Disorderly programming.
CALM analysis.
The future is already here

• Nearly all nontrivial systems are (or are becoming) distributed
• Programming distributed systems is hard
• Reasoning about them is harder
Outline

1. Disorderly Programming
2. The Bloom programming language
3. CALM Analysis and visualization
4. Challenge app: replicated shopping carts
Programming distributed systems
The state of the art

Order is pervasive in the stored program model

- Program state is an ordered array
- Program logic is a list of instructions with a PC
The state of the art

Order is pervasive in the stored program model

• Program state is an ordered array
• Program logic is a list of instructions with a PC
The state of the art

Order is pervasive in the stored program model

Parallelism and concurrency via retrofits:

• Threads
  – execute many copies of sequential instructions

• Event-driven programming
  – a single sequential program dispatches async processes
The state of the art

In distributed systems, order is

• expensive to enforce
• frequently unnecessary / overspecified
• easy to get wrong
The art of the state

Disorderly programming

Design maxim:

Think about the hard stuff. Forget about the easy stuff.
The art of the state

Disorderly programming

• Program state is unordered collections
• Program logic is an unordered set of rules
The art of the state

Disorderly programming

• Program state is unordered collections
• Program logic is an unordered set of rules
• Independence and concurrency are assumed
• Ordering of data or instructions are explicit, special-case behaviors
  – This is the hard stuff
<- bloom
BUD: Bloom Under Development

- Ruby internal DSL
- Set comprehension style of programming
- Fully declarative semantics
  - Based on Dedalus (Datalog + time)
Bloom Rules

multicast <- (message * members) do |mes, mem|
    [mem.address, mes.id, mes.payload]
end
Bloom Rules

multicast ← (message * members) do |mes, mem|
  [mem.address, mes.id, mes.payload]
end

<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?
Bud language features

• Module system
  – Encapsulation and composition via mixins
  – Separation of abstract interfaces and concrete implementations

• Metaprogramming and reflection
  – Program state is program data

• Pay-as-you-code schemas
  – Default is key => value
Operational model

Bloom Rules
atomic, local, deterministic

Local Updates
System Events
Network

Now
Next
Network
Writing distributed programs in Bloom
Abstract Interfaces and Declarations

module DeliveryProtocol
    state do
        interface input, :pipe_in,
            [:dst, :src, :ident] => [:payload]
        interface output, :pipe_sent,
            pipe_in.schema
    end
end
Abstract Interfaces and Declarations

module DeliveryProtocol
  state do
    interface input, :pipe_in,
      [:dst, :src, :ident] => [:payload]
    interface output, :pipe_sent,
      pipe_in.schema
  end
end
Concrete Implementations

```plaintext
module BestEffortDelivery
  include DeliveryProtocol

  state do
    channel :pipe-chan, pipe-in.schema
  end

  bloom :snd do
    pipe-chan <-- pipe-in
  end

  bloom :done do
    pipe-sent <= pipe-in
  end
end
```
Concrete Implementations

module BestEffortDelivery
  include DeliveryProtocol

state do
  channel :pipe_chan, pipe_in.schema
end

bloom :snd do
  pipe_chan <= pipe_in
end

bloom :done do
  pipe_sent <= pipe_in
end
end
Concrete Implementations

module BestEffortDelivery
  include DeliveryProtocol

  state do
    channel :pipe_chan, pipe_in.schema
  end

  bloom :snd do
    pipe_chan =~ pipe_in
  end

  bloom :done do
    pipe_sent <= pipe_in
  end
end
Concrete Implementations

module ReliableDelivery
include DeliveryProtocol
import BestEffortDelivery => :bed

state do
  table :buf, pipe_in.schema
  channel :ack, [:@src, :dst, :ident]
  periodic :clock, 2
end

bloom :remember do
  buf <= pipe_in
  bed.pipe_in <= pipe_in
  bed.pipe_in <= (buf * clock).lefts
end

bloom :send_ack do
  ack <= bed.pipe_chan do|p|
    [p.src, p.dst, p.ident]
  end
end

bloom :done do
  temp :msg_acked <= (buf * ack).lefts
  (:ident => :ident)
  pipe_sent <= msg_acked
  buf <= msg_acked
end
Concrete Implementations

module ReliableDelivery
include DeliveryProtocol
import BestEffortDelivery => :bed

state do
  table :buf, pipe_in.schema
  channel :ack, list:@src, dst, ident]
  periodic :clock, 2
end

bloom :remember do
  buf <= pipe_in
  bed.pipe_in <= pipe_in
  bed.pipe_in <= (buf * clock).lefts
end

bloom :send_ack do
  ack <= bed.pipe_chan do |p|
    [p.src, p.dst, p.ident]
  end
end

bloom :done do
  temp :msg_acked <= (buf * ack).lefts
  (:ident => :ident)
  pipe_sent <= msg_acked
  buf <= msg_acked
end
Concrete Implementations

module ReliableDelivery

include DeliveryProtocol
import BestEffortDelivery => :bed

state do
  table :buf, pipe_in.schema
  channel :ack, [@src, :dst, :ident]
  periodic :clock, 2
end

bloom :remember do
  buf <= pipe_in
  bed.pipe_in <= pipe_in
  bed.pipe_in <= (buf * clock).lefts
end

bloom :send_ack do
  ack <- bed.pipe_chan do |p|
  [p.src, p.dst, p.ident]
end
end

Asynchrony +
Nonmonotonicity
CALM Analysis
Monotonic Logic

The more you know, the more you know.

- E.g., select, project, join, union
  - (Filter, map, compose, merge)
- Pipelinable
- Insensitive to the ordering of inputs
Nonmonotonic Logic

Possible belief revision or retraction

(Un-firing missiles)

• E.g. aggregation, negation as complement
  – (Counting, taking a max/min, checking that an element isn’t in a hash)

• Blocking

• Ordering of inputs may affect results
Consistency

A family of correctness criteria for data-oriented systems

- Do my replicas all converge to the same final state?
- Am I missing any partitions?

State of the art: choose an extreme
Strong Consistency

Coordination: baked-in consistency via protocol
• Serializable reads and writes, atomic transactions, ACID
• 2PC, paxos, GCS, etc
• Establish a total order of updates

Problem: latency, availability
Strong Consistency
Loose Consistency

Application-specific correctness via “rules of thumb”

• Asynchronous replication
• Commutative (i.e., reorderable) operations
• Custom compensation / repair logic
• Tolerate all update orders

Problem: informal, fragile
Loose Consistency
CALM Conjecture

Consistency ``is'' logical monotonicity
CALM Analysis

• Logic programming dependency analysis applied to a distributed system

• Represent program as a directed graph
  – Nodes are collections
  – Arcs are rules
    • may be labeled asynchronous or nonmonotonic

• Tools identify “points of order”
  – Where different input orderings may produce divergent outputs
Points of order

Nonmonotonicity after asynchrony in dataflow

• Asynchronous messaging => nondeterministic message orders
• Nonmonotonic => order-sensitive
• Coordination may be required to ensure convergent results.
• Here (and only here) the programmer should reason about ordering!
Resolving points of order

1. Coordinate
   – Enforce an ordering over inputs
   – Or block till input set is completely determined
Resolving points of order

1. Coordinate
   - Enforce an ordering over inputs
   - Or block till input set is completely determined

2. Speculate
   - Track ``taint''
   - Tainted conclusions may be retracted
Resolving points of order

1. Coordinate
   - Enforce an ordering over inputs
   - Or block till input set is completely determined

2. Speculate
   - Track ``taint''
   - Tainted conclusions may be retracted

3. Rewrite
   - ``Push down'' points of order
     - Past async edges (localize coordination)
     - To edges with less flow (minimize coordination)
Resolving points of order

1. Coordinate
   - Enforce an ordering over inputs
   - Or block till input set is completely determined

2. Speculate
   - Track ``taint''
   - Tainted conclusions may be retracted

3. Rewrite
   - ``Push down'' points of order
     • Past async edges (localize coordination)
     • To edges with less flow (minimize coordination)
Shopping Carts
Example application: a replicated shopping cart

- Replicated to achieve high availability and low latency
- Clients are associated with unique session ids
- Add item, delete item and “checkout” operations

Challenge:
Ensure that replicas are “eventually consistent”
Carts done two ways

1. A “destructive” cart
   – Use a replicated KVS as a storage system
   – Client session id is the key
   – Value is an array representing cart contents
   – Checkout causes value array to be sent to client

2. A “disorderly” cart
   – Accumulate (and asynchronously replicate) a set of cart actions
   – At checkout, count additions and deletions, take difference
   – Aggregate the summary into an array “just in time”
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
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 do
  table :kvstate, [:key] => [:value]
end

bloom :mutate do
  kvstate <+ kvput { |s| [s.key, s.value] }
  kvstate <- (kvstate * kvput).lefts(:key => :key)
end

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

bloom :delete do
  kvstate <- (kvstate * kvdel).lefts(:key => :key)
end
A simple key/value store

module BasicKVS
  include KVSProtocol

state do
  table :kvstate, [:key] => [:value]
end

bloom :mutate do
  kvstate <+ kvput { |s| [s.key, s.value] }
  kvstate <- (kvstate * kvput).lefts(:key => :key)
end

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

bloom :delete do
  kvstate <- (kvstate * kvdel).lefts(:key => :key)
end
end
A simple key/value store

module BasicKVS
    include KVSProtocol

    state do
        table :kvstate, [:key] => [:value]
    end

    bloom :mutate do
        kvstate <+ kvput {s | [s.key, s.value]}
        kvstate <- (kvstate * kvput).lefts(:key => :key)
    end

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

    bloom :delete do
        kvstate <- (kvstate * kvdel).lefts(:key => :key)
    end
end

Nonmononic operations
A simple key/value store

module BasicKVS
include KVSProtocol

state do
  table :kvstate, [:key] => [:value]
end

bloom :mutate do
  kvstate <+ kvput {s | s.key, s.value}
  kvstate <- (kvstate * kvput).lefts(:key => :key)
end

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

bloom :delete do
  kvstate <- (kvstate * kvdel).lefts(:key => :key)
end
end

Nonmononic operations
module BasicKVS
include KVSProtocol

state do
  table :kvstate, [:key] => [:value]
end

bloom :mutate do
  kvstate <+ kvput {s | [s.key, s.value]}
  kvstate <- (kvstate * kvput).lefts(:key => :key)
end

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

bloom :delete do
  kvstate <- (kvstate * kvdel).lefts(:key => :key)
end
end

Nonmononic operations
module BasicKVS
include KVSProtocol

state do
  table :kvstate, [:key] => [:value]
end

bloom :mutate do
  kvstate <+ kvput {s | [s.key, s.value]}
  kvstate <- (kvstate * kvput).lefts(:key => :key)
end

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

bloom :delete do
  kvstate <- (kvstate * kvdel).lefts(:key => :key)
end
end

Nonmononic operations
``Destructive” Cart

module DestructiveCart
  include CartProtocol
  include KVSProtocol

  declare
  def queueing
    kvget <= action_msg { |a| [a.reqid, a.session] }  
    kvput <= action_msg do |a|
      if a.action == "Add" and not kvget_response.map{|b| b.key}.include? a.session
        [a.client, a.session, a.reqid, Array.new.push(a.item)]
      end
    end
  end

  old_state = join [kvget_response, action_msg], [kvget_response.key, action_msg.session]
  kvput <= old_state do |b, a|
    if a.action == "Add"
      [a.client, a.session, a.reqid, {b.value.clone.push(a.item)}]
    elsif a.action == "Del"
      [a.client, a.session, a.reqid, delete_one(b.value, a.item)]
    end
  end
  end

  declare
  def finish
    kvget <= checkout_msg {|c| [c.reqid, c.session] }  
    lookup = join([kvget_response, checkout_msg], [kvget_response.key, checkout_msg.session])
    response_msg <= lookup do |r, c|
      [r.client, r.server, r.key, r.value, nil]
    end
  end
end
``Destructive’’ Cart

module DestructiveCart
  include CartProtocol
  include KVSProtocol

declare
def queueing
  kvget <= action_msg { |a| [a.reqid, a.session] }
  kvput <= action_msg do |a|
    if a.action == "Add" and not kvget_response.map{|b| b.key}.include? a.session
      [a.client, a.session, a.reqid, Array.new.push(a.item)]
    end
  end
  end

old_state = join [kvget_response, action_msg], [kvget_response.key, action_msg.session]
kvput <= old_state do |b, a|
  if a.action == "Add"
    [a.client, a.session, a.reqid, (b.value.clone.push(a.item))]
  elsif a.action == "Del"
    [a.client, a.session, a.reqid, delete_one(b.value, a.item)]
  end
  end
  end

declare
def finish
  kvget <= checkout_msg { |c| [c.reqid, c.session] }
  lookup = join([kvget_response, checkout_msg], [kvget_response.key, checkout_msg.session])
  response_msg <= lookup do |r, c|
    [r.client, r.server, r.key, r.value, nil]
  end
  end
end
"Destructive" Cart

Diagram showing the process:
- S
  - client_checkout
    - checkout_msg
      - kvget
        - kvget_response, kvput, kvstate
      - response_msg
    - client_response
- T
  - kvdel
  - action_msg
    - +/-
``Destructive'' Cart

Asynchrony
``Destructive'' Cart

Asynchrony

Nonmonotonicity

client_checkout → client_action

checkout_msg → action_msg

kvget → kvget_response, kvput, kvstate

response_msg → client_response

S

kvdel

T
``Destructive’’ Cart

Asynchrony

Nonmonotonicity

Divergent Results?
Add coordination? E.g.,
- Synchronous replication
- Paxos
``Destructive” Cart

Add coordination? E.g.,
- Synchronous replication
- Paxos

\( n = |\text{client\_action}| \)
\( m = |\text{client\_checkout}| = 1 \)

\( n \) rounds of coordination
module DisorderlyCart
  include CartProtocol

state do
  table :cart_action, [:session, :reqid] => [:item, :action]
  scratch :action_cnt, [:session, :item, :action] => [:cnt]
  scratch :status, [:server, :client, :session, :item] => [:cnt]
end

declare

def saved
  cart_action <= action_msg { |c| [c.session, c.reqid, c.item, c.action] }
  action_cnt <= cart_action.group([cart_action.session, cart_action.item, cart_action.action], count (cart_action.reqid))
end

declare

def consider
  status <= (action_cnt * action_cnt * checkout_msg) do |a1, a2, c|
    if a1.session == a2.session and a1.item == a2.item and a1.session == c.session and a1.action == "Add" and a2.action == "Del"
      [c.client, c.server, a1.session, a1.item, a1.cnt - a2.cnt] if (a1.cnt - a2.cnt) > 0
    end
  end
  status <= (action_cnt * checkout_msg) do |a, c|
    if a.action == "Add" and not action_cnt.map{|d| d.item if d.action == "Del"}.include? a.item
      [c.client, c.server, a.session, a.item, a.cnt]
    end
  end
  end
  out = status.reduce({}) do |memo, i|
    memo[[i[0],i[1],i[2]]] ||= []; i[4].times {memo[[i[0],i[1],i[2]]] <<= i[3]}; memo
  end.to_a
  response_msg <= out { |k, v| k <= v}
end
end
``Disorderly Cart''

module DisorderlyCart
include CartProtocol
state do
  table :cart_action, [:session, :reqid] => [:item, :action]
  scratch :action_cnt, [:session, :item, :action] => [:cnt]
  scratch :status, [:server, :client, :session, :item] => [:cnt]
end

declare
def saved
  cart_action <= action_msg { |c| [c.session, c.reqid, c.item, c.action] }
  action_cnt <= cart_action.group([cart_action.session, cart_action.item, cart_action.action], count (cart_action.reqid))
end

declare
def consider
  status <= (action_cnt * action_cnt * checkout_msg) do |a1, a2, c|
    if a1.session == a2.session and a1.item == a2.item and a1.session == c.session and a1.action == "Add" and a2.action == "Del"
      [c.client, c.server, a1.session, a1.item, a1.cnt - a2.cnt] if (a1.cnt - a2.cnt) > 0
    end
  end
  status <= (action_cnt * checkout_msg) do |a, c|
    if a.action == "Add" and not action_cnt.map{|d| d.item if d.action == "Del"}.include? a.item
      [c.client, c.server, a.session, a.item, a.cnt]
    end
  end
end
out = status.reduce({}) do |memo, i|
  memo[[i[0],i[1],i[2]]] ||= []; i[4].times {memo[[i[0],i[1],i[2]]] <<= i[3]}; memo
end.to_a
response_msg <- out {k, v| k <= v}
end
Disorderly Cart Analysis
Disorderly Cart Analysis

Asynchrony
Disorderly Cart Analysis

Asynchrony

Nonmonotonicity
Disorderly Cart Analysis

Asynchrony

Nonmonotonicity

Divergent Results?
Disorderly Cart Analysis

$n = |\text{client}_\text{action}|$

$m = |\text{client}_\text{checkout}| = 1$

1 round of coordination
Replicated Disorderly Cart

Asynchronous (uncoordinated) replication
Replicated Disorderly Cart

Asynchronous (uncoordinated) replication

Still just 1 round of coordination
Summary

• Why disorderly?
  – Order is expensive and sometimes distracting

• When is order really needed?
  – When nonmonotonicity makes the computation order-dependent

• What is coordination for?
  – Re-establishing order, to guarantee consistency.

• CALM <~ Bloom
  – A disorderly programming language
  – Tools to identify points of order
More

Resources:

http://bloom.cs.berkeley.edu
http://bloom-lang.org

Writeups:

• Consistency Analysis in Bloom: A CALM and Collected Approach (CIDR’11)
• Dedalus: Datalog in Time and Space (Datalog2.0)
• The Declarative Imperative (PODS’10 Keynote address)
• Model-theoretic Correctness Criteria for Distributed Systems (in submission)
Queries?