consistency analysis in *bloom*

a *CALM* and collected approach

peter alvaro, neil conway, joseph m. hellerstein, william r. marczak
uc berkeley

the state of things

- distributed programming increasingly common
- hard^2
  - (parallelism + asynchrony + failure) × (software engineering)

choices

ACID

- general correctness via theoretical foundations
  - read/write: serializability
  - coordination/consensus

concerns: latency, availability

loose consistency

- app-specific correctness via design maxims
  - semantic assertions
  - custom compensation

concerns: hard to trust, test

desire: best of both worlds

- theoretical foundation for correctness under loose consistency
- embodiment of theory in a programming framework

progress

- CALM consistency (maxims ⇒ theorems)
- Bloom language (theorems ⇒ programming)

outline

- motivation: language-level consistency
- foundation: CALM theorem
- implementation: bloom prototype
- discussion: tolerating inconsistency taint


CALM

monotonicity

monotonic code

- info accumulation
  - *the more you know, the more you know*

non-monotonic code

- belief revision
  - *new inputs can change your mind*
  - e.g. aggregation, negation, state update

an aside

- double-blind review

an aside

- double-blind review
- pocket change

intuition

- counting requires waiting

intuition

- counting requires waiting
- waiting requires counting

CALM Theorem

- CALM: consistency and logical monotonicity
  - monotonic code ⇒ eventually consistent
  - non-monotonic ⇒ coordinate only at non-monotonic *points of order*

- conjectures at pods 2010
  - (web-search for "*the declarative imperative*")

- results submitted to pods 2011
  - Marczak, Alvaro, Hellerstein, Conway
  - Ameloot, Neven, Van den Bussche

practical implications

- compiler can identify non-monotonic “points of order”
  - inject coordination code
  - or mark uncoordinated results as “tainted”
- compiler can help programmer think about coordination costs
- easy to do this with the right language…

outline

- motivation: language-level consistency
- foundation: CALM theorem
- implementation: bloom prototype
- discussion: tolerating inconsistency taint

bloom

disorderly programming

- why is distributed programming hard?
  the von neumann legacy: obsession with order
  - state: ordered array
  - logic: ordered instructions, traversed by program counter

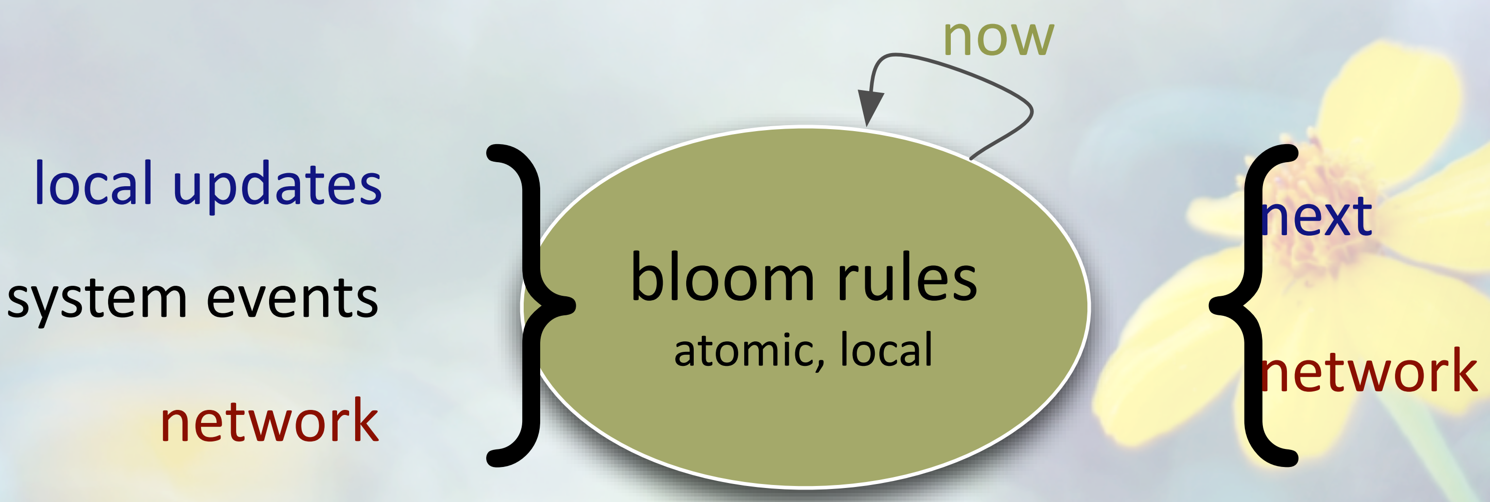
- disorderly programming
  - state: unordered collections
  - logic: unordered set of declarative statements

bud: bloom under development

- based in 5 years experience with Overlog
  - culmination: API-compliant HDFS++ implementation [Eurosys10]
- i got the itch to prototype a more usable language
  - dsl for distributed programming, embedded in ruby
  - interpreter: ~2300 lines of ruby
- bloom features
  - fully declarative semantics (based on *dedalus* temporal logic)
  - disorderly programming with pragmatics of modern language (ruby)
  - domain-specific code analysis

bloom operational model

- really a metaphor for dedalus logic
- each node runs independently
  - local clock, local data, local execution
- timestepped execution loop at each node



bloom statements

<collection> *<accumulator>* *<collection expression>*

bloom statements

<collection> *<accumulator>* *<collection expression>*

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

bloom statements

<collection> <accumulator> <collection expression>

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

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

bloom statements

<collection>

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

<accumulator>

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

<collection expression>

<collection>
`map, flat_map`
`reduce, group`
`join, natjoin, outerjoin`
`empty? include?`

toy example: delivery

```
# abstract interface
module DeliveryProtocol
  def state
    super
    interface input, :pipe_in,
                  ['dst', 'src', 'ident'], ['payload']
    interface output, :pipe_sent,
                  ['dst', 'src', 'ident'], ['payload']
  end
end
```

simple concrete implementation of the delivery protocol

Bloom

```
module BestEffortDelivery
  include DeliveryProtocol

  def state
    channel :pipe_chan,
      ['@dst', 'src', 'ident'], ['payload']
  end

  declare
  def snd
    pipe_chan <~ pipe_in
  end

  declare
  def done
    pipe_sent <= pipe_in
  end
end
```

an alternative implementation: reliable delivery

```
module ReliableDelivery
  include  BestEffortDelivery

  def state
    super
    table :pipe, ['dst', 'src', 'ident'], ['payload']
    channel :ack, ['@src', 'dst', 'ident']
    periodic :tock, 10
  end

  declare
  def remember_resend
    pipe <= pipe_in
    pipe_chan <~ join([pipe, tock]).map{|p, t| p }
  end

  declare
  def rcv
    ack <~ pipe_chan.map {|p| [p.src, p.dst, p.ident] }
  end

  declare
  def done
    apj = join [ack, pipe], [ack.ident, pipe.ident]
    pipe_sent <= apj.map {|a, p| p }
    pipe <- apj.map{|a, p| p}
  end
end
```

the payoff is in the paper

- case study: 2 replicated shopping cart implementations
  1. replicated key/value-store with “destructive” overwriting
  2. “disorderly” version that accumulates/replicates user actions
- demonstrates automatic consistency analysis
  - isolate points of order for coordination
  - highlights why the 2nd implementation is preferable to 1st
- tolerating inconsistency (autoPat)
  - identify “tainted” data in a program
  - automatically generate scaffolding for compensation logic

destructive cart

- full source in paper including replicated KVS

Bloom

```
module DestructiveCart
  include CartProtocol
  include KVSProtocol

  declare
  def do_action
    kvget <= action_msg.map{|a| [a.reqid, a.key]}
    kvput <= action_msg.map do |a|
      if a.action == "A"
        unless kvget_response.map{|b| b.key}.include? a.session
          [a.server, a.client, a.session, a.reqid, [a.item]]
        end
      end
    end
    old_state = join [kvget_response, action_msg],
      [kvget_response.key, action_msg.session]
    kvput <= old_state.map do |b, a|
      if a.action == "A"
        [a.server, a.client, a.session, a.reqid, b.value.push(a.item)]
      elsif a.action == "D"
        [a.server, a.client, a.session, a.reqid, delete_one(b.value,
a.item)]
      end
    end
  end
 declare
  def do_checkout
    kvget <= checkout_msg.map{|c| [c.reqid, c.session]}
    lookup = join [kvget_response, checkout_msg],
      [kvget_response.key, checkout_msg.session]
    response_msg <~ lookup.map do |r, c|
      [c.client, c.server, c.session, r.value]
    end
  end
end
```

disorderly cart

- full source in paper, including replication

```
module DisorderlyCart
  include CartProtocol
  include BestEffortDelivery

  def state
    table :cart_action, ['session', 'item', 'action', 'reqid']
    table :action_cnt, ['session', 'item', 'action'], ['cnt']
    scratch :status, ['server', 'client', 'session', 'item'], ['cnt']
  end

  declare
  def do_action
    cart_action <= action_msg.map do |c|
      [c.session, c.item, c.action, c.reqid]
    end
    action_cnt <= cart_action.group(
      [cart_action.session, cart_action.item, cart_action.action],
      count(cart_action.reqid))
  end
  declare
  def do_checkout
    del_items = action_cnt.map{|a| a.item if a.action == "Del"}
    status <= join([action_cnt, checkout_msg]).map do |a, c|
      if a.action == "Add" and not del_items.include? a.item
        [c.client, c.server, a.session, a.item, a.cnt]
      end
    end
    status <= join([action_cnt, action_cnt,
                    checkout_msg]).map do |a1, a2, c|
      if a1.session == a2.session and a1.item == a2.item and
         a1.session == c.session and
         a1.action == "A" and a2.action == "D"
        [c.client, c.server, c.session, a1.item, a1.cnt - a2.cnt]
      end
    end
   response_msg <~ status.group(
      [status.client, status.server, status.session],
      accum(status.cnt.times.map{status.item}))
  end
end
```

conclusion

- CALM theorem
  - what is coordination *for*? non-monotonicity.
  - pinpoint non-monotonic *points of order*
    - coordination or taint tracking
- Bloom
  - declarative, disorderly DSL for distributed programming
    - bud: organic Ruby embedding
  - CALM analysis of monotonicity
    - synthesize coordination/compensation
  - releasing to the dev community
    - friends-and-family next month
    - public beta, Fall 2011

more?

http://bloom.cs.berkeley.edu

thanks to:

Microsoft Research

Yahoo! Research

IBM Research

NSF

AFOSR

backup

influence propagation…?

- Technology Review TR10 2010:
  - *"The question that we ask is simple: is the technology likely to change the world?"*

- Fortune Magazine 2010 Top in Tech:
  - *"Some of our choices may surprise you."*
- Twittersphere:
  - *"Read this. Read this now."*

relative to LP and active DB

- "Unlike earlier efforts such as Prolog, active database languages, and our own Overlog language for distributed systems [16], Bloom is purely declarative: the syntax of a program contains the full specification of its semantics, and there is no need for the programmer to understand or reason about the behavior of the evaluation engine. Bloom is based on a formal temporal logic called Dedalus [3]."

why ruby?

- “Bud uses a Ruby-flavored syntax, but this is not fundamental; we have experimented with analogous Bloom embeddings in other languages including Python, Erlang and Scala, and they look similar in structure.”

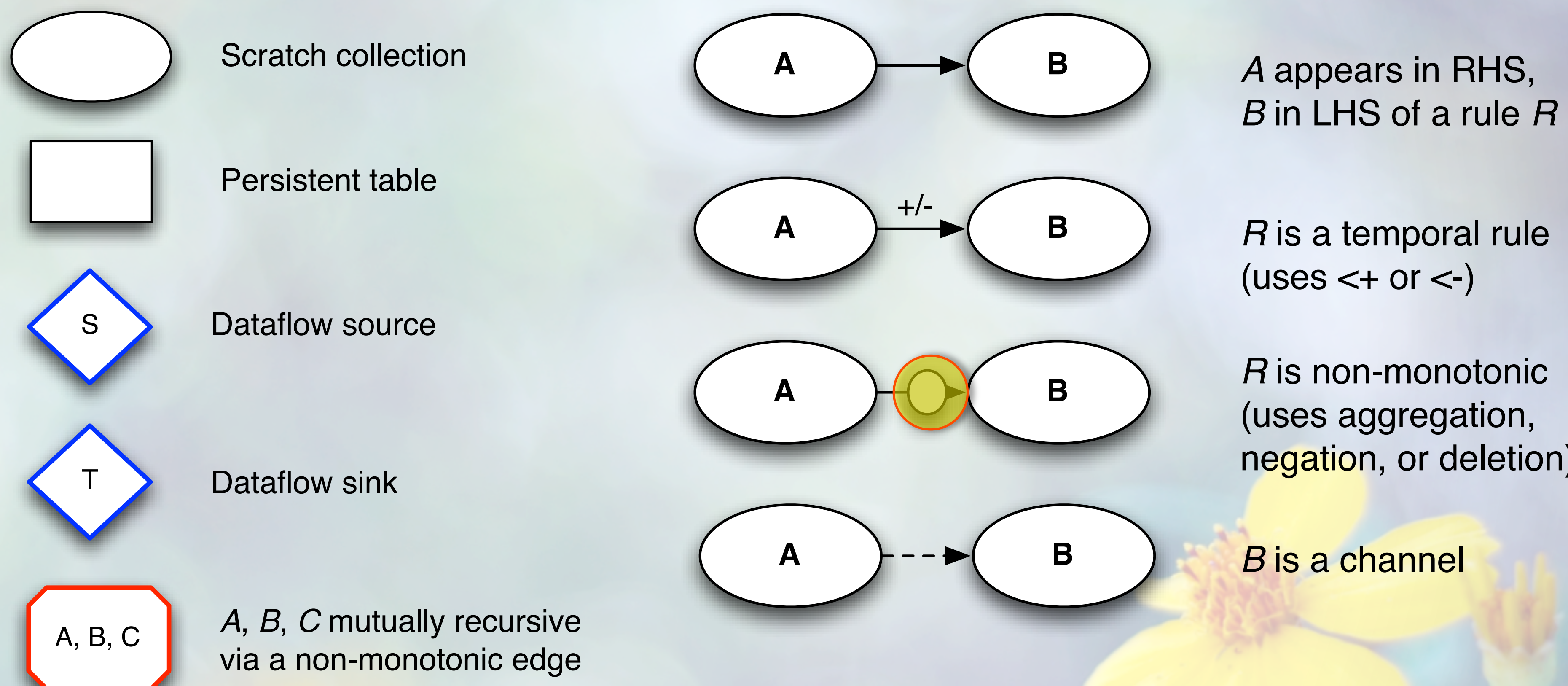
what about erlang?

- "we did a simple Bloom prototype DSL in Erlang (which we cannot help but call "Bloomerlang"), and there is a natural correspondence between Bloom-style distributed rules and Erlang actors. However there is no requirement for Erlang programs to be written in the disorderly style of Bloom. It is not obvious that typical Erlang programs are significantly more amenable to a useful points-of-order analysis than programs written in any other functional language. For example, ordered lists are basic constructs in functional languages, and without program annotation or deeper analysis than we need to do in Bloom, any code that modifies lists would need be marked as a point of order, much like our destructive shopping cart"

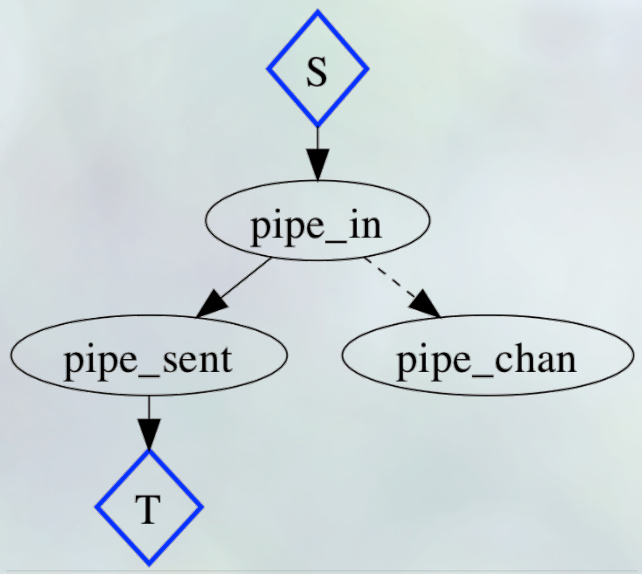
CALM analysis for traditional languages?

- We believe that Bloom's "disorderly by default" style encourages order-independent programming, and we know that its roots in database theory helped produce a simple but useful program analysis technique. While we would be happy to see the analysis "ported" to other distributed programming environments, it may be that design patterns using Bloom-esque disorderly programming are the natural way to achieve this.

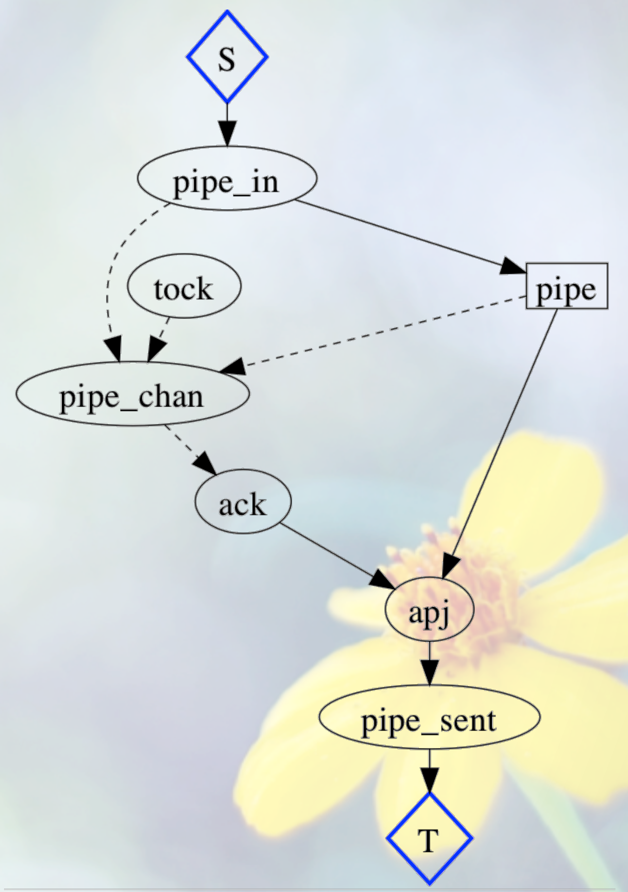
dependency graphs



dependency graphs

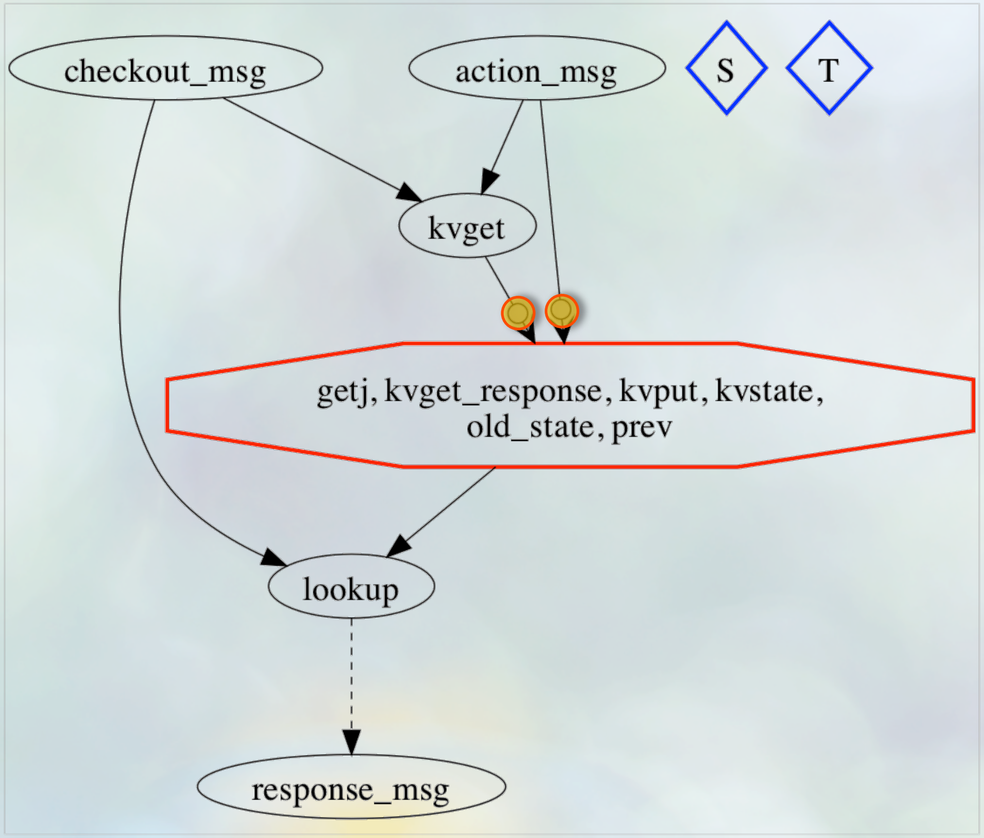


BestEffortDelivery

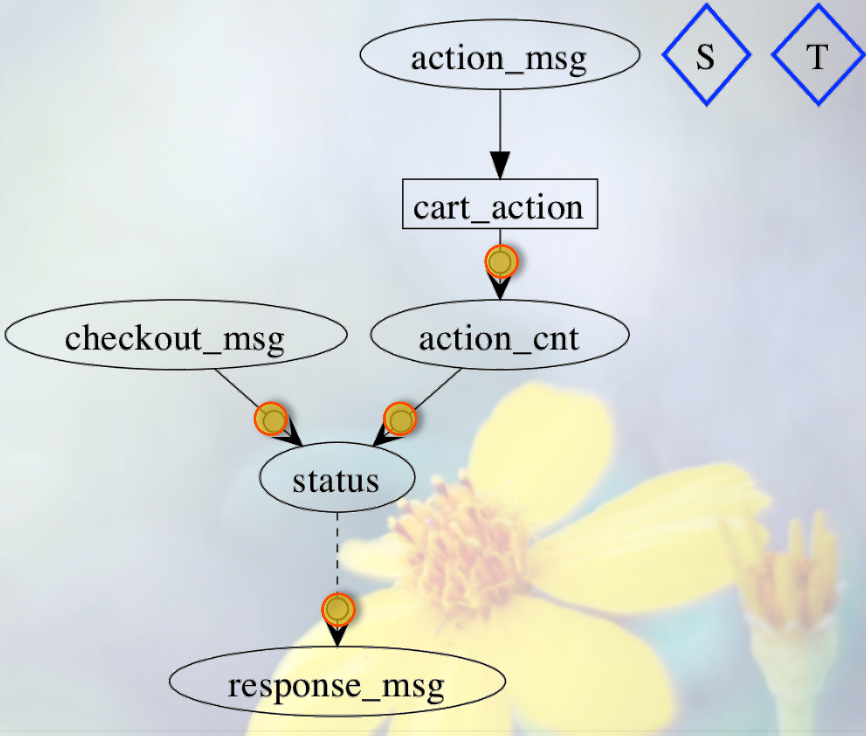


ReliableDelivery

2 cart implementations



destructive



disorderly

example analysis in paper: replicated shopping carts

- “destructive” cart implements a replicated key/value store
  - key: session id
  - value: array of the items in cart
  - add/delete “destructively” modify the value
- “disorderly” cart uses accumulation and aggregation
  - adds/deletes received/replicated monotonically
  - checkout requires counting up the adds/deletes
  - hence coordinate only at checkout time

Building on Quicksand

- Campbell/Helland CIDR 2009

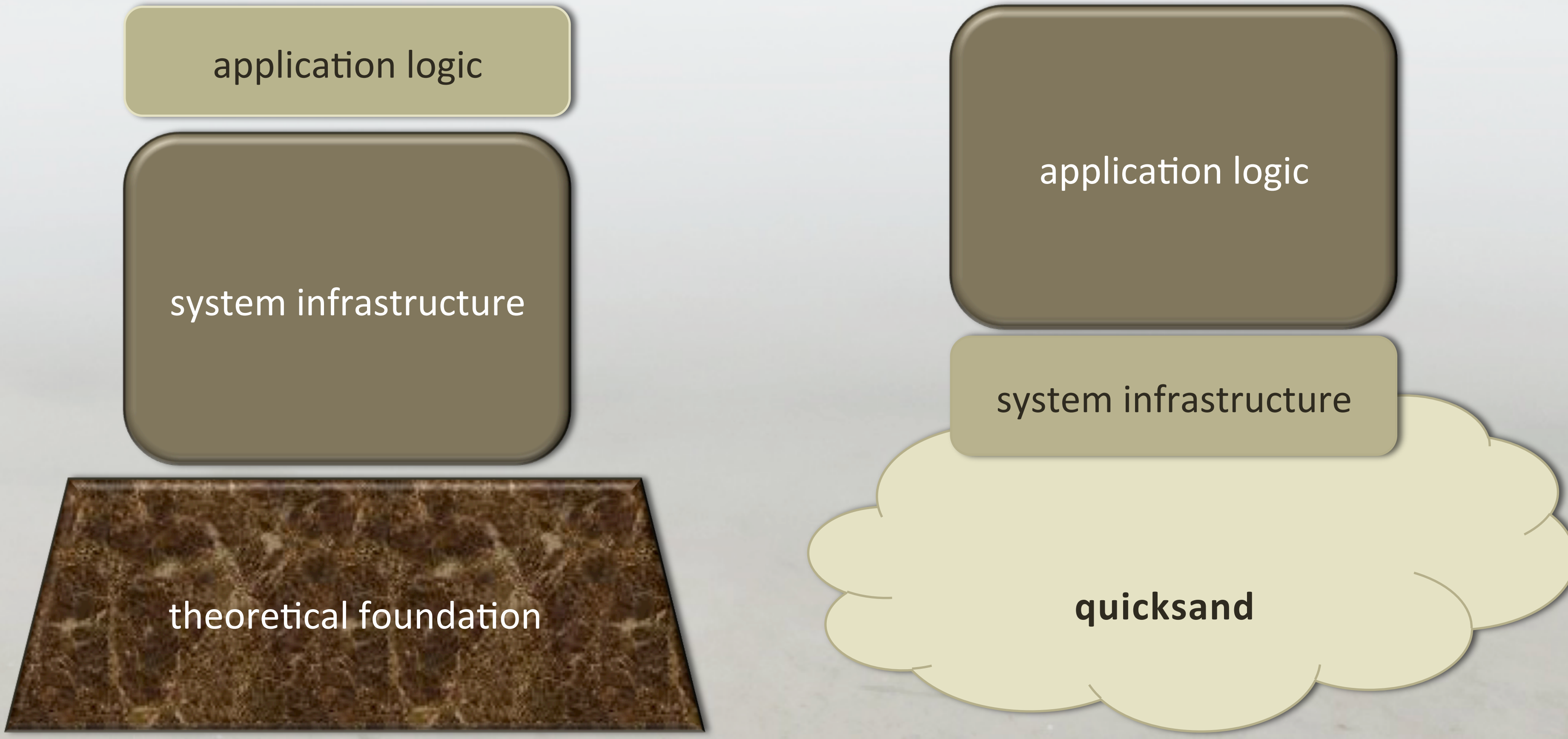
- goal: avoid coordination entirely
- maxim: memories, guesses and apologies

- can we use Bloom analysis to automate/prove correctness of this?
  - initial ideas so far

from quicksand & maxims to code & proofs

- "guesses": easy to see in dependency graph
  - any collection downstream of an uncoordinated point of order
  - compiler rewrites schema to add "taint" attribute to these
    - and rewrites rules to carry taint bit along
- "memories" at interfaces
  - compiler interposes table in front of any tainted output interface
- "apologies"
  - need to determine when "memory" tuples were inconsistent
  - idea: wrap tainted code blocks with "background" coordination check
    - upon success, garbage-collect relevant "memories"
    - upon failure, invoke custom "apology" logic to achieve some invariant
    - ideally, prove that inconsistent tuples + apology logic = invariant satisfied

the shift


application logic
system infrastructure
theoretical foundation
application logic
system infrastructure
quicksand

ruby embedding

- `class Bud`
  - "declare" methods for collections of Bloom statements
    - checked for legality, potentially optimized/rewritten
  - template methods for schemas and data
- all the usual Ruby goodness applies
  - rich dynamic type system
  - OO inheritance, mixins (~multiple inheritance), encapsulation
  - functional programming comprehension syntax
  - libraries for everything under the sun

a taste of ruby

inheritance
mixins
Enumerables and code blocks

```ruby
module MixMeIn
  def mixi
    "who do we appreciate"
  end
end

class SuperDuper
  def doit
    "a super duper bean"
  end
end
```

```ruby
class Submarine < SuperDuper
  include MixMeIn
  def doit
    "a yellow submarine"
  end
  def sing
    puts "we all live in " + doit
  end
  def chant(nums)
    out = nums.map { |n|  n*2 }
    puts out.inspect + " " + mixi
  end
end

s = Submarine.new
s.sing ; s.chant([1,2,3,4])
```

example app: shopping cart

- replicated for HA and low latency
- clients associated with unique session IDs
- add_item, deleted_item, checkout

- **challenge:** guarantee *eventual consistency* of replicas
- **maxim:** use commutative operations
  - c.f. Amazon Dynamo, Campbell/Helland "Building on Quicksand"
  - easier said than done!

abstract interfaces

Bloom

```
module CartClientProtocol
  def state
    interface input, :client_action,
      ['server', 'session', 'reqid'], ['item', 'action']
    interface input, :client_checkout,
      ['server', 'session', 'reqid']
    interface output, :client_response,
      ['client', 'server', 'session'], ['contents']
  end
end

module CartProtocol
  def state
    channel :action_msg,
      ['@server', 'client', 'session', 'reqid'],
      ['item', 'action']
    channel :checkout_msg,
      ['@server', 'client', 'session', 'reqid']
    channel :response_msg,
      ['@client', 'server', 'session'], ['contents']
  end
end
```

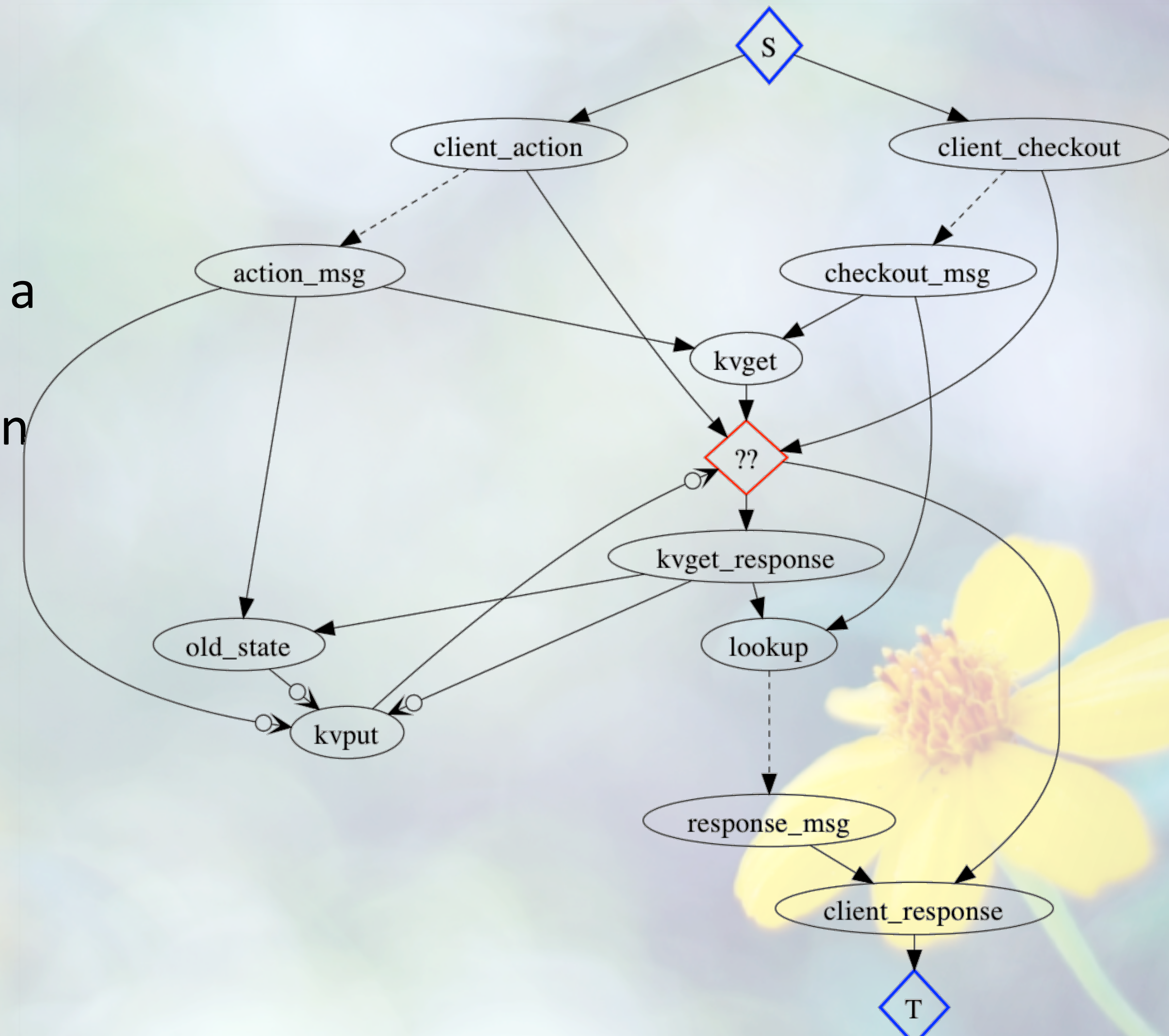
simple realization

Bloom

```
module CartClient
  include CartProtocol
  include CartClientProtocol

  declare
  def client
    action_msg <~ client_action.map do |a|
      [a.server, @local_addr, a.session, a.reqid, a.item,
a.action]
    end
    checkout_msg <~ client_checkout.map do |a|
      [a.server, @local_addr, a.session, a.reqid]
    end
    client_response <= response_msg
  end
end
```

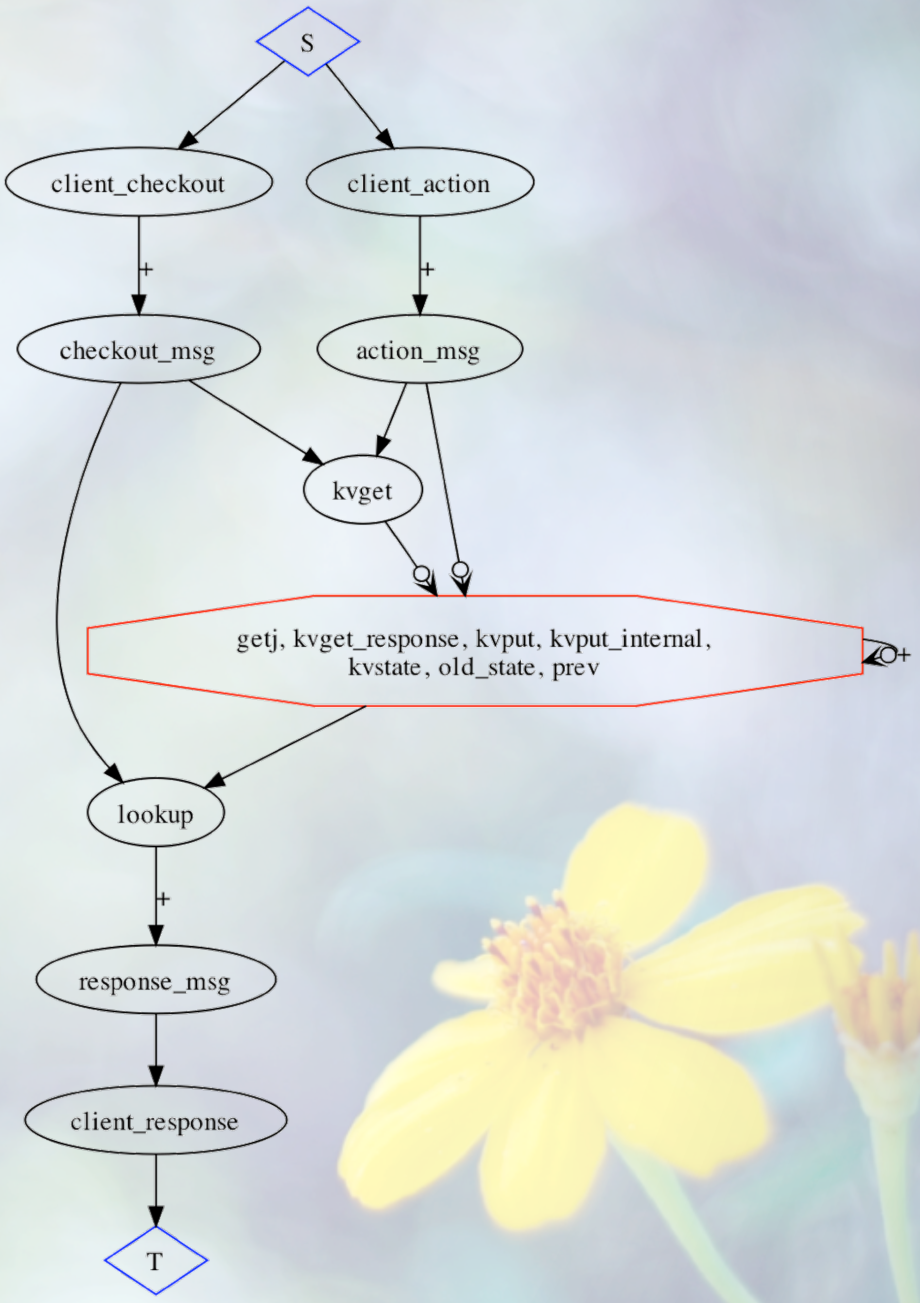
destructive cart

- disconnected because we haven't picked a kvs implementation yet

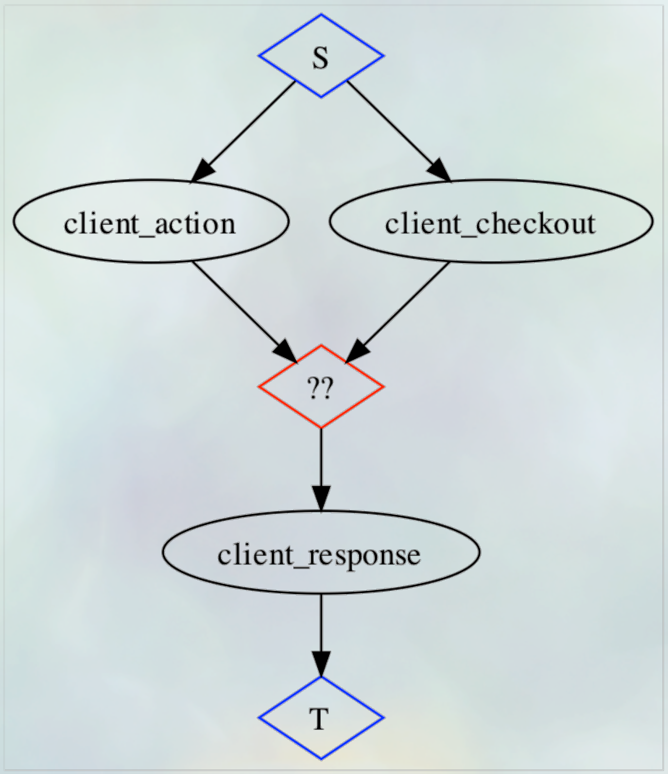


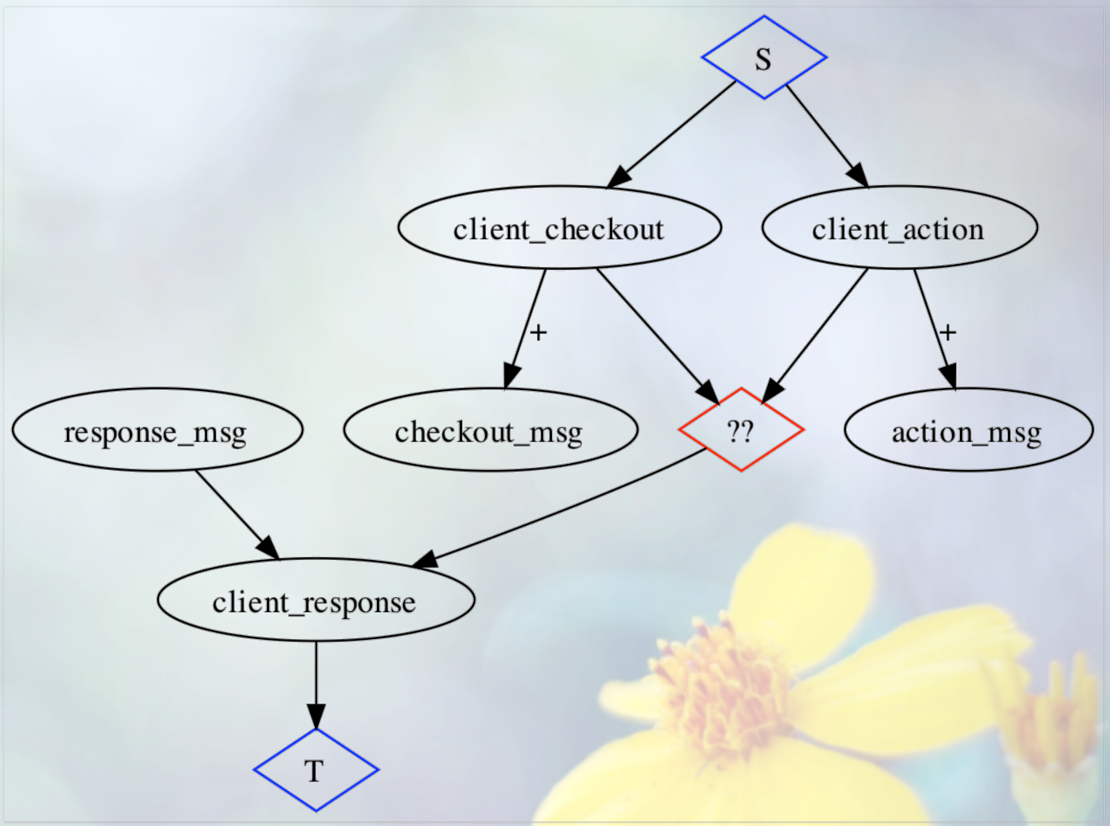
destructive cart

- basic KVS interposes a point of order into the dataflow



abstract and concrete clients





- note that concrete client is still underspecified: we haven't supplied an implementation of the cart yet!

simple key/ value store

```
module KVSProtocol
  def state
    super
    interface input, :kvput, ['client', 'key', 'reqid'],
                             ['value']
    interface input, :kvget, ['reqid'], ['key']
    interface output, :kvget_response, ['reqid'],
                             ['key', 'value']
  end
end
```

simple KVS

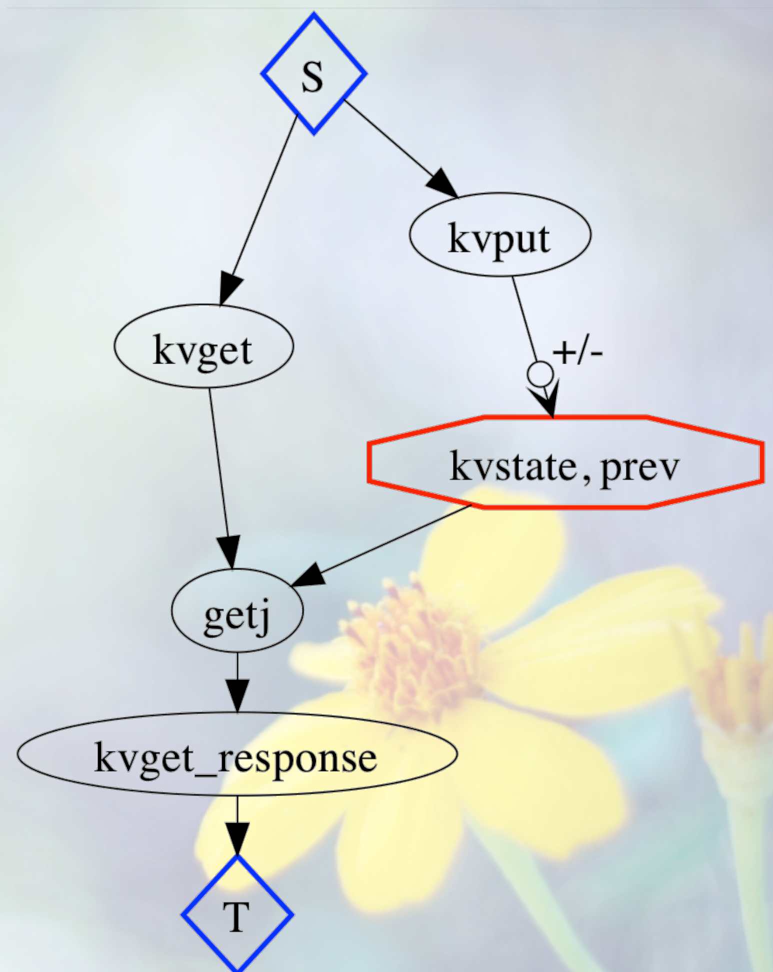
- no replication
- deletion on each put
  - gets worse with replication!

```
module BasicKVS
  include KVSProtocol

  def state
    table :kvstate, ['key'], ['value']
  end

  declare
  def do_put
    kvstate <+ kvput.map{|p| [p.key, p.value]}
    prev = join [kvstate, kvput],
                [kvstate.key, kvput.key]
    kvstate <- prev.map{|b, p| b}
  end

  declare
  def do_get
    getj = join [kvget, kvstate],
                [kvget.key, kvstate.key]
    kvget_response <= getj.map do |g, t|
      [g.reqid, t.key, t.value]
    end
  end
end
```

simple key/val store

- any path through kvput crosses both a point of order and a temporal edge.
- where's the non-monotonicity?
  - state update in the KVS
  - easy syntactic check!

```
kvstate <- prev.map{|b, p| b}
```

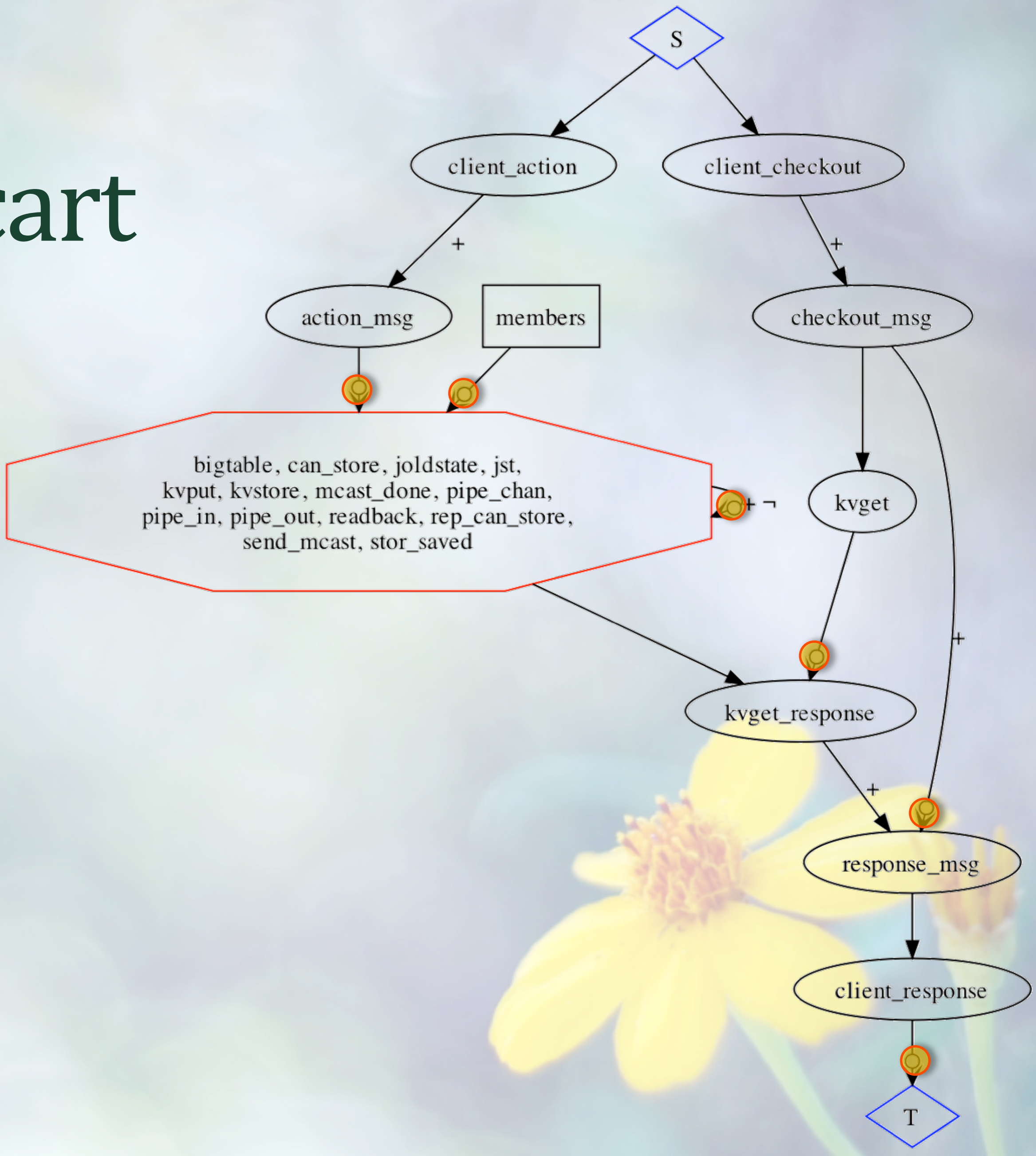


simple syntax check

```
module BasicKVS
  include KVSProtocol

  def state
    table :kvstate, ['key'], ['value']
  end

  declare
  def do_put
    kvstate <+ kvput.map{|p| [p.key, p.value]}
    prev = join [kvstate, kvput],
                [kvstate.key, kvput.key]
    # dude, it's here! (<-)
    kvstate <- prev.map{|b, p| b}
  end

  declare
  def do_get
    getj = join [kvget, kvstate],
                [kvget.key, kvstate.key]
    kvget_response <= getj.map do |g, t|
      [g.reqid, t.key, t.value]
    end
  end
end
```

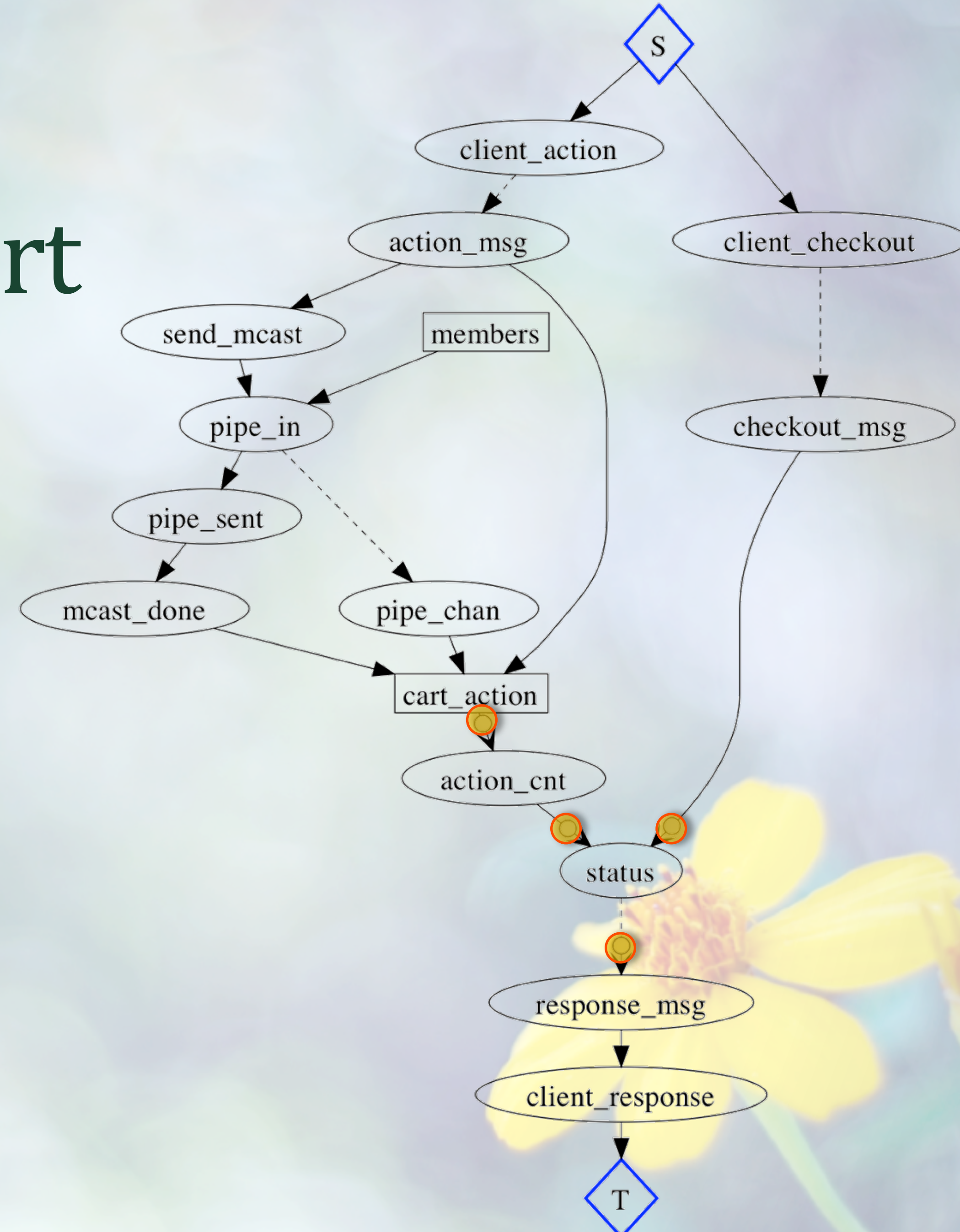
complete destructive cart

- analysis: bad news
  - coordinate on *each client action*
    - *add or delete*
  - coordinate on *each KVS replication*
- what if we skip coordination?
  - assert: actions are commutative
  - no way for compiler to check
  - and in fact it's wrong!



complete disorderly cart

- client actions *and cart replication* all monotonic
- point of order to handle *checkout* messages

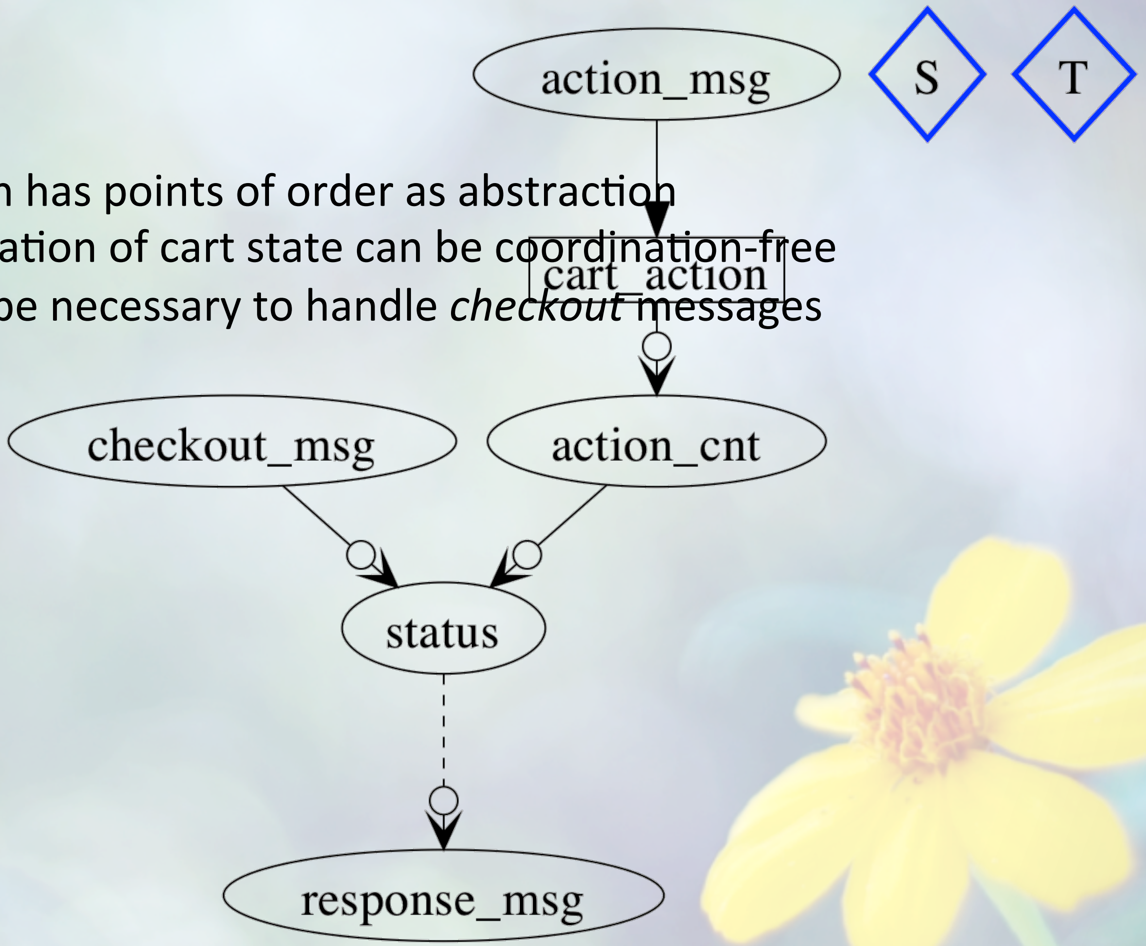


final analysis: destructive

- point of order on each client request for cart update
- this was visible even with the simplest KVS
  - only got worse with replication
- what to do?
  1. assert that operations commute, and leave as is
     - informal, bug-prone, subject to error creep over time
     - there's already a bug: deletes may arrive before adds at some replicas
  2. add a round of distributed coordination for each update
     - e.g. 2PC or Paxos
     - this makes people hate ACID
  3. best solution: a better cart abstraction!
     - *move* that point of order to a lower-frequency operation

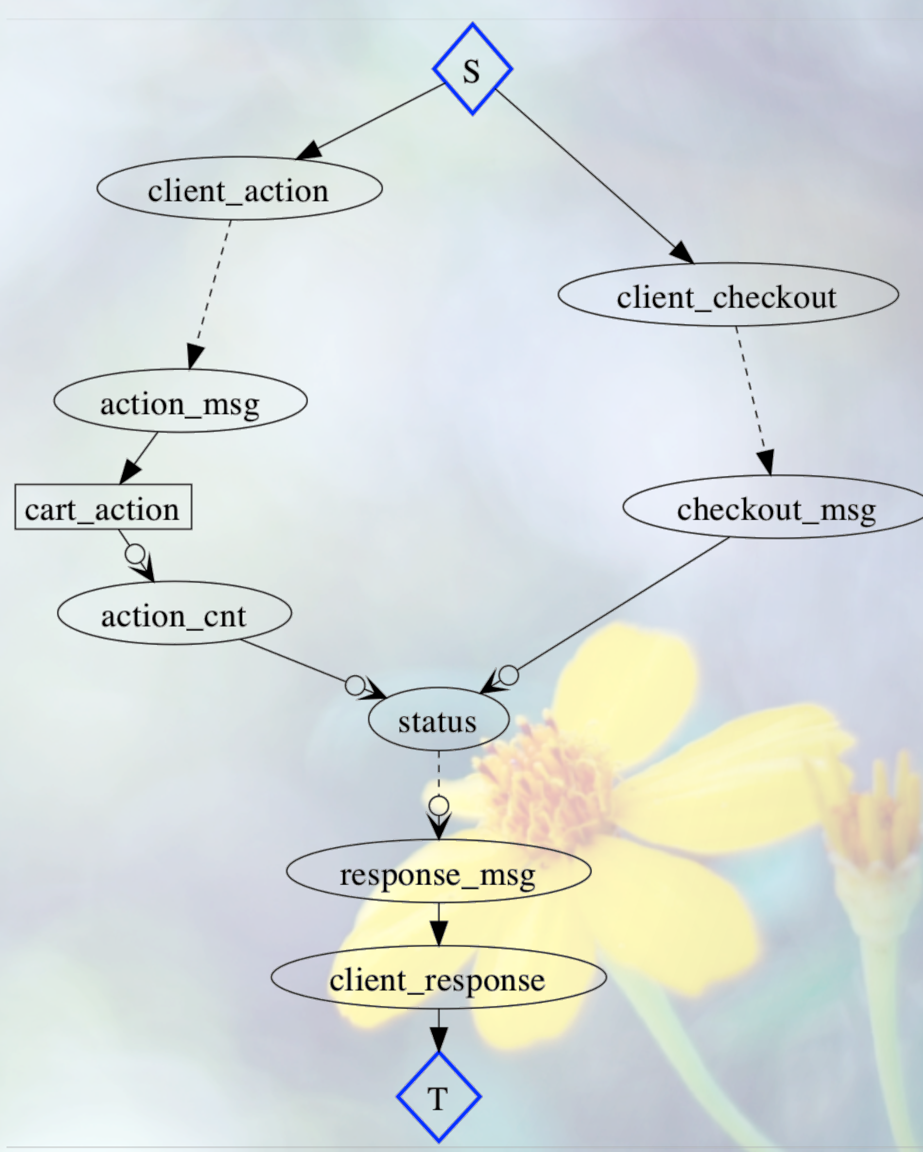
simple disorderly skeleton

concrete implementation has points of order as abstraction
client updates and replication of cart state can be coordination-free
some coordination may be necessary to handle *checkout* messages



… and its composition with the client code

- note points of order (circles) corresponding to aggregation



replication

We take the abstract class Multicast...

Bloom

```
module MulticastProtocol
  def state
    super
    table :members, ['peer']
    interface input, :send_mcast, ['ident'], ['payload']
    interface output, :mcast_done, ['ident'], ['payload']
  end
end

module Multicast
  include MulticastProtocol
  include DeliveryProtocol
  include Anise
  annotator :declare

  declare
  def snd_mcast
    pipe_in <= join([send_mcast, members]).map do |s, m|
      [m.peer, @addy, s.ident, s.payload]
    end
  end

  declare
  def done_mcast
    # override me
    mcast_done <= pipe_sent.map{|p| [p.ident, p.payload] }
  end
end
```

replication

... and extend the disorderly cart to use it (along with the concrete multicast implementation BestEffortDelivery)

Bloom

```
module ReplicatedDisorderlyCart
  include DisorderlyCart
  include Multicast
  include BestEffortDelivery

  declare
  def replicate
    send_mcast <= action_msg.map do |a|
      [a.reqid, [a.session, a.reqid, a.item, a.action]]
    end
    cart_action <= mcast_done.map {|m| m.payload }
    cart_action <= pipe_chan.map{|c| c.payload }
  end
end
```

final analysis: disorderly cart

- concrete implementation has points of order as abstraction
- client updates and replication of cart state can be coordination-free
- some coordination may be necessary to handle *checkout* messages