Bloom:

Big Systems,
Small Programs

Neil Conway
UC Berkeley

Distributed Computing

Programming Languages

Data prefetching

Register allocation

Loop unrolling

Function inlining

Optimization

Global coordination, waiting

Caching, indexing

Replication, data locality

Partitioning, load balancing

Undeclared variables

Type mismatches

Sign conversion mistakes

Warnings

Replica divergence

Inconsistent state

Deadlocks

Race conditions

Stack traces

`gdb`

Log files, `printf`

Debugging

Full stack visualization, analytics

Consistent global snapshots

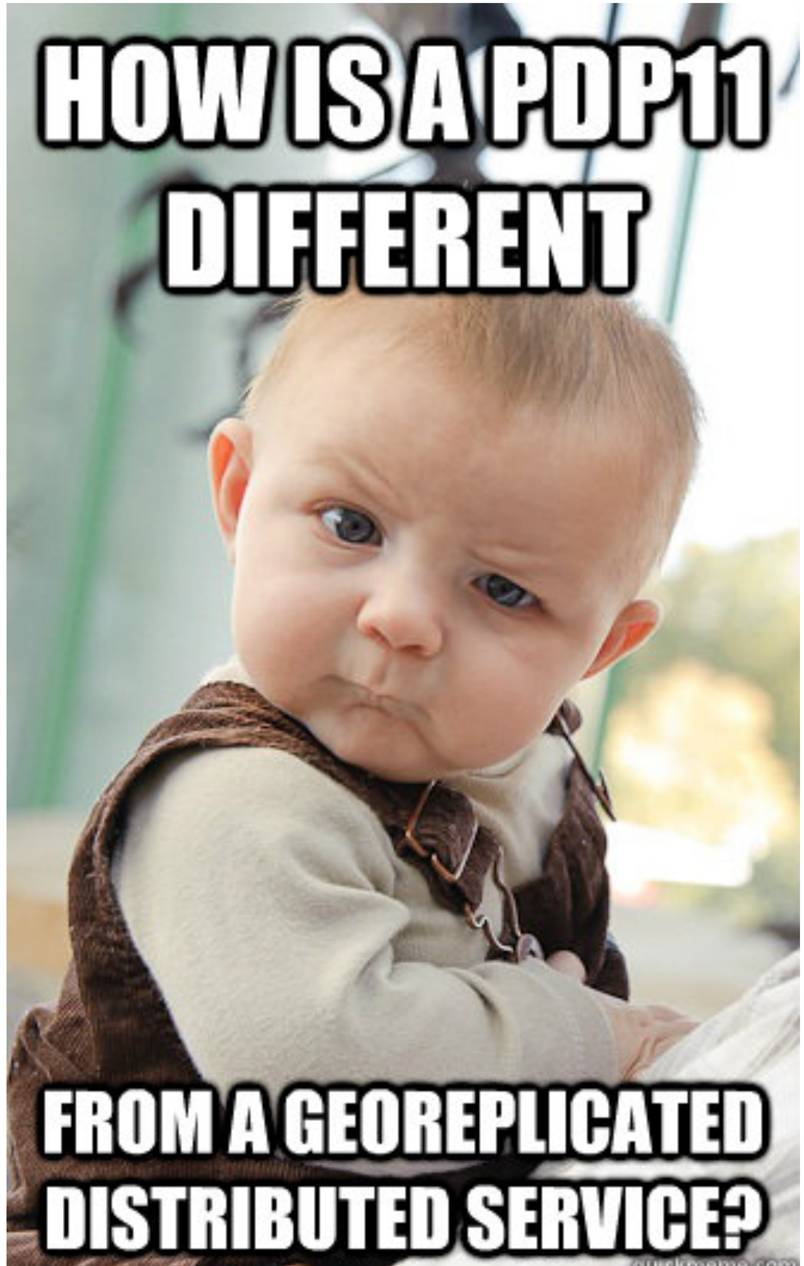
Provenance analysis

Developer **productivity** is a
major unsolved problem
in distributed computing.

We can do better!

… provided we're willing to make changes.

Design Principles


HOW IS A PDP11
DIFFERENT
FROM A GEOREPLICATED
DISTRIBUTED SERVICE?
quickmeme.com

Centralized Computing

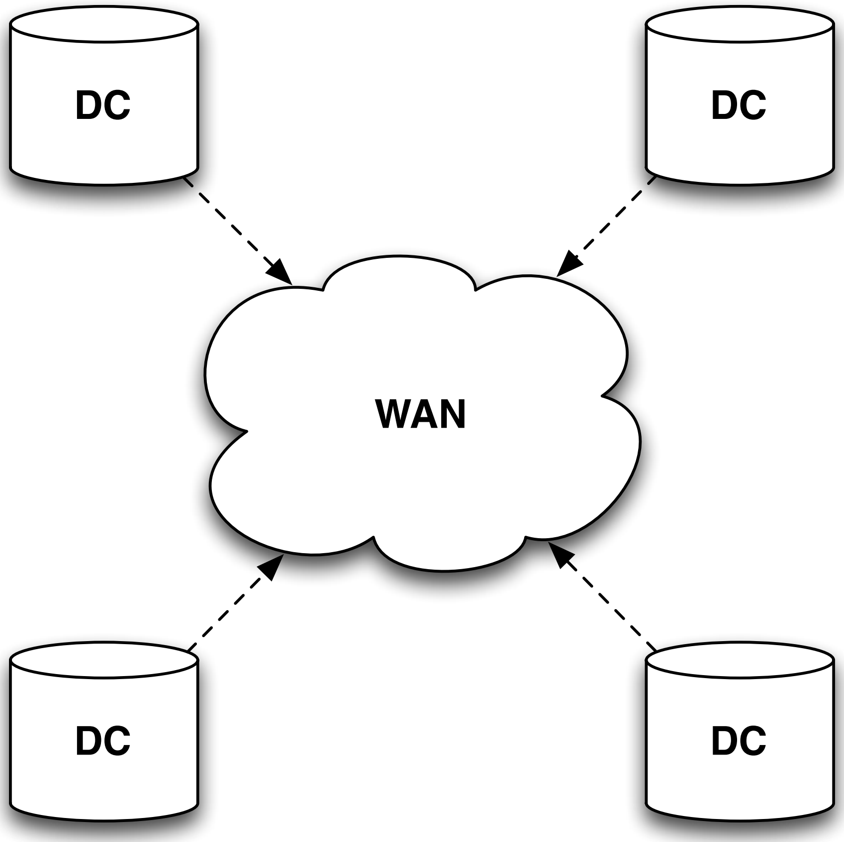
- Predictable latency
- No partial failure
- Single clock
  - **Global event order**

Taking Order For Granted

Global event order →

Data	(**Ordered**) array of bytes
Compute	(**Ordered**) sequence of instructions

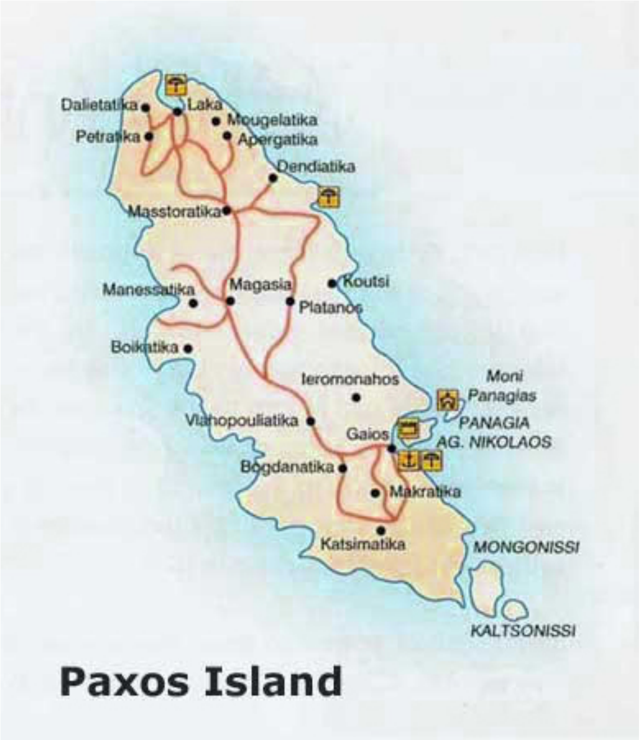
Distributed Computing



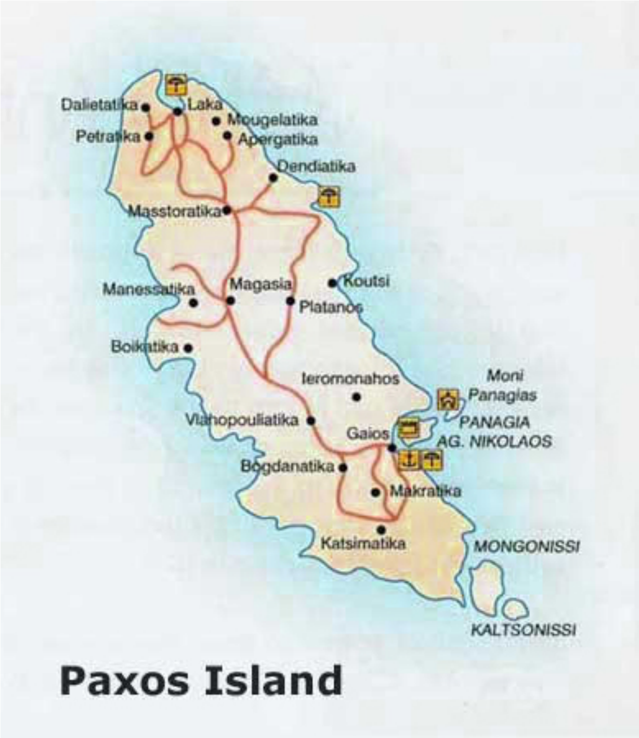
- Unpredictable latency
- Partial failures
- **No global event order**

Alternative #1:

Enforce global event order at all nodes ("Strong Consistency")



Alternative #1:
Enforce global event order at all nodes (“Strong Consistency”)



Problems:
- Availability (CAP)
- Latency

Alternative #2:
Ensure correct behavior
for any network order
(“Weak Consistency”)

Alternative #2:
Ensure correct behavior
for any network order
("Weak Consistency")

Problem:
With traditional languages,
this is **very difficult**.

The “ACID 2.0” Pattern

Associativity:

$$X \circ (Y \circ Z) = (X \circ Y) \circ Z$$

“batch tolerance”

Commutativity:

$$X \circ Y = Y \circ X$$

“reordering tolerance”

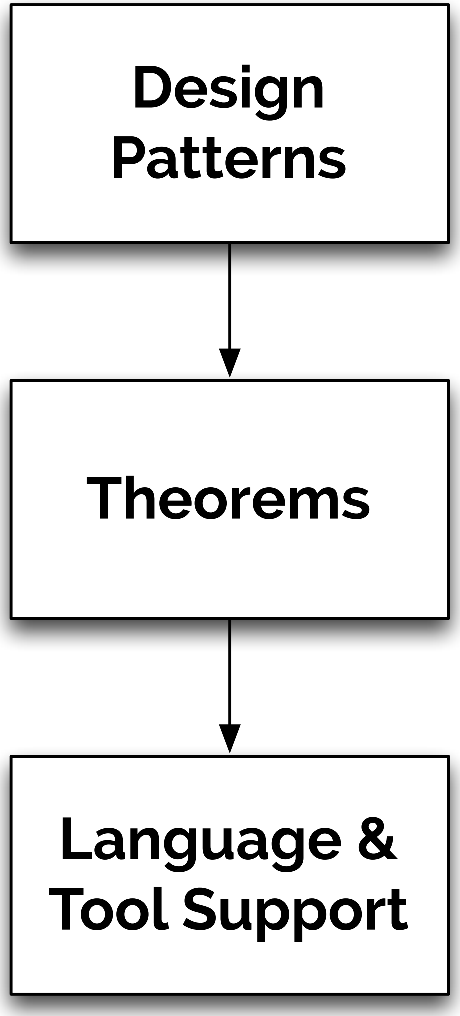
Idempotence:

$$X \circ X = X$$

“retry tolerance”

“When I see patterns in my programs, I consider it a sign of trouble … [they are a sign] that I'm using abstractions that aren't powerful enough.”

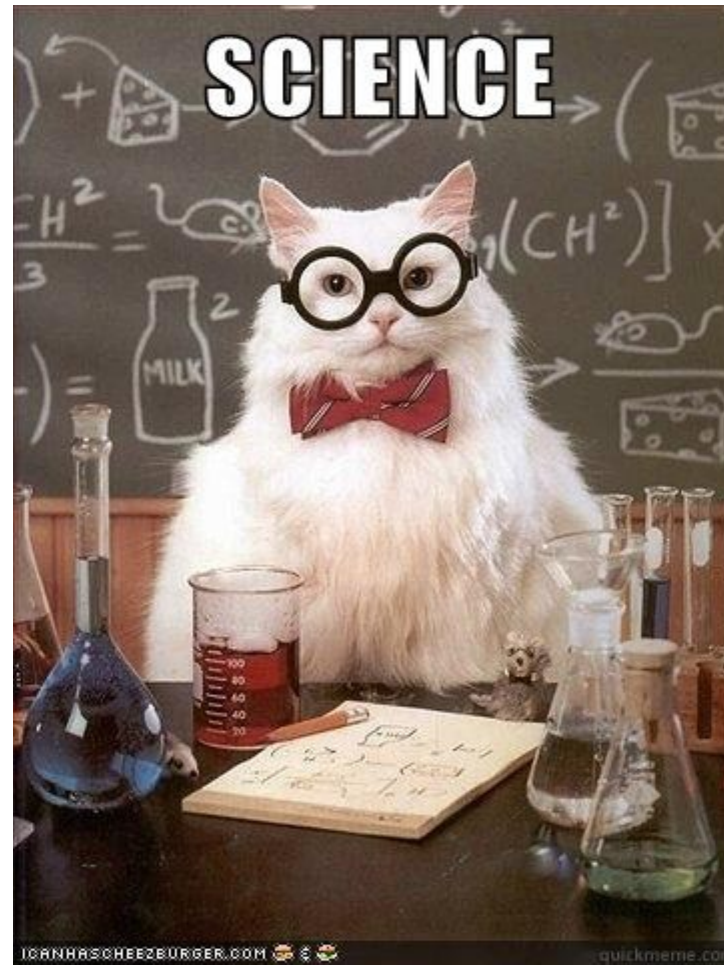
—Paul Graham



Bounded Join Semilattices

A triple $\langle S, \sqcup, \bot \rangle$ such that:

- S is a set
- $\sqcup$ is a binary operator ("*least upper bound*")
  - Induces a partial order on S: $x \leq_{\mathcal{S}} y$ if $x \sqcup y = y$
  - Associative, Commutative, and Idempotent
- $\forall x \in S$: $\bot \sqcup x = x$

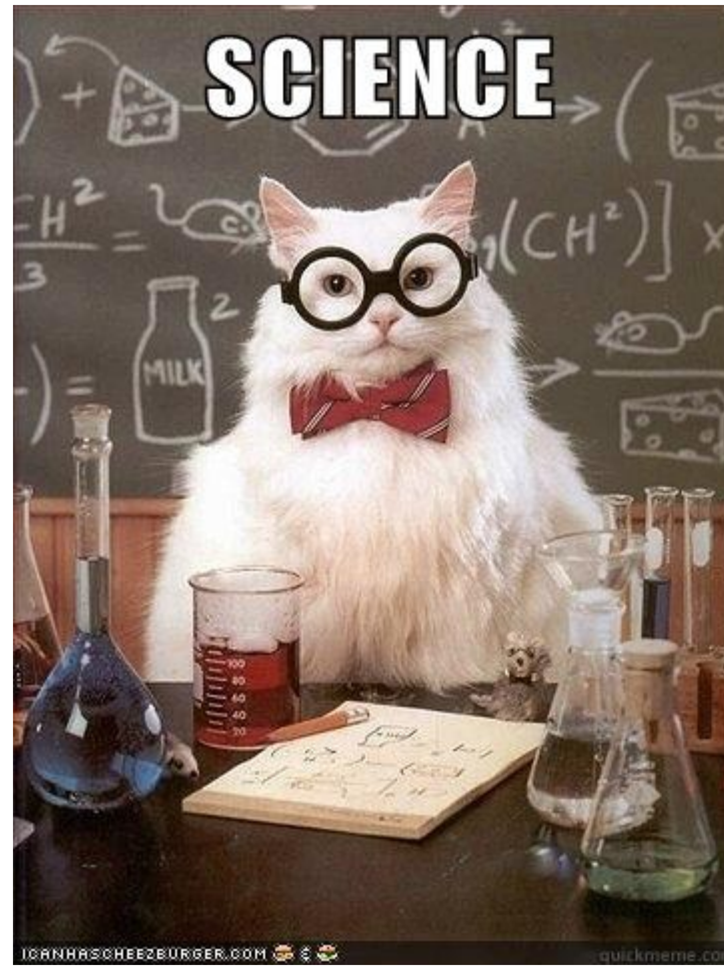


Bounded Join Semilattices

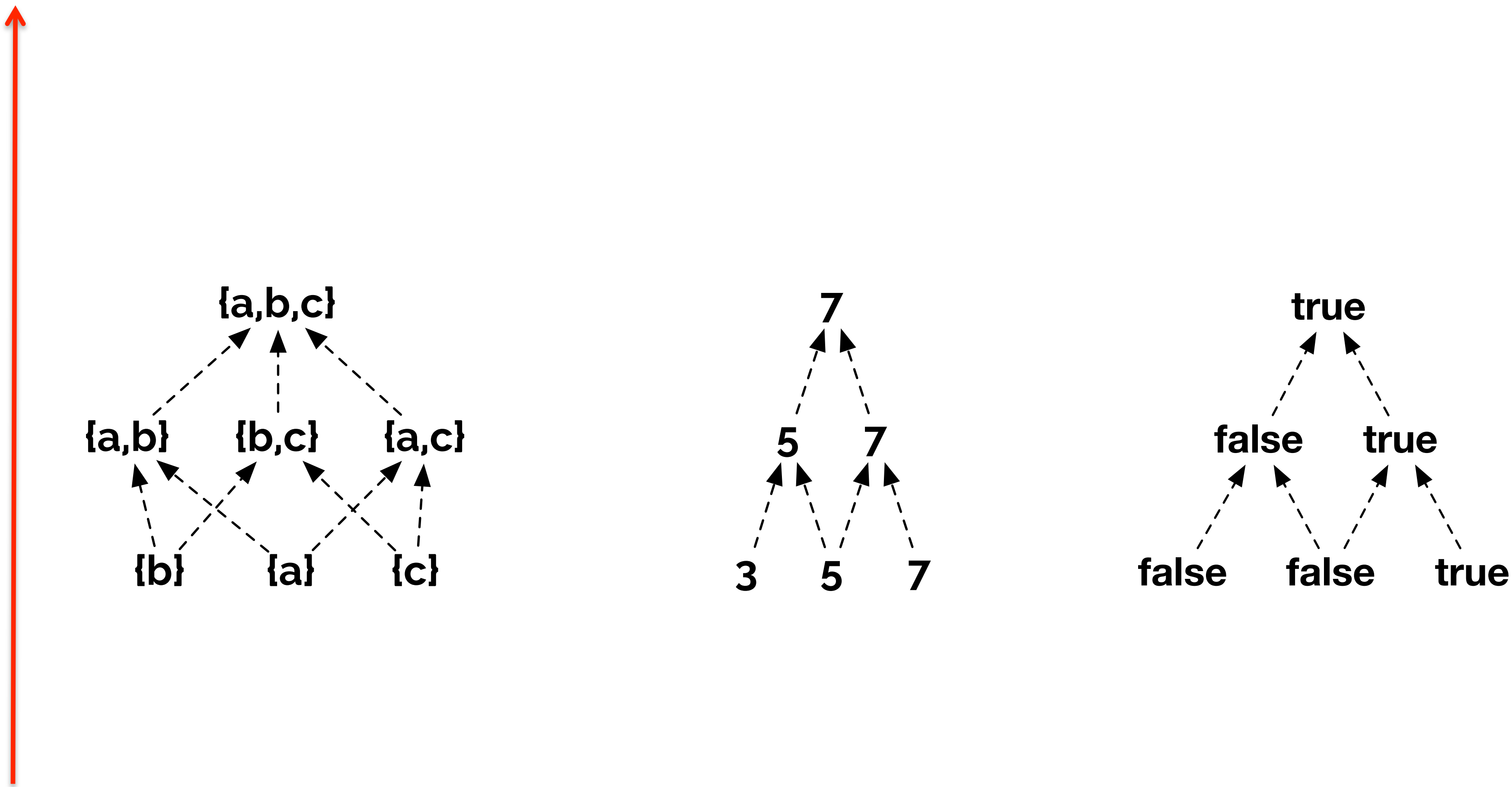
Lattices are objects that **grow** over time.

An **interface** with an ACID 2.0 `merge()` method

- Associative
- Commutative
- Idempotent



Time



Set
(Merge = *Union*)

Increasing Int
(Merge = *Max*)

Boolean
(Merge = *Or*)

CRDTs: Convergent Replicated Data Types

- e.g., registers, counters, sets, graphs, trees

Implementations:

- Statebox
- Knockbox
- riak_dt

Lattices represent
disorderly data.

How can we represent
disorderly computation?

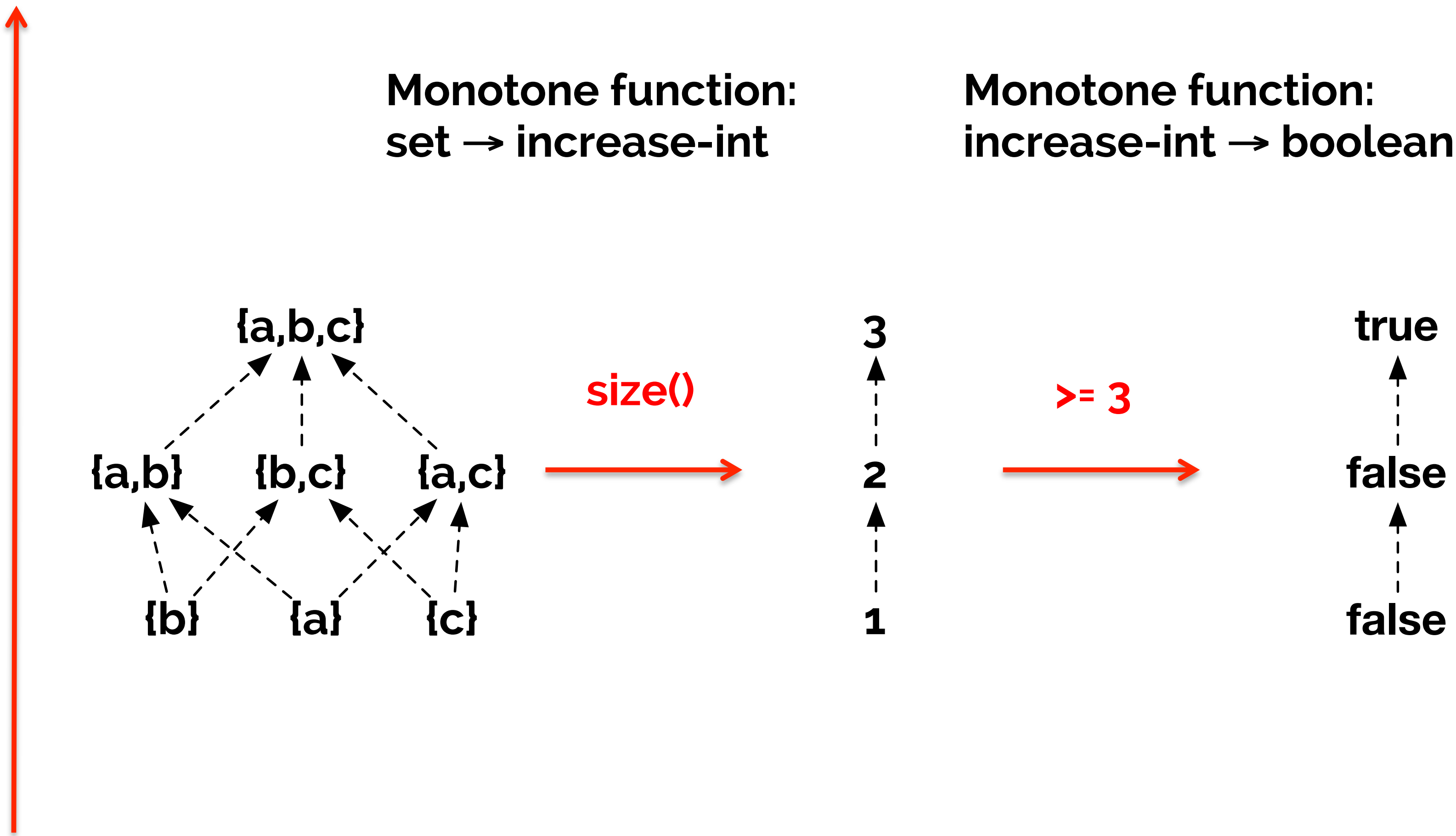
$f: S \to T$ is a ***monotone function*** iff:

$$\forall a, b \in S : a \leq_S b \Rightarrow f(a) \leq_T f(b)$$

Time

Monotone function:
set → increase-int

Monotone function:
increase-int → boolean

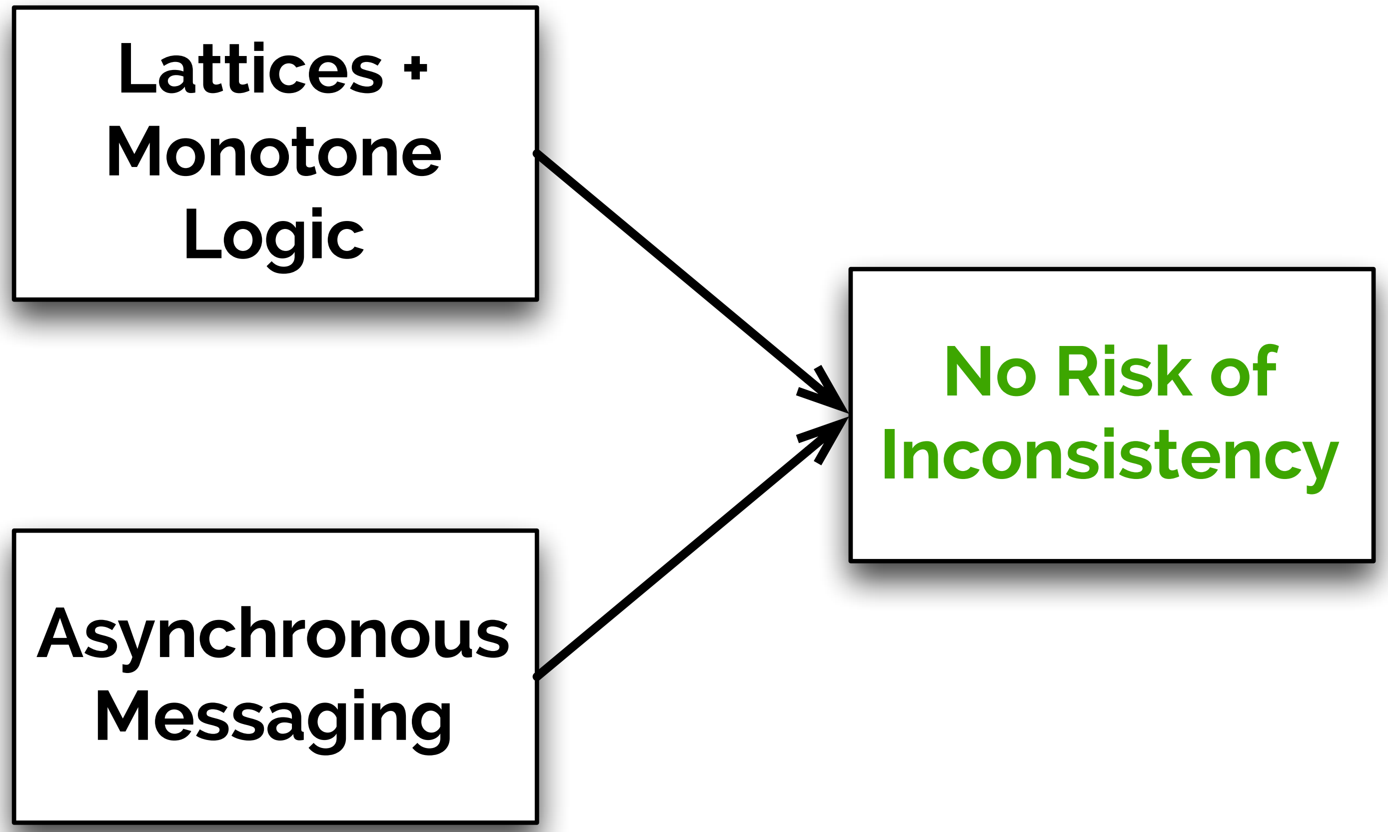


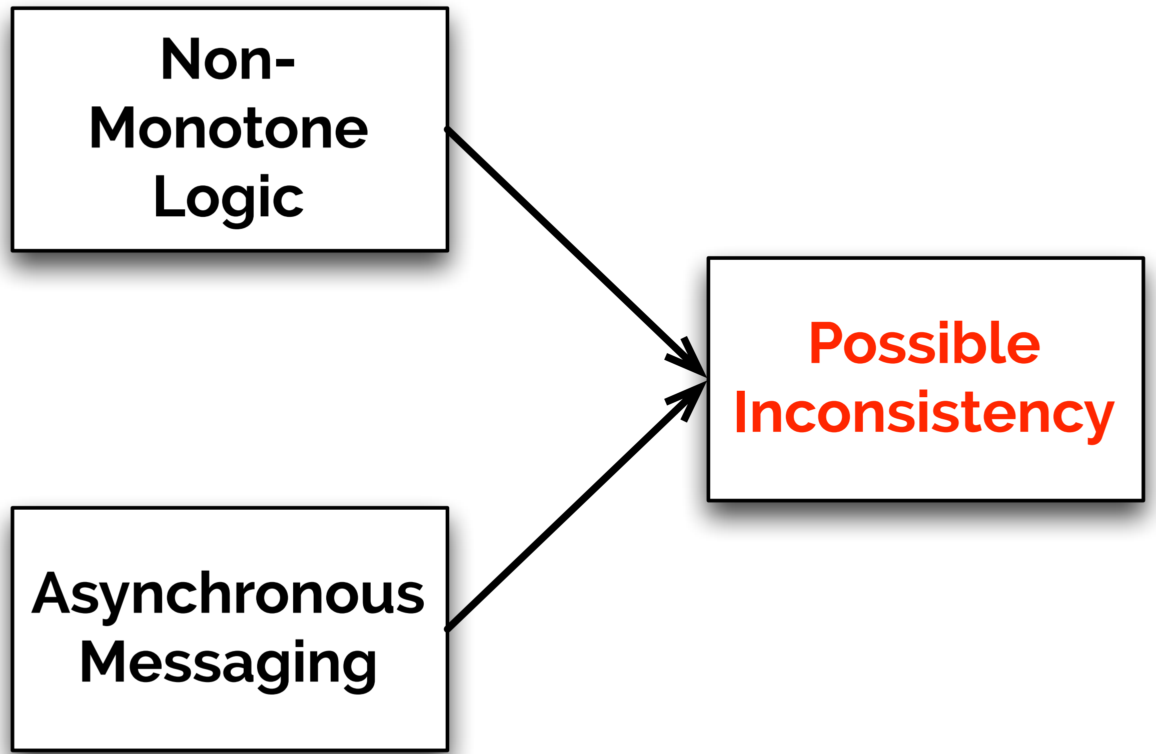
Set
(Merge = *Union*)

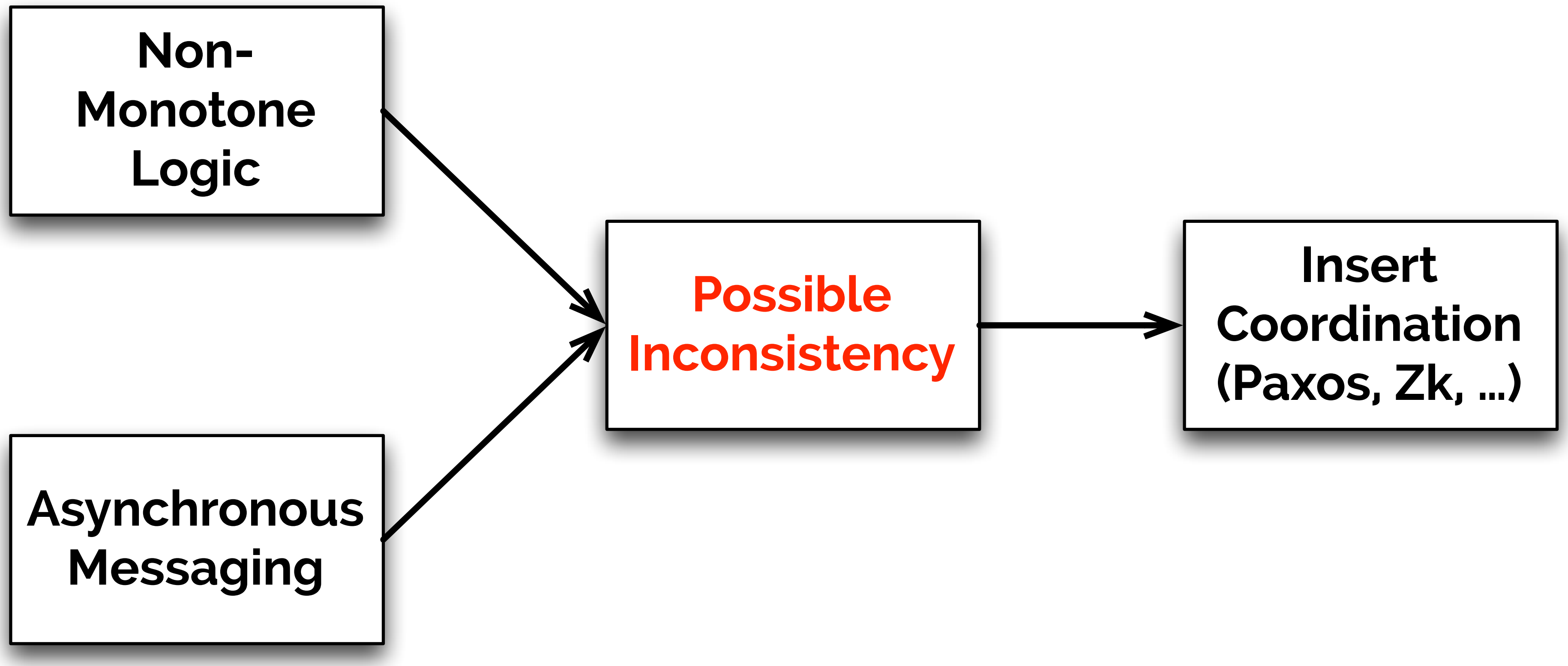
Increasing Int
(Merge = *Max*)

Boolean
(Merge = *Or*)

Consistency
As
Logical
Monotonicity

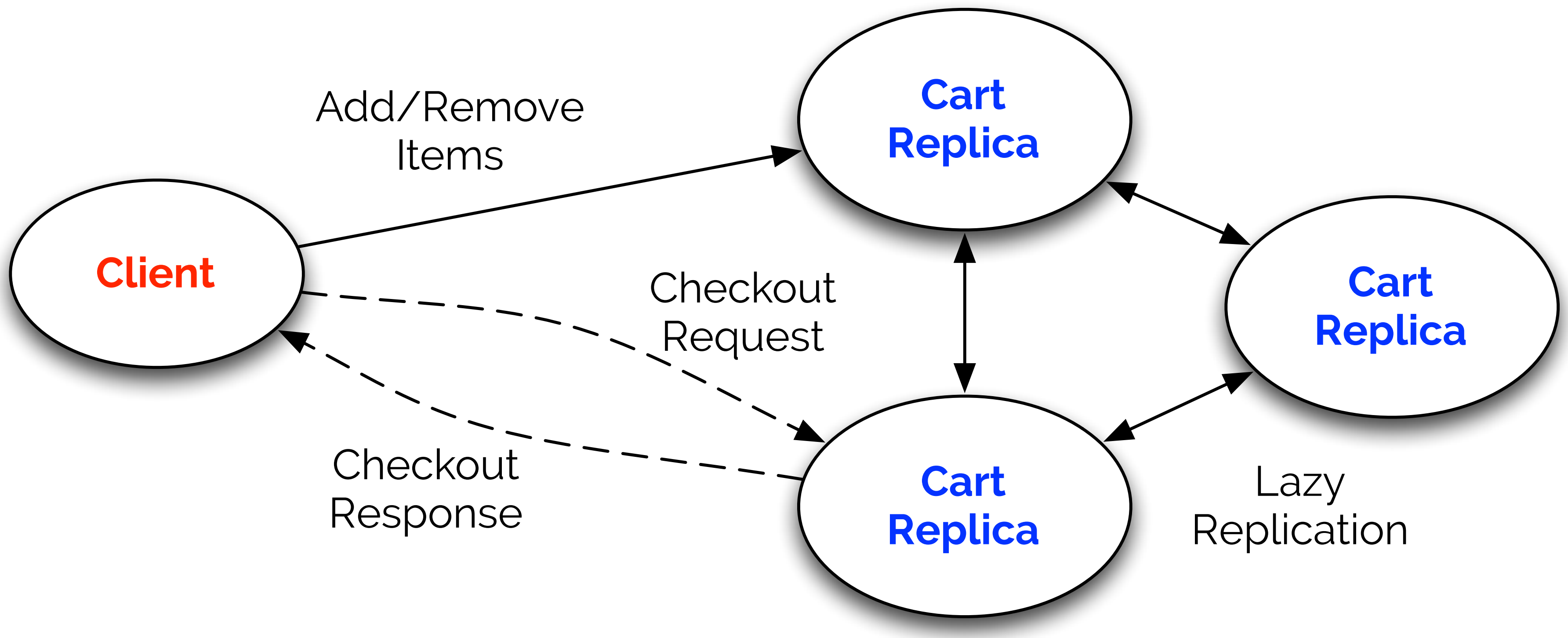

Lattices + Monotone Logic
Asynchronous Messaging
No Risk of Inconsistency

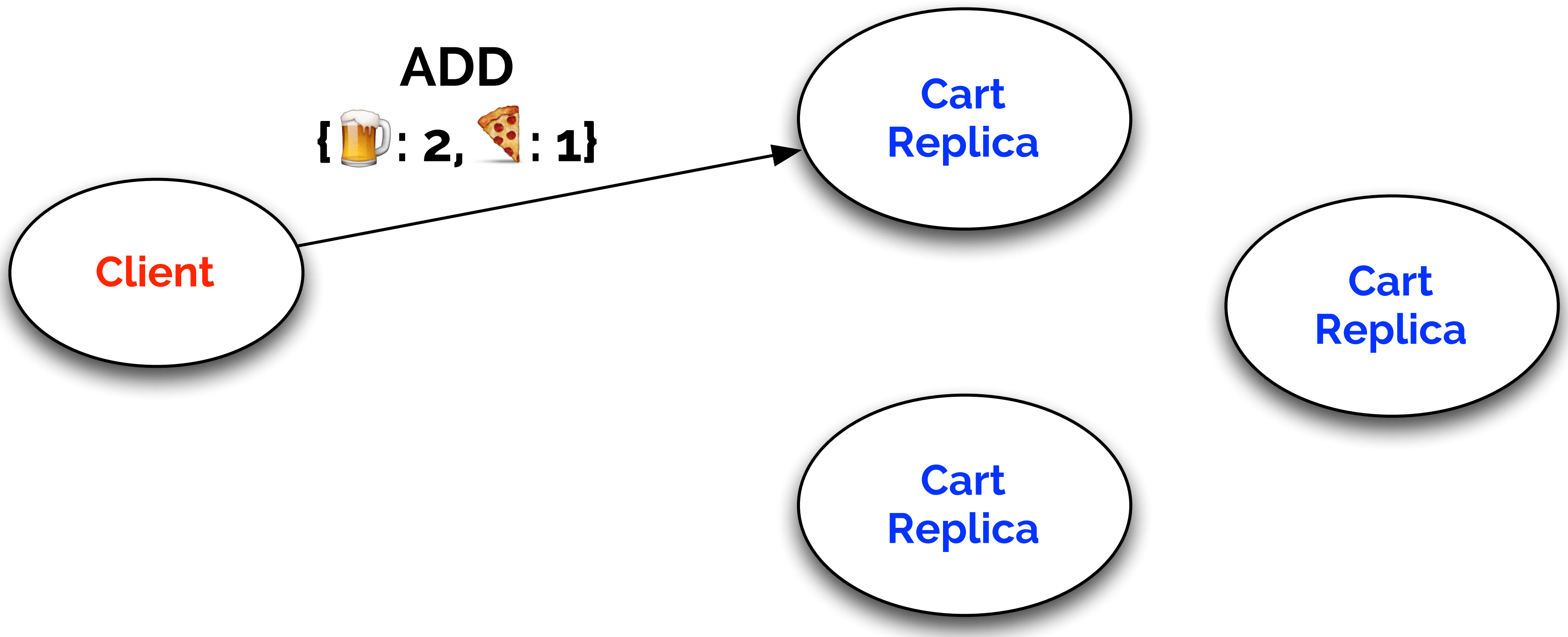

Non-Monotone Logic
Asynchronous Messaging
Possible Inconsistency

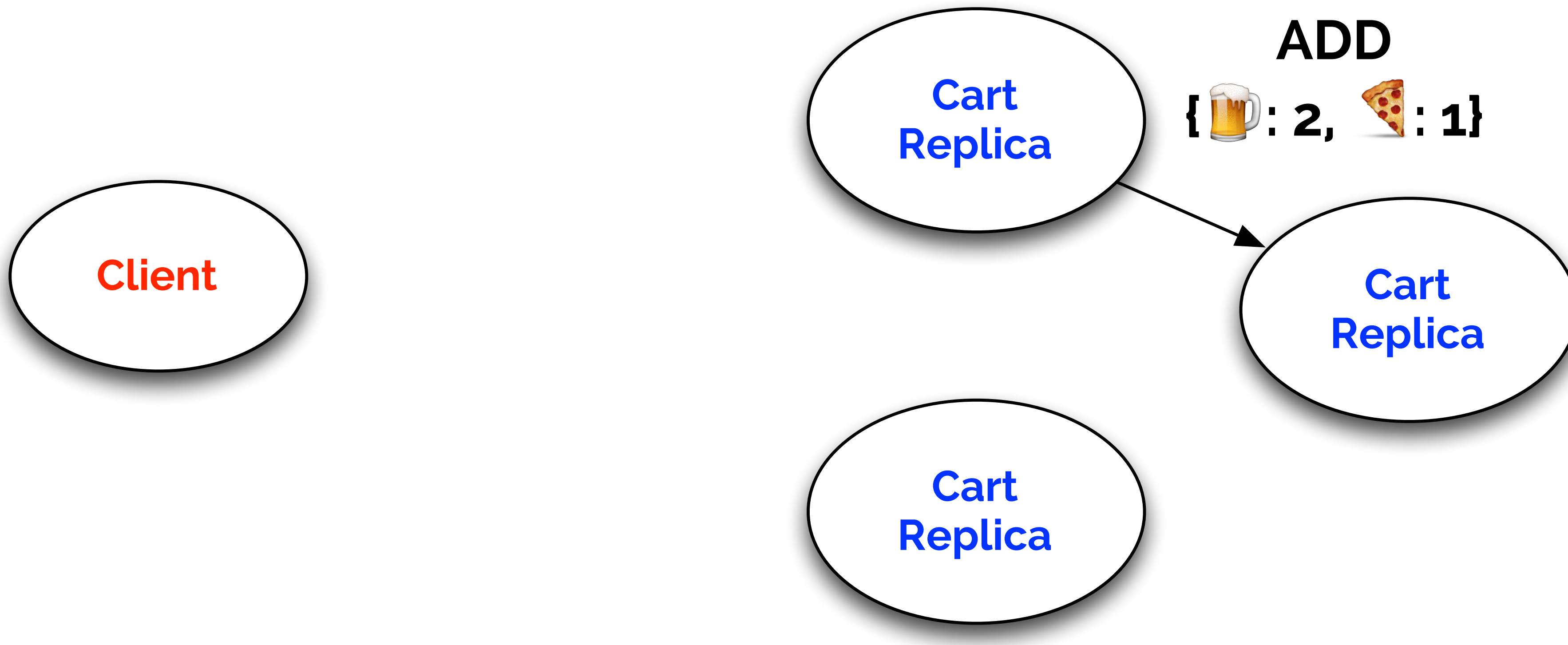


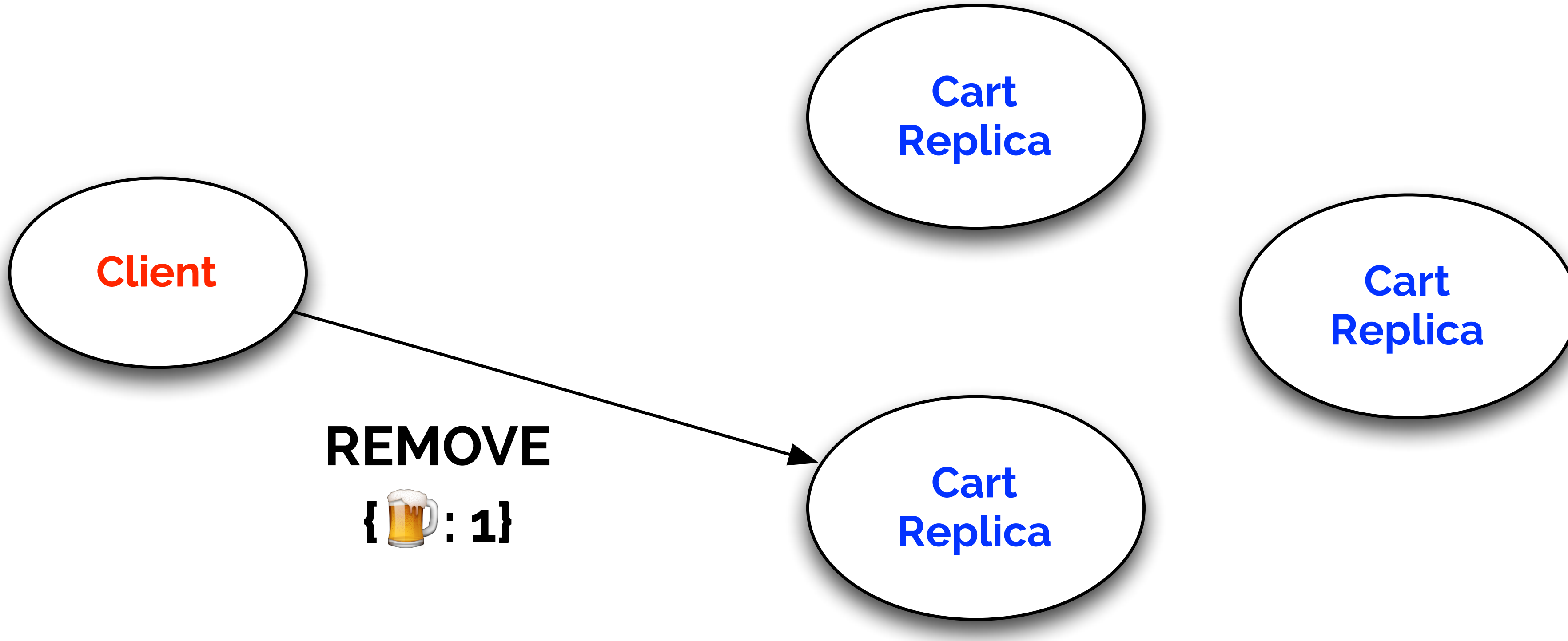
Non-Monotone Logic
Asynchronous Messaging
Possible Inconsistency
Insert Coordination (Paxos, Zk, …)

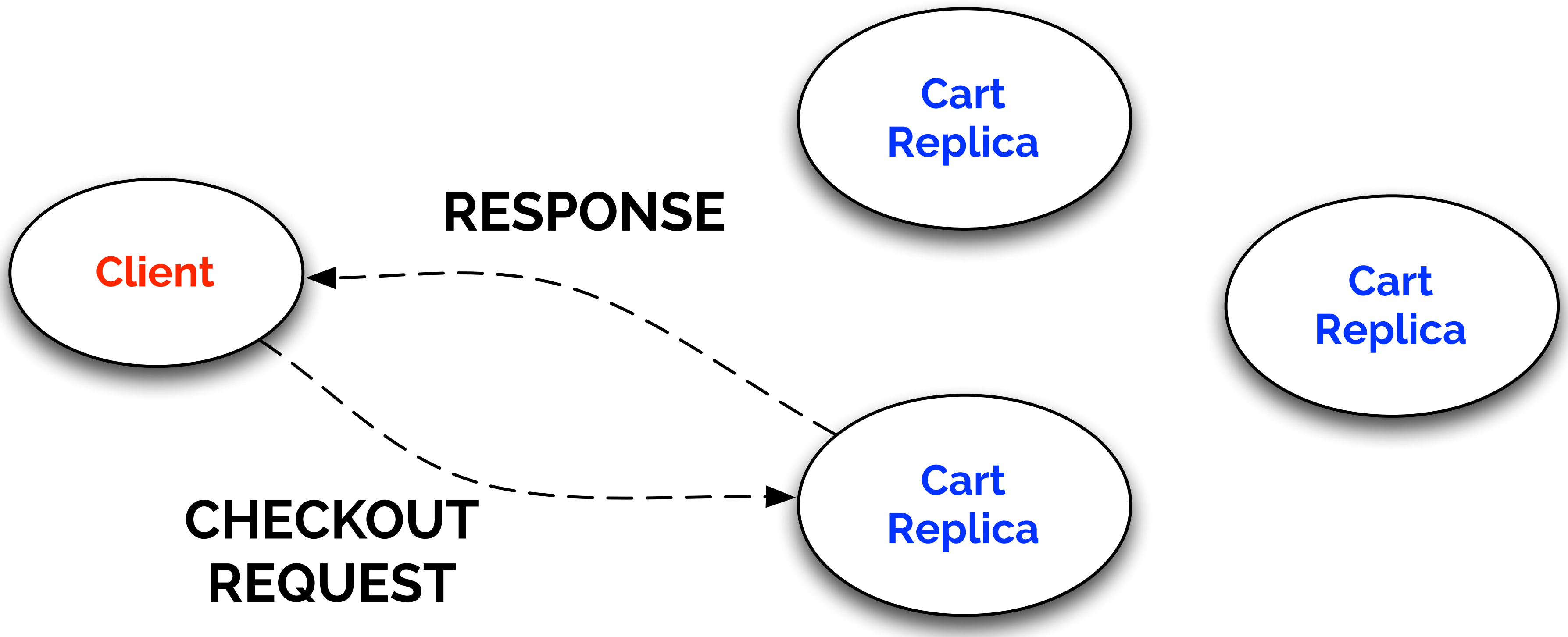
Case Study


Add/Remove
Items
Cart
Replica
Client
Checkout
Request
Cart
Replica
Checkout
Response
Cart
Replica
Lazy
Replication


ADD
{🍺: 2, 🍕: 1}
Client
Cart Replica
Cart Replica
Cart Replica


Client
Cart Replica
ADD
{🍺: 2, 🍕: 1}
Cart Replica
Cart Replica


Client
Cart
Replica
Cart
Replica
Cart
Replica
REMOVE
{🍺: 1}


Cart
Replica
RESPONSE
Client
Cart
Replica
Cart
Replica
CHECKOUT
REQUEST

Questions

1. Will cart replicas eventually converge?
   – "Eventual Consistency"

2. What will client observe on checkout?
   – Goal: checkout reflects all session activity

3. To achieve #1 and #2, how much ordering is required?

Design #1: Mutable State

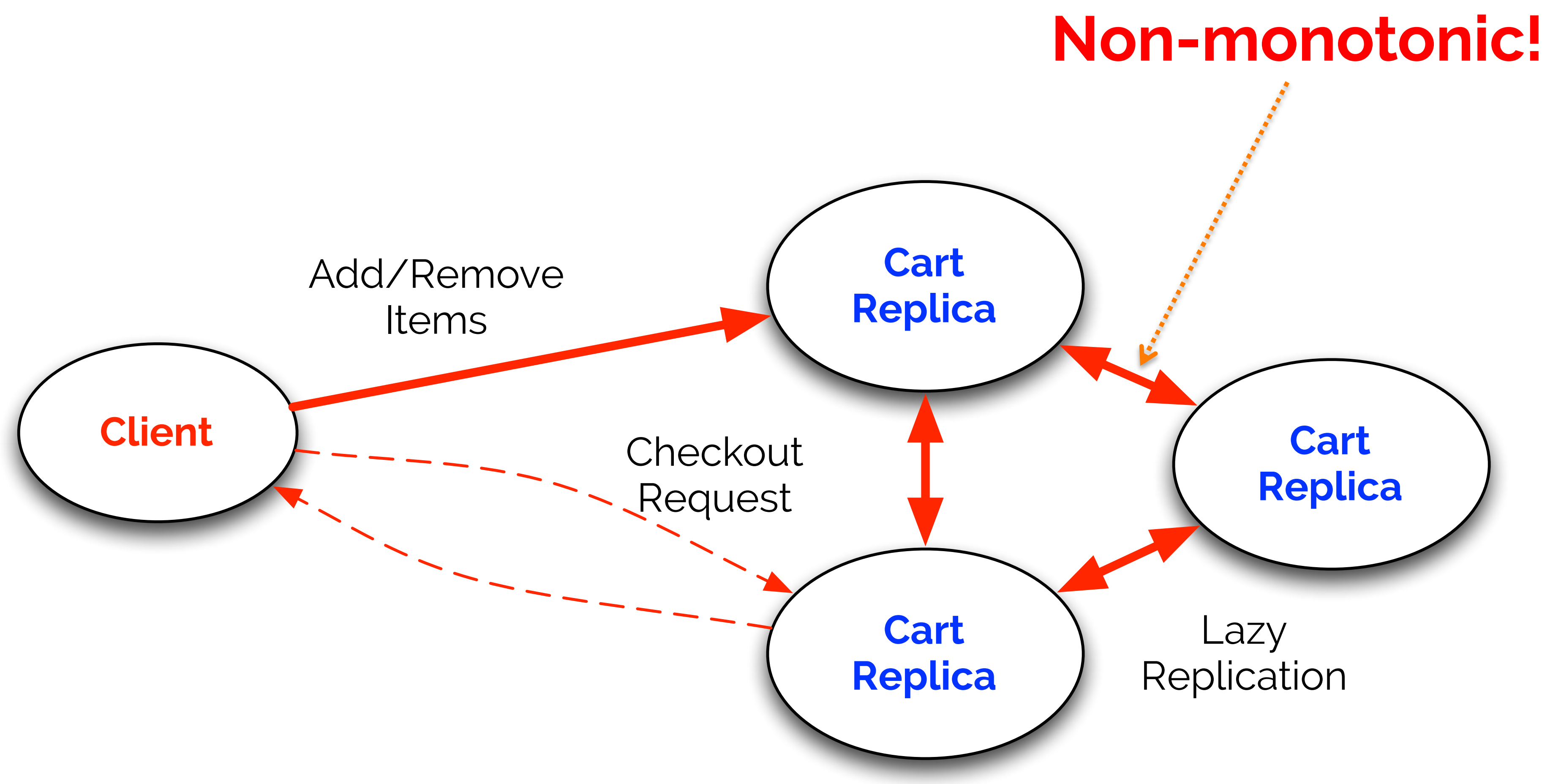
Add(item x, count c**):**

```
if kvs[x] exists:
  old = kvs[x]
  kvs.delete(x)
else
  old = 0
kvs[x] = old + c
```

Remove(item x, count c**):**

```
if kvs[x] exists:
  old = kvs[x]
  kvs.delete(x)
  if old > c
    kvs[x] = old - c
```

Non-monotonic!



Conclusion:
Every operation might require coordination!

Design #2: “Disorderly”

Add(item x, count c**)**:

```
Add x,c to add_log
```

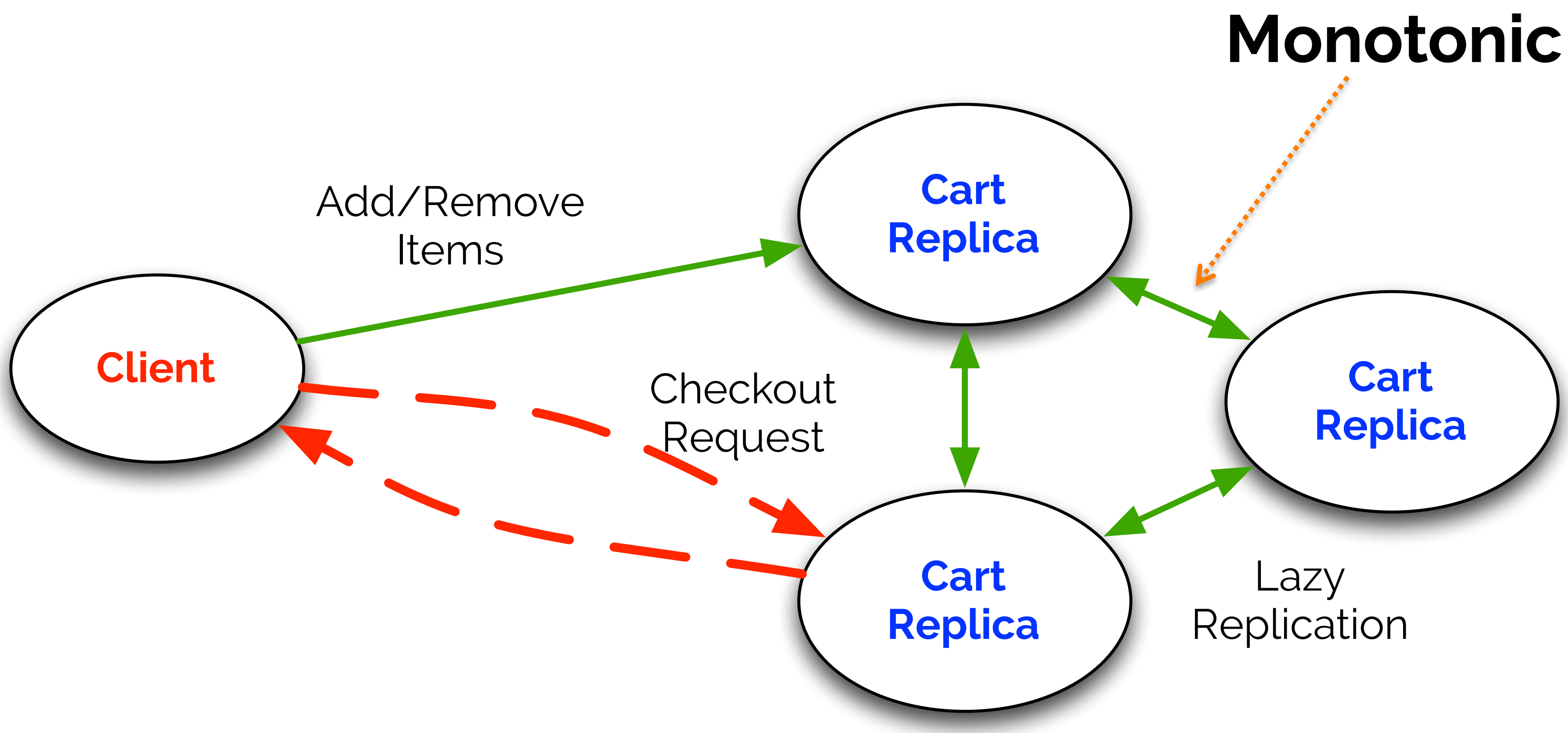
Remove(item x, count c**):**

```
Add x,c to del_log
```

Non-monotonic!

Checkout():

```
Group add_log by
item ID; sum counts.

Group del_log by
item ID; sum counts.

For each item, subtract
deletes from adds.
```



Conclusion:
Replication is safe;
might need to
coordinate on checkout

Takeaways

- **Avoid:** mutable state update
  Prefer: immutable data, monotone growth

- Major difference in coordination cost!
  - Coordinate once per operation vs.
    Coordinate once per checkout

- We'd like a type system for monotonicity

Language Design

Disorderly Programming

- Order-independent: **default**
- Order dependencies: **explicit**
- Order as part of the design process
- Tool support
  - Where is order needed? Why?


The Daily Telegraph

Rule of the mob

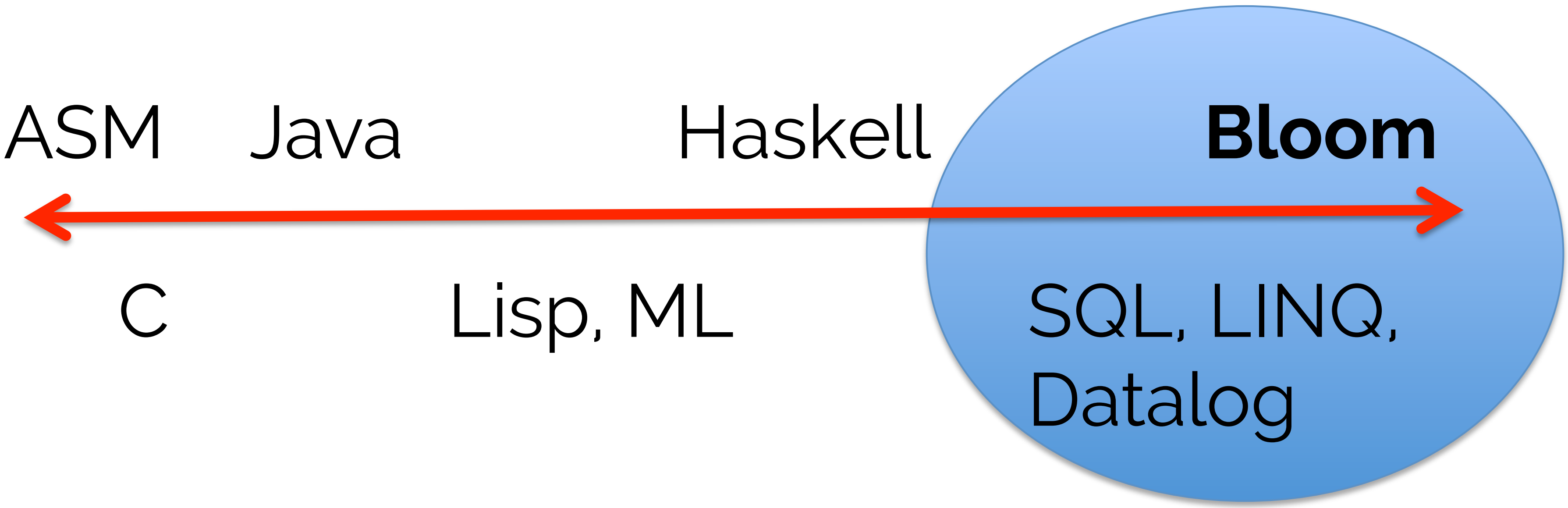
Lockdown in London after third day of riots

Police losing control of the capital's streets

Cameron ends holiday early as crisis deepens

Send in water cannon to clear streets, May is told

The Disorderly Spectrum



High-level
"Declarative"
Powerful optimizers

Processes that communicate via asynchronous message passing

Bloom ≈ declarative agents

Each process has a local database

Logical rules describe computation and communication ("SQL++")

Each agent has a database of **values** that changes over time.

All values have a **location** and **timestamp**.

If RHS is true
(`SELECT …`)

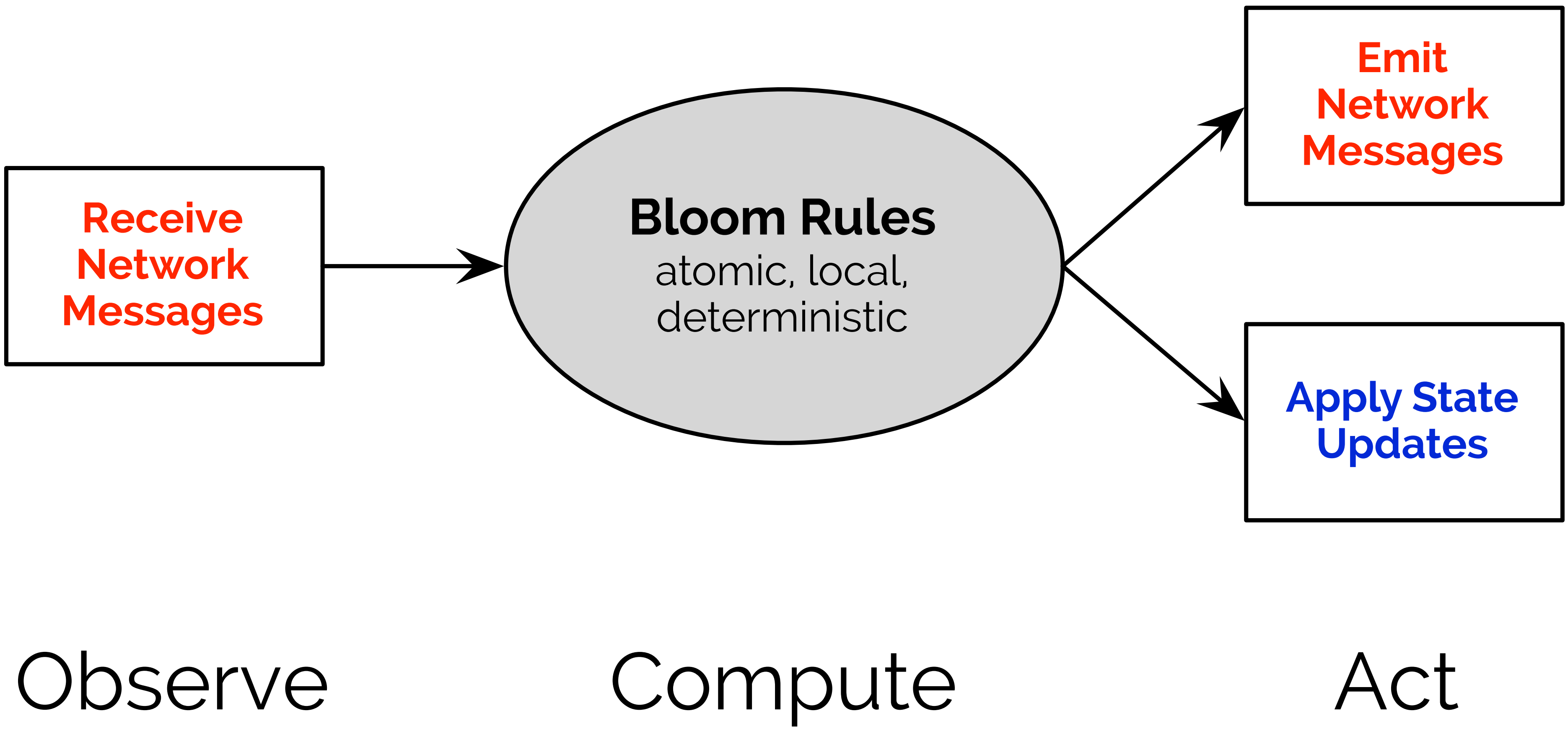
Then LHS is true
(`INTO lhs`)

left-hand-side <= *right-hand-side*

When and **where**
is the LHS true?

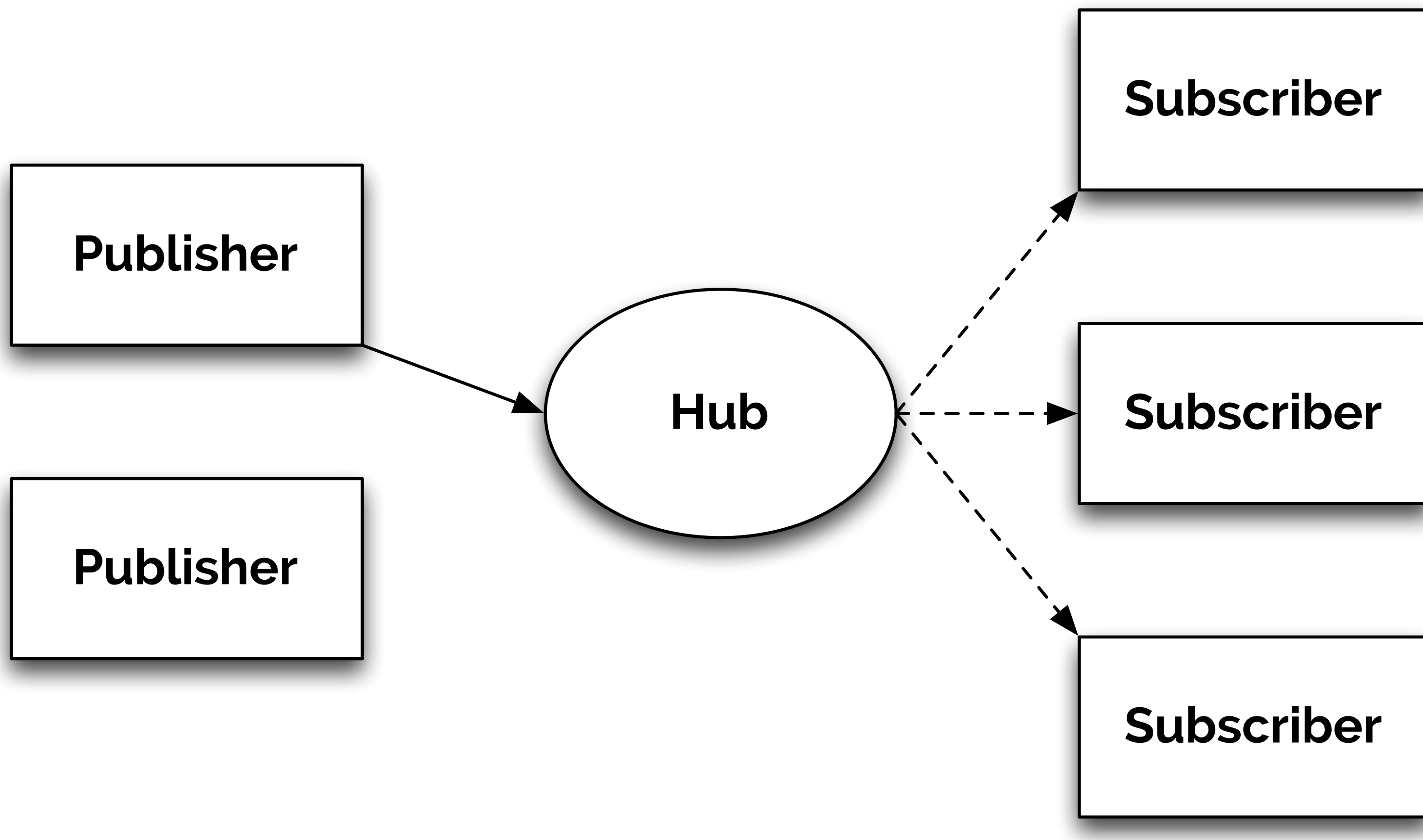
Temporal Operators

1. Same location, same timestamp	`<=`	Computation
2. Same location, next timestamp	`<+`	Persistence
	`<-`	Deletion
3. Different location, non-deterministic timestamp	`<~`	Communication



Receive Network Messages
Bloom Rules
atomic, local, deterministic
Emit Network Messages
Apply State Updates
Observe
Compute
Act

Our First Program: PubSub


Publisher
Publisher
Hub
Subscriber
Subscriber
Subscriber

Ruby DSL

```
class Hub
  include Bud

end
```

```
class Hub
  include Bud

  state do



  end






end
```

State declarations

```
class Hub
  include Bud

  state do




  end

  bloom do
                         Rules



  end
end
```

```
class Hub
  include Bud

  state do
    table    :sub,       [:client, :topic]



  end

  bloom do




  end
end
```

Schema

Persistent state: set of subscriptions

```
class Hub
  include Bud

  state do
    table   :sub,       [:client, :topic]
    channel :subscribe, [:@addr, :topic, :client]
    channel :pub,       [:@addr, :topic, :val]
    channel :event,     [:@addr, :topic, :val]
  end

  bloom do




  end
end
```

Network input, output

Destination address

```ruby
class Hub
  include Bud

  state do
    table   :sub,       [:client, :topic]
    channel :subscribe, [:@addr, :topic, :client]
    channel :pub,       [:@addr, :topic, :val]
    channel :event,     [:@addr, :topic, :val]
  end

  bloom do
    sub   <= subscribe {|s| [s.client, s.topic]}



  end
end
```

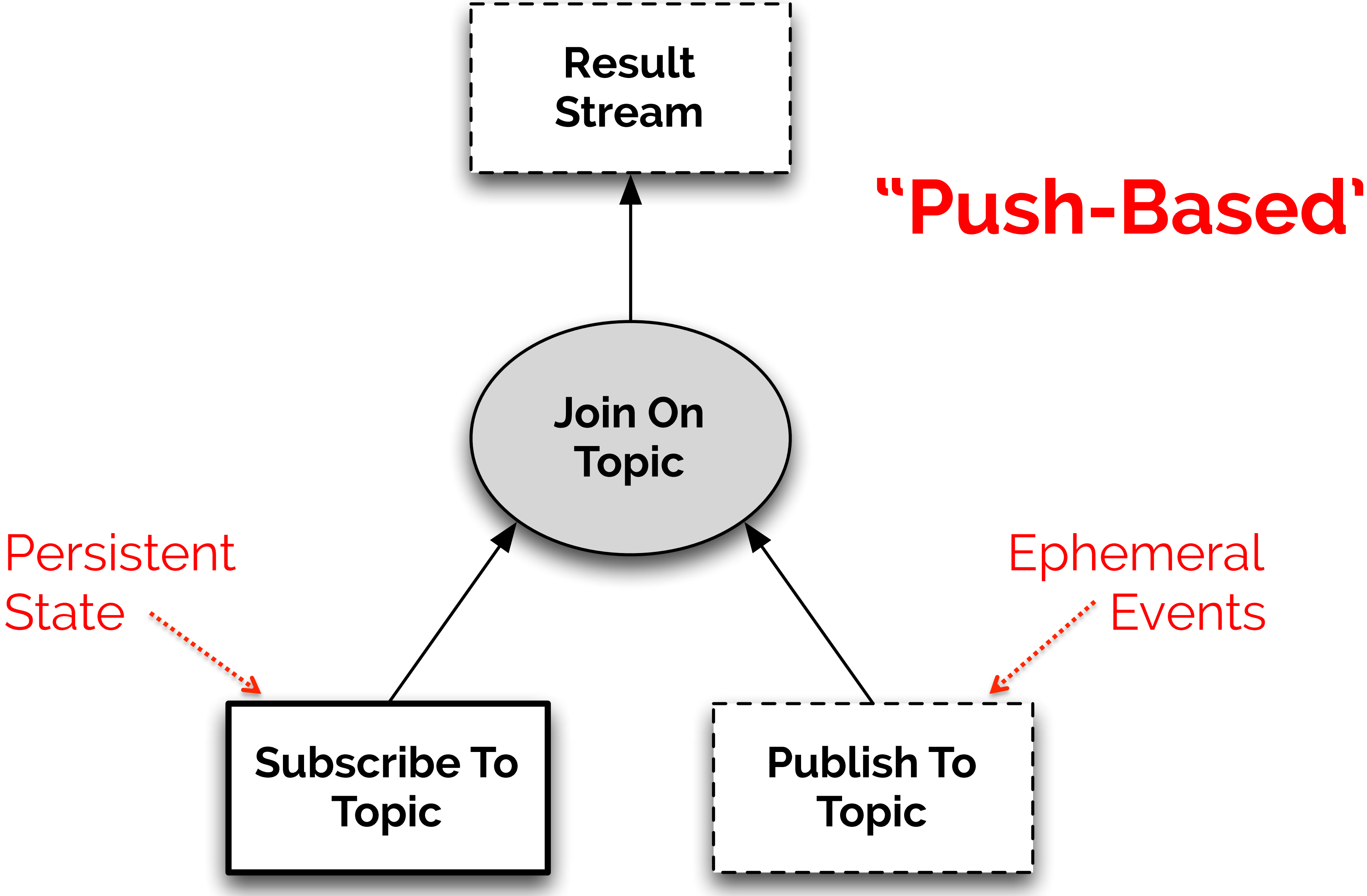
Remember subscriptions

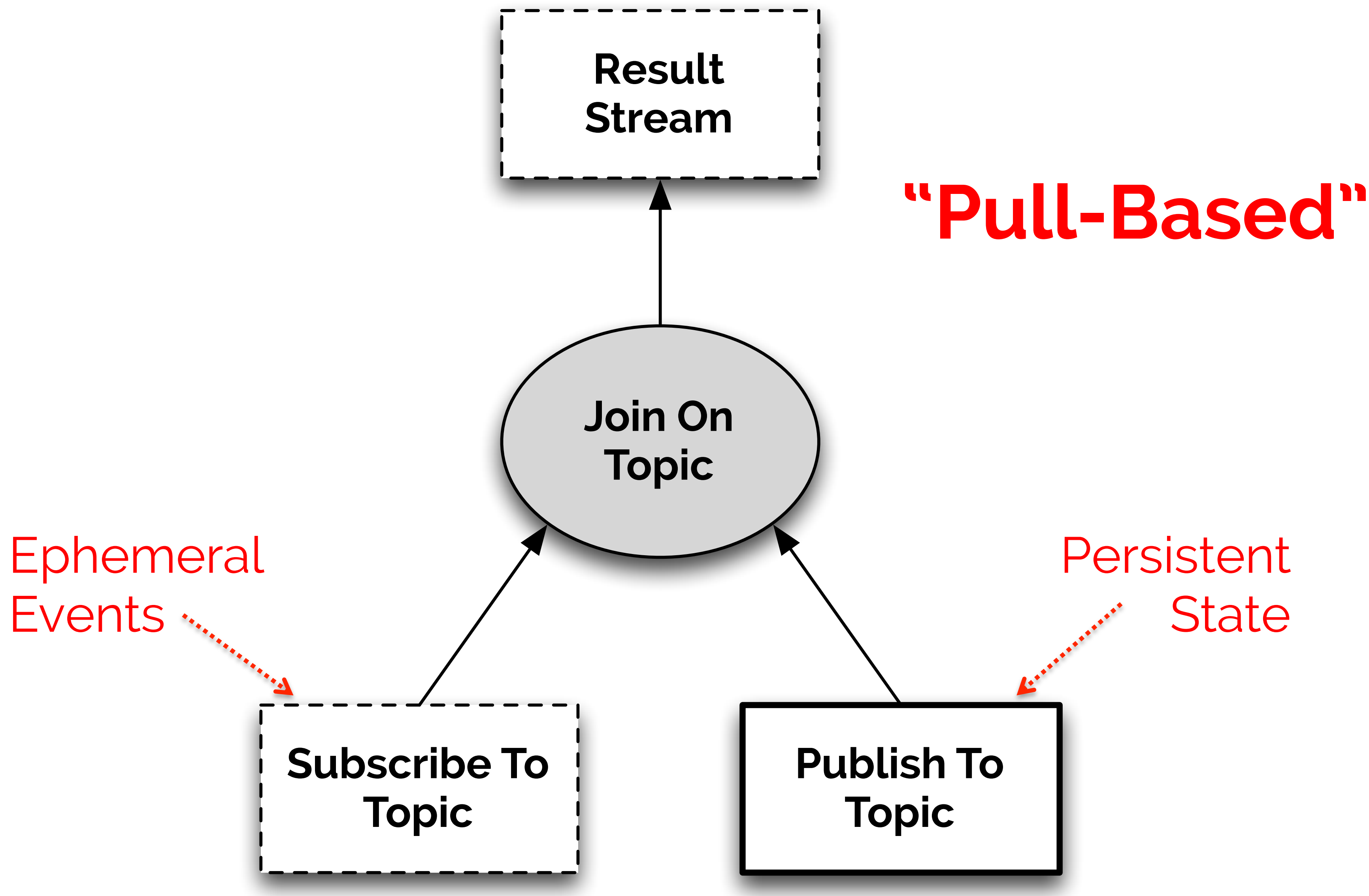
```ruby
class Hub
  include Bud

  state do
    table   :sub,       [:client, :topic]
    channel :subscribe, [:@addr, :topic, :client]
    channel :pub,       [:@addr, :topic, :val]
    channel :event,     [:@addr, :topic, :val]
  end

  bloom do
    sub   <= subscribe {|s| [s.client, s.topic]}
    event <~ (pub * sub).pairs(:topic => :topic) {|p,s|
      [s.client, p.topic, p.val]
    }
  end
end
```

Send events to subscribers

Join (as in SQL)

Join key


Result Stream
"Push-Based"
Join On Topic
Persistent State
Ephemeral Events
Subscribe To Topic
Publish To Topic


Result Stream
"Pull-Based"
Join On Topic
Ephemeral Events
Persistent State
Subscribe To Topic
Publish To Topic

```ruby
class Hub
  include Bud

  state do
    table   :sub,       [:client, :topic]
    channel :subscribe, [:@addr, :topic, :client]
    channel :pub,       [:@addr, :topic, :val]
    channel :event,     [:@addr, :topic, :val]
  end

  bloom do
    sub   <= subscribe {|s| [s.client, s.topic]}
    event <~ (pub * sub).pairs(:topic => :topic) {|p,s|
      [s.client, p.topic, p.val]
    }
  end
end
```

```ruby
class HubPull
  include Bud

  state do
    table   :pub,         [:topic, :val]
    channel :publish,     [:@addr, :topic, :val]
    channel :sub,         [:@addr, :topic, :client]
    channel :event,       [:@addr, :topic, :val]
  end

  bloom do
    pub   <= publish {|p| [p.topic, p.val]}
    event <~ (pub * sub).pairs(:topic => :topic) {|p,s|
      [s.client, p.topic, p.val]
    }
  end
end
```

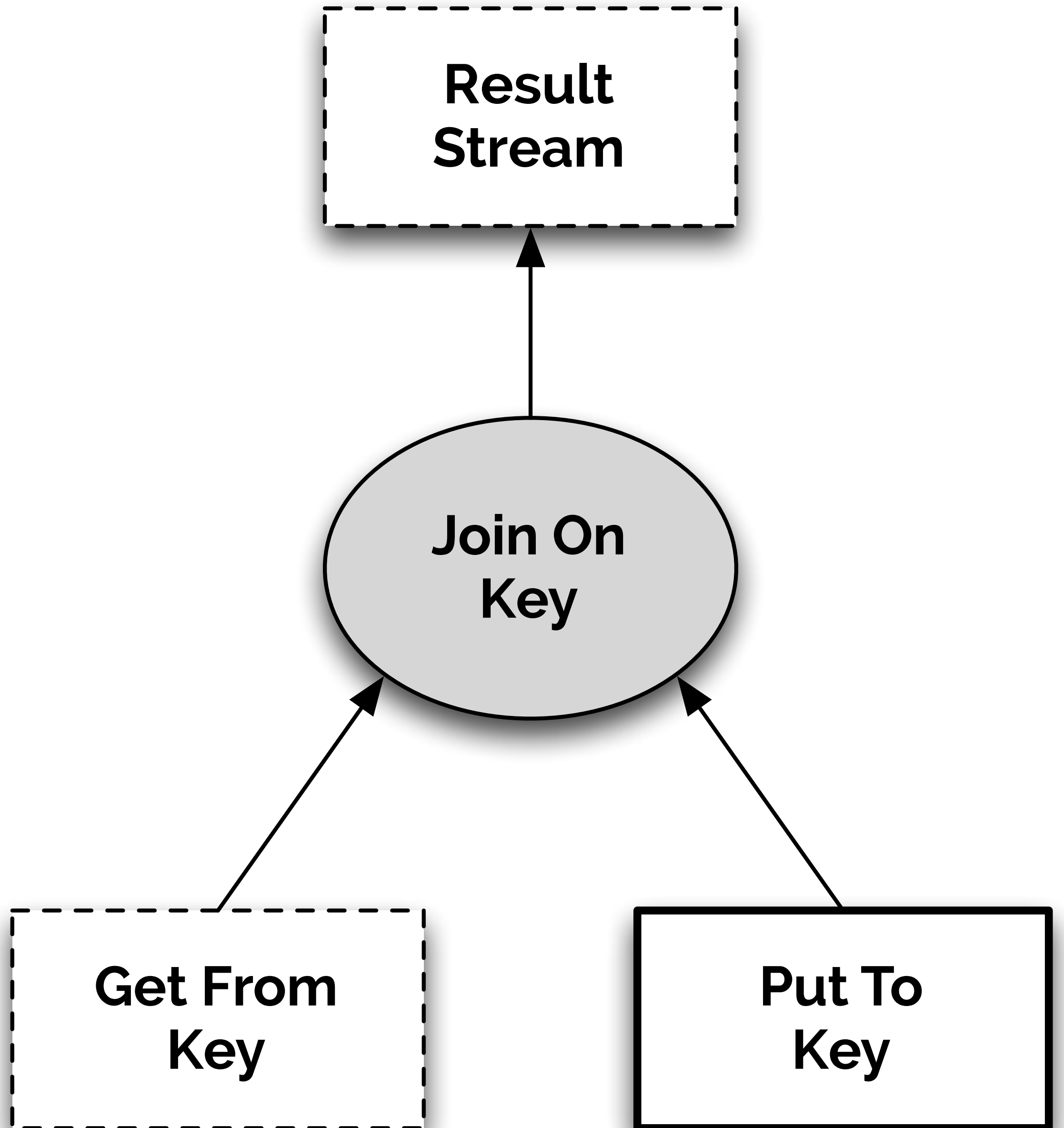
Suppose we keep only the most recent message for each topic ("last writer wins").

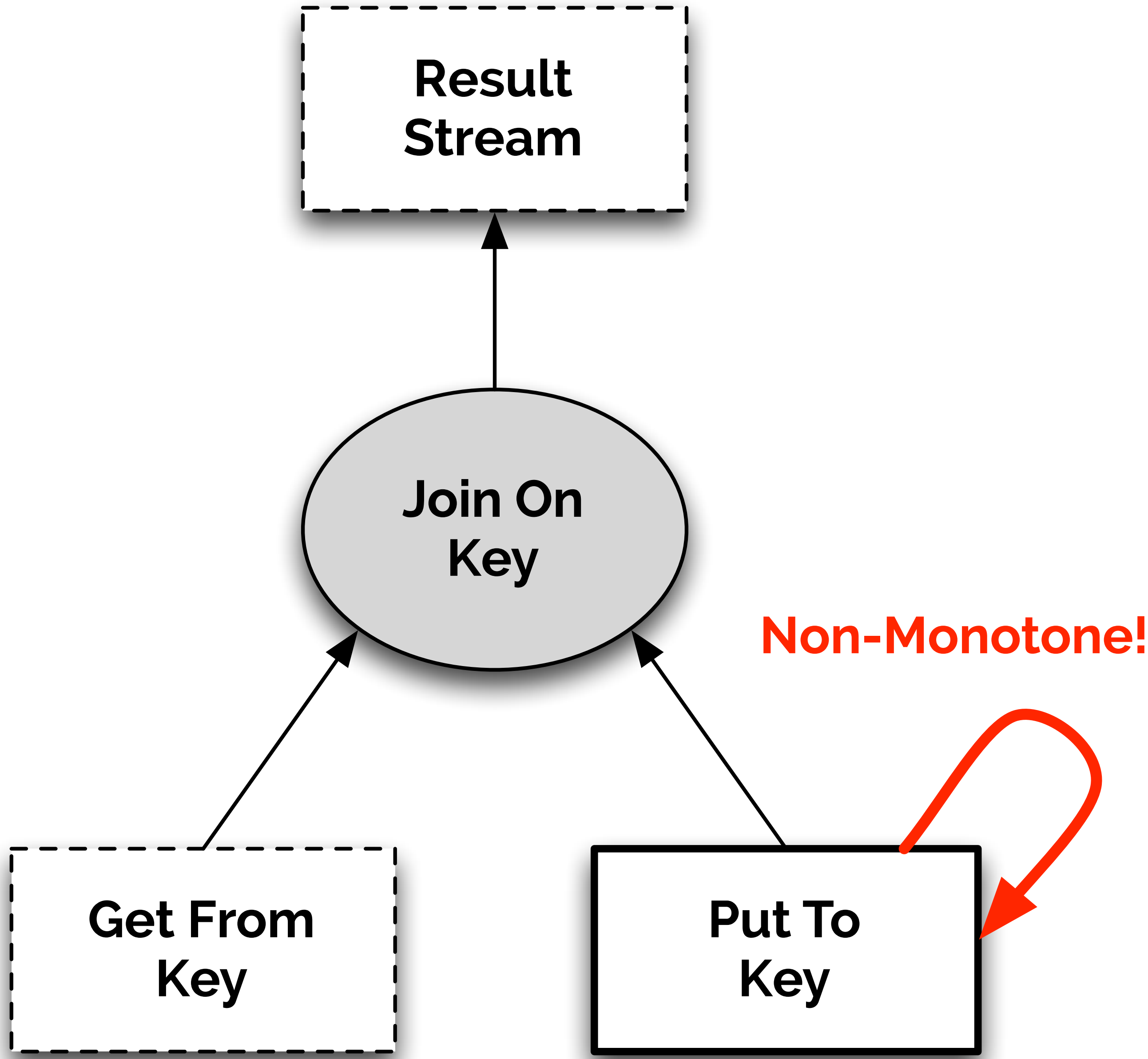
Rename:

Publish → Put
Subscribe → Get
Event → Reply
Pub → DB
Topic → Key

```
class KvsHub
  include Bud

  state do
    table   :db,          [:key, :val]
    channel :put,         [:@addr, :key, :val]
    channel :get,         [:@addr, :key, :client]
    channel :reply,       [:@addr, :key, :val]
end

  bloom do
    db    <+ put {|p| [p.key, p.val]}
    db    <- (db * put).lefts(:key => :key)
    reply <~ (db * get).pairs(:key => :key) {|d,g|
      [g.client, d.key, d.val]
    }
  end
end
```

Update = delete + insert


Result Stream
Join On Key
Get From Key
Put To Key



Result Stream
Join On Key
Non-Monotone!
Get From Key
Put To Key

```
class KvsHub
  include Bud

  state do
    table   :db,          [:key, :val]
    channel :put,         [:@addr, :key, :val]
    channel :get,         [:@addr, :key, :client]
    channel :reply,       [:@addr, :key, :val]
end

  bloom do
    db    <+ put {|p| [p.key, p.val]}
    db    <- (db * put).lefts(:key => :key)
    reply <~ (db * get).pairs(:key => :key) {|d,g|
      [g.client, d.key, d.val]
    }
  end
end
```

Takeaways

Bloom:

- Concise, high-level programs
- State update, asynchrony, and non-monotonicity are explicit in the syntax

Design Patterns:

- Communication vs. Storage
- Queries vs. Data
- Push vs. Pull

Actually not so different!

Conclusion

Traditional languages are not a good fit for modern distributed computing

Principle: Disorderly programs for disorderly networks

Practice: Bloom

- High-level, disorderly, declarative
- Designed for distribution

Thank You!

Twitter: `@neil_conway`

`gem install bud`

http://www.bloom-lang.net

Collaborators:

Peter Alvaro
Emily Andrews
Peter Bailis
David Maier
Bill Marczak
Joe Hellerstein
Sriram Srinivasan