Disorderly programming for a distributed world

Peter Alvaro
UC Berkeley
Joint work

• Peter Alvaro
• Neil Conway
• Joseph M. Hellerstein
• William R. Marczak

Thanks to

• National Science Foundation
• Air Force Office of Scientific Research
• Gifts from IBM, Microsoft and Yahoo! Research
The future is already here

• All 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 Von Neumann model

- Program state is an ordered array
- Program logic is a sequence of instructions
The state of the art

Order is pervasive in the Von Neumann model

Parallelism and concurrency via retrofits:
• Threads
• Event-driven programming
The state of the art

In distributed systems, order is
The state of the art

In distributed systems, order is

• expensive to enforce
The state of the art

In distributed systems, order is

• expensive to enforce

• often unnecessary
The state of the art

In distributed systems, order is
• expensive to enforce
• often unnecessary
• easy to get wrong
The state of the art
The art of the state

Disorderly programming
The art of the state

Disorderly programming:

Computation as transformation
The art of the state

Disorderly programming:

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

Disorderly programming:

• Independence and concurrency are assumed
• Ordering is \textit{explicit}
The art of the state

Disorderly programming

<- bloom
BUD: Bloom Under Development

• Ruby internal DSL
• Set comprehension style of programming
• Declarative semantics
Operational model

Bloom Rules
atomic, local, deterministic

Local Updates
System Events
Network
Bloom Rules

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

<Bloom Rules>

<persistent
table

<transient
scratch

<networked
channel

<transient
periodic

<one-way
interface

<Ruby

<Collection>

<Accumulator>

<From List>

<Expression>

<Collection>

<Accumulator>

<From List>

<Expression>

 persistent
table

 transient
scratch

 networked
channel

 transient
periodic

 one-way
interface

 <= now

 <+ next

 <- not next

 <= later

 (R * S)
join

 R.notin(S)
antijoin

Ruby
Bud language features

• Module system
  – Encapsulation and composition via mixins
  – Abstract interfaces, concrete implementations

• Metaprogramming and reflection
  – The program is data

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

• CALM Analysis
Writing distributed programs in Bloom
Abstract Interfaces and Declarations

```plaintext
module DeliveryProtocol
  state do
    interface input, :pipe_in,
      [:dst, :src, :ident] => [:payload]
    interface output, :pipe_sent, pipe_in.schema
    interface output, :pipe_out, pipe_in.schema
  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
      pipe_out <= pipe_chan
  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 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
CALM Analysis
Asynchronous messaging

You never really know
Asynchronous messaging
Monotonic Logic

The more you know, the more you know.
Monotonic Logic

A  B
C  D
E

select/filter

A
C
E
Monotonic Logic

\[
\begin{align*}
A & \quad B \\
C & \quad f(A) \quad f(B) \\
\end{align*}
\]
Monotonic Logic

A   B
C   D   E

B   F
D

join /
compose

B
D
Monotonic Logic is order-insensitive
Monotonic Logic is *pipelineable*
Nonmonotonic Logic

When do you know for sure?
Nonmonotonic Logic
Nonmonotonic logic is order-sensitive
Nonmonotonic logic is *blocking*
Nonmonotonic logic is *blocking*.

```
set minus
```

```
``Sealed``''
```
CALM Analysis

• Asynchrony => loss of order
• Nonmonotonicity => order-sensitivity
• Asynchrony ; Nonmonotonicity => Inconsistency
CALM Analysis

- Asynchrony => loss of order
- Nonmonotonicity => order-sensitivity
- Asynchrony ; Nonmonotonicity => Inconsistency

``Point of Order''
Resolving points of order
Resolving points of order

1. Ask for permission
Resolving points of order

1. Ask for permission

Coordination => strong consistency
Resolving points of order

1. Ask for permission
2. Ask for forgiveness
Resolving points of order

1. Ask for permission
2. Ask for forgiveness

Compensation, weak consistency
Resolving points of order

1. Ask for permission
2. Ask for forgiveness
3. Ask differently?

Rewrite to reduce consistency cost...
Shopping Carts
Replicated Shopping Carts

Replicated for high availability and low latency

Challenge:
Ensure that replicas are "eventually consistent"
Replicated Shopping Carts

module CartClientProtocol
  state do
    interface input, :client_action,
      [:server, :session, :reqid] => [:item, :action]
    interface input, :client_checkout,
      [:server, :session, :reqid]
    interface output, :client_response,
      [:client, :server, :session] => [:items]
  end
end
Replicated Shopping Carts

module CartClientProtocol
state do
  interface input, :client_action,
      [:server, :session, :reqid] => [:item, :action]
  interface input, :client_checkout,
      [:server, :session, :reqid]
  interface output, :client_response,
      [:client, :server, :session] => [:items]
end
end
Carts done two ways

1. A “destructive” cart
2. A “disorderly” cart
``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.action}) do |k,old,new| old + new
    end]
  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}.to_a.sort]
    end
  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.action}) do |
      k,old,new| old + new
    end]
  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}.to_a.sort]
  end
end
```

React to client updates

React to client checkout
Destructive Cart Analysis
Destructive Cart Analysis

Asynchrony

Nonmonotonicity

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

\[ n = |\text{client_action}| \]
\[ m = |\text{client_checkout}| = 1 \]

\( n \) rounds of coordination
``Disorderly Cart’’

module DisorderlyCart
include CartProtocol

state do
  table :action_log, [:session, :reqid] => [:item, :action]
  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.action] }
end

bloom :on_checkout do
  temp :checkout_log <= (checkout_msg * action_log).rights(:session => :session)
  item_sum <= checkout_log.group([action_log.session, action_log.item],
                               sum(action_log.action)) do |s|
    s if s.last > 0
  end
  session_final <= item_sum.group([:session], accum(:item), accum(:num))
  response_msg <= (session_final * checkout_msg).pairs(:session => :session) do |c,m|
    [m.client, m.server, m.session, c.items.zip(c.counts).sort]
  end
end
```
``Disorderly Cart’’

module DisorderlyCart
  include CartProtocol

state do
  table :action_log, [:session, :reqid] => [:item, :action]
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.action] }
end

bloom :on_checkout do
  temp :checkout_log <= (checkout_msg * action_log).rights(:session => :session)
  item_sum <= checkout_log.group([action_log.session, action_log.item],
                                 sum(action_log.action)) do |s|
    s if s.last > 0
  end
  session_final <= item_sum.group([:session], accum(:item), accum(:num))
  response_msg <= (session_final * checkout_msg).pairs(:session => :session) do |c,m|
    [m.client, m.server, m.session, c.items.zip(c.counts).sort]
  end
end
end
```
Disorderly Cart Analysis

\[ n = \mid \text{client_action} \mid \]
\[ m = \mid \text{client_checkout} \mid = 1 \]

1 round of coordination
Replicated Disorderly Cart

Asynchronous (uncoordinated) replication

Still just 1 round of coordination
Teaching ← bloom
Summary

• Why disorderly?
  – Order is a scarce (and distracting!) resource

• When is order really needed?
  – To resolve nonmonotonicity

• 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://boom.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?
Languages regarding languages

Other Languages

Bloom