PRACTICAL EVALUATION OF THE LASP PROGRAMMING MODEL AT LARGE SCALE

Christopher S. Meiklejohn,
Vitor Enes, Junghun Yoo, Carlos Baquero,
Peter Van Roy, Annette Bieniusa

Université catholique de Louvain,
Instituto Superior Técnico,
Universidade do Minho,
University of Oxford,
Technische Universität Kaiserslautern

UCL Université catholique de Louvain

UNIVERSITY OF OXFORD

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

LIGHTKONE

Lightweight computation for networks at the edge

DISTRIBUTED APPLICATIONS EVERYWHERE!

Example applications: rich-web and mobile

- Store state to operate quickly, refresh state with the server periodically
- Typically “throw” concurrent updates away when conflicting updates occur (last-writer-wins)
- Few provide the ability to operate offline

Nowadays, application developers must reason about:

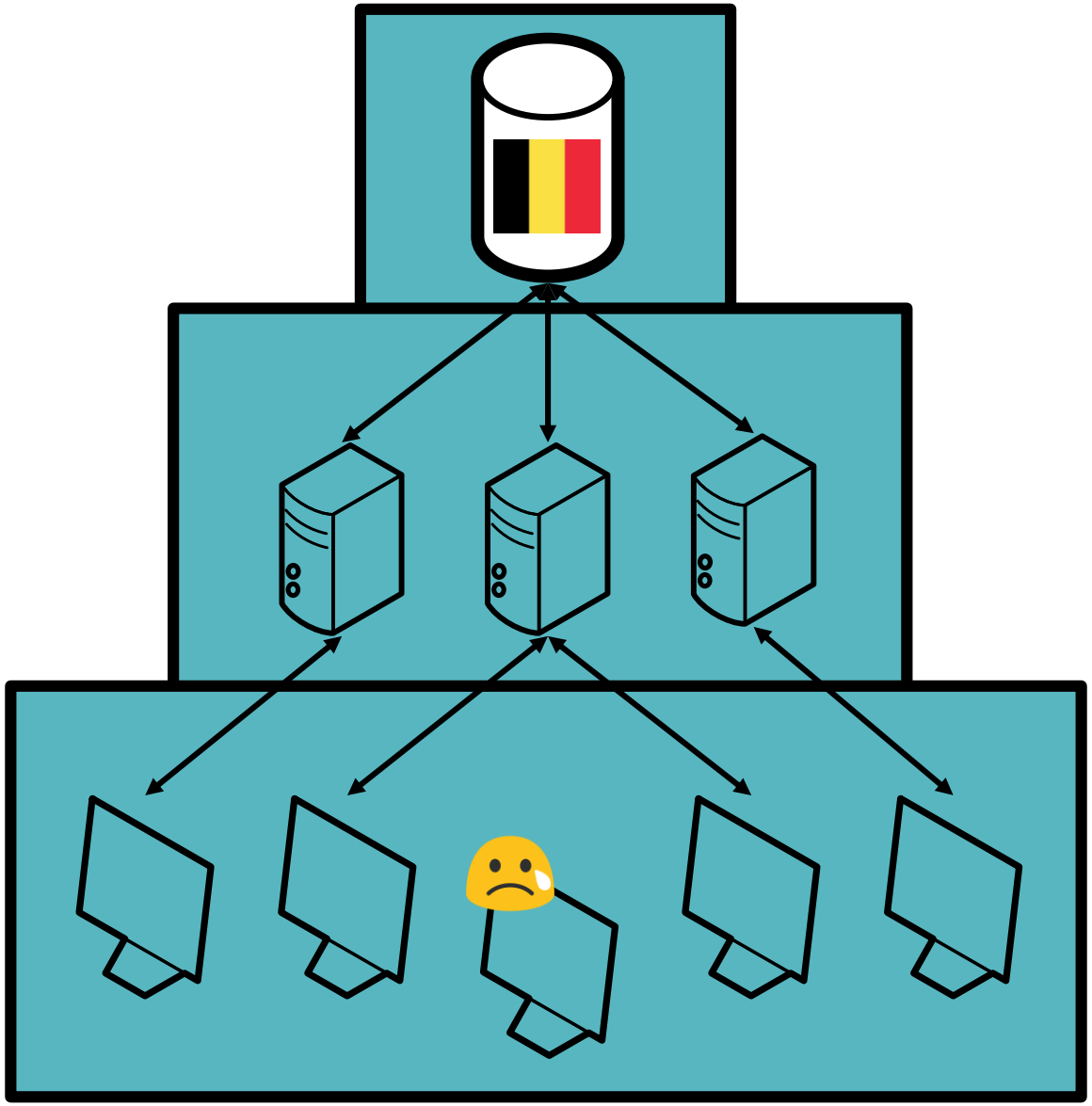
- Concurrent updates to shared state and conflict resolution
- Consistency of replicas
- Ordering of events
- Update visibility

TRADITIONAL ARCHITECTURE

- Communication through data center
- Application servers run business logic
- Clients must be online to operate

Analysis

- Application is easy to program
- Exhibits strong consistency
- Exhibits high latency (non-native)
- Exhibits low availability (DC-focused)

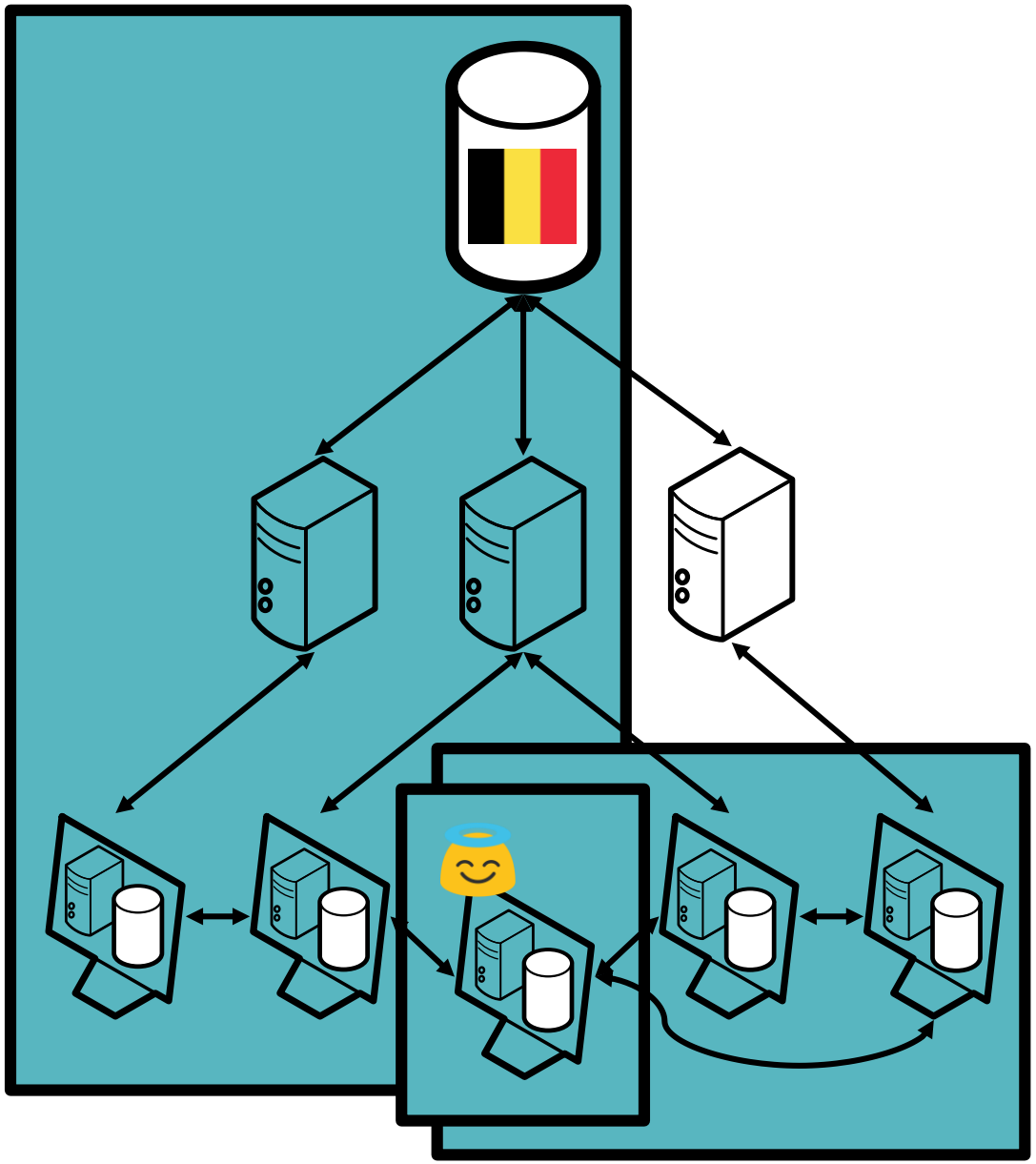


IDEAL ARCHITECTURE

- State replicated at the client
- Clients can communicate with other peers
- Clients can operate offline

Analysis

- Application is hard to program
- Exhibits weak consistency
- Exhibits low latency
- Exhibits high availability



PREVIOUS APPROACHES

Many systems and languages designed with scalability in mind

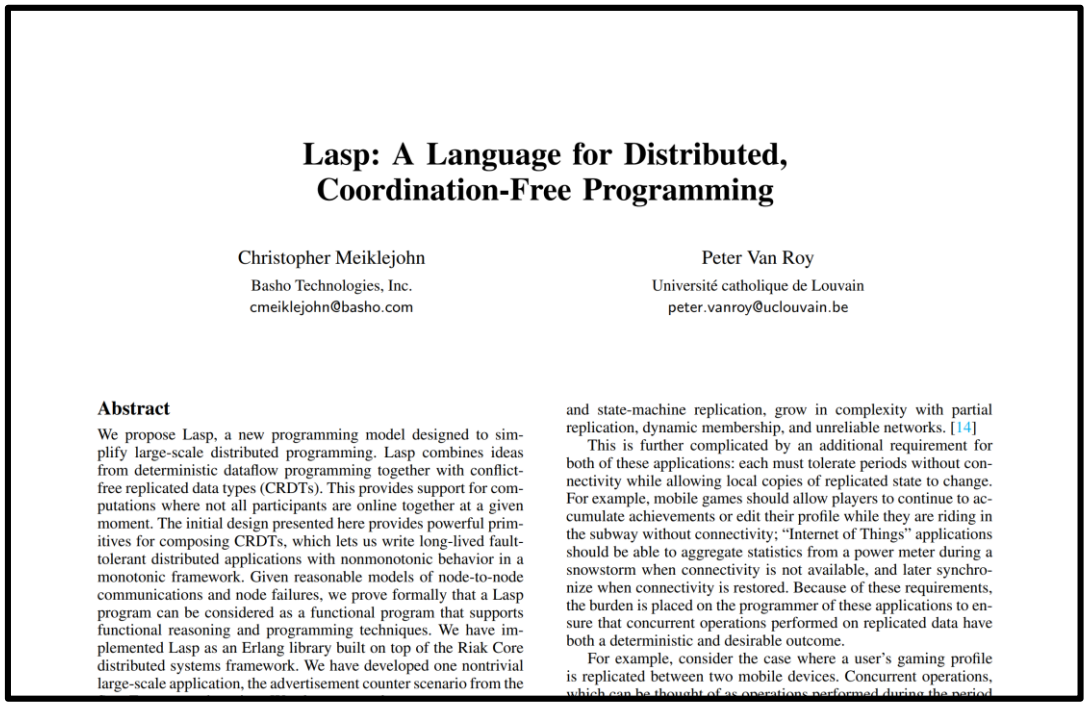
- Bayou (Terry et al. 1995)
- Bloom, Bloom_L (Alvaro et al. 2011, Conway et al. 2012)
- Cloud Types (Burckhardt et al. 2012), Global Sequence Protocol (Burckhardt et al. 2015)

Most, do not have evaluations demonstrating scalability in real world environments!

Demonstrating scalability of languages designed for scalability

- Non-trivial
- Rely on existing tooling, infrastructure which may be limited in scalability

LASP (PPDP '15)



Lasp: A Language for Distributed, Coordination-Free Programming

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

Abstract

We propose Lasp, a new programming model designed to simplify large-scale distributed programming. Lasp combines ideas from deterministic dataflow programming together with conflict-free replicated data types (CRDTs). This provides support for computations where not all participants are online together at a given moment. The initial design presented here provides powerful primitives for composing CRDTs, which lets us write long-lived fault-tolerant distributed applications with nonmonotonic behavior in a monotonic framework. Given reasonable models of node-to-node communications and node failures, we prove formally that a Lasp program can be considered as a functional program that supports functional reasoning and programming techniques. We have implemented Lasp as an Erlang library built on top of the Riak Core distributed systems framework. We have developed one nontrivial large-scale application, the advertisement counter scenario from the

and state-machine replication, grow in complexity with partial replication, dynamic membership, and unreliable networks. [14]

This is further complicated by an additional requirement for both of these applications: each must tolerate periods without connectivity while allowing local copies of replicated state to change. For example, mobile games should allow players to continue to accumulate achievements or edit their profile while they are riding in the subway without connectivity; "Internet of Things" applications should be able to aggregate statistics from a power meter during a snowstorm when connectivity is not available, and later synchronize when connectivity is restored. Because of these requirements, the burden is placed on the programmer of these applications to ensure that concurrent operations performed on replicated data have both a deterministic and desirable outcome.

For example, consider the case where a user's gaming profile is replicated between two mobile devices. Concurrent operations, which can be thought of as operations performed during the period

Declarative programming system that allows for distributed programming with co-designed runtime system

CRDTs: ADTs for distributed programming

- Data types containing a binary merge function for joining two replicas
- Used for value convergence under divergence introduced by concurrency

Functional programming model where CRDT is core data abstraction

LASP EXAMPLE

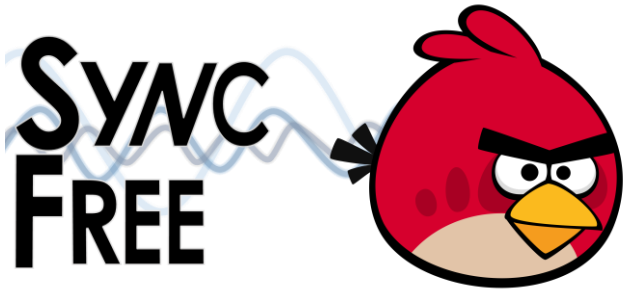
```
%% Create a set
A = declare(set)

%% Derive a new set
B = product(A, filter(P, A))

%% Create concurrent process
%% to insert into set
process do
      insert(A, random())
end
```

Creates a join-semilattice representation of a set (formalized as CRDT)

Creates a homomorphism to a join-semilattice B under image of product/filter

Concurrent additions produce a 'join' with A's state; triggers update of B

ADVERTISEMENT COUNTER



Industry use case from Rovio Entertainment

- Partner in SyncFree EU FP7 on coordination-free computation

Display advertisements while offline and track impressions

Disable advertisements when a threshold is reached

Interesting application requirements

- Replicated data, high contention
- Desire to scale to millions of clients
- Operation while client is disconnected

APPLICATION OUTLINE

1. Initialization

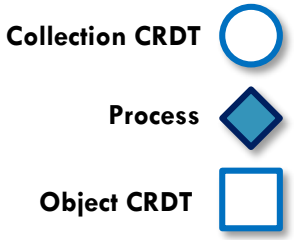
Create counters for each ad

2. Selection of displayable ads

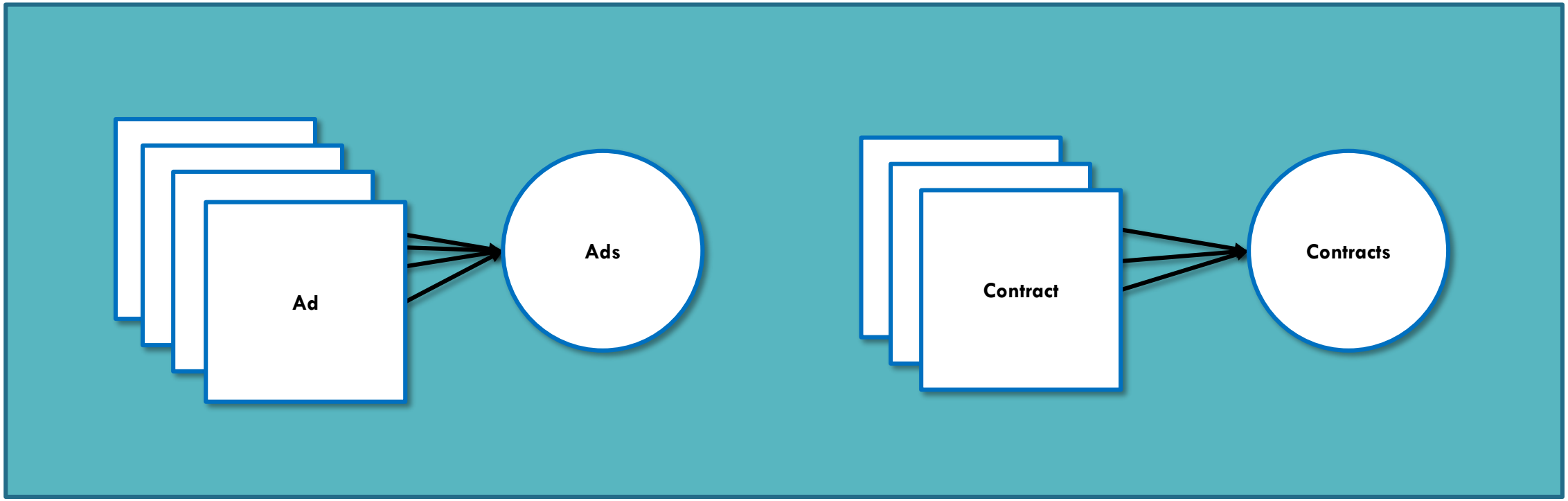
Filter set of ads into a set of advertisements that haven't met the threshold

3. Enforce invariant

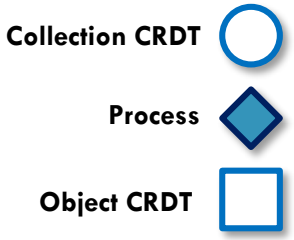
When a counter hits a threshold, remove it from the set of ads



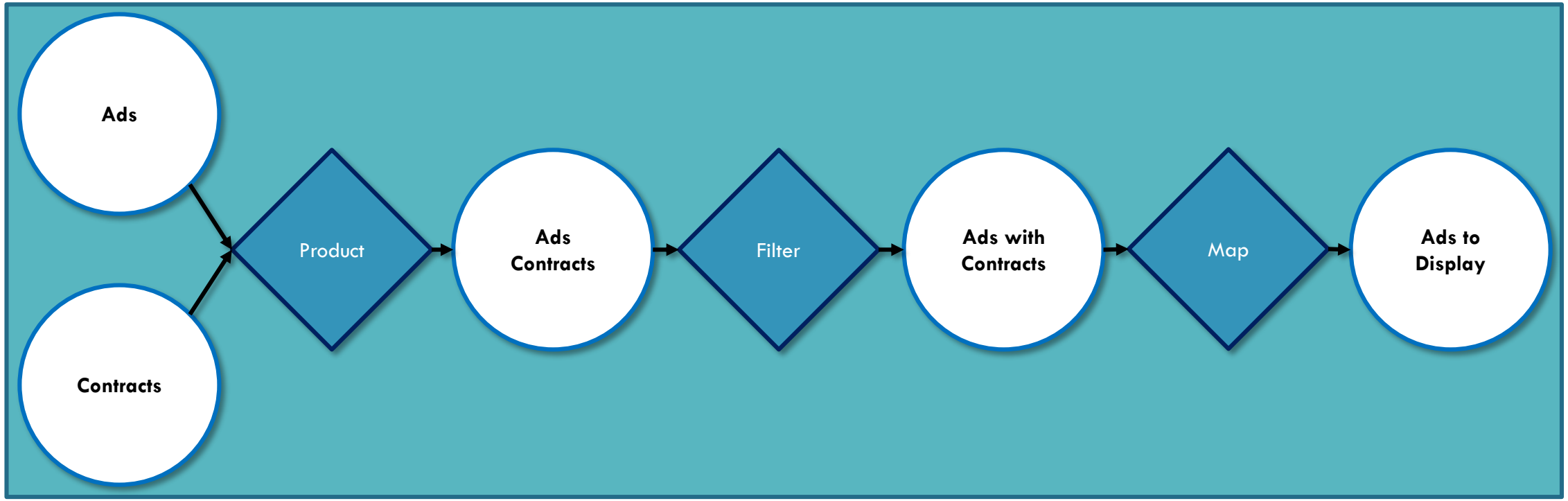
CREATION OF ADS AND CONTRACTS



Server: creates objects and inserts into collections

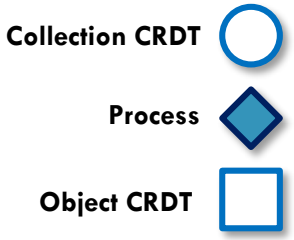


SELECTION OF DISPLAYABLE ADS

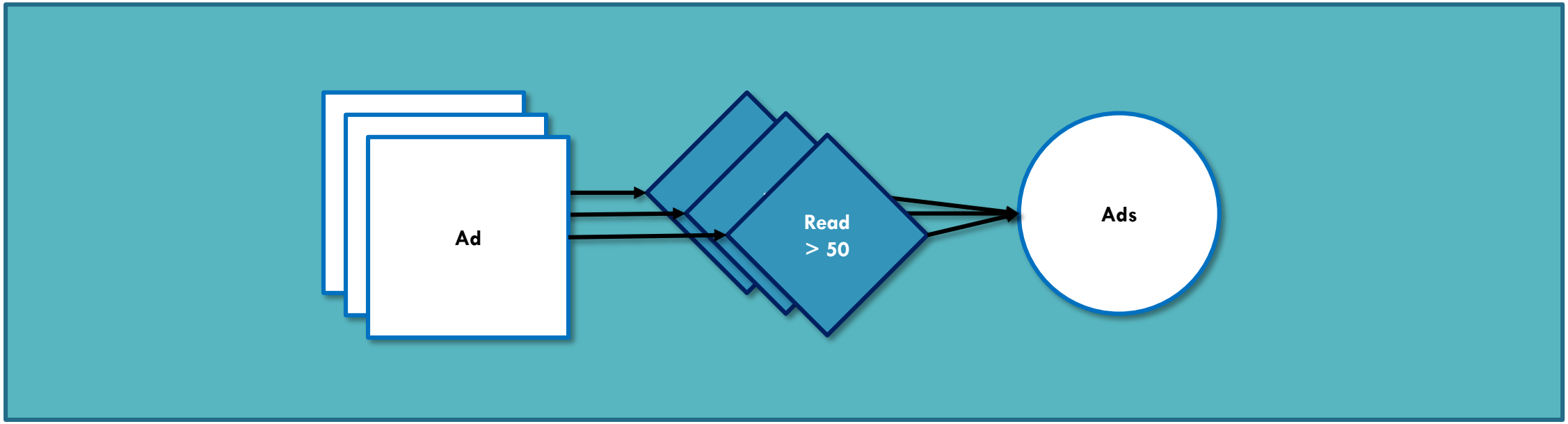


Server: constructs server dataflow

```
SELECT ads.id
FROM ads
INNER JOIN contracts
WHERE ads.id = contracts.ad_id
```



ENFORCEMENT OF INVARIANTS



Server: removes from collection on threshold reached

IMPLEMENTATION

Implementation was done using Distributed Erlang, a state-of-the-art production distributed runtime for the Erlang programming language

Lasp prototype written in Erlang

- Automatically propagates updates for replicated, shared data

[333 LOC] Server processes

- Create advertisement counters
- Disable advertisements at threshold

[276 LOC] Client processes

- Increment advertisement counters

50% of code is instrumentation

- Tracking state, logging updates, controlling experiment execution

ARCHITECTURE

We evaluate two architectures with two different runtime dissemination techniques for Lasp to see which yields the best scalability

Shared state for Lasp stored in KVS per node

- Variable identifiers point to locations in full replicated storage

Two cluster topologies

- Datacenter Lasp (Traditional)
  - One-hop DHT; structured overlay network
  - Clients communicate through server nodes
- Hybrid Gossip Lasp (Ideal)
  - Unstructured overlay network; partial membership
  - Inspired by the HyParView protocol

Two dissemination strategies

- State-based
  - Periodic, full state synchronization between peers via gossip
- Delta-based
  - Minimization of changes, sent to local peers in causal order
  - Not evaluated for DHT approach because of scalability in buffering updates for all local peers

EXPERIMENT CONFIGURATION

Experiments were run in the Amazon Cloud Computing environment; 2 experiments (at 30 minutes each) for each of the topologies and cluster sizes.

Amazon EC2

- 70 m3.2xlarge instances
- Subdivided using Apache Mesos via containers
  - Servers: 4 GB, 2 vCPU
  - Clients: 1 GB, 0.5 vCPU
- Experiment varied number of tasks launched by Mesos
  - 1 Erlang VM
  - 1 Lasp instance
  - 1 Unix Process

Environmental perturbations

- Tasks may be co-located
- Nodes communicate with each other through TCP
- Varying communication latencies between nodes
- Noisy-neighbors: might see effects from co-location

Conservative approximation to scalability

- Each task underapproximates the ability of modern mobile phones

EXPERIMENTAL WORKFLOW

Nondeterminism introduced from running on a production, industrial cloud environment was reduced by principled experimental workflow

Each node generates its own workflow, because a central task for workload generation slows down the system to the performance of the central task

1. Bootstrapping
   a) Cluster created
   b) Ensure single connected component
   c) Create advertisements
2. Simulation
   a) Each node begins generating its own workflow
   b) Periodically gossip state to local peers
3. Convergence
   a) Wait for all nodes to complete workload generation
   b) Wait for all nodes to see effect of the workload on all other nodes
4. Metrics Aggregation
   a) Perform metrics aggregation at all nodes
   b) Tear down cluster at end of the experiment

EXPERIMENTAL INFRASTRUCTURE

Technologies we built on top of, invented, or replaced to assist in the scalability of the Lasp runtime system

Apache Mesos

- Limited to 1,024 tasks
- Slow scaleup to 140 physical nodes
- Fast scaleup, for cost savings, triggered Mesos heartbeat lapses, disconnection, orphaned tasks

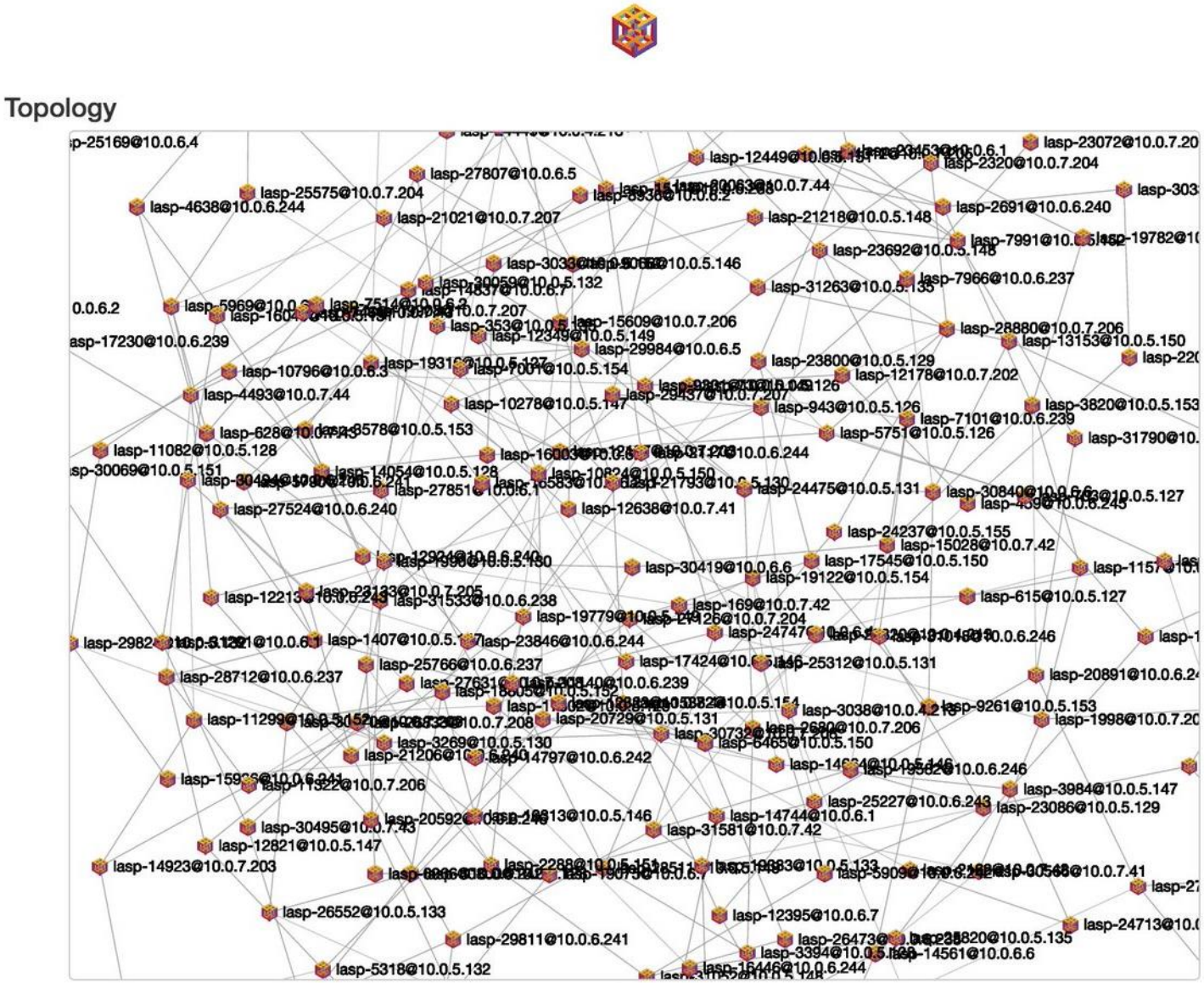
Sprinter (our contribution)

- Service discovery mechanism for task discovery
- Perform orchestration and experiment control
  a) Graph analysis for connectivity
  b) Delay experiment until single connected component
  c) Isolation reconnection
- Visual cluster debugger

Partisan (our contribution)

- Scalable replacement for Distributed Erlang
- Pluggable backends for different topologies
- Industry adoption
- Allow topology variation without application code change

CLUSTER VISUALIZER



WORKFLOW CRDT

Designing a coordination-free workflow management system for experiments using Lasp itself

Central orchestration of experiment problematic

- System only runs as fast as coordinator

Must have a barrier synchronization technique to prevent experiment running at different speeds at different nodes

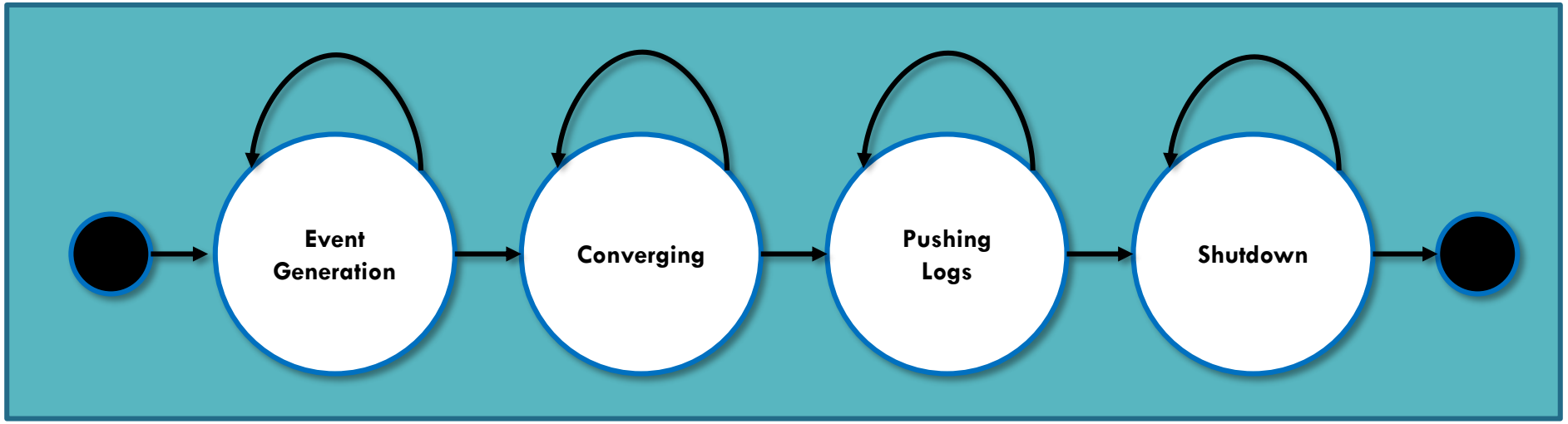
- Workload generation
- Blocking for event propagation and value convergence
- Log aggregation
- Shutdown

Uninstrumented workflow management CRDT

- Pairs of map lattices from node ids to boolean lattices
- Progress proceeds recursively as Booleans become true

WORKFLOW CRDT

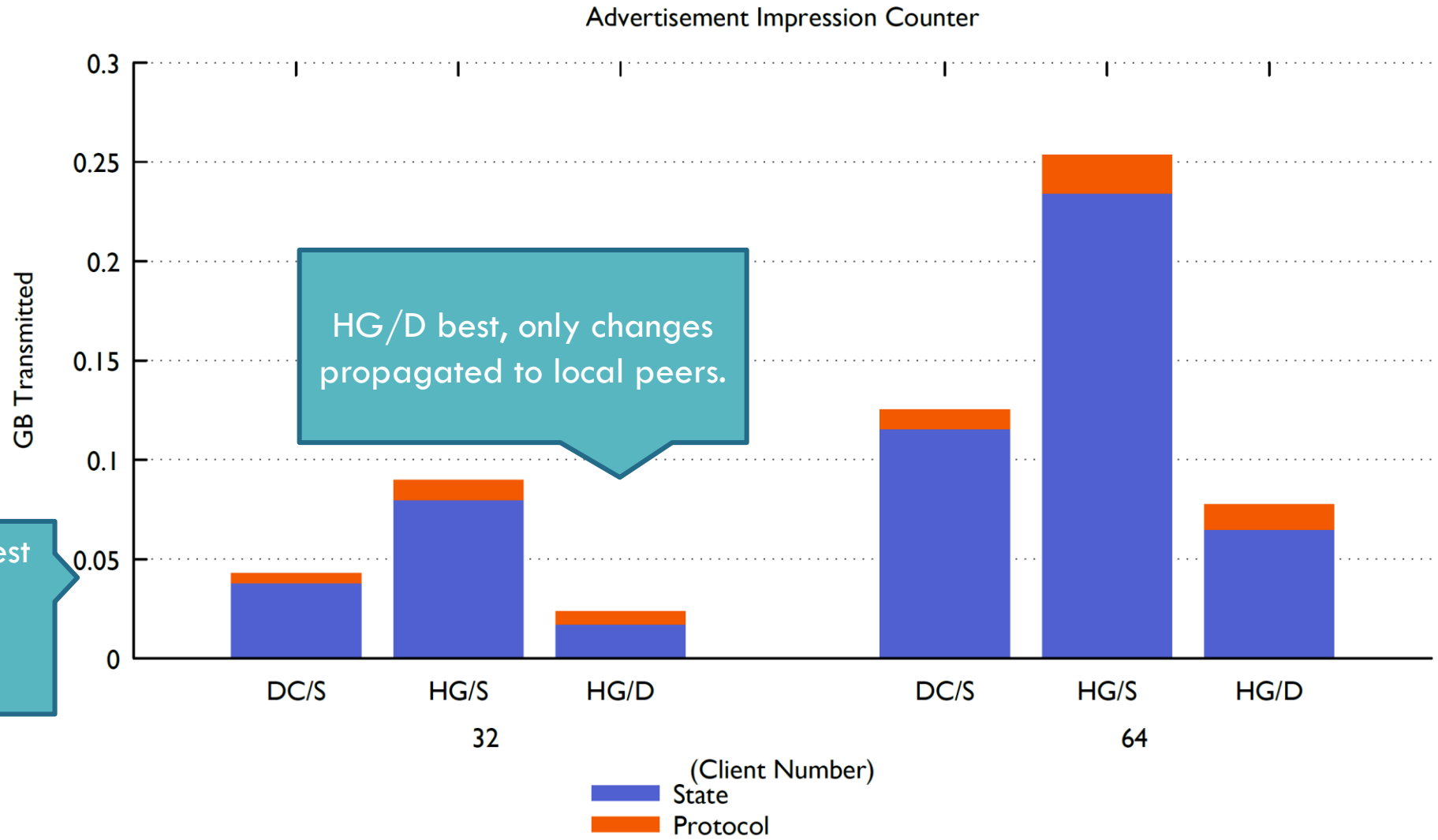
Nodes advance to the next stage when previous stage is complete.



Nodes spin on a stage until all nodes mark complete.

TOPOLOGIES

No delta evaluation for DC Lasp due to buffer overhead.

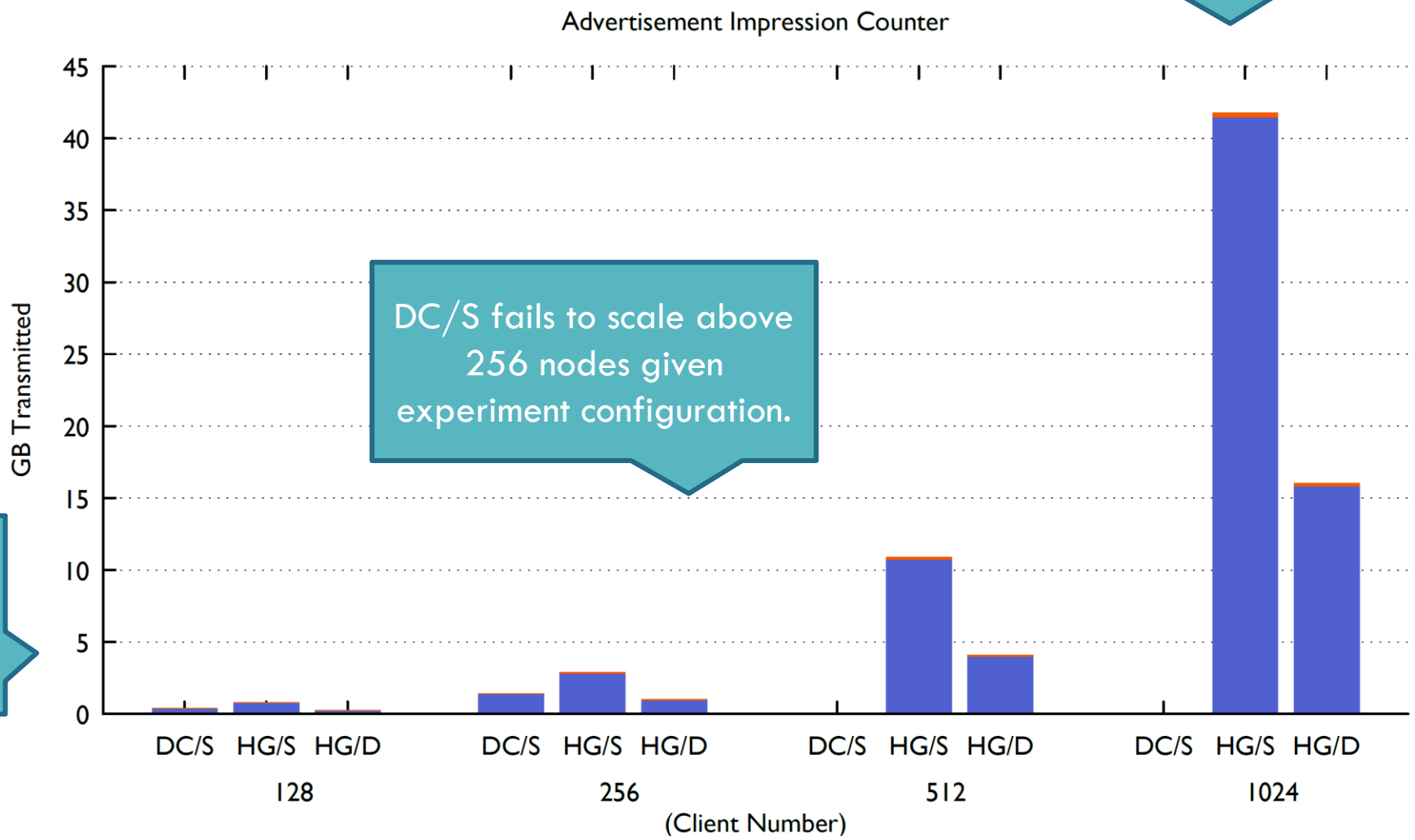


HG/D best, only changes propagated to local peers.

DC Lasp performs the best because lack of redundancy in communication.

SCALE

HG/S most expensive because of object transmission.



DC/S fails to scale above 256 nodes given experiment configuration.

Quadratic growth in lattice because of data structure – known solutions to reduce size.

TAKEAWAYS

Evaluating new designs for scalable systems will always be somewhat limited by the existing languages and tools we build on and be susceptible to problems in real world environments.

Existing tooling can be problematic

- Existing frameworks and tolling can arbitrarily alter performance, skew scalability to least scalable component

Visualizations are invaluable

- Assists in debugging, understanding behavior

Achieving reproducibility is non-trivial

- High-level abstractions provided by cloud are opaque

Performance can fluctuate

- VM placement, multiple levels of virtualization

Evaluations are expensive

- Real world evaluations take time, expensive in terms of resources, 9,900 EUR spend for few experiments