KEEP CALM AND QUERY ON

PROGRAMMING PRINCIPLES FOR A DISTRIBUTED ERA

JOE HELLERSTEIN
BERKELEY
TRIFACTA
OUTLINE
BY
ANALOGY
THE GLORY OF HISTORY
ELEGANTLY
TENACIOUS
FRUSTRATINGLY PERSISTENT
KEEP CALM AND CARRY ON

OUTLINE

HISTORY

TENACITY

FRUSTRATION

CALM
THE von NEUMANN MACHINE

• ORDER
  – LIST of Instructions
  – ARRAY of Memory

• THE STATE
  – Mutation in time

THE von NEUMANN MACHINE

• ORDER
  – LIST of Instructions
  – ARRAY of Memory

• THE STATE
  – Mutation in time
ORDER AND THE STATE
OUTLINE

HISTORY

TENACITY

FRUSTRATION

CALM
RACE
DISORDER
AND
THE STATE

http://www.flickr.com/photos/60057912@N00/4845067066/
THE TRANSACTION CONCEPT

• ORDER
  – Disorder across transactions
  – Illusion of order within transactions

• THE STATE
  – Registers, Memory
    • Isolation
  – Mutation in time
    • Atomicity

THE TRANSACTION CONCEPT

- ELEGANT THEORY
  - Serializability

- PRACTICAL ENGINEERING
  - A transparent illusion
    - Easy to ensure correct applications
    - Tricky to scale infrastructure

SUMMARY

- TRIUMPH OF ORDER
- TRIUMPH OF THE STATE

- ELEGANT ILLUSION OF ORDER AND STATE
  - FORMAL THEORY
  - NATURAL API
  - EFFICIENT IMPLEMENTATION
VON NEUMANN TRANSACTIONS CAP CALM
ORDER, THE STATE AND GLOBALIZATION

http://www.flickr.com/photos/29487767@N02/3574392846/
DISTRIBUTED COMPUTING IS THE NEW NORMAL

- ORDER IS TOO COSTLY
  - Synchronization
  - Coordination

- THE STATE IS HEARSAY
  - Delay
  - Failure
  - Partition
MIND THE CAP

THE CAP THEOREM
A NEGATIVE RESULT FOR A TIME OF DISILLUSIONMENT
• **DESIGN MAXIMS**
  - Commutative methods
  - Inverse methods
  - Free coupons

• **PRACTICAL ENGINEERING**
  - Pragmatic Systems
    • Easy to scale infrastructure
    • Tricky to ensure correct applications
**THE TRANSACTION CONCEPT**

- **ELEGANT THEORY**
  - Serializability

- **PRACTICAL ENGINEERING**
  - A transparent illusion
    - Easy to ensure correct applications
    - Tricky to scale infrastructure

**COPING WITH DISORDER**

- **DESIGN MAXIMS**
  - Commutative methods
  - Inverse methods
  - Apologies

- **PRACTICAL ENGINEERING**
  - Pragmatic Systems
    - Easy to scale infrastructure
    - Tricky to ensure correct applications
SUMMARY

ELEGANCE & ORDER

EXPENSIVE ILLUSIONS

MAXIMS & DISORDER

FRAGILE APPLICATIONS

MIND THE CAP
KEEP CALM AND QUERY ON

POSITIVE THINKING FOR THE CLOUDY FUTURE
THE TRANSACTION CONCEPT

- ELEGANT THEORY
  - Serializability

- PRACTICAL ENGINEERING
  - A transparent illusion
    - Easy to ensure correct applications
    - Tricky to scale infrastructure

COPING WITH DISORDER

- DESIGN MAXIMS
  - Commutative methods
  - Inverse methods
  - Apologies

- PRACTICAL ENGINEERING
  - Pragmatic Systems
    - Easy to scale infrastructure
    - Tricky to ensure correct applications
CALM

THEORY FOR APPLICATIONS

COMPILERS TRUMP INFRASTRUCTURE
ELEGANCE AND DISORDER

- **ELEGANT THEORY**
  - Maxims $\Rightarrow$ Theorems
    - Lattices
    - ♪ CALM Theorem

- **PRACTICAL ENGINEERING**
  - Theorems $\Rightarrow$ Compilers
    - $\sim$ bloom
    - CALM Analysis

http://www.flickr.com/photos/scobleizer/4870003098/sizes/l/in/photostream

CALM
Assiociative
  – \((X \circ Y) \circ Z = X \circ (Y \circ Z)\)
  – batch-insensitive

Commutative
  – \(X \circ Y = Y \circ X\)
  – order-insensitive

Idempotent
  – \(X \circ X = X\)
  – resend-insensitive

Distributed
  – acronym-insensitive
Associative
- \((X \circ Y) \circ Z = X \circ (Y \circ Z)\)
- batch-insensitive

Commutative
- \(X \circ Y = Y \circ X\)
- order-insensitive

Idempotent
- \(X \circ X = X\)
- resend-insensitive

Distributed
- acronym-insensitive
Storing an Integer

**VON NEUMANN**

```c
int ctr;

operator:= (x) {
    // assign
    ctr = x;
}
```

**ACID 2.0**

```c
int ctr;

operator<= (x) {
    // merge
    ctr = MAX(ctr, x);
}
```

**DISORDERLY INPUT STREAMS:**

2, 5, 6, 7, 11, 22, 44, 91
5, 7, 2, 11, 44, 6, 22, 91, 5
Storing an Integer

VON NEUMANN

ACID 2.0

DISORDERED INPUT STREAMS:
2, 5, 6, 7, 11, 22, 44, 91
5, 7, 2, 11, 44, 6, 22, 91, 5
PROGRESS

- **Lemma:**
  - ACID 2.0 $\Rightarrow$ monotonic

- **Lemma:**
  - ACID 2.0 $\Rightarrow$ confluent

- **Corollary:**
  - ACID 2.0 $\Rightarrow$ convergent
  - a.k.a. “Eventually Consistent”
    - No coordination!
CRDTs

Convergent Replicated Data Types

[Shapiro, et al. 2011]

- **Semilattice objects**
  - A class
  - `merge()` is ACID 2.0

- **Many examples:**
  - int w/ Max
  - set w/ Union
  - map w/ Insert
  - ...
CRDTs

convergent replicated data types
[Shapiro, et al. 2011]

Semilattices can also be defined algebraically: join and meet are associative, commutative, idempotent binary operations.
CRDTs
convergent replicated data types
[Shapiro, et al. 2011]

SCOPE DILEMMA

- SINGLE-OBJECT PROGRAMS?
- PROVE ACID 2.0
  - formalism?
  - unit testing?

Semilattices can also be defined algebraically: join and meet are associative, commutative, idempotent binary operations.
DESIRE: COMPOSITION

- PIECEWISE ANALYSIS
  - Multiple simple CRDTs
  - Each easy to test
  - Rules for composition

- SET LATTICES KNOWN
  - Database query languages
    - select/project/join rules
    - even with recursion!
  - Distributed Datalog
    - see P2, etc.

- CONSISTENCY?

http://www.flickr.com/photos/44606255@N00/370973576/
THE CALM THEOREM

CONSISTENCY AS LOGICAL MONOTONICITY
1. MONOTONICITY $\Rightarrow$ EVENTUAL CONSISTENCY

2. WHEN TO COORDINATE? NON-MONOTONE OPERATORS
THEORY \Rightarrow COMPILER

KEEPING CALM

COMPILEDERS
TRUMP INFRASTRUCTURE
ANACHRONISM
A THING BELONGING OR APPROPRIATE TO A PERIOD OTHER THAN THAT IN WHICH IT EXISTS

A *disorderly* language of lattices and mappings.

Encourages monotonicity.

Highlights non-monotonicity.

Designed for distribution.
<~ **bloom** operational model

- really a metaphor for a logic called **dedalus**
- each node runs independently
  - local clock, local data, local execution
- time-stepped execution loop at each node
Hello World in `<~ bloom`

```ruby
# a chat server
bloom do
  nodelist <= connect.map { |c| c.val }
  mcast `<~ (mcast*nodelist).pairs { |m,n| [n.key, m.val] }
end
```
Hello World in \textasciitilde\ bloom

\# a chat server
bloom do
  \texttt{nodelist} \leftarrow \texttt{connect.map \{ |c| c.val \}}
  \texttt{mcast} \leftarrow \texttt{mcast*nodelist}.pairs \{ lm, nl \}

sets of key/value pairs
Hello World in \textless~\textasciitilde\textgreater~\texttt{bloom}

\texttt{# a chat server}
\texttt{bloom do}
\texttt{nodelist} \texttt{<=} \texttt{connect.map \{lcl c.val\}}
\texttt{mcast} \texttt{<=} (\texttt{mcast* nodelist}).pairs \{ lm, nl \}

\texttt{instantaneous merge (union)}

\texttt{async merge (union)}
Hello World in `~ bloom`

```ruby
# a chat server
bloom do
  nodelist <= connect(map{|c| c.val})
  mcast   <= (mcast*nodelist).pairs { |m,n|
    [n.key, m.val]
  }
end
```

monotone functions
Monotone Function

A function $f$ is monotone if

$$x \leq y \implies f(x) \leq f(y)$$
Hello World in \(<\sim\) bloom

# a chat server
bloom do
  nodelist <= connect.map{|c| c.val}
  mcast   <= (mcast*nodelist).pairs { |m,n|
    [n.key, m.val]
  }
end

monotone functions
Tables and Channels

state do
  table :nodelist
  channel :connect
  channel :mcast
end

# a chat server
bloom do
  nodelist <= connect.map { |c| c.val }
  mcast ~> (mcast*nodelist).pairs { |m,n|
    [n.key, m.val]
  }
end

See Getting Started docs on github
Hello World in \(<\sim bloom\)

state do
  table : nodelist
  channel : connect
  channel : mcast
end

# a chat server
bloom do
  nodelist <= connect.map { lcl c.val }
  mcast ~> (mcast*nodelist).pairs { lm,nl
    [n.key, m.val]
  }
end

Lattice merge
+ Monotone Functions

MONOTONIC PROGRAM

hence
EVENTUALLY
CONSISTENT
More Lattices

```ruby
state do
  table :nodelist
  channel :connect
  channel :mcast
  lmax :cnt
  lbool :crowded
end

# a chat server
bloom do
  nodelist <= connect.map { |c| c.val }
  mcast <= (mcast*nodelist).pairs { |m,nl|
    [n.key, m.val]
  }
  cnt <= nodelist.group([], count())
  crowded <= cnt.gt(100)
end
```
state do
  table :nodelist
  channel :connect
  channel :mcast
  lmax :cnt      # integer with MAX
  lbool :crowded # bool with OR
end

# a chat server
bloom do
  nodelist <= connect.map { lci c.val }
  mcast  <= (mcast*nodelist).pairs { lm,nl
    [n.key, m.val]
  }
  cnt    <= nodelist.group([], count())
  crowded <= cnt.gt(100)
end

More Lattices
state do
  table :node
  channel :connect
  channel :mcast
  lmax :cnt  # integer with MAX
  lbool :crowded  # bool with OR
end

# a chat server
bloom do
  nodelist <= connect.map { |c| c.val }
  mcast <= (mcast*nodelist).pairs { |m,n|
    [m.key, n.val]
  }
  cnt <= nodelist.group([], count())
  crowded <= cnt.gt(100)
end

More Lattices

monotone functions across lattice types
Monotone Functions Across Lattice Types
state do

table:

channel: connect

channel: mcast

channel: disconnect

do end

bloom do

nodelist <= connect.map { |c| c.val }

mcast <~ (mcast * nodelist).pairs { |m,n| [n.key, m.val] }

nodelist <- disconnect.map { |c| c.val }

do end

Non-Monotonicity

Downstream of Asynchrony

Non-Monotone function
CALM Analysis

1. For any path through a Bloom module, label:
   - Asynchrony
   - Non-Monotonicity
   - Inconsistency

2. Compute labels transitively across modules

3. Identify code that needs coordination

4. Assess comm pattern, suggest coordination
   - 1-1, 1-many : ordered delivery
   - many-many   : order proxy, Paxos, etc.
Alvaro Diagrams

Basic KVS

Lattice KVS
module VectorClock
  state do
    lmap : my_vc
    lmap : next_vc
    scratch : in_msg, [:addr, :payload] => [:clock]
    scratch : out_msg, [:addr, :payload]
    scratch : out_msg_vc, [:addr, :payload] => [:clock]
  end

  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 { lml [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 { lml m.clock }
    my_vc <= next_vc
  end
end
Initially all clocks are zero.

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).
THE CALM THEOREM

COROLLARIES: WHY COORDINATE?

CRON CAUSALITY REQUIRED ONLY FOR NON-MONOTONICITY

COORDINATION COMPLEXITY HOW MUCH COORDINATION IS TRULY NEEDED FOR YOUR ALGORITHM?

FATEFUL TIME THE ONLY USE FOR “TIME” IS TO “SEAL FATE”.
SUMMARY:
UNITY

MAXIMIZE DISORDER AND UNDERSTAND ORDER’S ROLE

DISORDERLY CODE AND WHOLE-PROGRAM ANALYSIS

MIND THE CAP

<~ bloom

CALM
MORE CALM

• Papers
  – CALM/Bloom, CIDR ‘11
  – Bloom+Lattices, SOCC ‘12
  – BloomUnit, DBTest ’12

• Videos
  – Declarative Imperative, PODS ’10
  – Bloom, Lang.Next ’12
  – Bloom+Lattices, Basho Meetup ‘12
ACM SOCC NEXT TUES 10/16
10:45AM
SAN JOSE MARRIOT

DATA CONSISTENCY SESSION
• NEIL CONWAY ON LATTICE SUPPORT IN BLOOM
• PETER BAILIS ON POTENTIAL DANGERS OF CAUSAL CONSISTENCY

http://www.socc2012.org
KEEP CALM AND QUERY ON

POSITIVE THINKING FOR THE CLOUDY FUTURE

bloom-lang.org