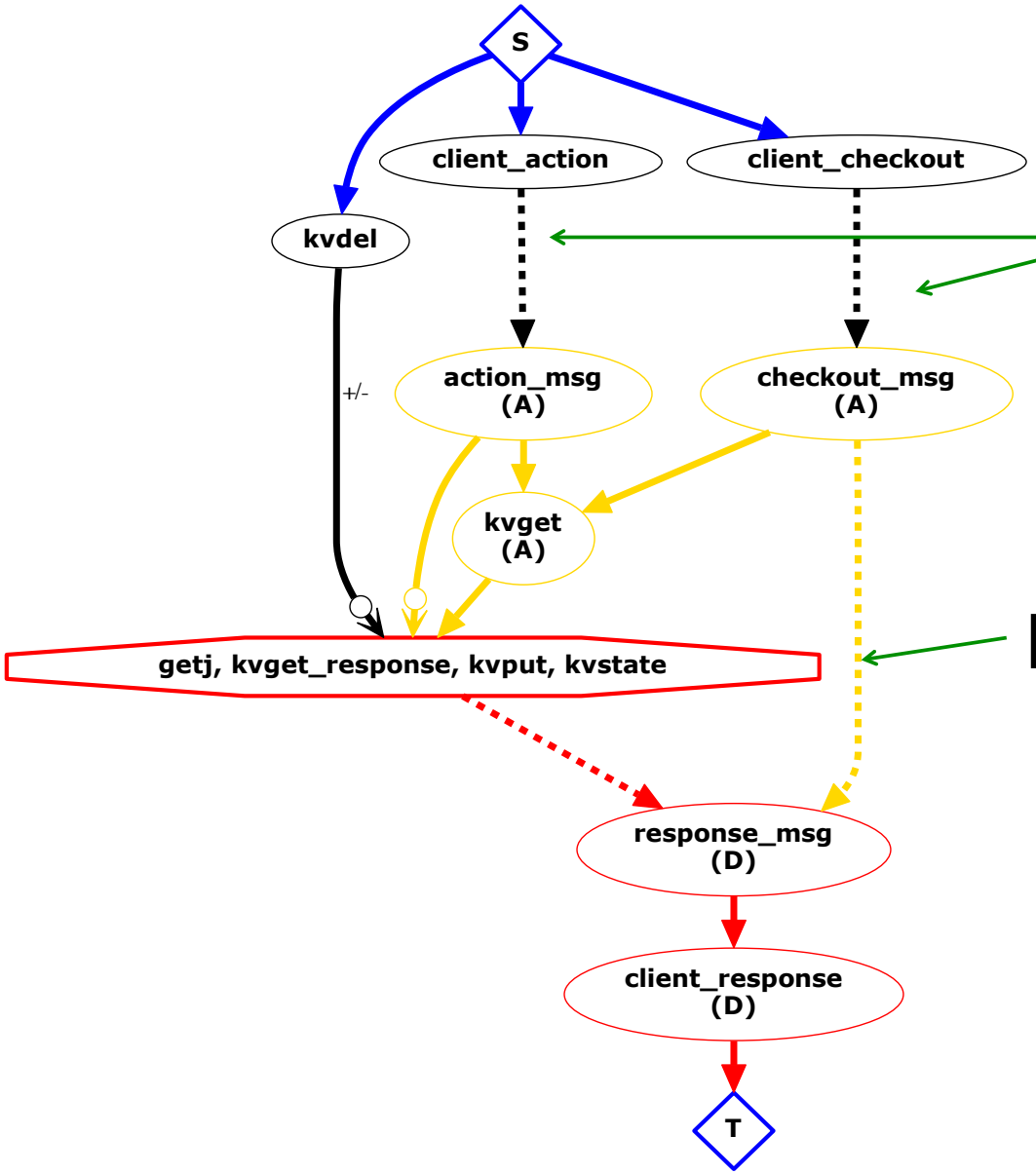
``destructive" cart



Asynchrony

Non-monotonicity

``destructive" cart

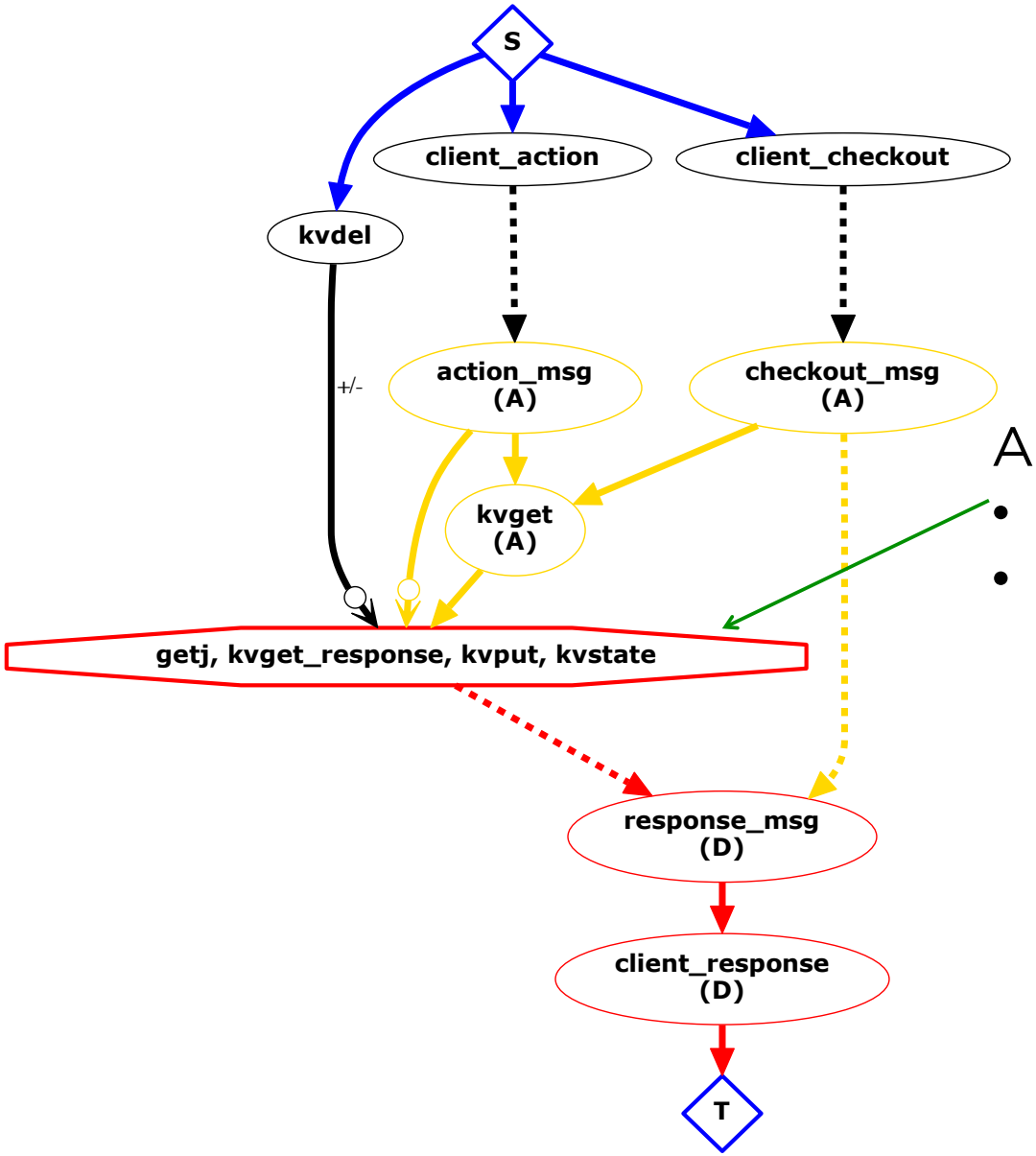


Asynchrony

Non-monotonicity

Divergent Results?

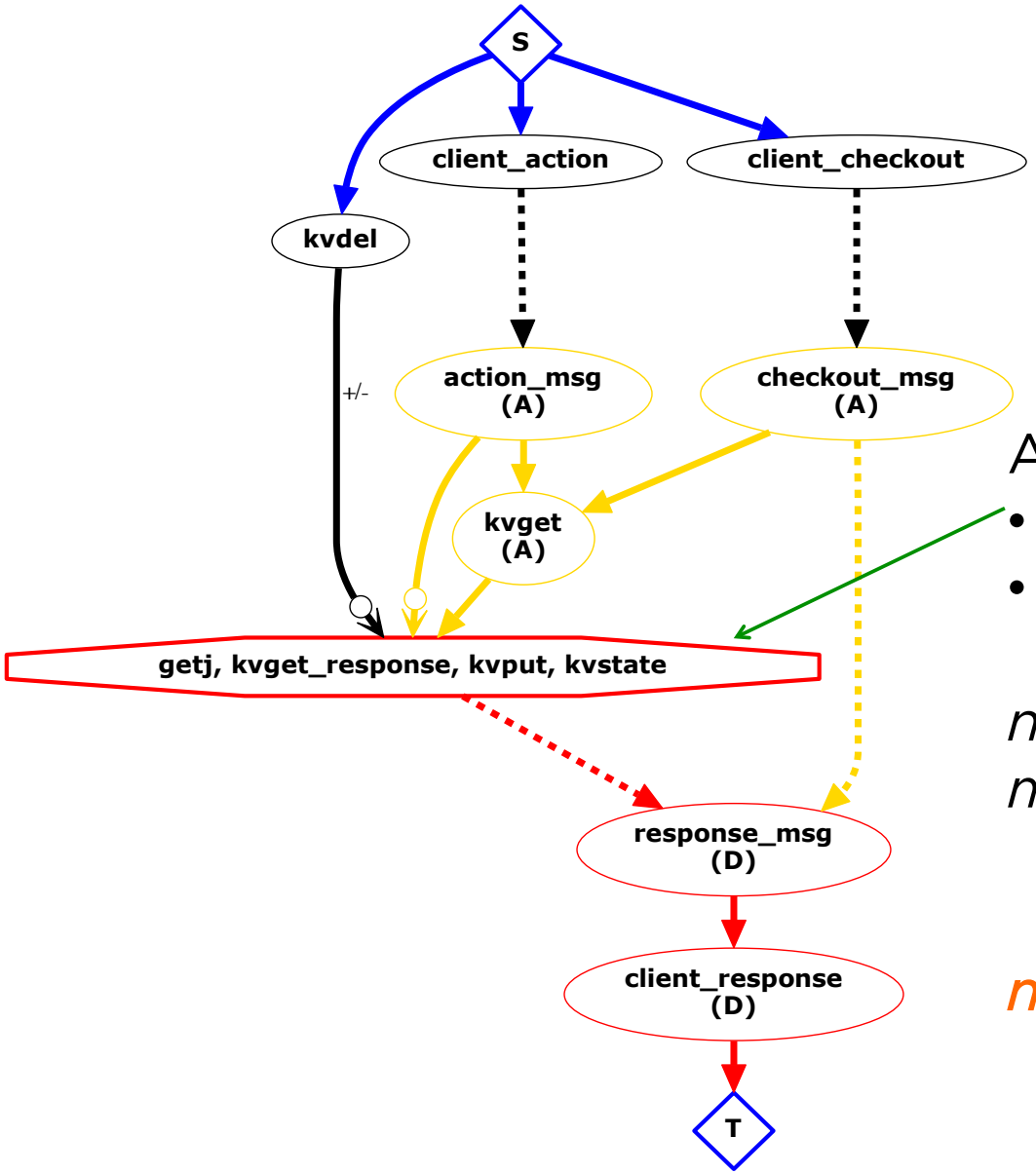
``destructive" cart



Add coordination; e.g.,

- synchronous replication
- Paxos

``destructive" cart



Add coordination; e.g.,

- synchronous replication
- Paxos

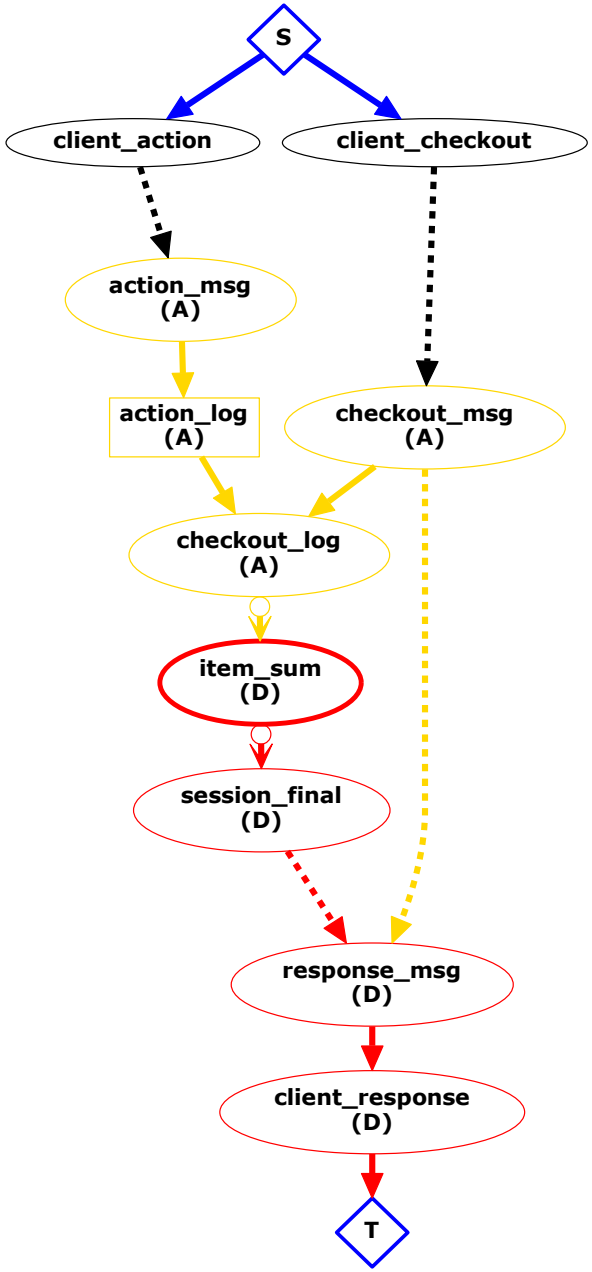
n = |client_action|
m = |client_checkout| = 1

n rounds of coordination

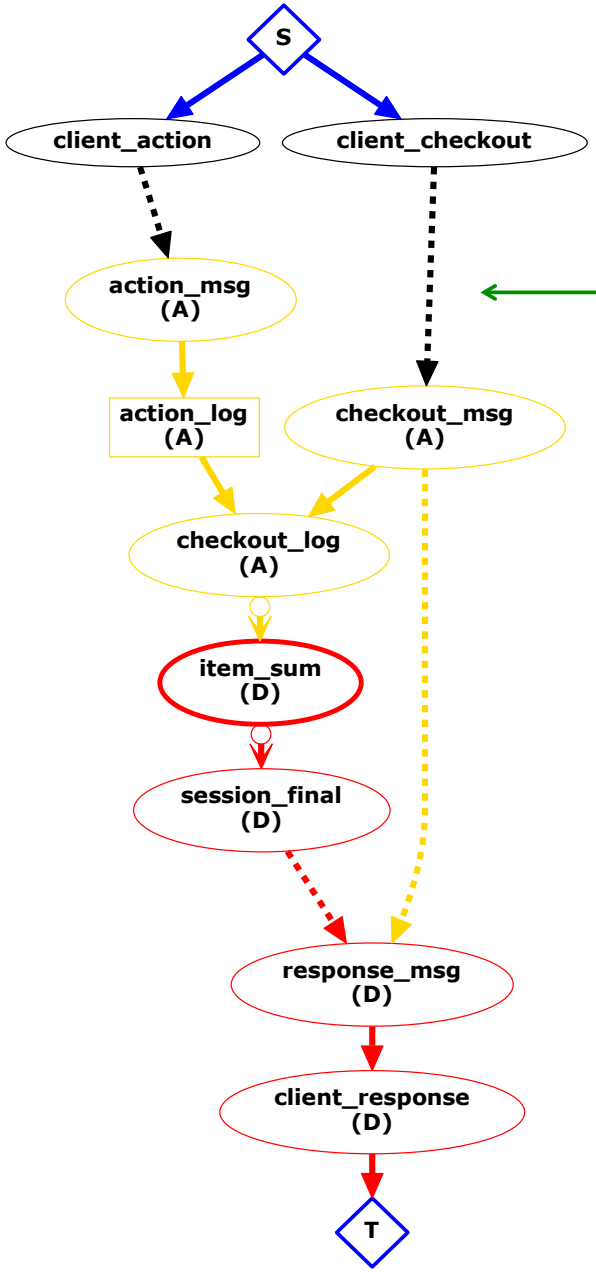
``disorderly cart"

```
module DisorderlyCart
  include CartProtocol

  state do
    table :action_log, [:session, :reqid] => [:item, :cnt]
    scratch :item_sum, [:session, :item] => [:num]
    scratch :session_final, [:session] => [:items, :counts]
  end

  bloom :on_action do
    action_log <= action_msg {|c| [c.session, c.reqid, c.item, c.cnt] }
  end

  bloom :on_checkout do
    temp :checkout_log <= (checkout_msg * action_log).rights(:session => :session)
    item_sum <= checkout_log.group([:session, :item], sum(:cnt)) do |s|
      s if s.last > 0 # Don't return items with non-positive counts.
    end
    session_final <= item_sum.group([:session], accum_pair(:item, :num))
    response_msg <~ (session_final * checkout_msg).pairs(:session => :session) do |c,m|
      [m.client, m.server, m.session, c.items.sort]
    end
  end
end
```

disorderly cart analysis

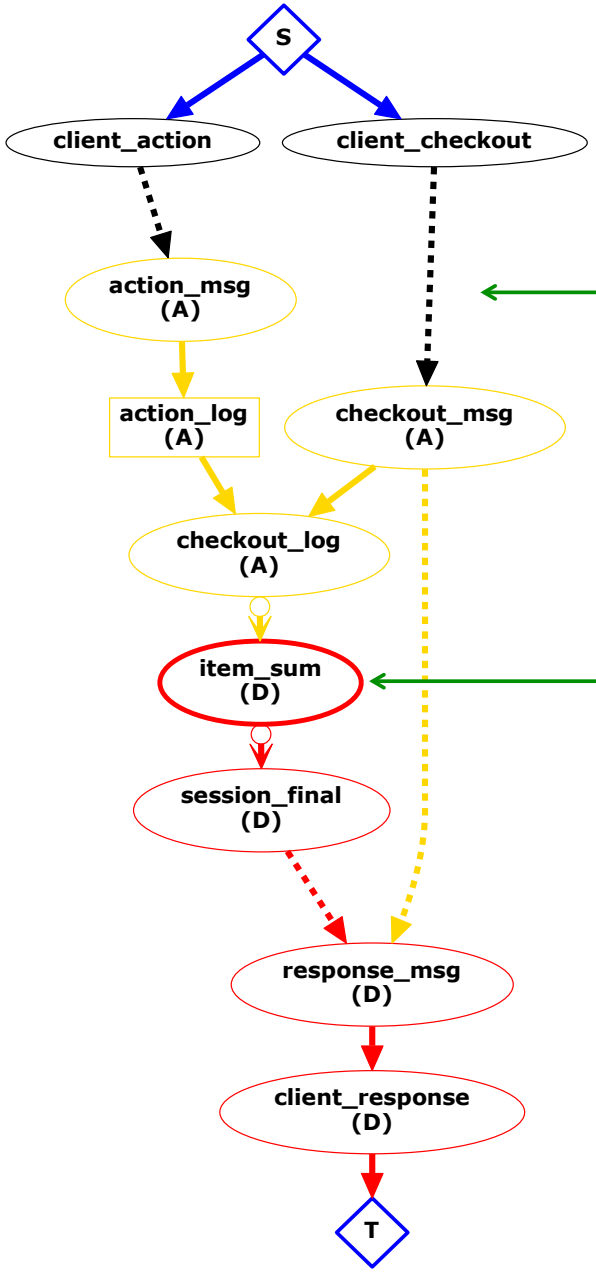


disorderly cart analysis



Asynchrony

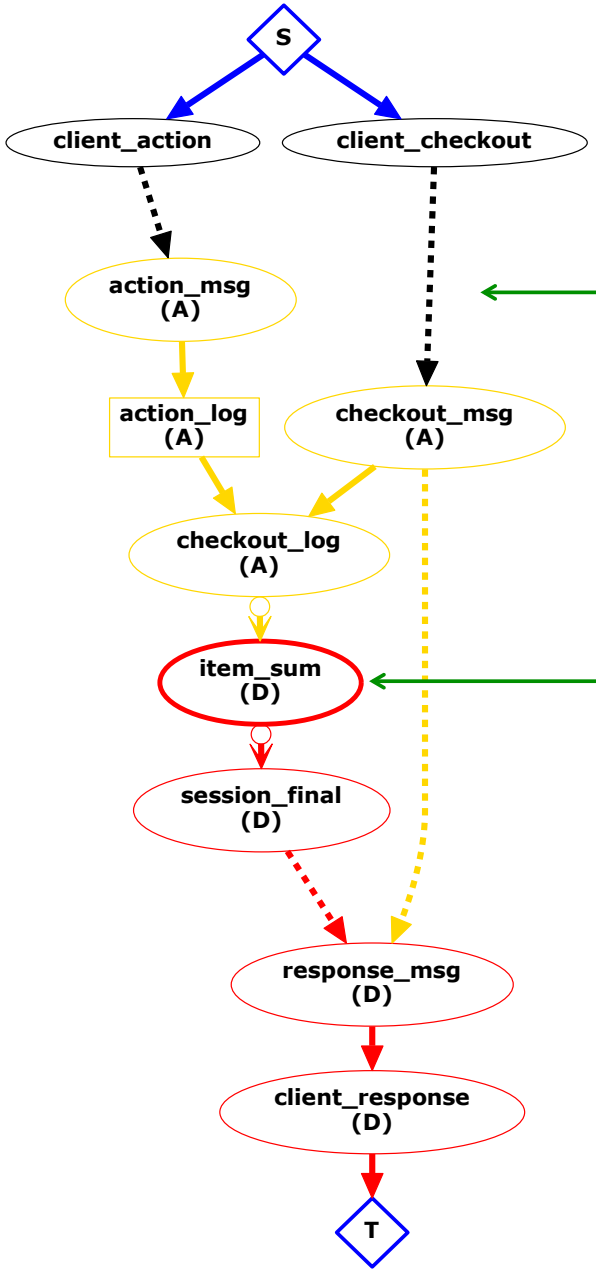
disorderly cart analysis



Asynchrony

Non-monotonicity

disorderly cart analysis

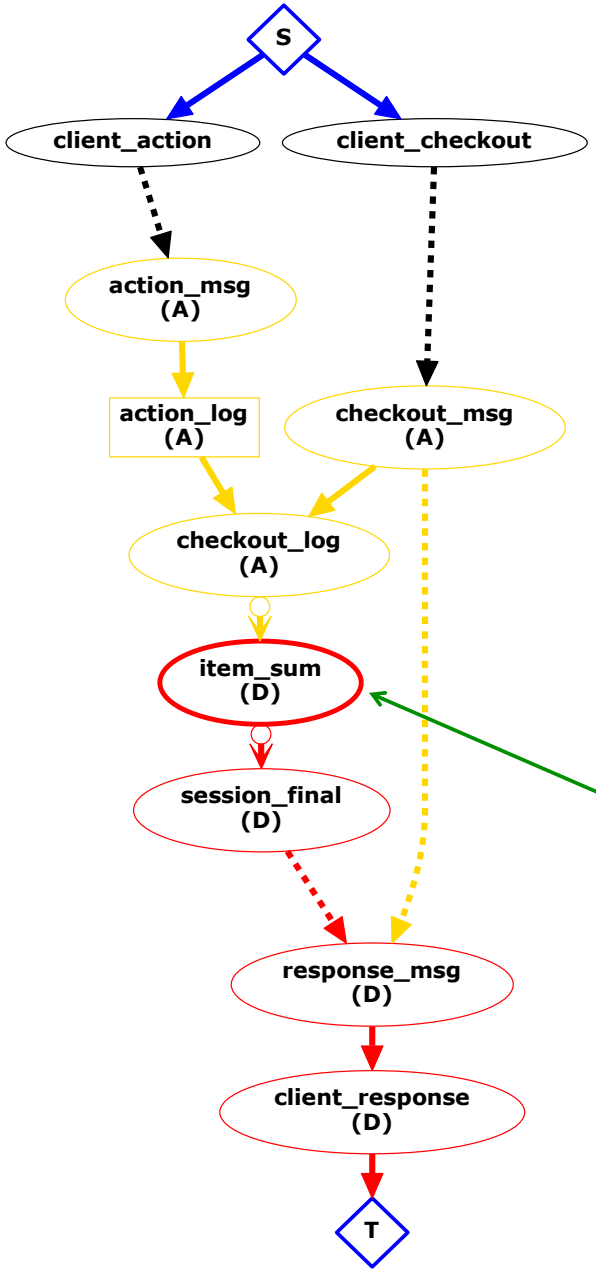


Asynchrony

Non-monotonicity

Divergent Results?

disorderly cart analysis



n = |client_action|
m = |client_checkout| = 1

m=1 round of coordination

OUTLINE

- Motivation
- CALM: Positive Theory
- Bloom: Disorderly Programming
  - Base Language
  - Lattices
  - Tools and Extensions

BEYOND COLLECTIONS

- What's so great about sets?
  - Order insensitive (union Commutes)
  - Batch insensitive (union Associates)
  - Retry insensitive (union "Idempotes")
- Design pattern: "ACID 2.0"
  - Can we apply the idea elsewhere?

BOUNDED JOIN SEMILATTICES

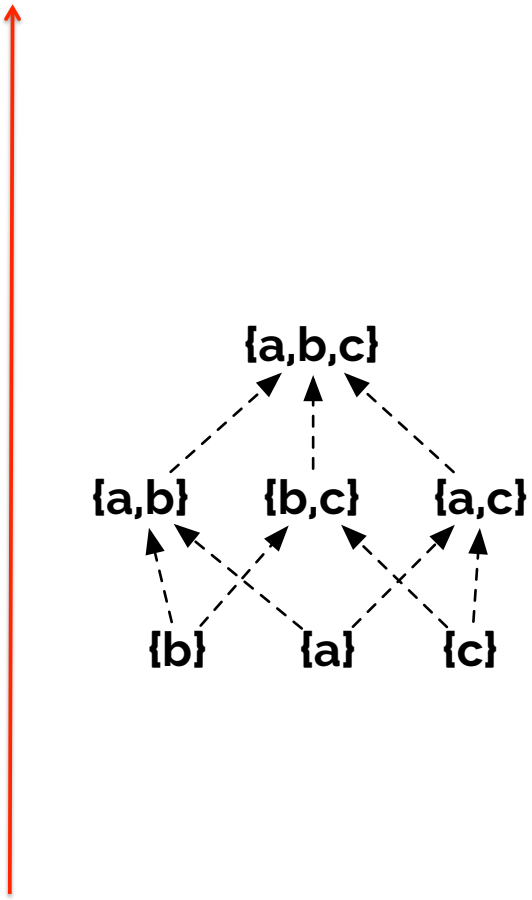
A pair $\langle S, \vee \rangle$ such that:

- S is a set
- $\vee$ is a binary operator ("*least upper bound*")
  - Associative, Commutative, and Idempotent
  - Induces a partial order on S:
    $x \leq_S y$ if $x \vee y = y$

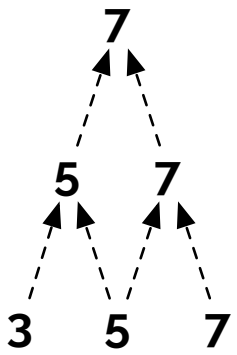
BOUNDED JOIN SEMILATTICES: PRACTICE

- Objects that grow over time
- Have an interface with an ACI merge method
  - Bloom's "Object <= expression"

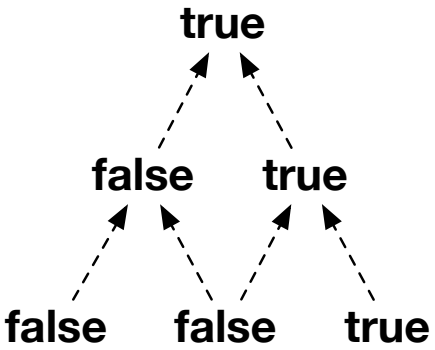
Time



Set
(Merge = *Union*)



Increasing Int
(Merge = *Max*)



Boolean
(Merge = *Or*)

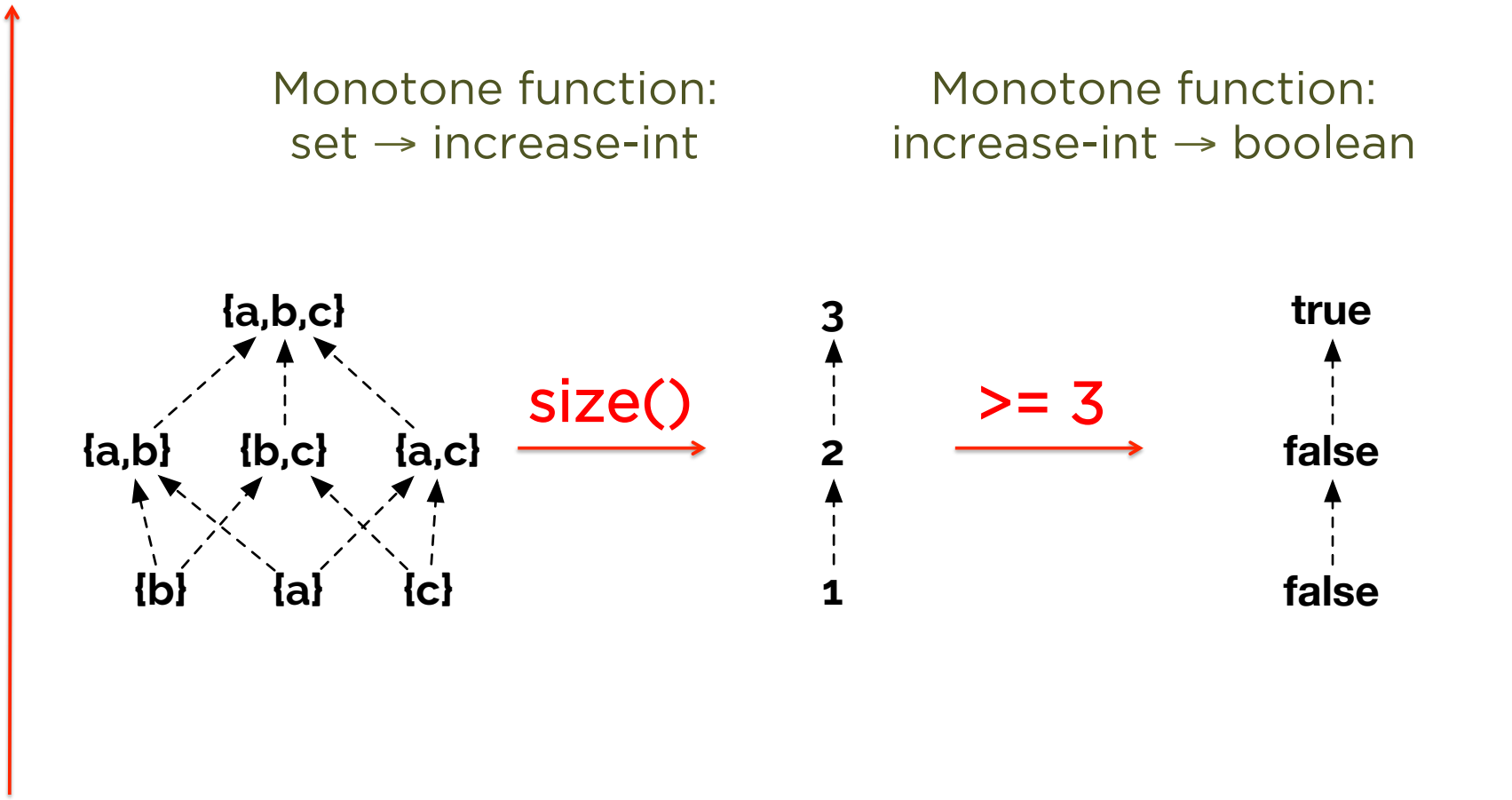
BEYOND OBJECTS

- Lattices represent disorderly data
- What about disorderly computation?

$f : S \to T$ is a ***monotone function*** iff:

$$f(a \vee_S b) = f(a) \vee_T f(b)$$

Time


Monotone function:
set → increase-int
Monotone function:
increase-int → boolean
{a,b,c}
{a,b}
{b,c}
{a,c}
{b}
{a}
{c}
size()
3
2
1
>= 3
true
false
false
Set
(Merge = *Union*)
Increasing Int
(Merge = *Max*)
Boolean
(Merge = *Or*)

BLOOM^L

- Bloom
- Collections => Lattices
- Monotone functions
- Non-monotone morphisms

[Conway, SOCC '12]

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.

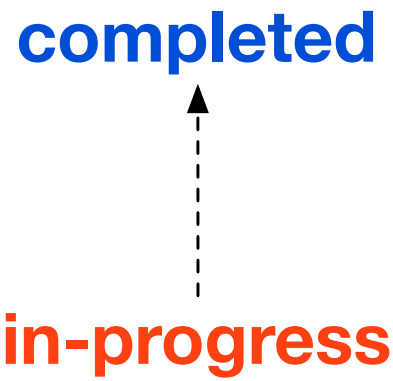
- Each time a process experiences an internal event, it increments its own logical clock in the vector by one.
- Each time a process prepares to send a message, it increments its own logical clock in the vector by one and then sends its entire vector along with the message being sent.

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).

ANOTHER EXAMPLE

Sealing the shopping cart

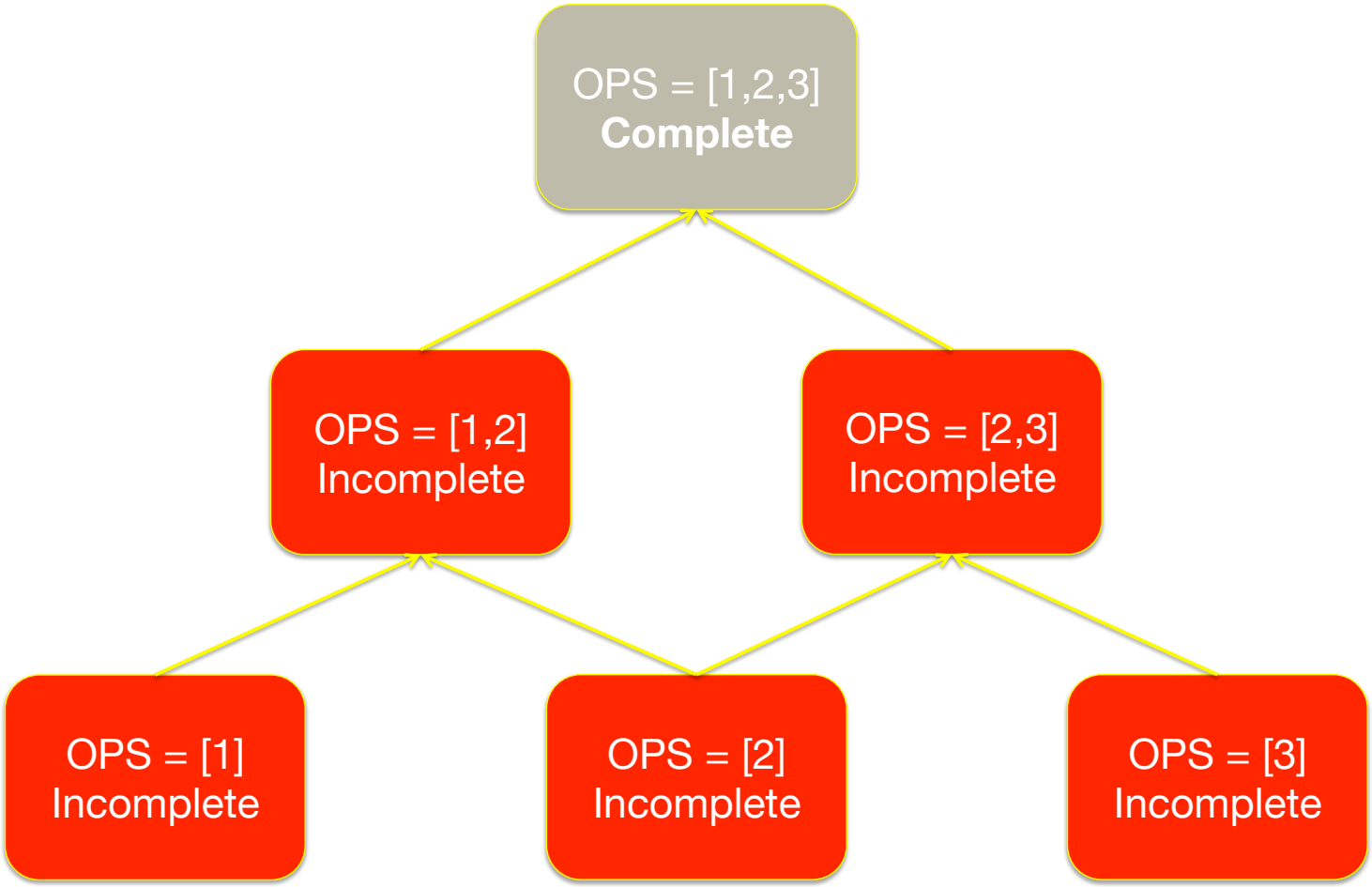
Observation:
Once a checkout occurs, no more shopping actions can be performed



Observation:

The client knows *locally* when a checkout can be processed “safely”

MONOTONE CHECKOUT



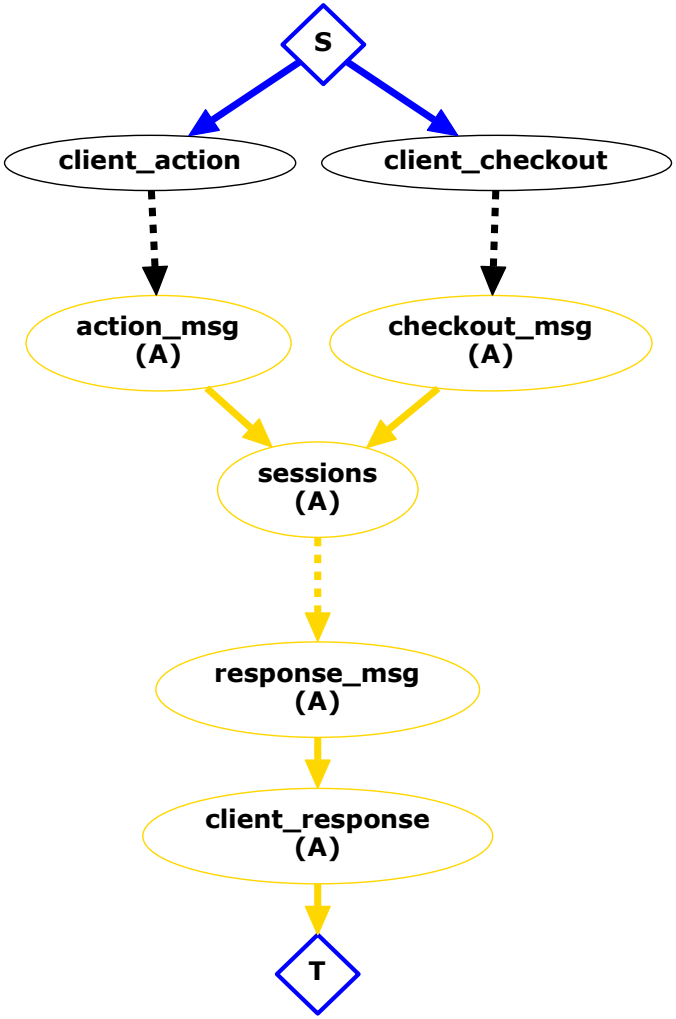
``monotone cart”

```
module MonotoneReplica
  include MonotoneCartProtocol

  state { lmap :sessions }

  bloom do
    sessions <= action_msg do |m|
      c = CartLattice.new({m.op_id => [ACTION_OP, m.item, m.cnt]})
      { m.session => c }
    end

    sessions <= checkout_msg do |m|
      c = CartLattice.new({m.op_id => [CHECKOUT_OP, m.lbound, m.addr]})
      { m.session => c }
    end

    response_msg <~ sessions.to_collection do |session, cart|
      cart.is_complete.when_true {
        [cart.checkout_addr, session, cart.summary]
      }
    end
  end
end
```

Accumulate

Wait for complete cart

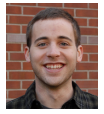
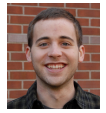
``monotone cart''



OUTLINE

- Motivation
- CALM: Positive Theory
- Bloom: Disorderly Programming
  - Base Language
  - Lattices
  - Tools and Extensions

TOOLS AND EXTENSIONS

- Blazes: Coordination Synthesis 
- BloomUnit: Declarative Testing 
- Edelweiss: Bloom and Grow 
- Beyond Confluence 
- Coordination-Avoiding Databases 

BLAZES

peter alvaro

- CALM Analysis & Coordination
  - Exploit *punctuations* for coarse-grained barriers
  - *Auto-synthesize* app-specific coordination
- Applications beyond Bloom
  - CALM for annotated “grey boxes” in dataflows
  - Applied to Bloom and to Twitter Storm

[Alvaro et al., ICDE13]

BLOOM UNIT: DECLARATIVE TESTING

peter alvaro

- Declarative Input/Output Specs
- Alloy-driven synthesis of interesting inputs
- CALM-driven collapsing of behavior space

[Alvaro et al., DBTest12]

EDELWEISS: BLOOM & GROW


neil conway

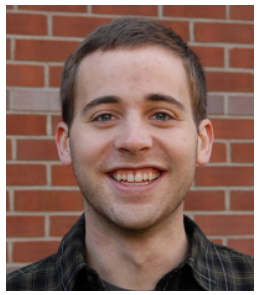
- Enforce the log-shipping pattern
- Bloom with *no deletion*
- Program-specific GC in Bloom?
  - Delivered message buffers
  - Persistent state eclipsed by new versions

[Conway et al., In Submission]

EXAMPLE PROGRAMS

	Number of Rules	
	Edelweiss	Bloom w/ Deletion
Reliable unicast	2	6
Reliable broadcast	2	10
Causal broadcast	6	15
Key-value store	5	23
Causal KVS	18	44
Atomic write transactions	5	14
Atomic read transactions	9	22

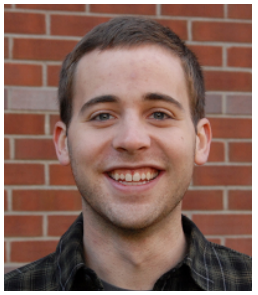
BEYOND CALM



peter bailis

- CALM focuses on eventual consistency
  - A “liveness” condition (eventually good)
- What about properties along the way?
  - “Safety” conditions (never bad)
- What about controlled non-determinism?
  - Consensus picks one winner, but needn’t be deterministic
- *Idea:* Confluence w.r.t. invariants

COORDINATION-AVOIDING DATABASES


peter bailis

- Faster databases with CALM?
- Yes! TPC-C with essentially no “locks”
- Outrageous performance/scalability

OUTLINE

- Motivation
- CALM: Positive Theory
- Bloom: Disorderly Programming

CALM DIRECTIONS

- Theory:
  - Formalize CALM for Bloom^L lattices
  - Harmonize the CALM proofs
  - Coordination "surface" complexity (expectation)
- Practice
  - Bloom 2.0: low latency, machine learning
  - Importing Bloom/CALM into current practice
    - Libraries, e.g. Immutable or Versioned memory
    - CALM program analysis for traditional languages.

STEPPING BACK

- CALM provides a framework
  - Disorderly opportunities
- Bloom as 1 concrete future direction
  - Well-suited to the domain
- Where to go next?

SW ENG OBSERVATIONS FROM (BIG) DATA

SW ENG OBSERVATIONS FROM (BIG) DATA

- Agility > Correctness
- Harbinger of things to come?
  - Design → Theory → Practice
- Concerns up the stack
  - Data-centric view of *all state*
  - Distribution (time!) as a primary concern

THOUGHTS

- Design patterns in the field
- Formalize and realize
- A great time for language design
  - DSLs and mainstream

MORE?

http://boom.cs.berkeley.edu

http://bloom-lang.org

hellerstein@berkeley.edu