Practical Evaluation of the
Lasp Programming Model at Large Scale

An Experience Report

Christopher S. Meiklejohn*
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Vitor Enes
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal

Junghun Yoo
University of Oxford
Oxford, United Kingdom

Carlos Baquero†
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Annette Bieniusa
Technische Universität Kaiserslautern
Kaiserslautern, Germany

ABSTRACT

Programming models for building large-scale distributed applications assist the developer in reasoning about consistency and distribution. However, many of the programming models for weak consistency, which promise the largest scalability gains, have little in the way of evaluation to demonstrate the promised scalability. We present an experience report on the implementation and large-scale evaluation of one of these models, Lasp, originally presented at PPDP '15, which provides a declarative, functional programming style for distributed applications. We demonstrate the scalability of Lasp's prototype runtime implementation up to 1024 nodes in the Amazon cloud computing environment. It achieves high scalability by uniquely combining hybrid gossip with a programming model based on convergent computation. We report on the engineering challenges of this implementation and its evaluation, specifically related to operating research prototypes in a production cloud environment.

1 INTRODUCTION

Once a specialized field for applications that required large data sets, large-scale distributed applications have become commonplace in our globalized society. Regardless of whether you are developing a rich-web application or a native mobile application, managing distributed data is challenging. For simplicity, developers today typically resort to using a single database that provides a form of strong[1] consistency. In essence, the database serves as shared memory for the clients in the system.

A single database is an obvious bottleneck as it introduces a serialization point for all operations; this restricts the possible throughput of the system. As developers strive to provide a near-native experience where operations appear to happen immediately, and since not all clients can be geographically located close to the database, application performance can suffer as users move farther from the database; or worse, when clients can't communicate with the database at all because they are offline. To provide good user experience, including high availability and low latency, developers are forced to integrate replication in the system design.

Systems that favor weak consistency scale better: data items can be locally replicated, locally mutated by the application, and their state can be disseminated asynchronously, outside of the critical path. Weak consistency allows applications to continue to operate while offline. While these systems provide for high scalability and high performance, programming with weak consistency can be a challenge for the application developer as updates to data items have no guarantee on update visibility or update order. Concurrency poses an additional problem, as updates happening concurrently at different replicas may be conflicting.

Numerous systems and programming models[2, 3, 6, 8, 11, 14, 17, 18] have been proposed for working with weak consistency,

---

[1]For instance, linearizability, where a value follows the real-time order of updates.

however few have seen adoption. Many of the systems have sound theoretical foundations, but few perform evaluations at scale to demonstrate the benefits in practice. We believe that the lack of these results comes from the difficulty in the required infrastructure for large-scale experiments, and the challenges in engineering an implementation of a theoretical model using existing software languages and libraries.

In this paper, we discuss the practical issues encountered when evaluating one of these programming models, Lasp [4, 5], originally presented at PPDP '15. Lasp is designed using a holistic approach where the programming model was co-designed with its runtime system to ensure scalability. We examine the challenges of engineering an implementation capable of scaling to a large number of nodes running in a public cloud environment, using a real world application scenario. Further, we report on the engineering challenges of demonstrating the scalability of the Lasp model. Our experience report substantiates that empirically validating scalability is non-trivial, regardless of the programming model.

2 ADVERTISEMENT COUNTER

Lasp was invented to ease the development of distributed applications with weak consistency. The advertisement counter scenario from Rovio Entertainment, creator of Angry Birds, is an ideal fit for Lasp. This application counts the total number of times each advertisement is displayed on all client mobile phones, up to a given threshold for each. The application has the following properties:

- **Replicated data.** Data is fully replicated to every client in the system. This replicated data is under high contention by each client in the system.
- **High scalability.** Clients resemble individual mobile phone instances of the application, so the application should scale up to millions of clients.
- **High availability.** Clients need to continue operation when disconnected as mobile phones frequently have periods of signal loss (offline operation).

As part of the large-scale evaluation done in the SyncFree project, and following the personal curiosity of the developers, we decided to invest resources in using industrial-strength engineering techniques to evaluate the scalability of this application running in a real world production cloud environment.

2.1 Lasp

Lasp [11] is a programming model that allows developers to write applications with Conflict-Free Replicated Data Types (CRDTs) [1, 16]. CRDTs are abstract data types, designed for use in concurrent and distributed programming, that have a binary merge operation to join any two replicas of a single CRDT. Under concurrent modification without coordination, different replicas of a single CRDT may diverge; the merge operation supports value convergence by ensuring that given enough communication, all replicas, without coordination, will converge to a single deterministic value regardless of the order that data is received and merged.

Historically, before CRDTs were introduced, ad-hoc merge functions were used, often with few formal guarantees. Later, after their development, programmers who wanted to use CRDTs in their applications would have two choices: either, using a single CRDT from

existing literature to store application state, fitting their problem to an existing data structure; or, building a custom CRDT that fits their application domain, which requires to ensure that the merge operation is both deterministic and convergent.

Lasp improves this choice in two ways:

- **Composition.** Lasp provides set-theoretic and functional combinators for composing CRDTs into larger CRDTs.
- **Monotonic conditional.** Lasp introduces a conditional operation that allows the execution of application logic based on monotonic conditions[2] on CRDTs.

These two concepts allow Lasp applications to be both transparently and arbitrarily distributed across a set of nodes without altering application behavior. For brevity, the reader is referred to [11] for a full treatment of the Lasp semantics.

The advertisement counter uses two data structures from Lasp: the Add-Wins Set CRDT[3], where elements can be arbitrarily removed and inserted without coordination and under concurrent add and remove operations the add will 'win'; and the Grow-Only Counter CRDT, which models a counter that only increments.

2.2 Overview

The design of the advertisement counter is roughly broken into three components.

- **Initialization.** When the advertisement counter application is first initialized, we first create Grow-Only Counters for each unique advertisement we want to track impressions for, and we then insert references to them into an initial Add-Wins Set of advertisements.
- **Selection of displayable advertisements.** We define a dataflow computation in Lasp that will derive an Add-Wins Set of advertisements to display to the clients based on advertisements that have valid "contracts": records that represent that an advertisement is allowed to be displayed at the current time (Figure 1).
- **Enforcing invariants.** Since clients increment each advertisement counter as advertisement impressions occur, when the target number of impressions is reached both the client and the server will fire a trigger to remove the advertisement counter from the set of advertisements, to prevent the advertisement from being further displayed. This can be done without coordination through the use of the Add-Wins Set.
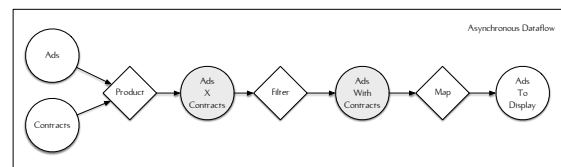

Figure 1: Asynchronous dataflow computation in Lasp that derives the set of displayable advertisements.

---

[2]Monotonicity implies that once a condition becomes true, it remains true; a monotonicity check can be done without distributed coordination.
[3]a.k.a. Observed-Remove Set

The advertisement counter has two important design choices, which makes its implementation in Lasp ideal.

- **Offline support.** As Angry Birds is a mobile application, there will be periods without connectivity. During this time, advertisements should still be displayable.
- **Lower-bound invariant.** Advertisements need to be displayed a minimum number of times; additional impressions are not problematic. This is a monotonic condition: once the condition is true, it remains true.

2.3 Implementation

The advertisement counter is broken into two components that work in concert. Both components track a single replica of a set of identifiers of displayable advertisements, and for each identifier a replica of an advertisement counter that tracks the total number of times the advertisement has been displayed to the user. Each node in our experiment runs either a single client or server process.

- **Server processes.** One or more server processes, each responsible for propagating their state to clients and disabling advertisements that have been displayed a minimum number of times by monotonically removing them from the set of displayable advertisements.
- **Client processes.** Many client processes that periodically propagate their state with other nodes, and increment their counter replicas based on a synthetic workload.

The prototype implementation of the Lasp programming model is built in the Erlang programming language and exposed to the user as an application library.

The fully instrumented Lasp advertisement counter client is implemented in 276 lines of Erlang code, and the fully instrumented advertisement counter server is 333 lines of Erlang code. Around 50% of this code is for instrumentation and orchestration, to ensure we can perform a full analysis of the application during experimentation. The Lasp runtime system takes care of cluster maintenance, data synchronization and storage, which are done manually in the previous approaches (ad-hoc merge or custom CRDT design).

3 SYSTEM ARCHITECTURE

To perform a real world evaluation of the advertisement counter, we implemented an efficient, scalable runtime system for Lasp. Lasp's runtime system is a highly-scalable eventually consistent data store with two different dissemination mechanisms (state-based vs. delta-based) and two different cluster topologies (datacenter vs. hybrid gossip). Lasp's programming model, presented in [11], sits above the data store and exposes a programming interface.

Datacenter Lasp [11] operates using a structured overlay network. Hybrid Gossip Lasp [12] uses an unstructured overlay network, and by design should achieve greater scalability and provide better fault-tolerance [15].

3.1 Datacenter Lasp

Datacenter Lasp refers to the prototype implementation of the runtime system presented with the programming model, at this conference two years ago [11].

In Datacenter Lasp, all CRDT state is both partitioned and replicated across several datacenter nodes. Client processes communicate directly with server processes that are running on datacenter nodes; client processes do not communicate amongst each other. Replication is used across datacenter nodes for fault tolerance, and partitioning/sharding is used for horizontal scalability: this is achieved through the use of consistent hashing and hash-space partitioning. In our experiments this is simplified and there is no partitioning, since the data set for our experiments never exceeds a single datacenter node's available capacity.

3.2 Hybrid Gossip Lasp

Hybrid Gossip Lasp is inspired by two Hybrid Gossip protocols, HyParView [10], and Plumtree [9]. In Hybrid Gossip Lasp, nodes are assembled in a peer-to-peer topology, where client processes can communicate either with server processes running on datacenter nodes or client processes. State is delivered transitively through other processes in the system: there is no need to communicate directly with a server process running on a datacenter node.

Hybrid Gossip Lasp uses a membership protocol heavily inspired by HyParView, to compute an overlay network containing all of the members in the cluster. The notable differences between the HyParView protocol and our membership protocol were the results of adapting the theoretical treatment in the HyParView paper to an actual implementation that was used for this experiment.

Specifically, the original HyParView protocol was evaluated in a low-churn environment, whereas our environment has much higher churn. *Churn* is defined as rate of node turnover, i.e., percentage of nodes leaving and being replaced by new nodes, per time unit. The higher churn in our environment was a byproduct of attempting to reduce experimentation time to save costs when operating large clusters: this allowed experiments that would normally take hours for cluster deployment and operations to be reduced to fractional hours at significant cost savings. For details on the modifications to the protocol, the reader is referred to [13].

3.3 Dissemination Protocols

The system supports two data dissemination protocols.

- **State-based.** Objects are locally updated through mutators that inflate the state. Objects are periodically sent to peers that merge the received object with their local state.
- **Delta-based.** Objects are locally updated by merging the state with the result of δ-mutators [1], called deltas, that compactly represent changed portions of state. These deltas are buffered locally and sent to each local peer in every propagation interval.

4 ENGINEERING SCALE

The Lasp semantics ensures that the runtime system is correct in theory for arbitrary distribution of the computation. However, engineering a scalable real-world system requires a significant amount of sophisticated tooling to ensure scalability both for deployment and for observability during execution. Near the end of the SyncFree project, we designed an experiment with the goal of scaling to 10 000 nodes. We finally achieved a scale of 1024 nodes at a total cloud computing cost of about €9000.

4.1 Experiment Configuration

For the purposes of the experiment, we used a total of 70 m3.2xlarge instances in the Amazon EC2 cloud computing environment, within the same region and availability zone. We used the Apache Mesos [7] cluster computing framework to subdivide each of these machines into smaller, fully-isolated machines using cgroups. Each virtual machine, representing a single Lasp node, communicated with other nodes in the cluster using TCP, and given the uniform deployment across all of the allocated instances, had varying latencies to other nodes in the system depending on their physical location.

When subdividing resources for the experiment, we allocated each server task 4 GB of memory with 2 virtual CPUs, and each client task 1 GB of memory, with 0.5 virtual CPUs. Here a *task* is a logical unit of computation that is executed on one virtual machine. We consider that these numbers vastly underrepresent the capabilities of modern mobile devices in widespread deployment today and therefore will lead to conservative results in the evaluation. We allocate more resources to servers, specifically in Datacenter Lasp mode, as servers are required to maintain connections to more nodes in the system; the advertisement counter does not require more resources between Datacenter and Hybrid Gossip modes.

4.2 Experimental Workflow

As running experiments in an unsimulated cloud environment can be challenging due to the inherent nondeterminism across different executions of the same experiment, we created a workflow targeted at reducing nondeterminism by controlling the experiments' setup and teardown procedures with detailed instrumentation for post-experimental analysis. We describe that workflow below.

- **Bootstrapping.** Initially, all of the server and client processes are bootstrapped and joined into a single cluster. The experiment does not begin until we ensure that all of the nodes in the system are connected and the connection graph forms a single connected component. Each node should be reachable by every other node in the system, either directly as a local neighbor, or indirectly via multi-hop. During this process, the system creates advertisement counters and the set of displayable ads.
- **Simulation.** Once we ensure the cluster is connected, each node starts collecting metrics and generating its own workload that randomly selects a counter to increment based on the set of displayable advertisements every predefined impression interval. Periodically, each process propagates local replicas with neighbor processes. It should be noted that each client has its own workload generator: using a centralized harness for running the experiment introduces coordination, which reduces the scalability of the system.
- **Convergence.** As each of the experiments has a controlled number of events that will be generated based on the number of clients participating in the system, the experiment continues to run until each node has observed the effects of all events: we refer to this process as convergence.
- **Metrics aggregation and archival.** Once convergence is reached, the experiment is complete. Each node, upon observing convergence begins uploading metrics recorded during the experiment to a central location: these logs are used for analysis of the runtime system. Once this process is complete, the experiment harness waits for the system to fully teardown the cluster before starting a subsequent run, to prevent state leakage between runs when reusing the same hardware to reduce costs.

4.3 Experimental Infrastructure

Evaluation of a large-scale distributed programming model is difficult. This is due to failures in the underlying frameworks that are used to provide mechanisms for deployment and operations, and because of inadequate tools required to observe the system during execution to ensure it is operating properly.

4.3.1 Apache Mesos. While experimentation shows Lasp scalability to 1024 nodes, we do not believe that this number is a firm upper limit. When attempting to run experiments with 2048 nodes we quickly ran into problems with the Apache Mesos cloud computing framework. One issue is that when attempting to bootstrap a cluster containing 70 instances too quickly, instances become disconnected and need to be manually reprovisioned. This required a slower cluster deployment where a cluster would be scaled from 35 instances, first to 50 instances, and then to 70 instances. As the 2048 experiment required 140 m3.2xlarge instances to operate, cluster deployment would take significantly longer.

When attempting to launch 2048 tasks in Mesos (with a single task representing a single application node), instances would become overloaded quickly and fail to respond to heartbeat messages: this triggered these instances being marked as offline by Mesos and the tasks orphaned. This would require restarting the experiment and reallocating the cluster to account for the lost tasks.

4.3.2 Sprinter. Once tasks were launched by Apache Mesos, we needed a mechanism for client processes to discover other client processes in the system and connect to them.

Therefore, we built an open source service discovery library called Sprinter that was used to fetch a list of running tasks from the Mesos framework, Marathon, and supply them to the system as targets to connect to. Sprinter also performs the following functions:

- **Graph analysis for connectedness.** Each node uploads its local membership view to Amazon S3. The first, lexicographically ordered, server periodically pulls this membership information and builds a local graph that is analyzed to determine if the graph contains all clients, and that the connection graph forms a single connected component.
- **Delay experiment for connectedness.** Based on graph analysis, the experiment's start is delayed until the connection graph forms a single connected component.
- **Periodic reconnection if isolated.** If a node becomes isolated from the cluster, it will rejoin the cluster, using the information provided by Marathon.

To assist in operator debugging of the experiments, a graphical tool was built to visualize the graph information from Sprinter along with extensive logging to the server node with information about cluster conditions.

4.3.3 Partisan. Distributed Erlang has known scalability problems when operated in the range of 50 or more nodes as it tracks full membership information in the cluster at each node and maintains

full connectivity between nodes using a single TCP connection that is used for both data transmission and heartbeat messages. Single connections are problematic because of head-of-line blocking when large messages are transmitted.

We knew that for the experiment to scale we would need: (1) to move away from Distributed Erlang, (2) to configure network topologies for both Datacenter Lasp and Hybrid Gossip Lasp in a single specification, and (3) to specify configurations at runtime without having to modify application code. To do this we built Partisan, an open source Erlang library that provides an alternative communication layer that eschews the use of Distributed Erlang. Partisan supports multiple network configurations and topologies: a client-server star topology, a full connectivity topology mirroring Distributed Erlang's, a static topology where per-node membership is explicitly maintained, and a random unstructured overlay membership protocol inspired by the HyParView membership protocol.

4.3.4 Workflow CRDT (W-CRDT). In our experiments, a central task could not be used to orchestrate the execution: early experiments demonstrated that the central task quickly became a bottleneck and slowed down execution to the speed of the central task. Therefore, we eliminated the central task.

However, without a central task performing orchestration, it becomes more difficult to control when nodes should perform certain actions. For example, after event generation is complete, we should wait for convergence before proceeding to metrics aggregation. Therefore, we needed a mechanism for asynchronously controlling the workflow of the application scenario.

We devised a novel data structure, called the *Workflow-CRDT* (W-CRDT), that is disseminated between nodes for controlling when certain actions should take place. This object is not instrumented by our runtime or included in any of the application logging, to prevent the structure itself from influencing the results of the experiment. The W-CRDT is a sequence of Grow-Only Map CRDTs, where each map is a function from opaque node identifiers to booleans. The sequence is implemented with the recursive Pair CRDT (similar to a recursive list type). The W-CRDT operates as follows:

- **Per node flag.** Each node's portion of a task to be completed is modeled as a flag; each node toggles its flag when it has completed its work.
- **Tasks as grow-only maps.** Each task that needs to be performed is represented by one grow-only map. When all the map's flags are true, the task is considered as complete. This corresponds to a barrier synchronization.
- **Sequential composition of tasks.** Each task can be sequenced with another task. A task starts when its preceding task has completed.
- **Workflow completion.** The workflow is considered complete when all of the tasks that make up the sequential composition are complete.

The W-CRDT is used to model the following sequential workflow in each experiment.

- **Perform event generation.** Once event generation is complete, nodes mark event generation complete.
- **Blocking for convergence.** Once convergence is reached, nodes mark convergence complete.

- **Log aggregation.** Once convergence is reached, nodes begin uploading their logs to a central location and mark log aggregation complete.
- **Shutdown.** Shutdown once log aggregation is complete.

5 EVALUATION

For Datacenter Lasp, we ran experiments using state-based dissemination, with a single server, and 32, 64, 128, 256 clients, forming with the server a star graph topology. For Hybrid Gossip Lasp, we ran experiments using both dissemination strategies, with a single server, and 32, 64, 128, 256, 512, and 1024 clients.

Each experiment was run twice, with the advertisement impression interval fixed at 10 seconds and the propagation interval at 5 seconds. The total number of impressions was configured to ensure that, in all executions, the experiment ran for 30 minutes.

Figure 2 and Figure 3 evaluate three different operational modes for Lasp, examining the state transmission for the duration of the experiment. Two Hybrid Gossip dissemination strategies, state-based and delta-based, are evaluated using a single overlay generated by the HyParView protocol. We also evaluated Datacenter Lasp, where clients propagate changes to the server using a state-based dissemination strategy. We did not evaluate delta-based for Datacenter Lasp, as it is unrealistic to believe that the server could buffer all changes in the system. In this evaluation, we scale up to 256 client processes: this is the largest number of client processes a single server could support in Datacenter Lasp. Hybrid Gossip scaled to 1024 nodes, before we ran into issues with Apache Mesos.

Datacenter Lasp performs the best in terms of state transmission when compared to Hybrid Gossip Lasp using the same dissemination strategy. This results from Datacenter Lasp have no redundancy at all: the star topology has a single point of failure that is used for communication between all nodes in the system. Delta-based dissemination demonstrates a clear advantage for Hybrid Gossip Lasp where redundancy is required to keep the system operating: state transmission can be reduced without sacrificing the fault-tolerance properties of the underlying overlay network. In terms of protocol transmission in Hybrid Gossip Lasp, delta-based dissemination performs better than state-based, even though it is a more complex protocol: in delta-based dissemination a process can track which updates have been seen by its neighbor processes and it will not disseminate an unchanged object, while in state-based dissemination an object is always propagated.

Our experiments confirm several design considerations made in Lasp. First, as demonstrated by the graphs, in the Datacenter Lasp model the transmission cost is reduced as there is no redundancy in messaging and subsequently no fault-tolerance. In this model, because of communication through a datacenter node, an update takes two hops to reach all clients in the system. However, this model has limited scalability because a centralized point, which could be partitioned and replicated across multiple servers, is used as a coordination point for all clients.

Hybrid Gossip Lasp adds additional redundancy by constructing a random overlay network using the HyParView protocol and gossiping state to local peers. This model has additional cost, but provides fault-tolerance through redundancy. In the worst case, an update will be observed by all nodes V after $\log |V|$ propagation
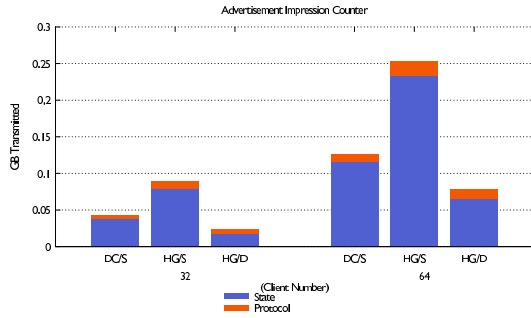
Figure 2: Comparison of state- and delta-based dissemination in both Datacenter and Hybrid Gossip Lasp with 32/64 clients.
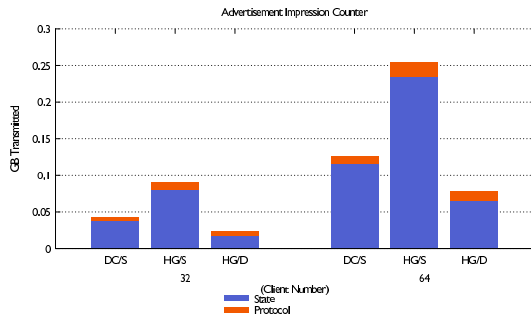

Figure 3: Comparison of state- and delta-based dissemination in both Datacenter and Hybrid Gossip Lasp with Datacenter Lasp ≤ 256 clients (limited in scalability) and Hybrid Gossip Lasp ≤ 1024 clients.

intervals, since in this topology the diameter is logarithmic on the number of nodes.

6 CONCLUSION

Designing new programming models for building large-scale distributed applications requires not only a solid theoretical design, but a well-engineered solution to demonstrate that the system can scale as advertised. Specifically, large-scale evaluations are plagued by the following problems.

- **Existing tooling can be problematic.** Existing infrastructure, frameworks, and languages can be treacherous as they can reduce the scalability of the system because of their design choices.
- **Visualizations are invaluable.** Visualizations assist in debugging the system in real time.
- **Achieving reproducibility is non-trivial.** Clouds provide high-level abstractions over machines, removing visibility into server location and isolation which makes controlled experiments difficult.
- **Performance can fluctuate.** Virtual machine placement and migration, compounded by a language VM layer, are factors that make performance measurement unpredictable.

Cost considerations also limit the statistical smoothing possible by running multiple experiments.

- **Evaluations are expensive.** To provide a real world evaluation, significant funding is required for the infrastructure resources and significant time is required for developing deployment tools and for debugging experiments.

Lasp's scalable design was achieved by taking a holistic approach: both the runtime system and programming model were designed to accommodate one another in a way that allows scalability. However, the effort required to demonstrate Lasp as both scalable and practical remained a non-trivial challenge.

REFERENCES

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2016. Delta State Replicated Data Types. *CoRR* abs/1603.01529 (2016). http://arxiv.org/abs/1603.01529
[2] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. 249–260.
[3] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global sequence protocol: A robust abstraction for replicated shared state. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[4] Christopher S. Meiklejohn. 2017. Lasp Language Documentation. https://lasp-lang.org. (2017).
[5] Christopher S. Meiklejohn. 2017. Lasp Language Source Repository. https://github.com/lasp-lang. (2017).
[6] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 1.
[7] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.
[8] Lindsey Kuper and Ryan R Newton. 2014. Joining forces: toward a unified account of LVars and convergent replicated data types. In *5th Workshop on Determinism and Correctness in Parallel Programming (WoDet 2014)*.
[9] João Leitão, Jose Pereira, and Luìs Rodrigues. 2007. Epidemic broadcast trees. IEEE, 301–310.
[10] João Leitão, José Pereira, and Luìs Rodrigues. 2007. HyParView: A membership protocol for reliable gossip-based broadcast. IEEE, 419–429.
[11] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. ACM, 184–195.
[12] Christopher Meiklejohn and Peter Van Roy. 2015. Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation. In *Reliable Distributed Systems Workshop (SRDSW), 2015 IEEE 34th Symposium on*. IEEE, 62–67.
[13] Christopher S Meiklejohn and Peter Van Roy. 2017. Loquat: A framework for large-scale actor communication on edge networks. In *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*. IEEE, 563–568.
[14] Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. 2016. I now pronounce you reactive and consistent: handling distributed and replicated state in reactive programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems*. ACM, 1–8.
[15] Rodrigo Rodrigues and Peter Druschel. 2010. Peer-to-peer systems. *Commun. ACM* 53, 10 (2010), 72–82.
[16] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of convergent and commutative replicated data types*. Technical Report RR-7506. INRIA.
[17] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981
[18] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. *Managing update conflicts in Bayou, a weakly connected replicated storage system*. Vol. 29. ACM.