

# Programming Language Support for Loose Consistency in Distributed Systems

Peter Alvaro, Neil Conway, Joe Hellerstein, **Bill Marczak**



# Consistency

- Replicate for robustness
- Replication → nondeterminism
  - Consistency: all replicas have same state
- “Strong” consistency (e.g., 2PC, Paxos)
  - Global sync on every read/write/message
  - High price
- “Loose” (eventual) consistency
  - Write program to avoid sync (i.e., commutative)
  - But few existing tools to help

# Use Logic

- Distributed first-order logic programming
  - Similar performance & OOM less code vs imperative [P2, BOOM]
  - Apply existing analyses to new domain

# CALM Principle

*“Distributed monotonic logic is eventually consistent; distributed non-monotonic logic requires distributed coordination logic (a barrier) for eventual consistency”*

- Monotonic: state accumulation
- Non-monotonic: deleting, counting, summarizing ( $\forall$ )

# CALM Principle

*“Distributed monotonic logic is eventually consistent; distributed non-monotonic logic requires distributed coordination logic (a barrier) for eventual consistency”*

- Monotonic: state accumulation
- Non-monotonic: deleting, counting, summarizing ( $\forall$ )

Based on *stratification* analysis from Logic Programming

# BLOOM Language

- Based on Dedalus: formal temporal logic (extends Datalog)
  - Computation : **deduction** over **sets (collections)** of tuples
  - Each tuple has “logical time”
    - Local deduction: “now” or “next” (updates)
    - Messages: **non-deterministic future time**
  - CALM is a **static analysis**
- BLOOM: Bridging language between imperative & logic programming
- BUD (“BLOOM Under Development”): DSL in Ruby
  - Integrates with Ruby code
  - Familiar syntax & encapsulation

# Outline

- **Flavor of BLOOM: Shopping cart**
- CALM Analysis
- A better shopping cart

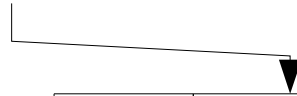
# BLOOM Language

collection



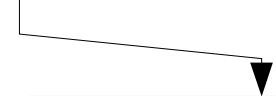
<b>table</b>	Persist across timesteps
<b>scratch</b>	Temporary; store for one timestep
<b>channel</b>	Remote scratch

op



<b>&lt;=</b>	Now
<b>&lt;+</b>	Future
<b>&lt;-</b>	Delete

collection expr



map
filter
join
not include
group

# BLOOM Language

collection

op

collection expr

<b>table</b>	Persist across timesteps
<b>scratch</b>	Temporary; store for one timestep
<b>channel</b>	Remote scratch

<b>&lt;=</b>	Now
<b>&lt;+</b>	Future
<b>&lt;-</b>	Delete

map
filter
join
not include
group

Non-Monotonicity

# Shopping Cart

- Single shopper; replicated cart on servers
- Server KVS stores array with cart contents
  - {Client, [Item\_1, ..., Item\_n]}
- Operations supported:
  - Add(X):  $\text{KVS}[\text{Client}] = \text{KVS}[\text{Client}] \cup \{X\}$
  - Delete(X):  $\text{KVS}[\text{Client}] = \text{KVS}[\text{Client}] - \{X\}$
  - Checkout(): Send  $\text{KVS}[\text{Client}]$  to Client

# Schema

**table** :bigtable, ['key', 'value']

**scratch** :kvput, ['server', 'key', 'reqid', 'value']

Key-Value  
Store  
Interface

**channel** :action\_msg, 0, ['server', 'client', 'item',  
                          'action', 'reqid']

**table** :client\_action, 0, ['server', 'client', 'item',  
                          'action', 'reqid']

Cart Update

# Schema

```
table :bigtable, ['key', 'value']
```

```
scratch :kvput, ['server', 'key', 'reqid', 'value']
```

```
channel :action_msg, 0, ['server', 'client', 'item',  
                           'action', 'reqid']
```

```
table :client_action, 0, ['server', 'client', 'item',  
                           'action', 'reqid']
```

Key-Value  
Store  
Interface

Cart Update

The table mapping keys → values

# Schema

```
table :bigtable, ['key', 'value']
```

```
scratch :kvput, ['server', 'key', 'reqid', 'value']
```

Key-Value  
Store  
Interface

```
channel :action_msg, 0, ['server', 'client', 'item',  
                          'action', 'reqid']
```

```
table :client_action, 0, ['server', 'client', 'item',  
                          'action', 'reqid']
```

Cart Update

Interface to insert into KVS (like function call)

# Schema

```
table :bigtable, ['key', 'value']
```

```
scratch :kvput, ['server', 'key', 'reqid', 'value']
```

Key-Value  
Store  
Interface

```
channel :action_msg, 0, ['server', 'client', 'item',  
                          'action', 'reqid']
```

Cart Update

```
table :client_action, 0, ['server', 'client', 'item',  
                          'action', 'reqid']
```

Channel to pass cart updates from client to a server replica

# Schema

```
table :bigtable, ['key', 'value']
```

```
scratch :kvput, ['server', 'key', 'reqid', 'value']
```

Key-Value  
Store  
Interface

```
channel :action_msg, 0, ['server', 'client', 'item',  
                          'action', 'reqid']
```

Cart Update

```
table :client_action, 0, ['server', 'client', 'item',  
                          'action', 'reqid']
```

Persists the messages received through the channel

# Shopping Cart

```
declare def store
```

```
  kvput <= join([bigtable, action_msg]).map do |b, a|  
    if b.key == a.client  
      if a.action == "Add"  
        [a.server, a.client, a.reqid,  
         b.value.push(a.item)]  
      elsif a.action == "Del"  
        [a.server, a.client, a.reqid,  
         b.value.reject{|bv| bv == a.item}]  
      end  
    end  
  end  
end  
end
```

All rules are located inside a **declare** block.  
Allows programmer to organize rules of similar purpose; subclasses can override.

# Shopping Cart

```
declare def store
  kvput <= join([bigtable, action_msg]).map do |b, a|
    if b.key == a.client
      if a.action == "Add"
        [a.server, a.client, a.reqid,
         b.value.push(a.item)]
      elsif a.action == "Del"
        [a.server, a.client, a.reqid,
         b.value.reject{|bv| bv == a.item}]
      end
    end
  end
end
```

*For each (persisted) cart update (action\_msg), look up the client's cart in the KVS (bigtable), and return a [tuple] with the item added or deleted.*

# Shopping Cart

```
declare def store
  kvput <= join([bigtable, action_msg]).map do |b, a|
    if b.key == a.client
      if a.action == "Add"
        [a.server, a.client, a.reqid,
         b.value.push(a.item)]
      elsif a.action == "Del"
        [a.server, a.client, a.reqid,
         b.value.reject{|bv| bv == a.item}]
      end
    end
  end
end
end
end
```

*For each (persisted) cart update (action\_msg), look up the client's cart in the KVS (bigtable), and return a [tuple] with the item added or deleted. **In the same timestep, add this collection to***

# Shopping Cart

```
declare def store
  kvput <= join([bigtable, action_msg]).map do |b, a|
    if b.key == a.client
      if a.action == "Add"
        [a.server, a.client, a.reqid,
         b.value.push(a.item)]
      elsif a.action == "Del"
        [a.server, a.client, a.reqid,
         b.value.reject{|bv| bv == a.item}]
      end
    end
  end
end
end
end
```

*For each (persisted) cart update (action\_msg), look up the client's cart in the KVS (bigtable), and return a [tuple] with the item added or deleted. In the same timestep, add this collection to **the KVS (kvput)***

# Shopping Cart

```
declare def store
  kvput <= join([bigtable, action_msg]).map do |b, a|
    if b.key == a.client
      if a.action == "Add"
        [a.server, a.client, a.reqid,
         b.value.push(a.item)]
      elsif a.action == "Del"
        [a.server, a.client, a.reqid,
         b.value.reject{|bv| bv == a.item}]
      end
    end
  end
end
end
end
```

Seems monotonic, but what about `kvput`?

# Key Value Store

```
bigtable <+ kvput.map do |k|  
  [k.key, k.value]  
end
```

*For every interface call to add a key-value pair (kvput), add it to the KVS in the next timestep (<+)*

# Key Value Store

```
bigtable <+ kvput.map do |k|  
  [k.key, k.value]  
end
```

*For every interface call to add a key-value pair (kvput), add it to the KVS in the next timestep (<+)*

```
bigtable <- join([bigtable, kvput]).map do |b, k|  
  if b.key == k.key  
    b  
  end  
end
```

*For every interface call to add a key-value pair (kvput), delete (<-) any existing value associated with the key (preserves FD)*

# Key Value Store

```
bigtable <+ kvput.map do |k|  
  [k.key, k.value]  
end
```

*For every interface call to add a key-value pair (kvput), add it to the KVS in the next timestep (<+)*

```
bigtable <- join([bigtable, kvput]).map do |b, k|  
  if b.key == k.key  
    b  
  end  
end
```

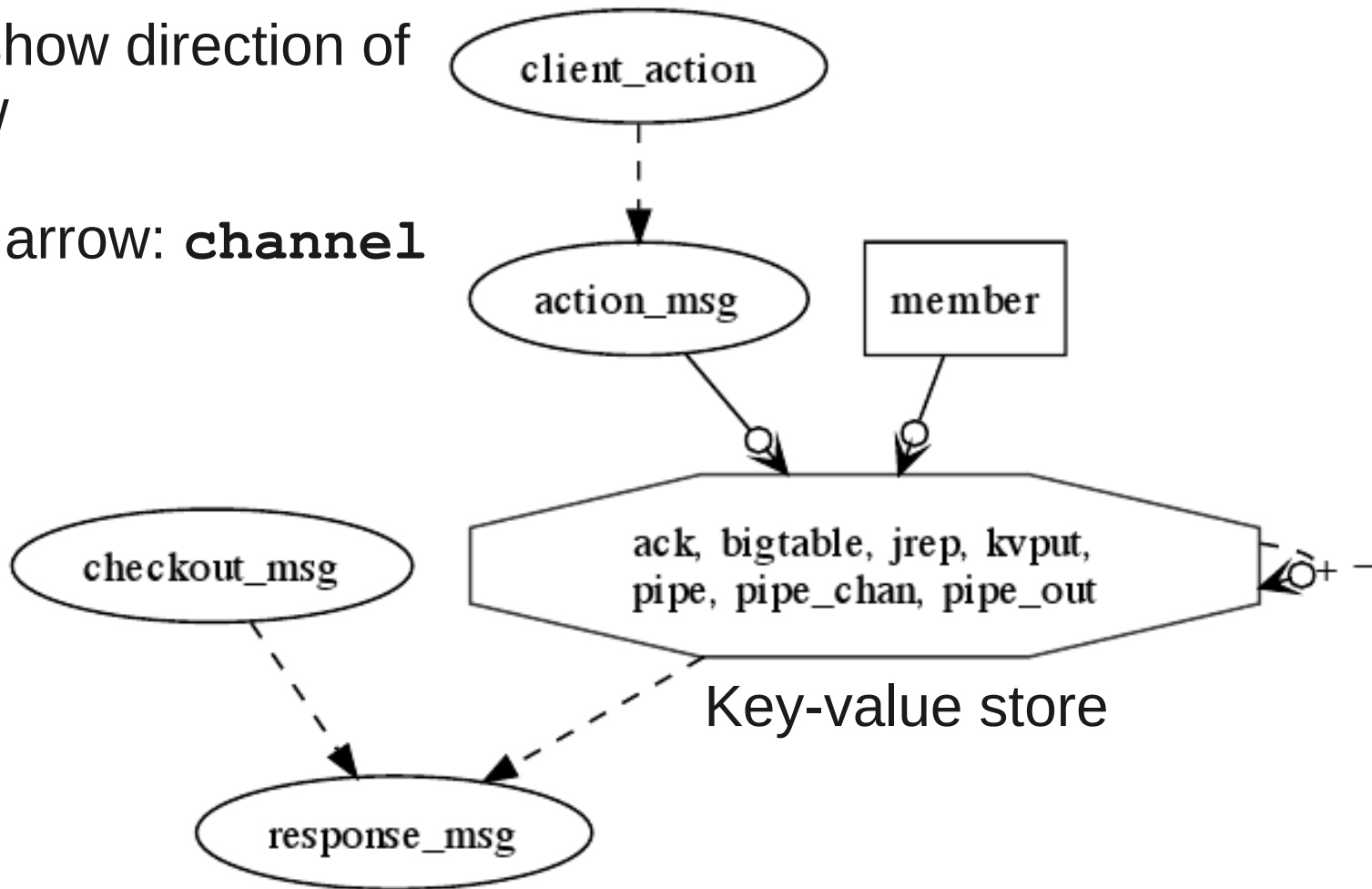
*For every interface call to add a key-value pair (kvput), delete (<-) any existing value associated with the key (preserves FD)*

**Non-monotonic**

# CALM Analysis

Arrows show direction of data flow

Dashed arrow: **channel**

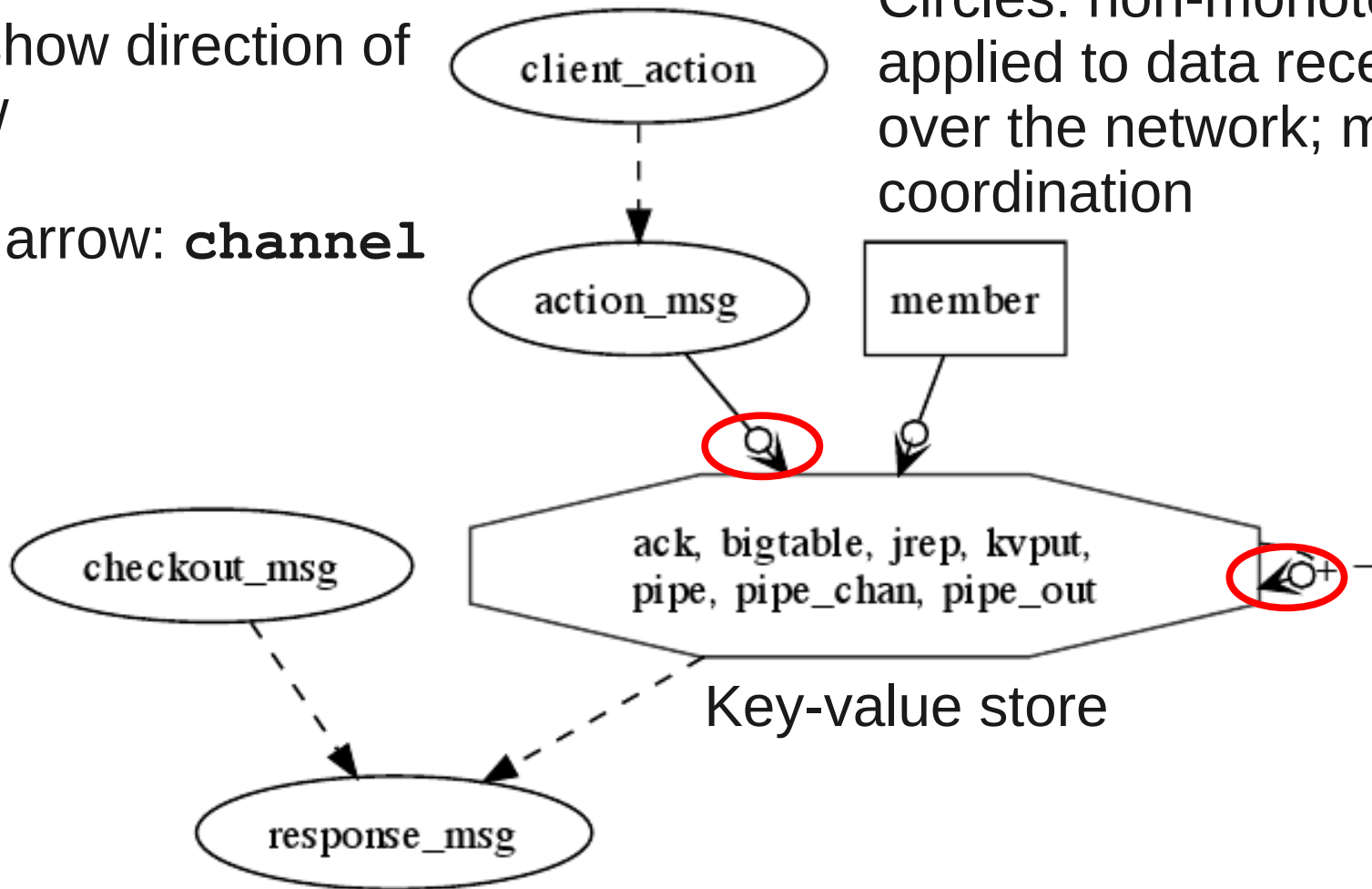


# CALM Analysis

Arrows show direction of data flow

Dashed arrow: **channel**

Circles: non-monotonic logic applied to data received over the network; may need coordination



# CALM Analysis: Conclusions

- Non-monotonic logic **on every cart update**
- Conservative solution: totally order all cart updates (i.e., Paxos, 2PC)
- Do we need to? No, but:
  - Deletions may be re-ordered before their insertions
  - Checkouts may be re-ordered before all insertions and deletions have been received

# CALM Analysis: Conclusions

- Non-monotonic logic **on every cart update**
- Conservative solution: totally order all cart updates (i.e., Paxos, 2PC)
- Do we need to? No, but:
  - Deletions may be re-ordered before their insertions
  - Checkouts may be re-ordered before all insertions and deletions have been received

**We overused non-monotonicity**

# CALM Analysis: Conclusions

- Non-monotonic logic **on every cart update**
- Conservative solution: totally order all cart updates (i.e., Paxos, 2PC)
- Do we need to? No, but:
  - Deletions may be re-ordered before their insertions
  - Checkouts may be re-ordered before all insertions and deletions have been received

How did our analysis work?

# Problems with Non-Monotonicity

- “not in”/“group” over data received from a channel
  - Can't decide until you have all the facts
- “<-” operator over data received from a channel
  - Deletion may not commute with operation using deleted data
- Channels not guarded by tables
  - Data may arrive “too early” (recall a channel is a scratch – data only lasts 1 timestep)

# Problems with Non-Monotonicity

- “not in”/“group” over data received from a channel
  - Can't decide until you have all the facts
- “<-” operator over data received from a channel
  - Deletion may not commute with operation using deleted data
- Channels not guarded by tables
  - Data may arrive “too early” (recall a channel is a scratch – data only lasts 1 timestep)

Analysis puts circles for any of the above

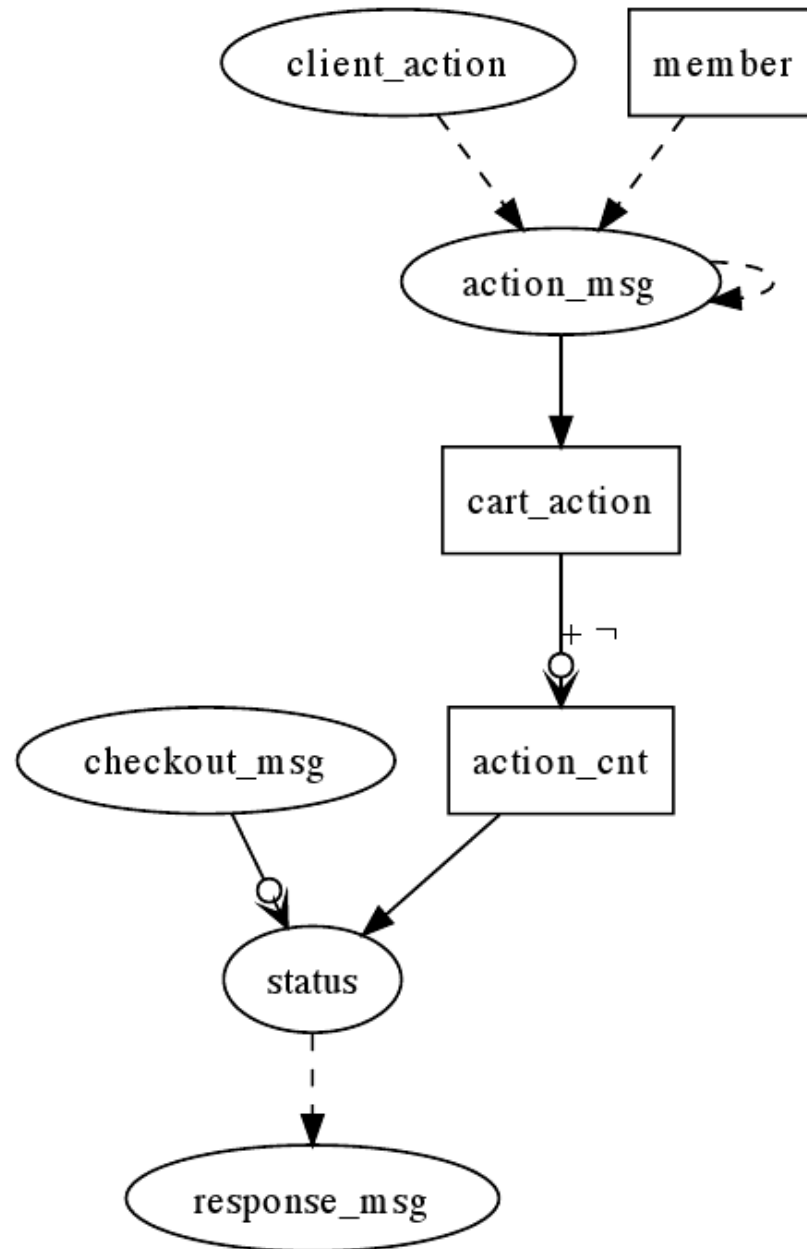
# Why Monotonicity is OK

- If all channels guarded by tables, and we don't delete, then any tuple ever received from the network always exists
- If no “not in”/“group” or deletion, then no existing results change as we receive more data
- Everything once true is always true – accumulation

# A Better Shopping Cart

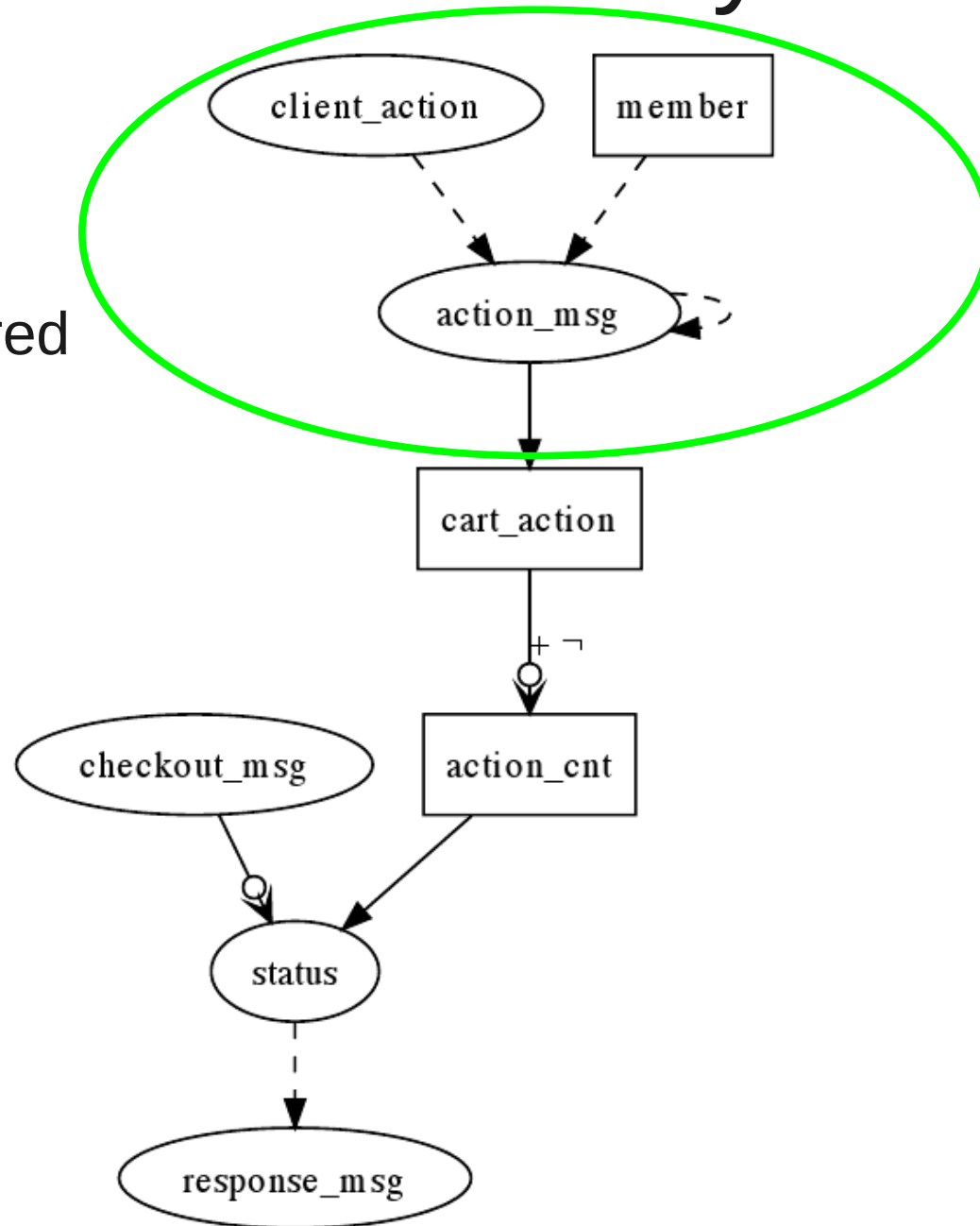
- Accumulate additions & deletions (monotonic)
  - i.e., append to a log
- Summarize once on checkout (non-monotonic)
  - Cancel additions with deletions in log

# Better Cart: Analysis



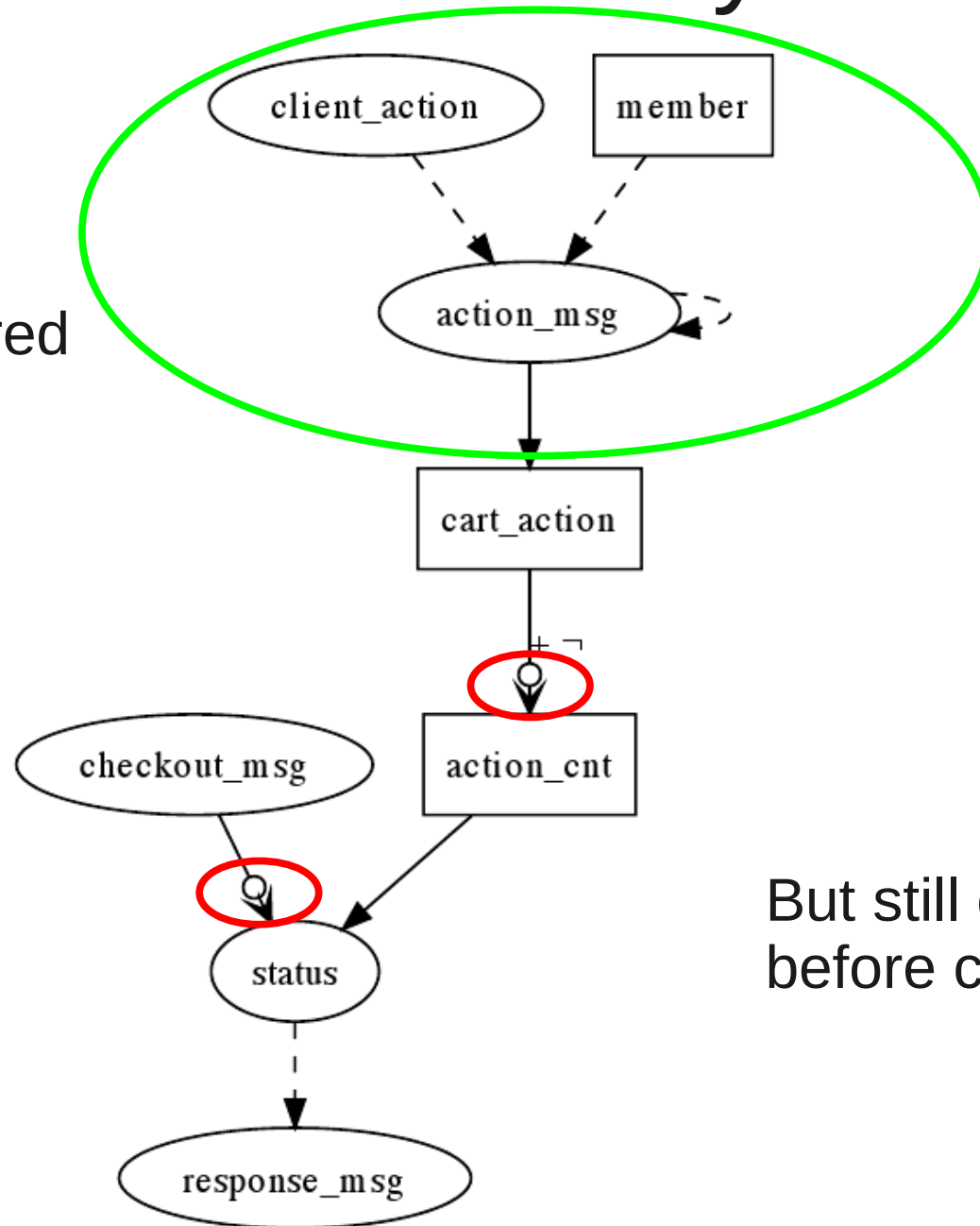
# Better Cart: Analysis

Cart updates and replication is now monotonic; no coordination required here



# Better Cart: Analysis

Cart updates and replication is now monotonic; no coordination required here



But still coordination before checkout

# Conclusion

- CALM Principle: Maximize monotonicity
- BLOOM: Bridging language between imperative & logic programming
- Analysis: Indicates possible need for coordination

# Feedback

- Analysis:
  - Important?
  - Relevant related work?
  - Applicability to PL community?
- Language:
  - Major features missing?
  - Intuitive syntax?
  - Would you want to think/write in this way?

# Programming Language Support for Loose Consistency in Distributed Systems

Peter Alvaro, Neil Conway, Joe Hellerstein, **Bill Marczak**



<http://boom.cs.berkeley.edu>

```

declare
def accumulate
  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 summarize
  status <= join([action_cnt, checkout_msg]).map do |a, c|
    if a.action == "Add" and not
      action_cnt.map{|d| d.id if d.action == "Del"}.include? a.id
      [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 == "Add" and
      a2.action == "Del"
      [a1.session, a1.item, a1.cnt - a2.cnt]
    end
  end
end

declare
def communicate
  response_msg <+ join([status, checkout_msg]).map do |s, c|
    if s.session == c.session
      [c.client, c.server, s.session, s.item, s.cnt]
    end
  end

  action_msg <+ join([action_msg, member]).map do |a, m|
    unless member.map{|nm| nm.player}.include? a.client
      [m.player, a.server, a.session, a.item, a.action, a.reqid]
    end
  end
end
end

```