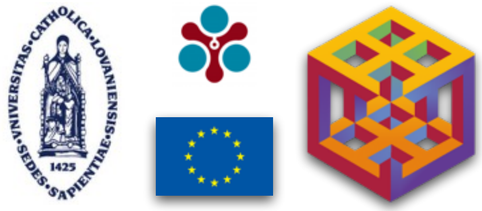
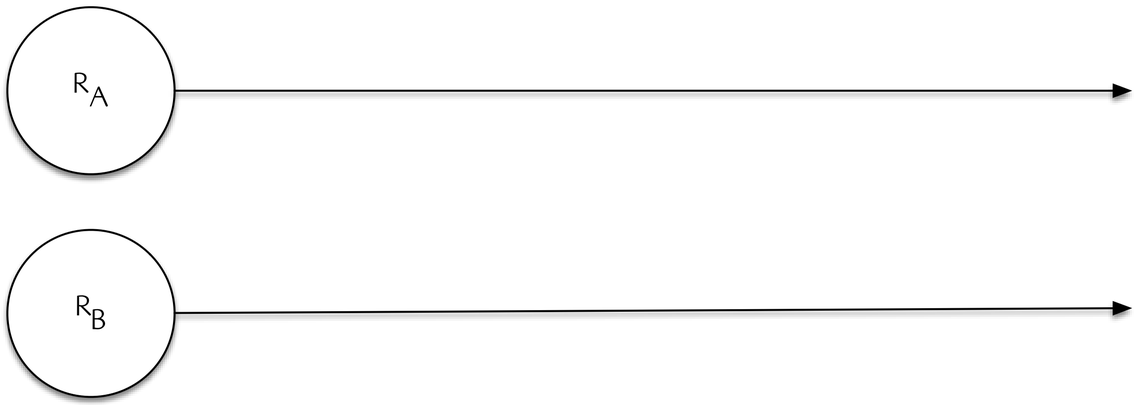
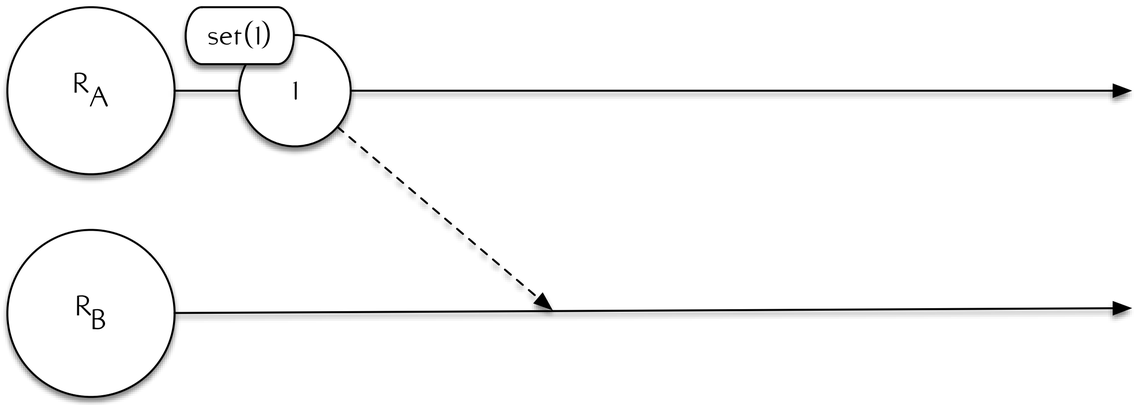
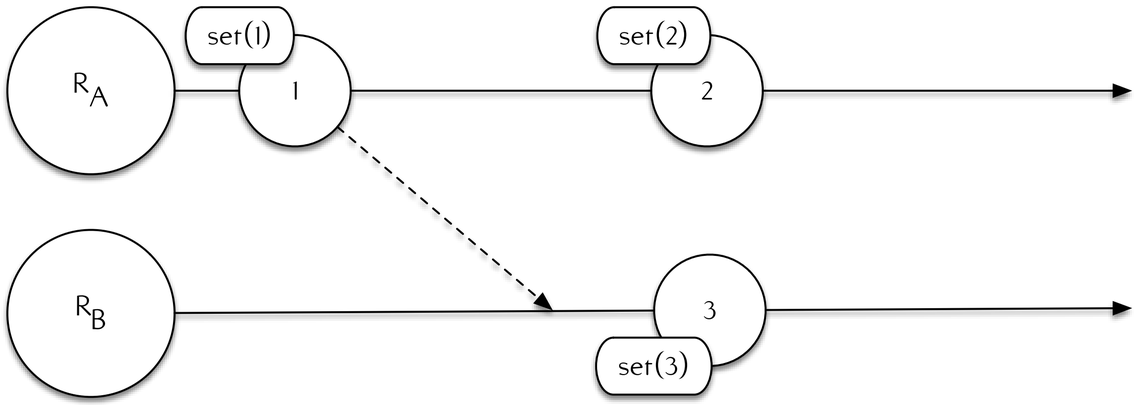
Designing and Evaluating a Distributed Computing Language Runtime

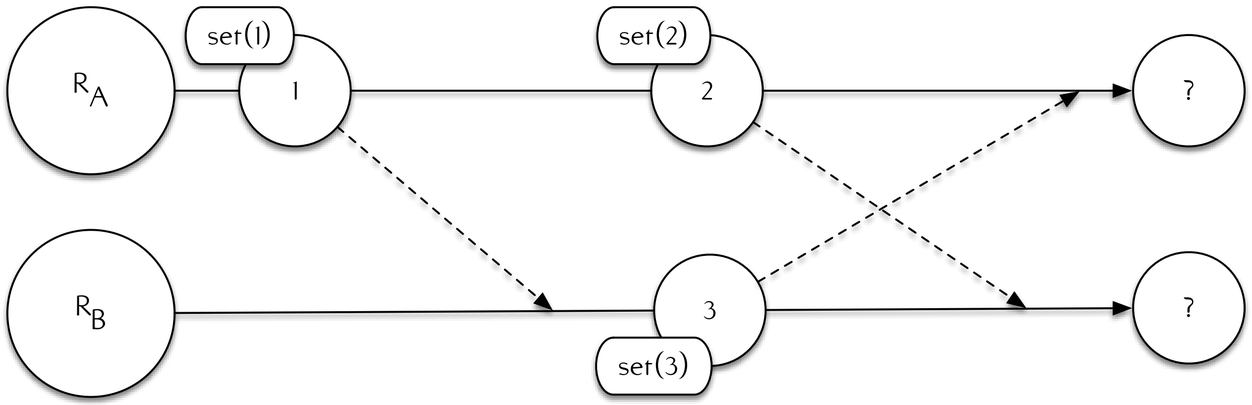
Christopher Meiklejohn (@cmeik)
Université catholique de Louvain, Belgium




R_A
R_B


set(1)
R_A
1
R_B


set(1)
set(2)
R_A
1
2
R_B
3
set(3)


R_A
set(1)
1
set(2)
2
?
R_B
3
set(3)
?

Synchronization

- To enforce an order
  Makes programming easier

Synchronization

- To enforce an order
  Makes programming easier
- Eliminate accidental nondeterminism
  Prevent race conditions

Synchronization

- To enforce an order
  Makes programming easier
- Eliminate accidental nondeterminism
  Prevent race conditions
- Techniques
  Locks, mutexes, semaphores, monitors, etc.

Difficult Cases

- “Internet of Things”,
  Low power, limited memory and connectivity

Difficult Cases

- “Internet of Things”,
  Low power, limited memory and connectivity
- Mobile Gaming
  Offline operation with replicated, shared state

Weak Synchronization

- Can we achieve anything **without synchronization**? Not really.

Weak Synchronization

- Can we achieve anything **without synchronization**?
  Not really.
- **Strong Eventual Consistency (SEC)**
  “Replicas that deliver the same updates have equivalent state”

Weak Synchronization

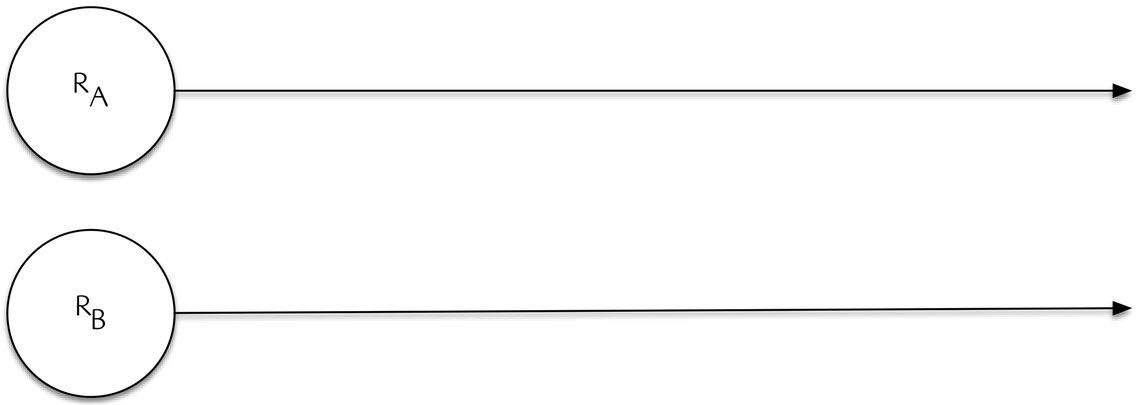
- Can we achieve anything **without synchronization**?
  Not really.
- Strong Eventual Consistency (SEC)
  “Replicas that deliver the same updates have equivalent state”
  - Primary requirement
    Eventual replica-to-replica communication

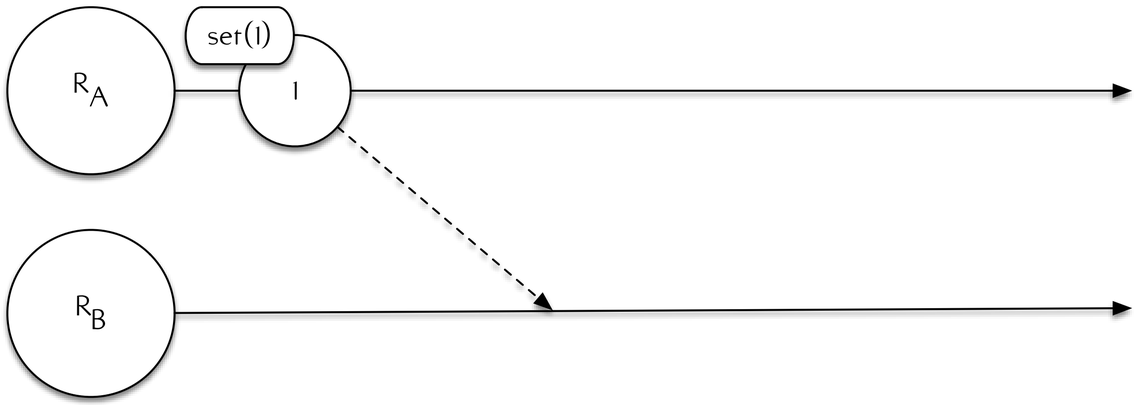
Weak Synchronization

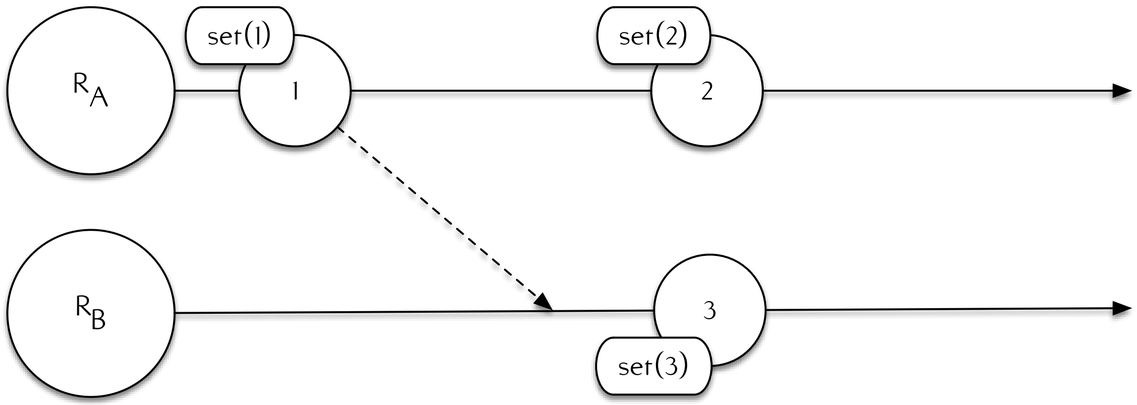
- Can we achieve anything **without synchronization**?
  Not really.
- **Strong Eventual Consistency (SEC)**
  “Replicas that deliver the same updates have equivalent state”
  - **Primary requirement**
    Eventual replica-to-replica communication
  - **Order insensitive!** (Commutativity)

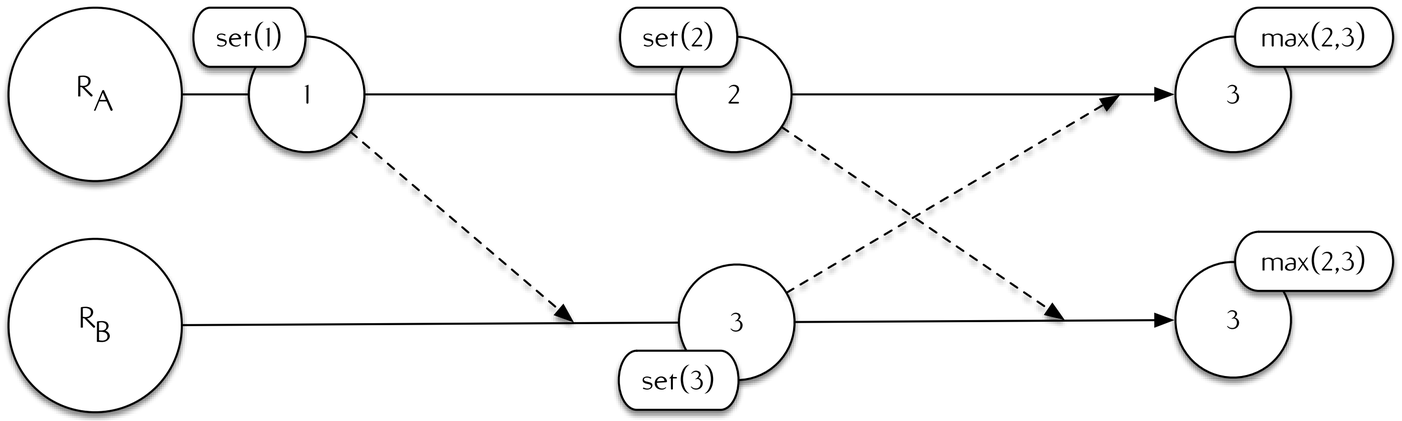
Weak Synchronization

- Can we achieve anything **without synchronization**?
  Not really.
- **Strong Eventual Consistency (SEC)**
  “Replicas that deliver the same updates have equivalent state”
  - **Primary requirement**
    Eventual replica-to-replica communication
  - **Order insensitive!** (Commutativity)
  - **Duplicate insensitive!** (Idempotent)


R_A
R_B


set(1)
R_A
1
R_B


set(1)
set(2)
R_A
1
2
R_B
3
set(3)


R_A
set(1)
1
set(2)
2
max(2,3)
3
R_B
3
set(3)
max(2,3)
3

How can we succeed with **Strong Eventual Consistency?**

Programming SEC

1. Eliminate accidental nondeterminism
   (ex. deterministic, modeling non-monotonic operations monotonically)

Programming SEC

1. Eliminate accidental nondeterminism
   (ex. deterministic, modeling non-monotonic operations monotonically)

2. Retain the properties of functional programming
   (ex. confluence, referential transparency over composition)

Programming SEC

1. Eliminate accidental nondeterminism
   (ex. deterministic, modeling non-monotonic operations monotonically)

2. Retain the properties of functional programming
   (ex. confluence, referential transparency over composition)

3. Distributed, and fault-tolerant runtime
   (ex. replication, membership, dissemination)

Programming SEC

1. Eliminate accidental nondeterminism
   (ex. deterministic, modeling non-monotonic operations monotonically)

2. Retain the properties of functional programming
   (ex. confluence, referential transparency over composition)

3. Distributed, and fault-tolerant runtime
   (ex. replication, membership, dissemination)

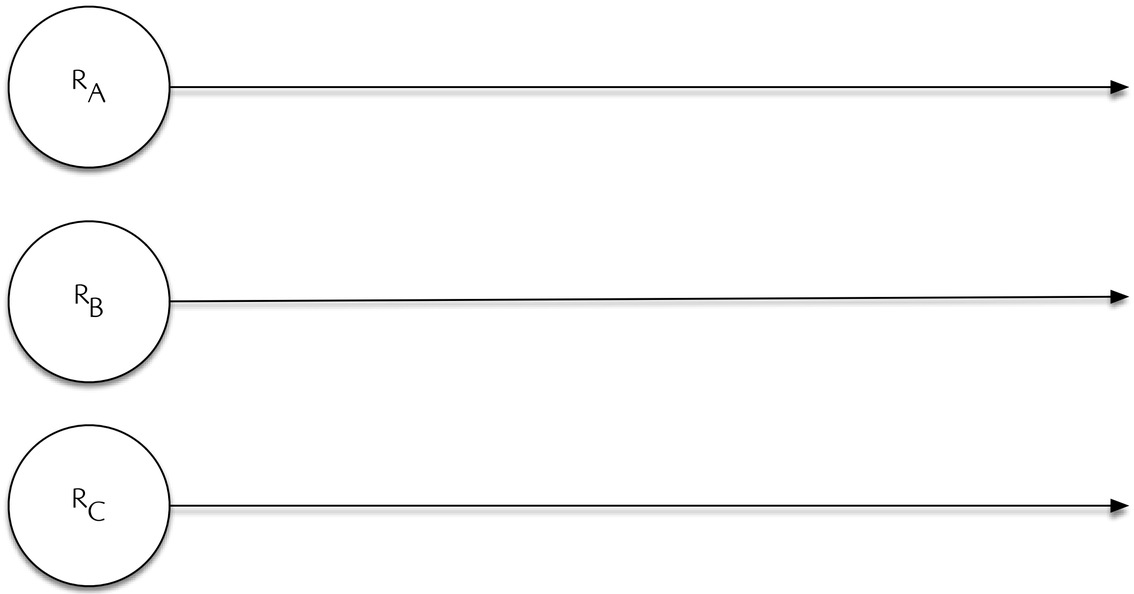
Convergent Objects
Conflict-Free
Replicated Data Types

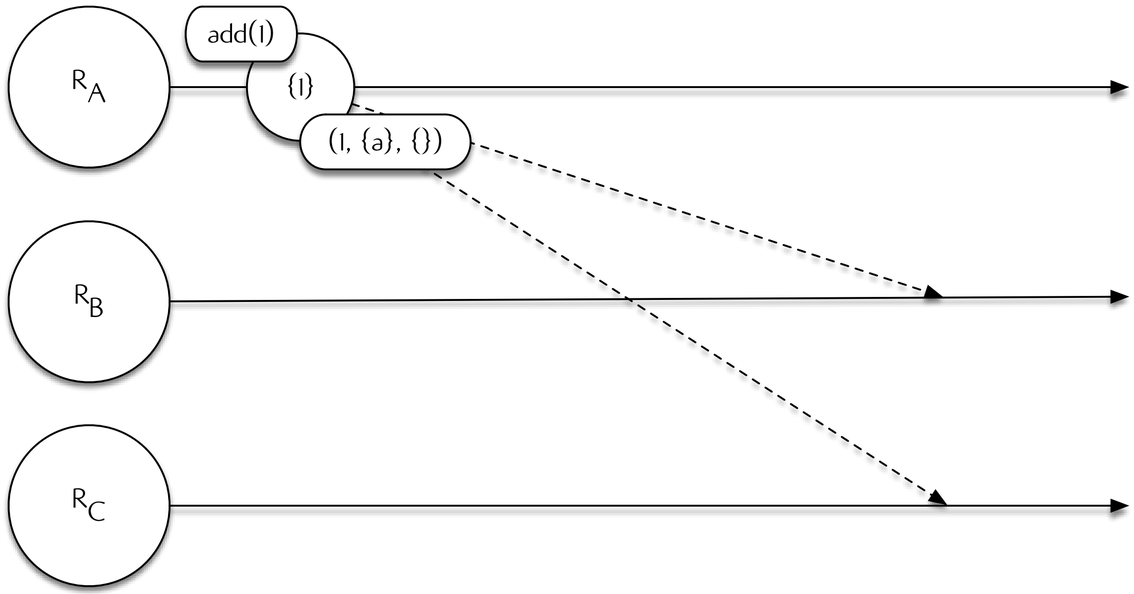
Conflict-Free Replicated Data Types

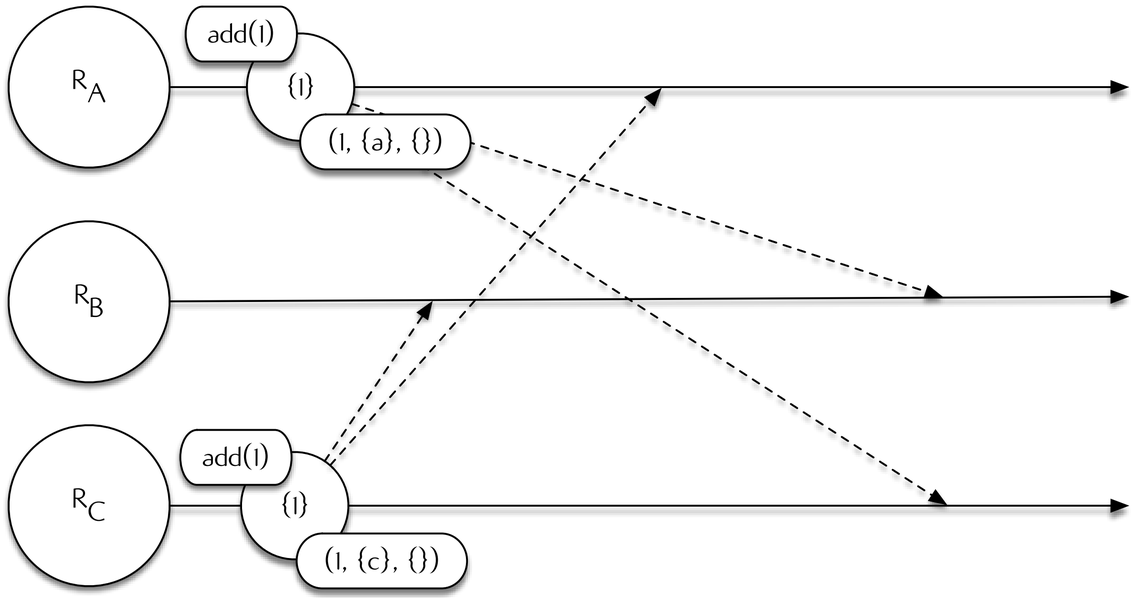
- **Many types exist with different properties**
  Sets, counters, registers, flags, maps, graphs

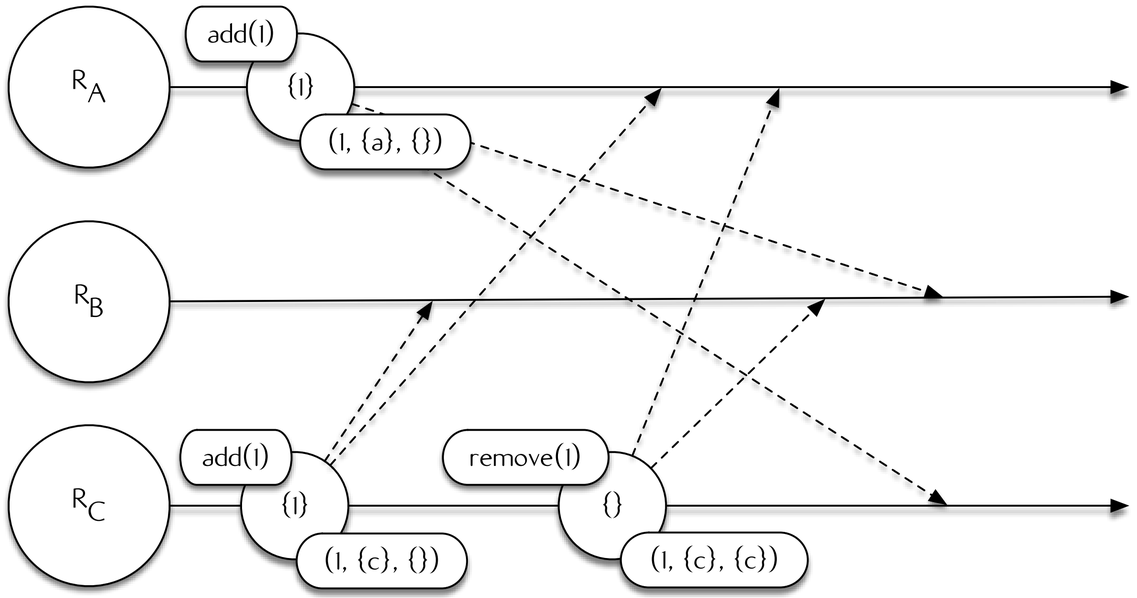
Conflict-Free Replicated Data Types

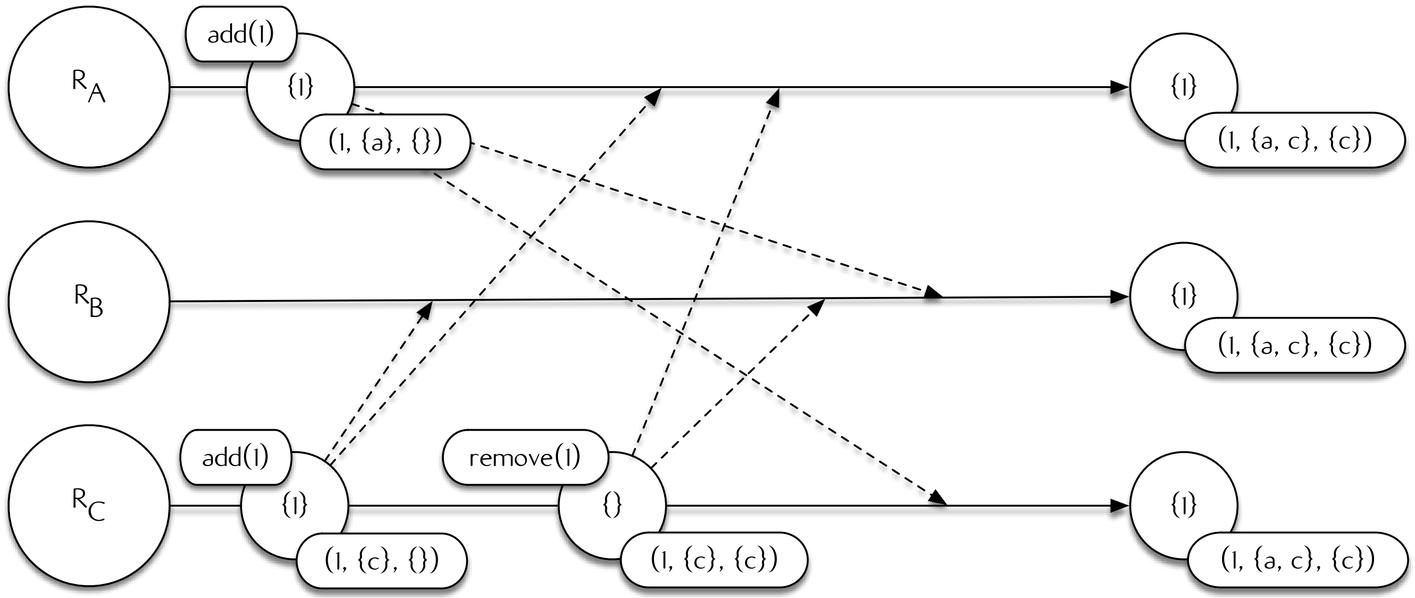
- **Many types exist with different properties**
  Sets, counters, registers, flags, maps, graphs
- **Strong Eventual Consistency**
  Instances satisfy SEC property **per-object**


R_A
R_B
R_C


add(1)
R_A
{1}
(1, {a}, {})
R_B
R_C


add(1)
R_A
{1}
(1, {a}, {})
R_B
add(1)
R_C
{1}
(1, {c}, {})


R_A
R_B
R_C
add(1)
{1}
(1, {a}, {})
add(1)
{1}
(1, {c}, {})
remove(1)
{}
(1, {c}, {c})


R_A
add(1)
{1}
(1, {a}, {})
{1}
(1, {a, c}, {c})
R_B
{1}
(1, {a, c}, {c})
R_C
add(1)
{1}
(1, {c}, {})
remove(1)
{}
(1, {c}, {c})
{1}
(1, {a, c}, {c})

Programming SEC

1. Eliminate accidental nondeterminism
   (ex. deterministic, modeling non-monotonic operations monotonically)

2. Retain the properties of functional programming
   (ex. confluence, referential transparency over composition)

3. Distributed, and fault-tolerant runtime
   (ex. replication, membership, dissemination)

Convergent Programs
Lattice Processing

Lattice Processing (Lasp)

- **Distributed dataflow**
  Declarative, functional programming model

Lattice Processing (Lasp)

- **Distributed dataflow**
  Declarative, functional programming model
- **Convergent data structures**
  Primary data abstraction is the CRDT

Lattice Processing (Lasp)

- **Distributed dataflow**
  Declarative, functional programming model
- **Convergent data structures**
  Primary data abstraction is the CRDT
- **Enables composition**
  Provides functional composition of CRDTs that **preserves the SEC property**

```
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

```
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

```
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

```
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

```
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

Programming SEC

1. Eliminate accidental nondeterminism
   (ex. deterministic, modeling non-monotonic operations monotonically)

2. Retain the properties of functional programming
   (ex. confluence, referential transparency over composition)

3. Distributed, and fault-tolerant runtime
   (ex. replication, membership, dissemination)

Distributed Runtime
Selective Hearing

Selective Hearing

- Epidemic broadcast based runtime system
  Provide a runtime system that can scale to large numbers of nodes, that is resilient to failures and provides efficient execution

Selective Hearing

- Epidemic broadcast based runtime system
  Provide a runtime system that can scale to large numbers of nodes, that is resilient to failures and provides efficient execution
- Well-matched to Lattice Processing (Lasp)

Selective Hearing

- Epidemic broadcast based runtime system
  Provide a runtime system that can scale to large numbers of nodes, that is resilient to failures and provides efficient execution
- Well-matched to Lattice Processing (Lasp)
  - Epidemic broadcast mechanisms provide **weak ordering** but are resilient and efficient

Selective Hearing

- Epidemic broadcast based runtime system
  Provide a runtime system that can scale to large numbers of nodes, that is resilient to failures and provides efficient execution
- Well-matched to Lattice Processing (Lasp)
  - Epidemic broadcast mechanisms provide weak ordering but are resilient and efficient
  - Lasp's programming model is tolerant to message re-ordering, disconnections, and node failures

Selective Hearing

- Epidemic broadcast based runtime system
  Provide a runtime system that can scale to large numbers of nodes, that is resilient to failures and provides efficient execution
- Well-matched to Lattice Processing (Lasp)
  - Epidemic broadcast mechanisms provide weak ordering but are resilient and efficient
  - Lasp's programming model is tolerant to message re-ordering, disconnections, and node failures
- "Selective Receive"
  Nodes selectively receive and process messages based on interest.

Layered Approach

Layered Approach

- Membership
  Configurable membership protocol which can operate in a client-server or peer-to-peer mode

Layered Approach

- Membership
  Configurable membership protocol which can operate in a client-server or peer-to-peer mode

- Broadcast (via Gossip, Tree, etc.)
  Efficient dissemination of both program state and application state via gossip, broadcast tree, or hybrid mode

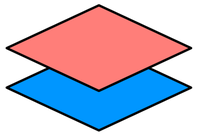
Layered Approach

- Membership
  Configurable membership protocol which can operate in a client-server or peer-to-peer mode

- Broadcast (via Gossip, Tree, etc.)
  Efficient dissemination of both program state and application state via gossip, broadcast tree, or hybrid mode

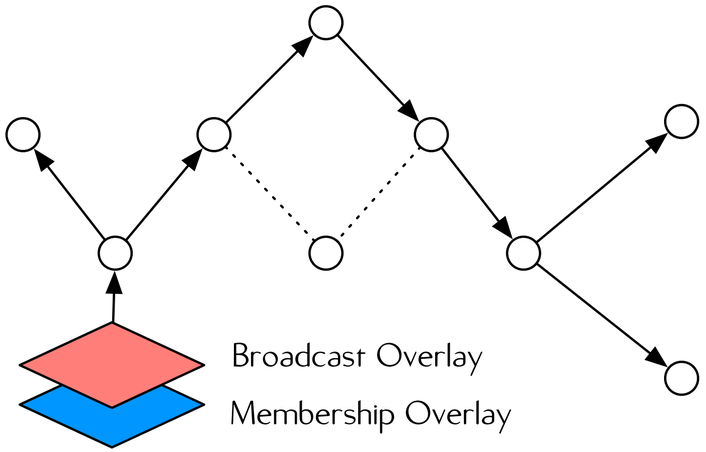
- Auto-discovery
  Integration with Mesos, auto-discovery of Lasp nodes for ease of configurability

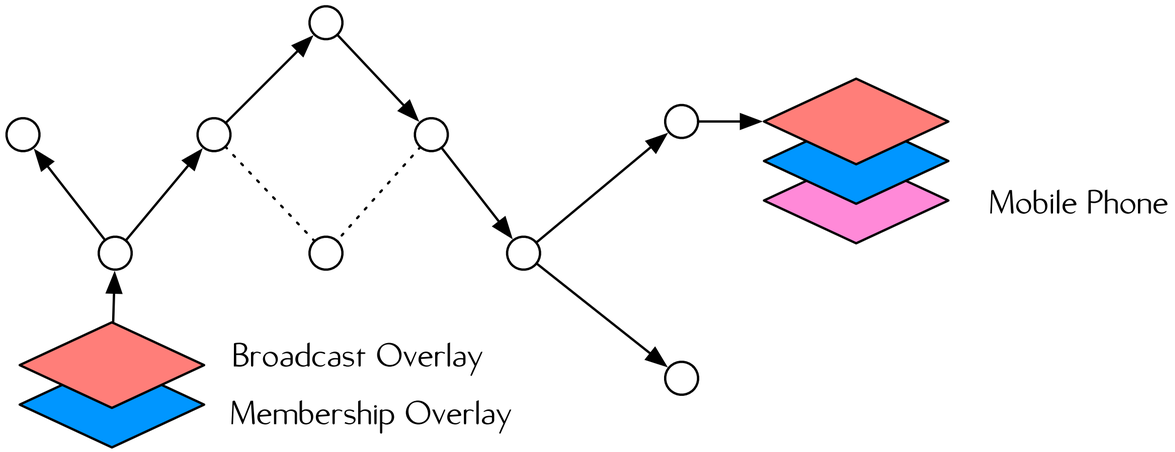

Membership Overlay

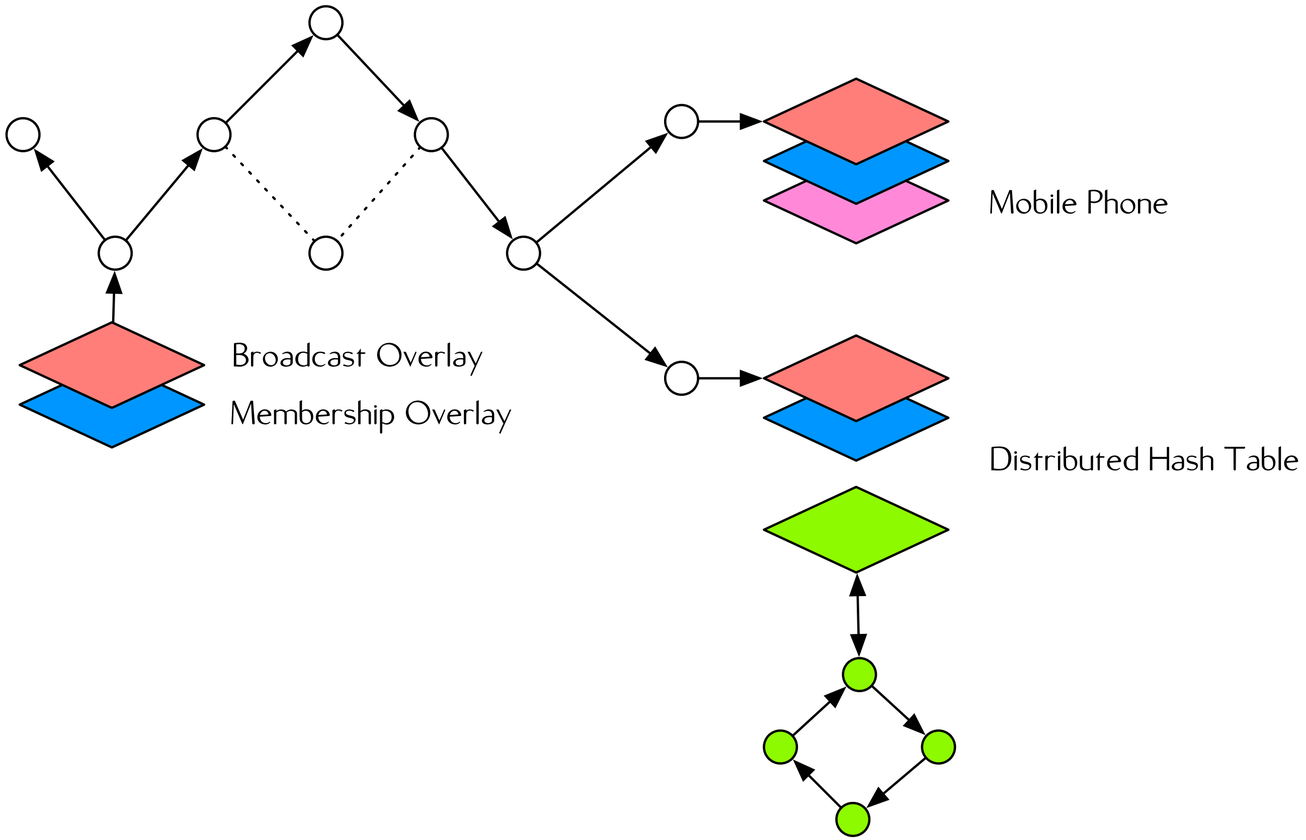


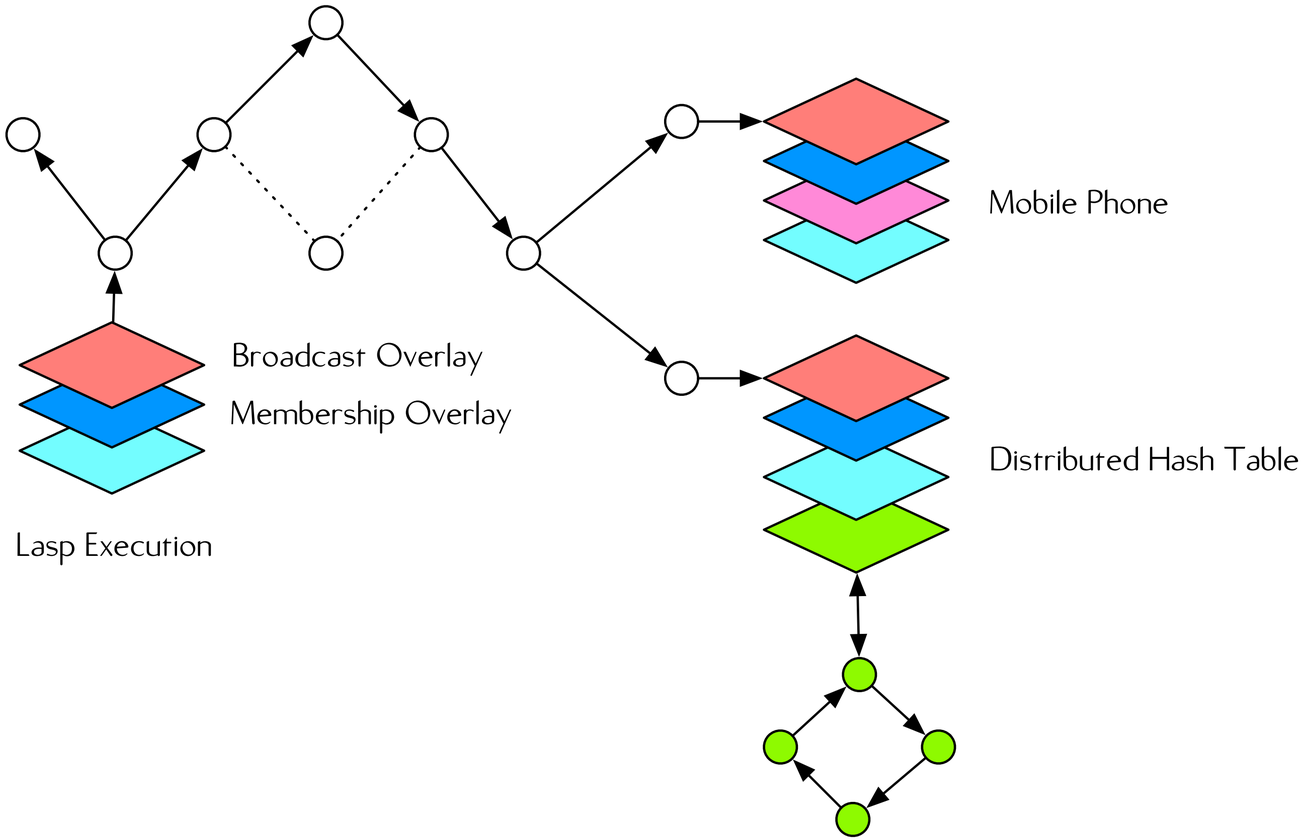
Broadcast Overlay

Membership Overlay


Broadcast Overlay
Membership Overlay


Mobile Phone
Broadcast Overlay
Membership Overlay


Mobile Phone
Broadcast Overlay
Membership Overlay
Distributed Hash Table


Mobile Phone
Broadcast Overlay
Membership Overlay
Distributed Hash Table
Lasp Execution

Programming SEC

1. Eliminate accidental nondeterminism
   (ex. deterministic, modeling non-monotonic operations monotonically)

2. Retain the properties of functional programming
   (ex. confluence, referential transparency over composition)

3. Distributed, and fault-tolerant runtime
   (ex. replication, membership, dissemination)

What can we build?

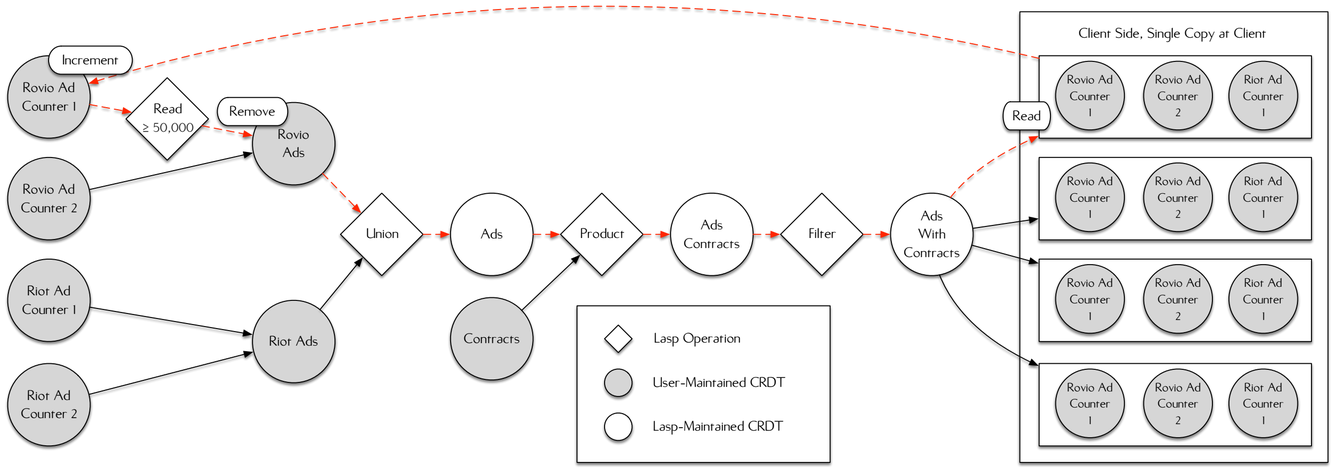
Advertisement Counter

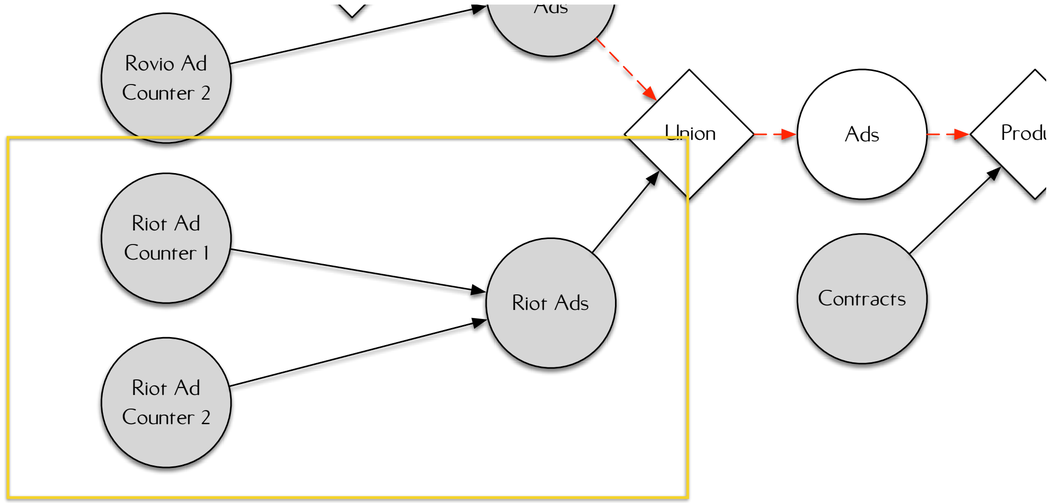
Advertisement Counter

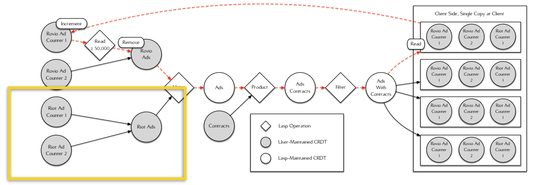
- Mobile game platform selling advertisement space
  Advertisements are paid according to a minimum number of impressions

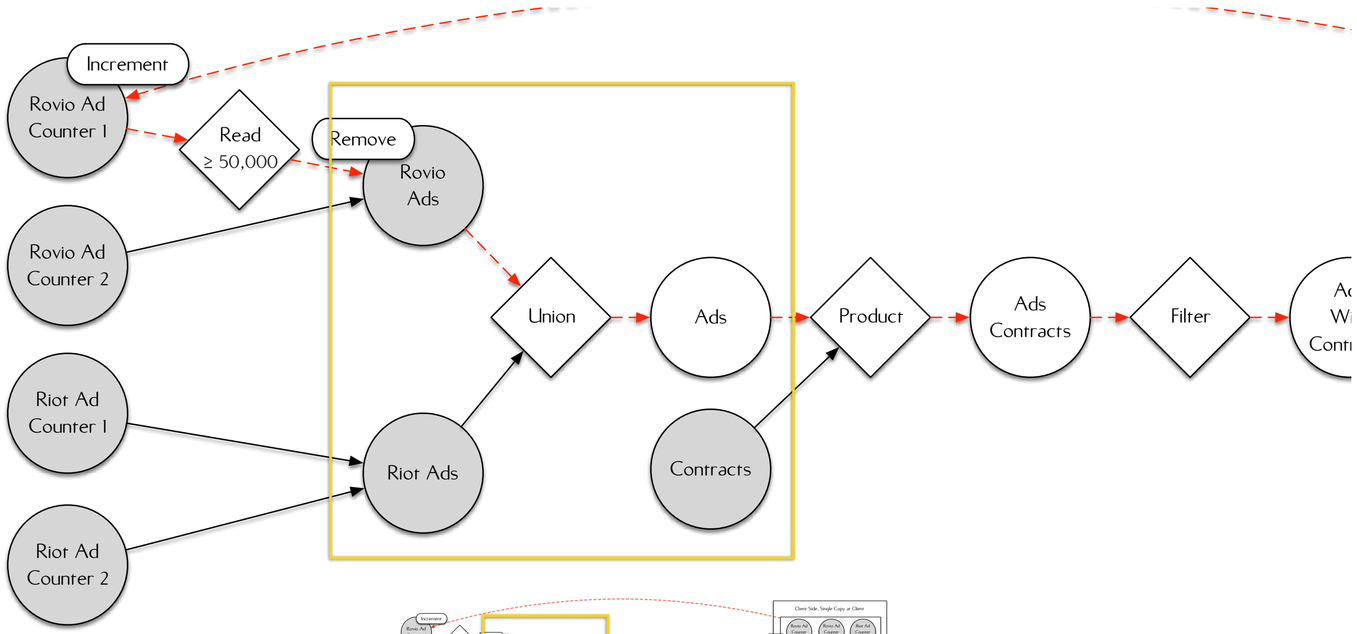
Advertisement Counter

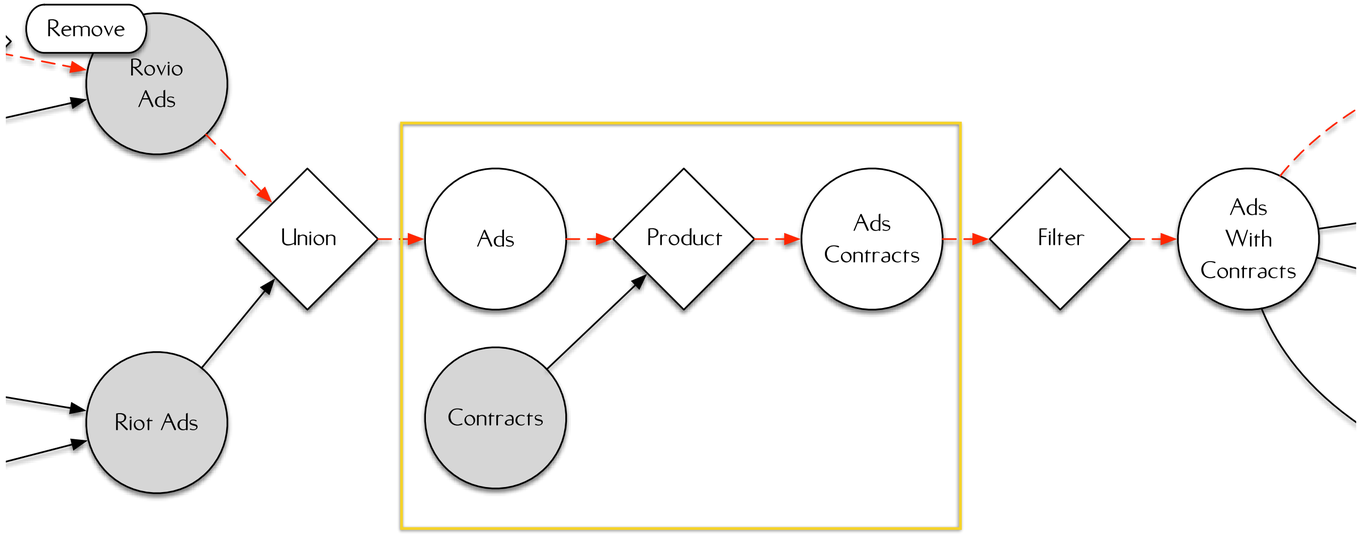
- Mobile game platform selling advertisement space
  Advertisements are paid according to a minimum number of impressions
- Clients will go offline
  Clients have limited connectivity and the system still needs to make progress while clients are offline

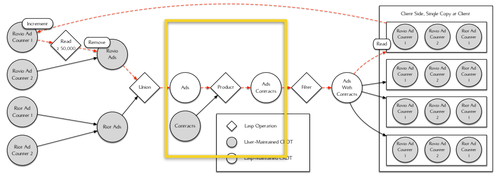

Increment
Rovio Ad Counter 1
Read ≥ 50,000
Remove
Rovio Ads
Rovio Ad Counter 2
Riot Ad Counter 1
Riot Ads
Riot Ad Counter 2
Union
Ads
Contracts
Product
Ads Contracts
Filter
Ads With Contracts
Read
Client Side, Single Copy at Client
Rovio Ad Counter 1
Rovio Ad Counter 2
Riot Ad Counter 1
Lasp Operation
User-Maintained CRDT
Lasp-Maintained CRDT

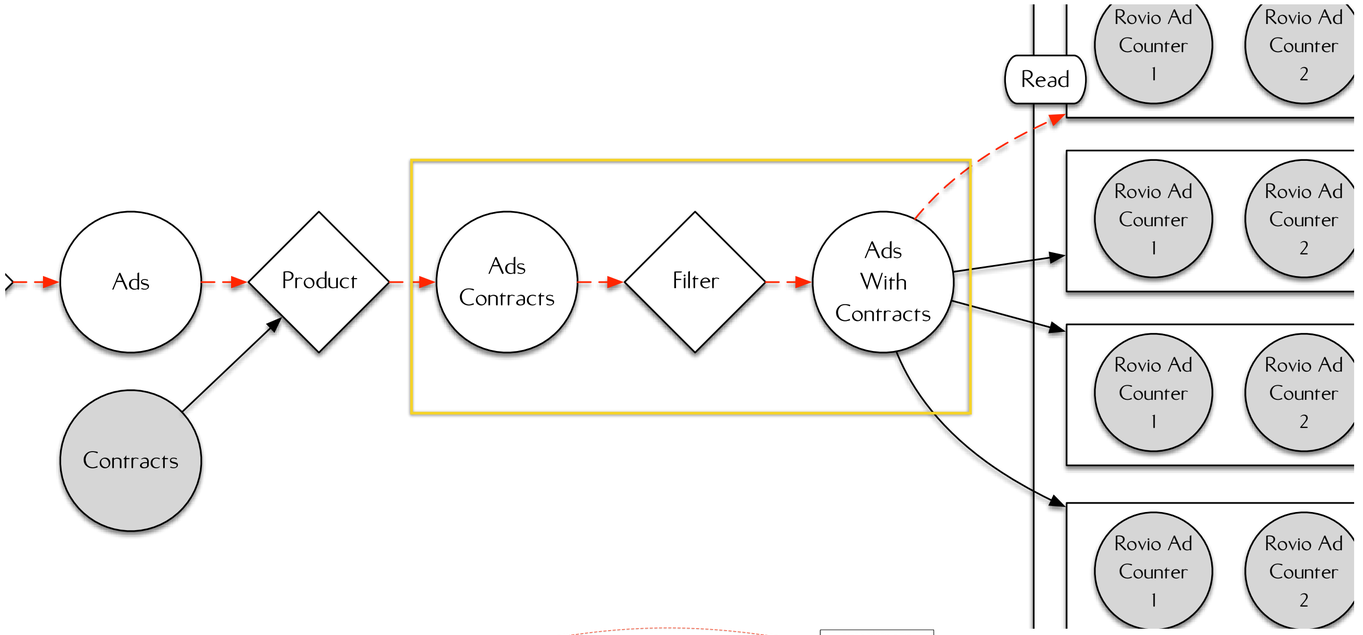

Rovio Ad
Counter 2
Riot Ad
Counter 1
Riot Ad
Counter 2
Riot Ads
Union
Ads
Contracts
Prodı

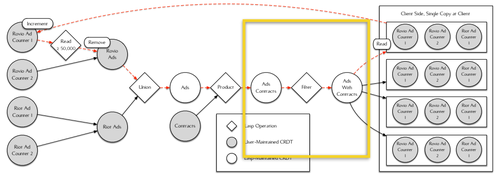


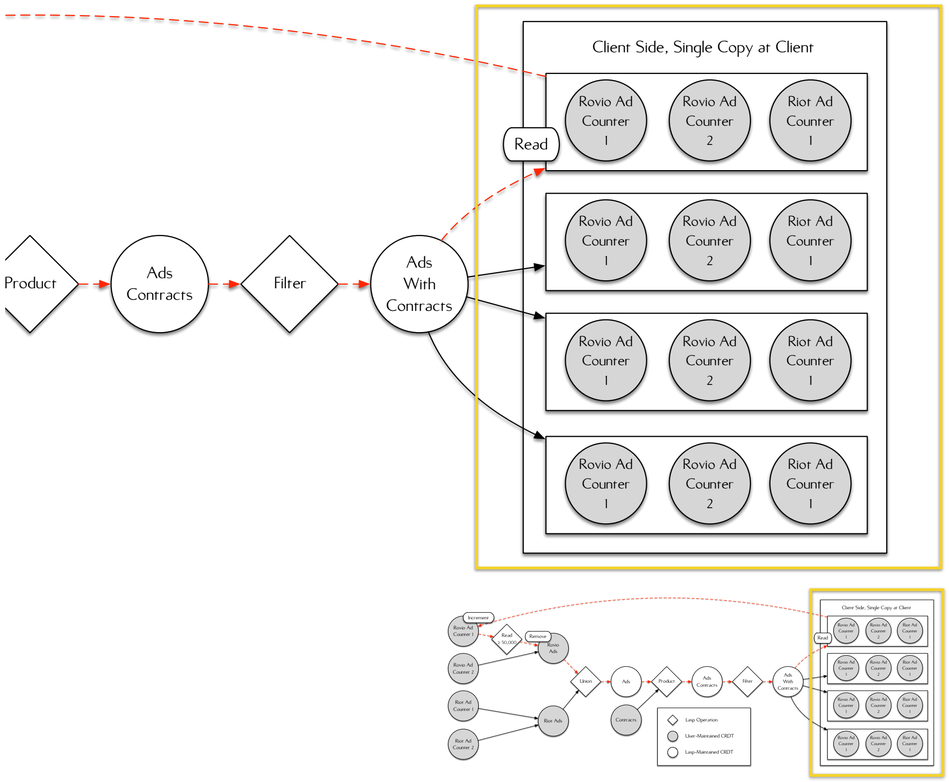

Increment
Rovio Ad Counter 1
Read ≥ 50,000
Remove
Rovio Ads
Rovio Ad Counter 2
Union
Ads
Product
Ads Contracts
Filter
Riot Ad Counter 1
Riot Ads
Contracts
Riot Ad Counter 2

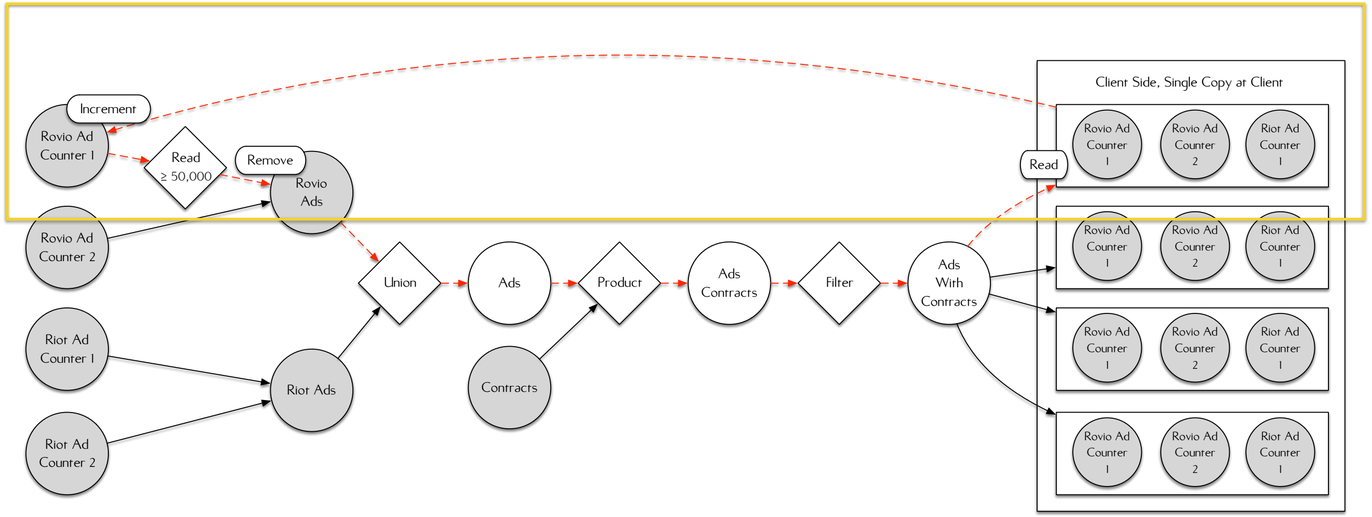

Remove
Rovio Ads
Union
Ads
Product
Ads Contracts
Filter
Ads With Contracts
Riot Ads
Contracts

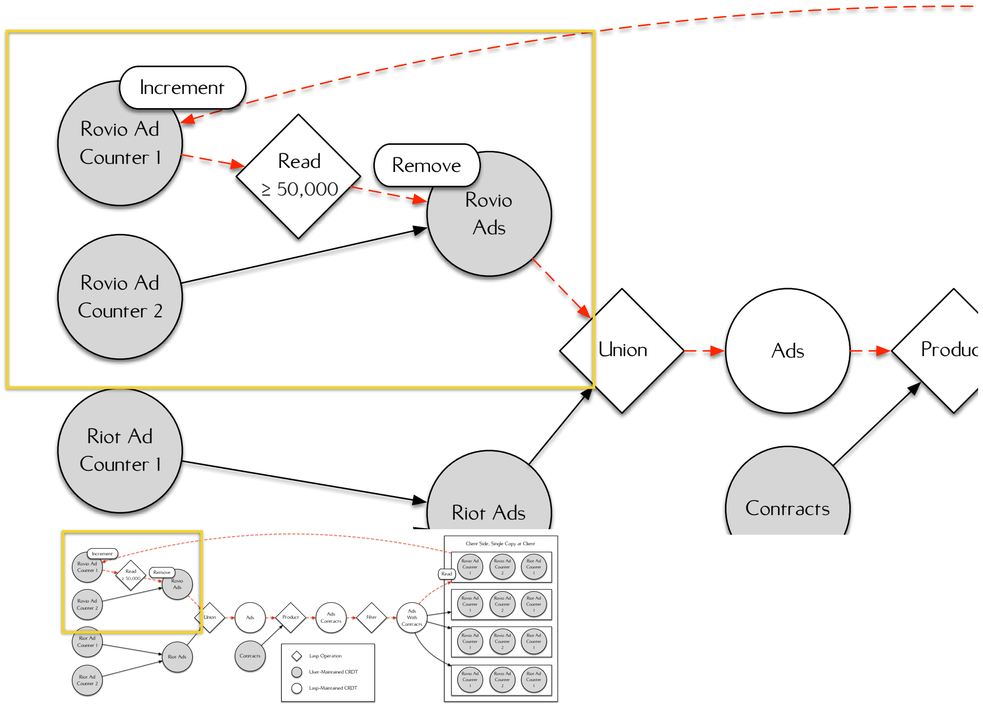


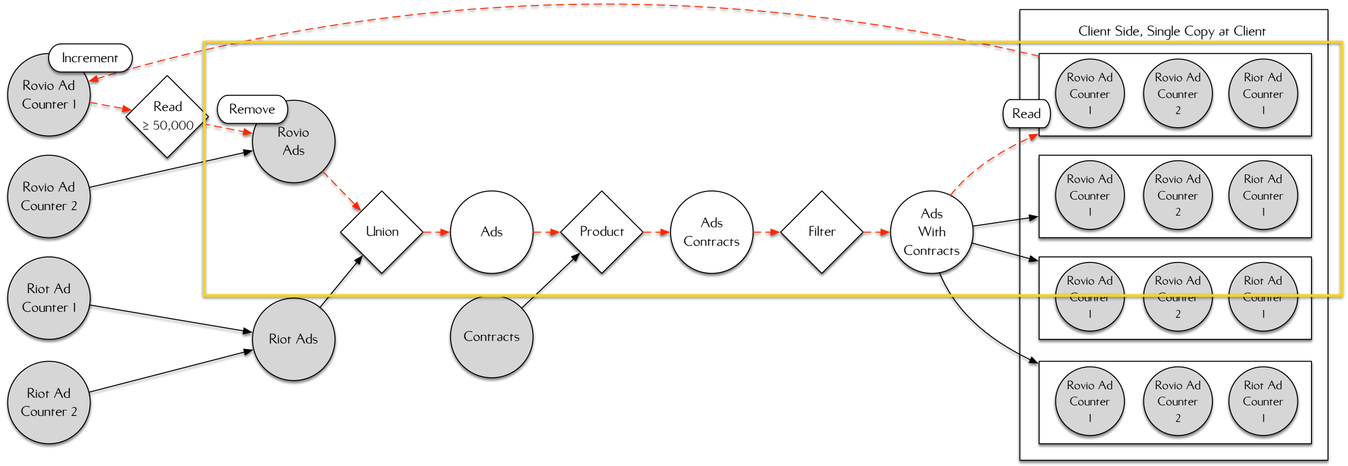

Ads
Contracts
Product
Ads
Contracts
Filter
Ads
With
Contracts
Read
Rovio Ad
Counter
1
Rovio Ad
Counter
2
Rovio Ad
Counter
1
Rovio Ad
Counter
2
Rovio Ad
Counter
1
Rovio Ad
Counter
2
Rovio Ad
Counter
1
Rovio Ad
Counter
2

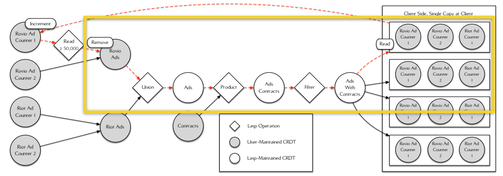



Client Side, Single Copy at Client
Read
Rovio Ad Counter 1
Rovio Ad Counter 2
Riot Ad Counter 1
Product
Ads Contracts
Filter
Ads With Contracts


Client Side, Single Copy at Client
Increment
Rovio Ad Counter 1
Read
≥ 50,000
Remove
Rovio Ads
Rovio Ad Counter 2
Riot Ad Counter 1
Riot Ad Counter 2
Riot Ads
Union
Ads
Contracts
Product
Ads Contracts
Filter
Ads With Contracts
Read
Rovio Ad Counter 1
Rovio Ad Counter 2
Riot Ad Counter 1


Increment
Rovio Ad Counter 1
Read ≥ 50,000
Remove
Rovio Ads
Rovio Ad Counter 2
Union
Ads
Produc
Riot Ad Counter 1
Riot Ads
Contracts


Client Side, Single Copy at Client
Increment
Rovio Ad Counter 1
Read ≥ 50,000
Remove
Rovio Ads
Rovio Ad Counter 2
Riot Ad Counter 1
Riot Ads
Riot Ad Counter 2
Union
Ads
Contracts
Product
Ads Contracts
Filter
Ads With Contracts
Read
Rovio Ad Counter 1
Rovio Ad Counter 2
Riot Ad Counter 1



Evaluation
Initial Evaluation

Background
Distributed Erlang

- Transparent distribution
  Built-in, provided by Erlang/BEAM, cross-node message passing.

Background
Distributed Erlang

- Transparent distribution
  Built-in, provided by Erlang/BEAM, cross-node message passing.
- Known scalability limitations
  Analyzed in academic in various publications.

Background
Distributed Erlang

- Transparent distribution
  Built-in, provided by Erlang/BEAM, cross-node message passing.
- Known scalability limitations
  Analyzed in academic in various publications.
  - Single connection
    Head of line blocking.

Background
Distributed Erlang

- Transparent distribution
  Built-in, provided by Erlang/BEAM, cross-node message passing.
- Known scalability limitations
  Analyzed in academic in various publications.
  - Single connection
    Head of line blocking.
  - Full membership
    All-to-all failure detection with heartbeats and timeouts.

Background
Erlang Port Mapper Daemon

- Operates on a known port
  Similar to Solaris sunrpc style portmap: known port for mapping to dynamic port-based services.

Background
Erlang Port Mapper Daemon

- Operates on a known port
  Similar to Solaris sunrpc style portmap: known port for mapping to dynamic port-based services.

- Bridged networking
  Problematic for cluster in bridged networking with dynamic port allocation.

Experiment Design

- Single application
  Advertisement counter example from Rovio Entertainment.

Experiment Design

- Single application
  Advertisement counter example from Rovio Entertainment.
- Runtime configuration
  Application controlled through runtime environment variables.

Experiment Design

- Single application
  Advertisement counter example from Rovio Entertainment.
- Runtime configuration
  Application controlled through runtime environment variables.
- Membership
  Full membership with Distributed Erlang via EPMD.

Experiment Design

- Single application
  Advertisement counter example from Rovio Entertainment.
- Runtime configuration
  Application controlled through runtime environment variables.
- Membership
  Full membership with Distributed Erlang via EPMD.
- Dissemination
  State-based object dissemination through anti-entropy protocol (fanout-based, PARC-style.)

Experiment Orchestration

- Docker and Mesos with Marathon
  Used for deployment of both EPMD and Lasp application.

Experiment Orchestration

- Docker and Mesos with Marathon
  Used for deployment of both EPMD and Lasp application.
- Single EPMD instance per slave
  Controlled through the use of host networking and HOSTNAME: UNIQUE constraints in Mesos.

Experiment **Orchestration**

- Docker and Mesos with Marathon
  Used for deployment of both EPMD and Lasp application.
- Single EPMD instance per slave
  Controlled through the use of host networking and HOSTNAME: UNIQUE constraints in Mesos.
- Lasp
  Local execution using host networking: connects to local EPMD.

Experiment **Orchestration**

- **Docker and Mesos with Marathon**
  Used for deployment of both EPMD and Lasp application.
- **Single EPMD instance per slave**
  Controlled through the use of host networking and HOSTNAME: UNIQUE constraints in Mesos.
- **Lasp**
  Local execution using host networking: connects to local EPMD.
- **Service Discovery**
  Service discovery facilitated through clustering EPMD instances through **Sprinter.**

Ideal Experiment

- Local Deployment
  High thread concurrency when operating with lower node count.

Ideal Experiment

- Local Deployment
  High thread concurrency when operating with lower node count.
- Cloud Deployment
  Low thread concurrency when operating with a higher node count.

Results
Initial Evaluation

Initial Evaluation

- Moved to DC/OS exclusively
  Environments too different: too much work needed to be adapted for things to work correctly.

Initial Evaluation

- Moved to DC/OS exclusively
  Environments too different: too much work needed to be adapted for things to work correctly.
- Single orchestration task
  Dispatched events, controlled when to start and stop the evaluation and performed log aggregation.

Initial Evaluation

- Moved to DC/OS exclusively
  Environments too different: too much work needed to be adapted for things to work correctly.
- Single orchestration task
  Dispatched events, controlled when to start and stop the evaluation and performed log aggregation.
  - Bottleneck
    Events immediately dispatched: would require blocking for processing acknowledgment.

Initial Evaluation

- Moved to DC/OS exclusively
  Environments too different: too much work needed to be adapted for things to work correctly.
- Single orchestration task
  Dispatched events, controlled when to start and stop the evaluation and performed log aggregation.
  - Bottleneck
    Events immediately dispatched: would require blocking for processing acknowledgment.
  - Unrealistic
    Events do not queue up all at once for processing by the client.

Lasp Difficulties

- Too expensive
  2.0 CPU and 2048 MiB of memory.

Lasp Difficulties

- Too expensive
  2.0 CPU and 2048 MiB of memory.
- Weeks spent adding instrumentation
  Process level, VM level, Erlang Observer instrumentation to identify heavy CPU and memory processes.

Lasp Difficulties

- Too expensive
  2.0 CPU and 2048 MiB of memory.
- Weeks spent adding instrumentation
  Process level, VM level, Erlang Observer instrumentation to identify heavy CPU and memory processes.
- Dissemination too expensive
  1000 threads to a single dissemination process (one Mesos task) leads to backed up message queues and memory leaks.

Lasp Difficulties

- Too expensive
  2.0 CPU and 2048 MiB of memory.
- Weeks spent adding instrumentation
  Process level, VM level, Erlang Observer instrumentation to identify heavy CPU and memory processes.
- Dissemination too expensive
  1000 threads to a single dissemination process (one Mesos task) leads to backed up message queues and memory leaks.
- Unrealistic
  Two different dissemination mechanisms: thread to thread and node to node: one is synthetic.

EPMD Difficulties

- Nodes become unregistered
  Nodes randomly unregistered with EPMD during execution.

EPMD Difficulties

- Nodes become unregistered
  Nodes randomly unregistered with EPMD during execution.

- Lost connection
  EPMD loses connections with nodes for some arbitrary reason.

EPMD Difficulties

- Nodes become unregistered
  Nodes randomly unregistered with EPMD during execution.

- Lost connection
  EPMD loses connections with nodes for some arbitrary reason.

- EPMD task restarted by Mesos
  Restarted for an unknown reason, which leads Lasp instances to restart in their own container.

Overhead Difficulties

- Too much state
  Client would ship around 5 GiB of state within 90 seconds.

Overhead Difficulties

- Too much state
  Client would ship around 5 GiB of state within 90 seconds.

- Delta dissemination
  Delta dissemination only provides around a 30% decrease in state transmission.

Overhead Difficulties

- Too much state
  Client would ship around 5 GiB of state within 90 seconds.

- Delta dissemination
  Delta dissemination only provides around a 30% decrease in state transmission.

- Unbounded queues
  Message buffers would lead to VMs crashing because of large memory consumption.

Evaluation

Rearchitecture

Ditch Distributed Erlang

- Pluggable membership service
  Build pluggable membership service with abstract interface initially on EPMD and later migrate after tested.

Ditch Distributed Erlang

- Pluggable membership service
  Build pluggable membership service with abstract interface initially on EPMD and later migrate after tested.
- Adapt Lasp and Broadcast layer
  Integrate pluggable membership service throughout the stack and librate existing libraries from distributed Erlang.

Ditch Distributed Erlang

- Pluggable membership service
  Build pluggable membership service with abstract interface initially on EPMD and later migrate after tested.
- Adapt Lasp and Broadcast layer
  Integrate pluggable membership service throughout the stack and librate existing libraries from distributed Erlang.
- Build service discovery mechanism
  Mechanize node discovery outside of EPMD based on new membership service.

Partisan
(Membership Layer)

- Pluggable protocol membership layer
  Allow runtime configuration of protocols used for cluster membership.

Partisan
(Membership Layer)

- Pluggable protocol membership layer
  Allow runtime configuration of protocols used for cluster membership.
- Several protocol implementations:

Partisan
(Membership Layer)

- Pluggable protocol membership layer
  Allow runtime configuration of protocols used for cluster membership.
- Several protocol implementations:
  - Full membership via EPMD.

Partisan
(Membership Layer)

- Pluggable protocol membership layer
  Allow runtime configuration of protocols used for cluster membership.
- Several protocol implementations:
  - Full membership via EPMD.
  - Full membership via TCP.

Partisan
(Membership Layer)

- Pluggable protocol membership layer
  Allow runtime configuration of protocols used for cluster membership.
- Several protocol implementations:
  - Full membership via EPMD.
  - Full membership via TCP.
  - Client-server membership via TCP.

Partisan
(Membership Layer)

- Pluggable protocol membership layer
  Allow runtime configuration of protocols used for cluster membership.
- Several protocol implementations:
  - Full membership via EPMD.
  - Full membership via TCP.
  - Client-server membership via TCP.
  - Peer-to-peer membership via TCP (with HyParView)

Partisan
(Membership Layer)

- Pluggable protocol membership layer
  Allow runtime configuration of protocols used for cluster membership.
- Several protocol implementations:
  - Full membership via EPMD.
  - Full membership via TCP.
  - Client-server membership via TCP.
  - Peer-to-peer membership via TCP (with HyParView)
- Visualization
  Provide a force-directed graph-based visualization engine for cluster debugging in real-time.

Partisan
(Full via EPMD or TCP)

- **Full membership**
  Nodes have full visibility into the entire graph.

Partisan
(Full via EPMD or TCP)

- Full membership
  Nodes have full visibility into the entire graph.
- Failure detection
  Performed by peer-to-peer heartbeat messages with a timeout.

Partisan
(Full via EPMD or TCP)

- Full membership
  Nodes have full visibility into the entire graph.
- Failure detection
  Performed by peer-to-peer heartbeat messages with a timeout.
- Limited scalability
  Heartbeat interval increases when node count increases leading to false or delayed detection.

Partisan
(Full via EPMD or TCP)

- Full membership
  Nodes have full visibility into the entire graph.
- Failure detection
  Performed by peer-to-peer heartbeat messages with a timeout.
- Limited scalability
  Heartbeat interval increases when node count increases leading to false or delayed detection.
- Testing
  Used to create the initial test suite for Partisan.

Partisan
(Client-Server Model)

- Client-server membership
  Server has all peers in the system as peers; client has only the server as a peer.

Partisan
(Client-Server Model)

- Client-server membership
  Server has all peers in the system as peers; client has only the server as a peer.
- Failure detection
  Nodes heartbeat with timeout all peers they are aware of.

Partisan
(Client-Server Model)

- Client-server membership
  Server has all peers in the system as peers; client has only the server as a peer.
- Failure detection
  Nodes heartbeat with timeout all peers they are aware of.
- Limited scalability
  Single point of failure: server; with limited scalability on visibility.

Partisan
(Client-Server Model)

- Client-server membership
  Server has all peers in the system as peers; client has only the server as a peer.
- Failure detection
  Nodes heartbeat with timeout all peers they are aware of.
- Limited scalability
  Single point of failure: server; with limited scalability on visibility.
- Testing
  Used for baseline evaluations as “reference” architecture.

Partisan
(HyParView, default)

- Partial view protocol
  Two views: active (fixed) and passive (log n); passive used for failure replacement with active view.

Partisan
(HyParView, default)

- Partial view protocol
  Two views: active (fixed) and passive (log n); passive used for failure replacement with active view.

- Failure detection
  Performed by monitoring active TCP connections to peers with keep-alive enabled.

Partisan
(HyParView, default)

- Partial view protocol
  Two views: active (fixed) and passive (log n); passive used for failure replacement with active view.

- Failure detection
  Performed by monitoring active TCP connections to peers with keep-alive enabled.

- Very scalable (10k+ nodes during academic evaluation)
  However, probabilistic; potentially leads to isolated nodes during churn.

Sprinter
(Service Discovery)

- Responsible for clustering tasks
  Uses Partisan to cluster all nodes and ensure connected overlay network: reads information from Marathon.

Sprinter
(Service Discovery)

- Responsible for clustering tasks
  Uses Partisan to cluster all nodes and ensure connected overlay network: reads information from Marathon.
- Node local
  Operates at each node and is responsible for taking actions to ensure connected graph: required for probabilistic protocols.

Sprinter
(Service Discovery)

- Responsible for clustering tasks
  Uses Partisan to cluster all nodes and ensure connected overlay network: reads information from Marathon.
- Node local
  Operates at each node and is responsible for taking actions to ensure connected graph: required for probabilistic protocols.
- Membership mode specific
  Knows, based on the membership mode, how to properly cluster nodes and enforces proper join behaviour.

Debugging Sprinter

- S3 archival
  Nodes periodically snapshot their membership view for analysis.

Debugging **Sprinter**

- S3 archival
  Nodes periodically snapshot their membership view for analysis.
- Elected node (or group) analyses
  Periodically analyses the information in S3 for the following:

Debugging **Sprinter**

- S3 archival
  Nodes periodically snapshot their membership view for analysis.
- Elected node (or group) analyses
  Periodically analyses the information in S3 for the following:
  - Isolated node detection
    Identifies isolated nodes and takes corrective measures to repair the overlay.

Debugging **Sprinter**

- S3 archival
  Nodes periodically snapshot their membership view for analysis.
- Elected node (or group) analyses
  Periodically analyses the information in S3 for the following:
  - Isolated node detection
    Identifies isolated nodes and takes corrective measures to repair the overlay.
  - Verifies symmetric relationship
    Ensures that if a node knows about another node, the relationship is symmetric: prevents I know you, but you don’t know me.

Debugging **Sprinter**

- S3 archival
  Nodes periodically snapshot their membership view for analysis.
- Elected node (or group) analyses
  Periodically analyses the information in S3 for the following:
  - Isolated node detection
    Identifies isolated nodes and takes corrective measures to repair the overlay.
  - Verifies symmetric relationship
    Ensures that if a node knows about another node, the relationship is symmetric: prevents I know you, but you don't know me.
  - Periodic alerting
    Alerts regarding disconnected graphs so external measures can be taken, if necessary.

Evaluation

Next Evaluation

Evaluation **Strategy**

- Deployment and runtime configuration
  Ability to deploy a cluster of node and configure simulations at runtime.

Evaluation **Strategy**

- Deployment and runtime configuration
  Ability to deploy a cluster of node and configure simulations at runtime.
- Each simulation:

Evaluation **Strategy**

- Deployment and runtime configuration
  Ability to deploy a cluster of node and configure simulations at runtime.
- Each simulation:
  - Different application scenario
    Uniquely execute a different application scenario at runtime based on runtime configuration.

Evaluation **Strategy**

- Deployment and runtime configuration
  Ability to deploy a cluster of node and configure simulations at runtime.
- Each simulation:
  - Different application scenario
    Uniquely execute a different application scenario at runtime based on runtime configuration.
  - Result aggregation
    Aggregate results at end of execution and archive these results.

Evaluation **Strategy**

- Deployment and runtime configuration
  Ability to deploy a cluster of node and configure simulations at runtime.
- Each simulation:
  - Different application scenario
    Uniquely execute a different application scenario at runtime based on runtime configuration.
  - Result aggregation
    Aggregate results at end of execution and archive these results.
  - Plot generation
    Automatically generate plots for the execution and aggregate the results of multiple executions.

Evaluation **Strategy**

- Deployment and runtime configuration
  Ability to deploy a cluster of node and configure simulations at runtime.
- Each simulation:
  - Different application scenario
    Uniquely execute a different application scenario at runtime based on runtime configuration.
  - Result aggregation
    Aggregate results at end of execution and archive these results.
  - Plot generation
    Automatically generate plots for the execution and aggregate the results of multiple executions.
- Minimal coordination
  Work must be performed with minimal coordination, as a single orchestrator is a scalability bottleneck for large applications.

Completion **Detection**

- “Convergence Structure”
  Uninstrumented CRDT of grow-only sets containing counters that each node manipulates.

Completion **Detection**

- “Convergence Structure”
  Uninstrumented CRDT of grow-only sets containing counters that each node manipulates.
- Simulates a workflow
  Nodes use this operation to simulate a lock-stop workflow for the experiment.

Completion **Detection**

- “Convergence Structure”
  Uninstrumented CRDT of grow-only sets containing counters that each node manipulates.
- Simulates a workflow
  Nodes use this operation to simulate a lock-stop workflow for the experiment.
  - Event Generation
    Event generation toggles a boolean for the node to show completion.

Completion **Detection**

- “Convergence Structure”
  Uninstrumented CRDT of grow-only sets containing counters that each node manipulates.
- Simulates a workflow
  Nodes use this operation to simulate a lock-stop workflow for the experiment.
  - Event Generation
    Event generation toggles a boolean for the node to show completion.
  - Log Aggregation
    Completion triggers log aggregation.

Completion **Detection**

- “Convergence Structure”
  Uninstrumented CRDT of grow-only sets containing counters that each node manipulates.
- Simulates a workflow
  Nodes use this operation to simulate a lock-stop workflow for the experiment.
  - Event Generation
    Event generation toggles a boolean for the node to show completion.
  - Log Aggregation
    Completion triggers log aggregation.
  - Shutdown
    Upon log aggregation completion, nodes shutdown.

Completion **Detection**

- “Convergence Structure”
  Uninstrumented CRDT of grow-only sets containing counters that each node manipulates.
- Simulates a workflow
  Nodes use this operation to simulate a lock-stop workflow for the experiment.
  - Event Generation
    Event generation toggles a boolean for the node to show completion.
  - Log Aggregation
    Completion triggers log aggregation.
  - Shutdown
    Upon log aggregation completion, nodes shutdown.
- External monitoring
  When events complete execution, nodes automatically begin the next experiment.

Results
Next Evaluation

Results Lasp

- Single node orchestration: bad
  Not possible once you exceed a few nodes: message queues, memory, delays.

Results **Lasp**

- Single node orchestration: bad
  Not possible once you exceed a few nodes: message queues, memory, delays.

- Partial Views
  Required: rely on transitive dissemination of information and partial network knowledge.

Results Lasp

- Single node orchestration: bad
  Not possible once you exceed a few nodes: message queues, memory, delays.

- Partial Views
  Required: rely on transitive dissemination of information and partial network knowledge.

- Results
  Reduced Lasp memory footprint to 75MB; larger in practice for debugging.

Results Partisan

- Fast churn isolates nodes
  Need a repair mechanism: random promotion of isolated nodes; mainly issues of symmetry.

Results Partisan

- Fast churn isolates nodes
  Need a repair mechanism: random promotion of isolated nodes; mainly issues of symmetry.
- FIFO across connections
  Not per connection, but protocol assumes across all connections leading to false disconnects.

Results Partisan

- Fast churn isolates nodes
  Need a repair mechanism: random promotion of isolated nodes; mainly issues of symmetry.
- FIFO across connections
  Not per connection, but protocol assumes across all connections leading to false disconnects.
- Unrealistic system model
  You need per message acknowledgements for safety.

Results Partisan

- Fast churn isolates nodes
  Need a repair mechanism: random promotion of isolated nodes; mainly issues of symmetry.
- FIFO across connections
  Not per connection, but protocol assumes across all connections leading to false disconnects.
- Unrealistic system model
  You need per message acknowledgements for safety.
- Pluggable protocol helps debugging
  Being able to switch to full membership or client-server assists in debugging protocol vs. application problems.

Latest Results

- Reproducibility at 300 nodes for full applications
  Connectivity, but transient partitions and isolated nodes at 500 - 1000 nodes (across 140 instances.)

Latest Results

- **Reproducibility at 300 nodes for full applications**
  Connectivity, but transient partitions and isolated nodes at 500 - 1000 nodes (across 140 instances.)

- **Limited financially and by Amazon**
  Harder to run larger evaluations because we're limited financially (as a university) and because of Amazon limits.

Latest Results

- **Reproducibility at 300 nodes for full applications**
  Connectivity, but transient partitions and isolated nodes at 500 - 1000 nodes (across 140 instances.)

- **Limited financially and by Amazon**
  Harder to run larger evaluations because we're limited financially (as a university) and because of Amazon limits.

- **Mean state reduction per client**
  Around 100x improvement from our PaPoC 2016 initial evaluation results.

Plat à emporter

- Visualizations are important!
  Graph performance, visualize your cluster: all of these things lead to easier debugging.

Plat à emporter

- Visualizations are important!
  Graph performance, visualize your cluster: all of these things lead to easier debugging.
- Control changes
  No Lasp PR accepted without divergence, state transmission, and overhead graphs.

Plat à emporter

- Visualizations are important!
  Graph performance, visualize your cluster: all of these things lead to easier debugging.
- Control changes
  No Lasp PR accepted without divergence, state transmission, and overhead graphs.
- Automation
  Developers use graphs when they are easy to make: lower the difficulty for generation and understand how changes alter system behaviour.

Plat à emporter

- Visualizations are important!
  Graph performance, visualize your cluster: all of these things lead to easier debugging.
- Control changes
  No Lasp PR accepted without divergence, state transmission, and overhead graphs.
- Automation
  Developers use graphs when they are easy to make: lower the difficulty for generation and understand how changes alter system behaviour.
- Make work easily testable
  When you test locally and deploy globally, you need to make things easy to test, deploy and evaluate (for good science, I say!)

Thanks!

Christopher Meiklejohn
@cmeik
http://www.lasp-lang.org
http://github.com/lasp-lang