Lasp: A Language for Distributed, Eventually Consistent Computations with CRDTs

(Work in progress report)

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

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

Abstract

We propose Lasp, a novel programming model aimed to simplify correct, large-scale, distributed programming. Lasp leverages ideas from distributed dataflow programming extended with convergent data types. This provides support for computations where not all participants are online together at a given moment through Lasp's "convergent by design" applications. Lasp provides a familiar functional programming semantics, built on top of distributed systems infrastructure, targeted at the Erlang runtime system.

The initial Lasp design presented in this report supports synchronization free programming using convergent data types. It combines the expressiveness of these data types together with powerful primitives for composing them. This design lets us write long-lived fault-tolerant distributed applications with non-monotonic behavior. We show how to implement one nontrivial large-scale application, the ad counter scenario from the SyncFree project.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Distributed data structures

Keywords Eventual Consistency, Commutative Operations, Erlang

1. Introduction

Synchronization of data across systems is becoming increasingly expensive and impractical when running at the scale required by "Internet of Things" [12] applications and large online mobile games.[1] Not only does the time required to coordinate with an ever growing number of clients increase with each additional client, but techniques that rely on coordination of shared state, such as Paxos and state-machine replication, grow in complexity with partial replication, dynamic membership, and unreliable networks. [7]

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, 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.

Recently, a formalism has been proposed by Shapiro et al. for supporting deterministic resolution of individual objects that are acted upon concurrently in a distributed system. These data types, referred to as conflict-free replicated data types (CRDTs), provide a property formalized as Strong Eventual Consistency: given all updates to a given object are eventually delivered in a distributed system, all copies of that object will converge to the same state. [13]

While strong eventual consistency is a highly desirable property for a distributed system because operations may arrive at a given replica reordered or duplicated without negatively affecting convergence, it has been demonstrated that the composition of arbitrary CRDTs is non-trivial. [4, 6, 8, 10]

To achieve this goal, we propose a novel programming model aimed at simplifying correct, large-scale, distributed programming, called Lasp[2]. [1] This model provides the ability to use operations from functional programming to deterministically compose CRDTs into larger computations that observe the strong eventual consistency property. This model builds on our previous work, Derflow and Derflow$_L$ [5, 11], a system that provides a distributed, fault-tolerant lattice variable store powering a deterministic concurrency programming model.

We propose the following contributions:

- **Monotonic read:** We provide a `read` operation for CRDTs, which ensures that once a given value is read, all future read operations observe a value causally equivalent or later than the previous read.
- **Functional programming operations:** We provide standard functional programming operations lifted to operate over CRDTs: `map`, `filter`, and `fold`.

[1] Rovio, developer of the popular "Angry Birds" game franchise reported that during the month of December 2012 they had 263 million active users. This does not account for users who play the game on multiple devices, which is an even larger number of devices requiring some form of shared state in the form of statistics, metrics, or leaderboards. [2]

[2] Inspired by LISP's etymology of "LISt Processing", our fundamental data structure is a join-semilattice, hence Lasp.

PaPoC'15, April 21, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3537-9/15/04…$15.00.
http://dx.doi.org/10.1145/2745947.2745954

- **Set-theoretic operations:** We provide set-theoretic operations lifted to operate over CRDTs: `product`, `union`, and `intersection`.
- **Prototype implementation:** We also provide a prototype implementation of Lasp, implemented as an Erlang library using the Riak Core [9] distributed systems framework.

2. Lasp

Lasp operations create processes that connect all replicas of two or more CRDTs.

Each of these processes track the monotonic growth of the internal CRDT state at each replica, and maintain a functional semantics between the state of the input and output instances. The process correctly transforms the internal metadata of the input CRDT to compute the metadata of the output CRDT.[3]

For example, the Lasp `map` operation can be used to connect two instances of the Observed-Remove Set CRDT. [13] The Observed-Remove Set CRDT models arbitrary non-monotonic operations, such as additions and removals of the same element, monotonically, in order to guarantee convergence with concurrent operations at different replicas.[4]

In this example, whenever an element e is added or removed from the input set, the mapped version $f(e)$ is correctly added or removed from the output set. The other operations provided by Lasp are analogous: the user visible behavior is the normal result of the function or set-theoretic operation.

2.1 Semantics

In this report, we provide an example of the semantics of Lasp: the semantics of the `map` operation over the Observed-Remove Set. We focus on the Observed-Remove Set because it is the least-complex CRDT which serves as a general building block for applications.[5]

Each CRDT in Lasp has the appearance of a single CRDT which evolves monotonically over time as update operations are issued. This single CRDT forms a stream of indefinite length, of which a prefix of length n is known, with s' representing future values of the stream. Each value of s is the state of a state-based CRDT.

Definition 2.1 (Streams)**.** A stream is a sequence of infinite length of which only a finite prefix n is known at any given time, while s' represents future values of the stream.

$$s = s_0|s_1|s_2|\ldots|s_{n-1}|s'$$

Definition 2.2 (Observed-Remove Set)**.** The Observed-Remove Set state is a set of triples, where each triple has one value v, with metadata consisting of add set a and remove set r.

$$s_i = \{(v, a, r), (v', a', r'), \ldots\}$$

Additionally, we formalize the `query` function over the Observed-Remove Set, which returns the user-visible value of the data structure, removing all metadata.

Definition 2.3 (Query Operation of an Observed-Remove Set)**.** Presence of a value v in a given Observed-Remove Set, s_i, is determined by comparison of the remove set with the add set. If the remove set is a subset of the add set, the value is in the set.

$$\{v \mid \forall(v, a, r) \in s_i, r \subset a\}$$

Definition 2.4 (Map)**.** The map procedure defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. For each element, the value is separated from the metadata, the function f is applied to the value, and the metadata is attached to the resulting value, $f(v)$.

Algorithm 1 Map algorithm

```
procedure MAP($s, f, t$)
    for all $s_i \in s$ do
        $e \leftarrow \{\}$
        for all $(v, a, r) \in s_i$ do
            $fv \leftarrow f(v)$
            if $\exists a', r'.(fv, a', r') \in e$ then
                $e \leftarrow e \setminus \{(fv, a', r')\} \cup \{(fv, a \cup a', r \cup r'\}$
            else
                $e \leftarrow e \cup \{(fv, a, r)\}$
            end if
        end for
        $t_i \leftarrow e$
    end for
end procedure
```

Figure 1 provides an example of applying the `map` function to an Observed-Remove Set. In this example, the user does not need to program against the internal data structure of each CRDT, only the non-monotonic external representation, as the Lasp runtime handles the metadata mapping automatically.

```
%% Create initial set.
{ok, S1} = lasp:declare(riak_dt_orset),

%% Add elements to initial set and update.
{ok, _} = lasp:update(S1, {add_all, [1,2,3]}, a),

%% Create second set.
{ok, S2} = lasp:declare(riak_dt_orset),

%% Apply map operation between S1 and S2.
{ok, _} = lasp:map(S1, fun(X) -> X * 2 end, S2).
```

Figure 1: Map operation applied to an Observed-Remove Set. In this example, the application developer does not have to program using the internal structure of the CRDT, given the Lasp map operation is lifted to operate over the internal state. We ignore the return values of these functions, given the brevity of the example.

3. Advertisement Counter Example

One of the use cases for our language is supporting clients that need to operate without connectivity. For example, imagine a provider of mobile games that sells advertisement space within their games.

In this example, the correctness criteria is such:

- Clients will go offline: consider mobile devices such as cellular phones that experience periods without connectivity. In the event the client is offline, advertisements should still be able to be displayed to the user.
- Advertisements need to be displayed a minimum number of times, additional impressions, within a certain bound, is not problematic.

[3] The internal metadata of each CRDT is responsible for ensuring correct convergence; this requires that this transformation be deterministic at each replica.

[4] It is paramount that the metadata transformation is performed correctly, or replicas of the `map` operation will not converge correctly.

[5] For instance, the Grow-Only Set does not allow removals, the Two-Phase Set only allows one removal of a given item, and the Observed-Remove Set without tombstones adds additional complexity in the form of optimizations, which lie outside of the core language semantics.

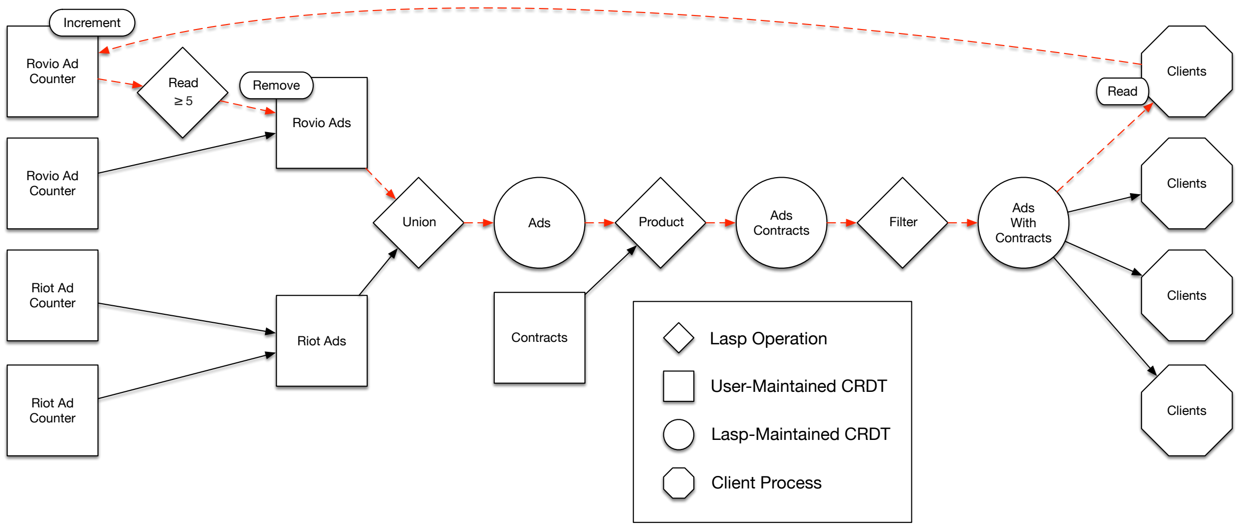

Figure 2: Eventually consistent advertisement counter. In this example, the dotted line represents the monotonic flow of information for one counter.

Figure 2 visualizes an eventually consistent advertisement counter written in Lasp. In this example, squares represent primitive CRDTs, where circles represent CRDTs that are maintained through composition using Lasp operations. Additionally, Lasp operations are represented as diamonds, and edges represent the monotonic flow of information in the Lasp application.

Our advertisement counter operates as follows:

- Advertisement counters are grouped by vendor.
- All advertisement groups area combined into one list of advertisements using a `union` operation.
- Advertisements are joined with active "contracts" into a list of displayable advertisements using both the `product` and `filter` operations.
- Each client reads the list of active advertisements when displaying an advertisement.
- For each advertisement displayed, each client updates the associated advertisement counter.
- As a counter hits five advertisement impressions, the advertisement is "disabled" by removing it from the list of advertisements.

The implementation of this advertisement counter is completely monotonic and synchronization-free. Adding and removing ads, adding and removing contracts, and disabling ads when their contractual number of views is achieved are all modeled as the monotonic growth of state in CRDTs connected by active processes. Programmer-visible non-monotonicity is represented by monotonic metadata in the CRDTs. The initial Lasp design, which supports only programming with zero synchronization and optimistic replication, has sufficient functionality to model this application.

4. Conclusion and Future Work

We introduced the Lasp programming model and motivated its use for large-scale computation over replicated data. Our future plans for Lasp include identifying optimizations for more efficient state propagation, exploring stronger consistency models, and optimizing distribution, and replica placement for better fault-tolerance and reduced latency in computations. Our ultimate goal is for Lasp to become a general purpose language for building large-scale distributed applications in which synchronization is used as little as possible.

A. Code Availability

All of the code discussed will be available on GitHub under the Apache 2.0 License at `http://github.com/cmeiklejohn/lasp`.

Acknowledgments

Thanks to Peter Alvaro, Sean Cribbs, Scott Lystig Fritchie, and Andrew Stone for their valuable feedback. The research leading to these results has been partially funded by the SyncFree Project [3] in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^{o} 609551.

References

[1] Lasp source code repository. `https://github.com/cmeiklejohn/lasp`. Accessed: 2015-02-14.

[2] 263 Million Monthly Active Users In December. `http://www.rovio.com/en/news/blog/261/263-million-monthly-active-users-in-december/`. Accessed: 2015-02-13.

[3] SyncFree: Large-scale computation without synchronisation. `https://syncfree.lip6.fr`. Accessed: 2015-02-13.

[4] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 23. ACM, 2013.

[5] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM, 2014.

[6] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak DT map: a composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 1. ACM, 2014.

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[8] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[9] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[10] C. Meiklejohn. On the composability of the Riak DT map: expanding from embedded to multi-key structures. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 13. ACM, 2014.

[11] C. Meiklejohn. Eventual Consistency and Deterministic Dataflow Programming. *8th Workshop on Large-Scale Distributed Systems and Middleware*, 2014.

[12] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[13] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, et al. A comprehensive study of convergent and commutative replicated data types. 2011.