Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation

Christopher Meiklejohn
Basho Technologies, Inc.
Bellevue, WA
Email: cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium
Email: peter.vanroy@uclouvain.be

***Abstract*—We present a new programming model for large-scale mobile and "Internet of Things" style distributed applications. The model consists of two layers: a language layer based on the Lasp language with a runtime layer based on epidemic broadcast. The Lasp layer provides deterministic coordination-free computation primitives based on conflict-free replicated data types (CRDTs). The epidemic broadcast layer is based on the Plumtree protocol. It provides a communication framework where clients may only have a partial view of membership and may not want to participate in or have knowledge of all active computations. We motivate the new model with a nontrivial mobile application, a distributed ad counter, and we give the model's formal semantics.**

I. Introduction

Traditional approaches to synchronization increasingly have problems when clients become geographically distributed and more numerous. Specifically, they do not operate within the acceptable latency requirements of most consumer-facing applications.[1] This problem is further complicated by the recent addition of two new classes of large-scale Internet applications: "Internet of Things" sensor networks and mobile applications. "Internet of Things" sensor networks rely on tiered aggregation networks that leverage devices with limited connectivity, limited power, and limited local storage capacity. Mobile applications usually operate with replicated state and allow offline modifications to this state, placing the onus on the application developer to resolve concurrent modifications to replicated data items.

In previous work, we have proposed a solution to the problem of large-scale, coordination-free programming, namely the Lasp programming model [2], [3]. Lasp uses functional programming operations to deterministically compose conflict-free replicated data types (CRDTs) [4]. CRDTs are guaranteed to converge under concurrent operations to replicated state. The composition of CRDTs into larger applications preserves these convergence properties. This means that applications can make progress while offline, propagating their state upstream as connectivity becomes available, and are resilient to both re-ordering and replay of messages. We have implemented the Lasp programming model on a consistent-hashed ring in a datacenter (see Section II-C). However, this architecture is inappropriate for edge computing because ring management is increasingly difficult for growing numbers of limited nodes with intermittent connectivity.

[1]For example, Amazon estimated that every 100ms in latency resulted in a 1% sales loss [1].

This paper proposes a new programming model for edge computing applications such as Internet of Things and mobile applications. The model combines an execution layer based on Lasp with a gossip layer based on epidemic broadcast. These two layers work well together: gossip is adapted to loosely coupled systems and Lasp is adapted to the properties of the gossip layer. Using gossip provides improved placement of application state and computations with that state across a large and variable set of nodes. Using Lasp provides an inherent ability to continue computing despite frequent node disconnections, node failures, and message reordering. The gossip layer is based on previous work on epidemic broadcast trees [5], which provides the efficient and reliable delivery of messages to clusters containing large and dynamically variable numbers of nodes. This paper presents and motivates the programming model and gives a formal semantics of its execution. We are currently implementing and evaluating the model. To our knowledge, this paper is the first to articulate a general purpose programming model using epidemic broadcast as the basis for the language's runtime.

The paper is structured as follows. Section II introduces the concepts we build on: CRDTs, Lasp and its ring-based implementation, and epidemic broadcast trees. Section III gives a motivating example, namely a distributed ad counter for mobile applications. Section IV presents Selective Hearing, which combines the Lasp execution model with a new distribution model based on gossip. Section V gives the formal semantics that defines Lasp execution on the gossip layer. Section VI relates our new model with other models of group management and execution. Section VII concludes and explains how we intend to continue this work.

II. Background

In this section we review Conflict-free Replicated Data Types, Lasp, and Epidemic Broadcast Trees.

A. Conflict-free Replicated Data Types (CRDTs)

CRDTs are data structures designed for use in replicated, distributed computations. They come in a variety of flavors, such as maps, sets, counters, registers, and flags, and they provide a programming interface that is similar to their sequential counterparts. They are designed to capture concurrency properly: for example, by guaranteeing deterministic convergence after concurrent additions of the same element at two different replicas of a replicated set.

One variant of these data structures is formalized in terms of bounded join-semilattices. Regardless of the type of mutation performed on these data structures and whether that function results in a change that is externally non-monotonic, state is always monotonically increasing and two states are always *join*-able via a binary operation that computes a *supremum*, or least upper bound. To provide an example, adding to a set is always monotonic, but removing an element from a set is non-monotonic. CRDT-based sets, such as the Observed-Remove Set (OR-Set) used in our example, model non-monotonic operations, such as the removal of an item from a set, in a monotonic manner. To properly capture concurrent operations that occur at different replicas of the same objet, individual operations, as well as the actors that generate those operations, must be uniquely identified in the state.

The combination of monotonically advancing state, in addition to ensuring that replicas can converge via a deterministic merge operation, provides a strong convergence property: with a deterministic replica-to-replica communication protocol that guarantees that all updates are eventually seen by all replicas, multiple replicas of the same object are guaranteed to deterministically converge to the same value. Shapiro et al. have formalized this property as Strong Eventual Consistency (SEC) in [4].

To demonstrate this property, we look at an example. In this example, a small circle represents an operation at a given replica and a dotted line represents a message sharing that state with another replica, where it is merged in with its current state.

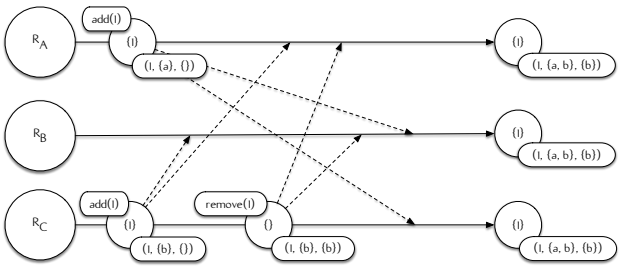

Figure 1: *Example of resolving concurrent operations with an Observed-Remove Set (OR-Set). In this example, concurrent operations are represented via unique identifiers at each replica.*

Figure 1 is an example of the Observed-Remove Set (OR-Set) CRDT. This set uses unique identifiers derived at each replica and represents state at each replica as a triple of values, a set of unique identifiers for each element addition and a set of unique identifiers for each element removal. When removing an element, removals remove all of the "observed" additions, so under concurrent additions and removals, the set biases towards additions.

B. Lasp

Lasp is a programming model that uses CRDTs as its primary data type [2], [3]. Lasp allows programmers to build applications using CRDTs while ensuring that the composition of the CRDTs also observes the same strong convergence properties (SEC) as the individual objects do. Lasp provides this by ensuring that the monotonic state of each object maintains a homomorphism with the program state.[2]

The relevant contribution of the Lasp programming model is the **process**. In Lasp, processes are used to connect two or more instances of CRDTs. One example of a Lasp process is the *filter* operation over sets: as the input set is mutated, the filter function is reevaluated, resulting in a new value for the output. Lasp processes ensure this transformation is both monotonic and deterministic.

C. Ring-Based Distribution Model for Lasp

The Lasp programming model was initially designed and implemented in Erlang [6] using the Riak Core distributed systems library. The Riak Core library provides a framework for building applications in the style of the original Dynamo system as described by DeCandia et al. in 2007 [1]. Riak Core provides library functions for cluster management, dynamic membership, failure detection and state management.

This Lasp implementation uses Dynamo-style partitioning of application state and computations: consistent hashing and hash-space partitioning are used to distribute copies of each variable and Lasp process across nodes in a cluster to ensure high availability and fault tolerance. Replication of each variable's state, and the instantiation of Lasp processes, are performed between adjacent nodes in a cluster and quorum-based operations are used to read and modify variables and the result of computations in the system. Additionally, an anti-entropy protocol is deployed alongside the quorum-based operations to ensure reliable delivery of all messages in the system.

While this model of distribution is designed for fault-tolerance and high-availability, it is inherently skewed towards clusters where both the work and latency distribution across the cluster is uniform.

D. Epidemic Broadcast Trees

Epidemic Broadcast Trees [5], or more specifically the Plumtree protocol, is an efficient, **reliable broadcast** protocol. This approach combines techniques from two previous approaches to reliable broadcast: deterministic tree-based broadcast protocols that have low complexity in message size and are therefore less fault-tolerant, and gossip protocols that have higher complexity in message size but are tolerant to faults.

To achieve efficient and fault-tolerant reliable broadcast, the protocol implements a hybrid approach for each message that is composed of two phases: given a unique identifier for the message, first push the message identifier and payload to nodes contained by the leaves of the broadcast tree, known as the **eager push** phase; then, push only the message identifier to a random sampling of other nodes known by the peer service, known as the **lazy push** phase. If any of the nodes do not receive a message they have learned about through the **lazy push** phase within a designated timeout period, they request this message by identifier from a randomly picked peer in the overlay network.

[2]For more information about how this transformation is performed and maintained, the reader is referred to [2], [3] and [6].

The Plumtree protocol starts off with a random sampling of nodes, selected from a peer service, placed in the **eager** set. As the protocol evolves, nodes are moved from the **eager** set to the **lazy** set as duplicate messages are received. This process of pruning the **eager** set is how the protocol computes the spanning tree that will be used for the eager phase of message broadcast.

III. Motivating Example

We now present an application scenario to motivate our programming model. Figure 2 visualizes an eventually consistent advertisement counter written in Lasp as originally presented in [2], [3]. In this example, shaded circles represent primitive CRDTs and white circles represent CRDTs that are maintained through composition using Lasp operations. Additionally, Lasp operations are represented as diamonds, and directed edges represent the monotonic flow of information in the Lasp application.

Our advertisement counter operates as follows:

- Advertisement counters are grouped by vendor and 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 periodically reads the list of active advertisements from the server and stores a copy locally. When displaying an advertisement, clients choose an advertisement from this local list and increment the counter. This allows clients to make progress while offline, but still correctly capture the number of advertisement impressions.
- Clients periodically synchronize their counters with the server. As a counter hits 50,000 advertisement impressions, the advertisement is "disabled" by removing it from the list of active advertisements.

A. Selection of Consistency Protocol

Our original Lasp design focused on the replication of all objects across a hash-space partitioned ring using consistent hashing. This design is problematic for the advertisement counter. In the advertisement counter, not all objects may want to adhere to the same consistency protocol. We examine two cases in Figure 2.

Advertisement Counters Each advertisement counter stored at the client is a local copy that is mutated when advertisements are viewed locally. In this case, we do not want to replicate this object at the client or across other clients; we may want to only store a single copy of this counter and periodically synchronize with the server accepting that any impressions between synchronization periods may be lost. In this example fault-tolerance is provided through periodic synchronization periods with the server.

Advertisement Transformations At the server, the transformation using **union**, **product**, **filter** is performed on a weakly consistent store with quorum-based operations. In this case, an administrator may want to use state-machine replication techniques via a system like Zookeeper, or a traditional RDMBS, such as PostgreSQL to store this information.

Given the desired flexibility, we propose that single nodes in the gossip model of this paper can themselves be implemented either as Dynamo-style rings with quorum operations, state-machine replication, or a single register.

IV. Selective Hearing

We now present our new distribution model for Lasp, called Selective Hearing. This algorithm supports large-scale computation with Lasp where clients can incrementally contribute results to computations and selectively receive results of computations as needed. The semantics for this algorithm removes the notion of "replicas" as previously presented in [2], [3] and reasons about a single copy of a data item. This "single copy" is an abstraction over the consistency protocol used to maintain it; for example, it may be maintained with state-machine replication, quorum-based operations on a weakly consistent store, or operations on a single register.

The system model consists of a set of nodes supporting Lasp operations that are implemented using epidemic broadcast. Each node is uniquely identified and tracks a monotonic counter that is incremented with each operation. Nodes can join or leave at any time. Nodes fail by crashing and all messages in the system are eventually delivered to all correct nodes by the epidemic broadcast protocol (reliable broadcast). Crashed nodes disappear from the system; whenever a node recovers it chooses a new identifier and reinitializes its monotonic counter at zero. We can therefore summarize the system model as two layers:

A. Lasp Layer

Each node can initiate one of the following operations:

- $declare(t)$: Given type information t, return a new unique variable identifier i that contains the type information and broadcasts this identifier to all nodes. We do not specify in more detail how t is encoded.
- $read(i, p, c)$: Test whether variable i's value v satisfies predicate $p(v)$. If so, call the continuation $c(v)$. If not, add (p, c) to the interest set for variable i. This information will be used by the $bind$ operation when the variable is bound.
- $bind(i, v)$: Broadcast value v, which is then merged with the existing value of variable i on all nodes that have an interest set for i. For these nodes, test all predicates and invoke the corresponding continuation for each predicate that succeeds.

Both predicate p and continuation c have one argument v. Any standard semantics for predicates (i.e., boolean functions) and continuations may be used; for brevity we do not give these semantics since they are not a contribution of this paper. Note that these three operations are designed to commute pairwise for any variable i (see Section V-C). This is an essential property since the gossip layer cannot guarantee delivery order.

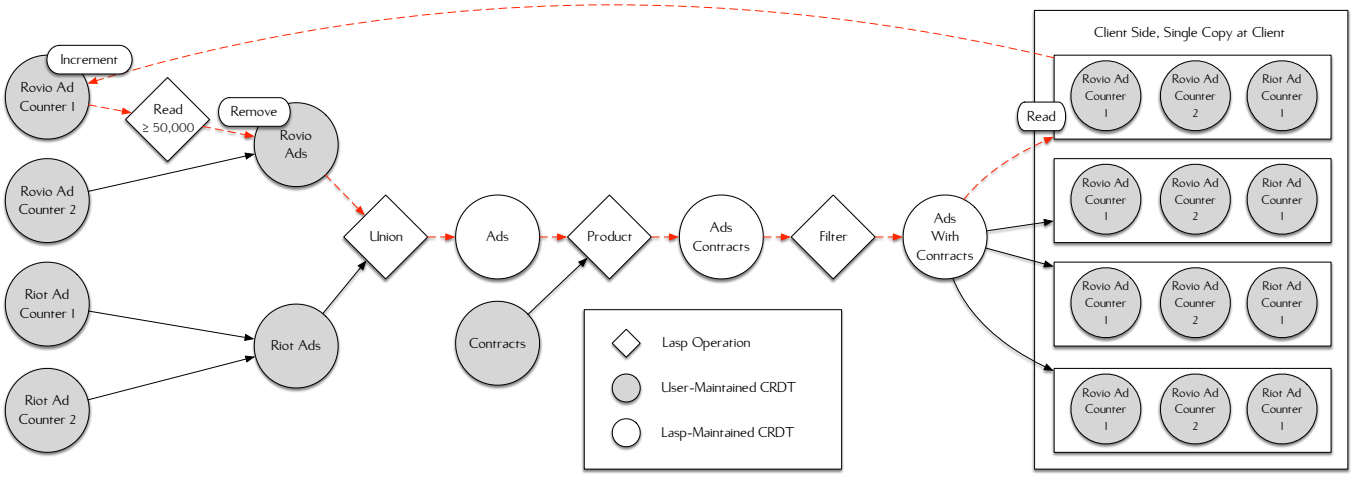

Figure 2: *Eventually consistent advertisement counter as presented in [2], [3]. The dotted line represents the monotonic flow of information for one update. In this example, clients contain only their portion of the shared state they are modifying. Only one copy of the partial counters on the client is stored, it is not replicated in the cluster.*

B. Gossip Layer

The gossip layer implements the Plumtree epidemic broadcast protocol. It efficiently implements the broadcast operations required by the Lasp layer. The broadcast operations are not ordered, i.e., a node may receive broadcasts in any order and different nodes may receive them in different orders. Because of the Strong Eventual Consistency property of CRDTs, this does not affect correctness. Furthermore, this allows an important optimization that reduces the computations needed to implement the bind operation.

Bind operations initiated on each node are numbered consecutively via a node-level monotonic counter. Since each variable's successive values are inflations of a lattice, a bind operation that is delivered on a node does not have to invoke local computation if another bind with a greater value has already been delivered.

The Plumtree protocol relies on three properties for fault-tolerant message delivery: (1) Each message can be uniquely identified: given the lazy push phase of the protocol broadcasts only message identifiers, a node is required to know whether that message has been received or not. (2) Nodes must store a history of all messages received. (3) When receiving an identifier for a message, a node must be able to determine if it has already been subsumed by a previous one.

To meet these requirements, we maintain a monotonic clock at each node and store a version vector for each CRDT. This version vector is used to uniquely identify the message when broadcast and allows us to identify messages that have been subsumed by other messages without comparison of payload. By leveraging a per object version vector that is incremented as mutations are performed to each object, we can store a history of all messages received with vector as wide as the number of participating actors in the system.

V. Semantics

We give the formal semantics of the operations in the Lasp layer in terms of the gossip layer's broadcast operation.

A. Node State

The system consists of a set of nodes, where the state of each node is a three-tuple $(\sigma, \delta_i, \delta_v)$. Here, σ is the known variables set, δ_i is the interest set, and δ_v is the known values set. The execution of each node is a sequence of states:

$$(\sigma^{(0)}, \delta_i^{(0)}, \delta_v^{(0)}) = (\{\}, \{\}, \{\}) \to \cdots \to (\sigma^{(k)}, \delta_i^{(k)}, \delta_v^{(k)}) \to \cdots \tag{1}$$

The sets are initially empty; the k-th state is denoted by superscript (k). We now define the content of each set.

The *known variables set* σ contains the unique variable identifiers known at the node:

$$\sigma = \{i_0, i_1, \ldots\} \tag{2}$$

The *interest set* δ_i contains information about the variables that the node is interested in, i.e., for which a read operation has been invoked but not yet resolved by the arrival of a new value that satisfies the read predicate. For each variable, the set contains the variable identifier i and a set of pairs of a one-argument predicate p and a one-argument continuation c. When the node receives a new value, then each predicate is evaluated, and for those that succeed the continuation is invoked.

$$\delta_i = \{(i_0, \{(p_0, c_0), \ldots\}),\ (i_1, \{(p_1, c_1), \ldots\}), \ldots\} \tag{3}$$

The *known values set* δ_v contains a set of pairs (i, v) of variable identifiers i and their highest values v observed on the node:

$$\delta_v = \{(i_0, v_0), (i_1, v_1), \ldots\} \tag{4}$$

B. Basic Invariants on Node State

We assume that node states obey the following invariants:

- The known variables set σ and the known values set δ_v both grow monotonically: variables will only be added and never removed to both sets.

$$\sigma^{(k)} \subseteq \sigma^{(k+1)} \qquad \delta_v^{(k)} \subseteq \delta_v^{(k+1)} \tag{5}$$

- For any interest set δ_i and known values set δ_v, their identifiers will be contained in some future state of σ[3].

$$\begin{aligned} &\forall k.\ \exists n.\ n > k \Rightarrow \pi_i(\delta_i^{(k)}) \subseteq \sigma^{(n)} \\ &\forall k.\ \exists n.\ n > k \Rightarrow \pi_i(\delta_v^{(k)}) \subseteq \sigma^{(n)} \end{aligned} \tag{6}$$

C. Operations

All Lasp operations are initiated on one node and may have effects on all nodes; we denote the initiating node by a subscript k. We specify what each operation does on a node state $(\sigma, \delta_i, \delta_v)$ to compute the subsequent state $(\sigma', \delta_i', \delta_v')$; any set that is not mentioned does not change value. In addition to local operations, some operations do a broadcast using the gossip layer; we assume the broadcast message is delivered to *all* nodes including the sending node. We specify what each receiving operation does.

declare The operation $i = declare(t)$ returns a new unique variable identifier i. The operation has the following local specification:

$$i = declare_k(t) : u = unique() \wedge i = (u, t) \tag{7}$$

The variable identifier is a pair of a unique constant u and type information t. The operation then broadcasts the variable identifier with the following specification (the notation $declare_k^j(i)$ means that node k broadcasts to node j). This adds the variable identifier to the known variables σ:

$$declare_k^j(i) : \sigma' = \sigma \cup \{i\} \tag{8}$$

read The operation $read(i, p, c)$ tests whether $p(v)$ holds for the current value v of variable i. If it does, the continuation is invoked as $c(v)$. If not, (p, c) is added to the interest set of variable i, which will delay the invocation until a binding arrives that sufficiently increases i's value.

$$\begin{aligned} read_k(i, p, c) : &(\exists v.(i, v) \in \delta_v \wedge p(v) \Rightarrow c(v) \\ &;\ (\exists s.(i, s) \in \delta_i \Rightarrow \\ &\quad s_n = s \cup \{(p, c)\}; s_n = \{(p, c)\}) \\ &\delta_i' = \delta_i \setminus \{(i, \_)\} \cup \{(i, s_n)\} \\ &\sigma' = \sigma \cup \{i\}) \end{aligned} \tag{9}$$

bind The operation $bind(i, v)$ updates the current value stored in δ_v by doing a join with v. The operation has the following local specification:

$$bind_k(i, v) : \text{true} \tag{10}$$

The operation is then broadcast and has the following specification on all other nodes:

$$\begin{aligned} bind_k^j(i, v) : &(\exists v'.(i, v') \in \delta_v \Rightarrow v_n = v \sqcup v'; v_n = v) \\ &\delta_v' = \delta_v \setminus \{(i, \_)\} \cup \{(i, v_n)\} \\ &\sigma' = \sigma \cup \{i\} \\ &\exists s.(i, s) \in \delta_i \Rightarrow \\ &\quad s_{sat} = \{(p, \_) \in s \mid p(v_n)\} \\ &\quad \delta_i' = \delta_i \setminus \{(i, \_)\} \cup \{(i, s \setminus s_{sat})\} \\ &\quad \forall (\_, c) \in s_{sat} : c(v_n) \end{aligned} \tag{11}$$

[3]We define π as the standard projection.

In variable i's interest set, all pending read operations (all pairs (p, c) in s) are checked. Those for which p succeeds are removed from s and their continuation c is invoked.

D. Processes

A Lasp process is defined as a recursive function that uses the Lasp operations. Figure 3 shows the execution of a Lasp process running the *filter* operation. The example runs on four nodes, (1), (2), (3), and (4).

Subfigure (a) is a Lasp program that creates two instances of the Grow-Only Set (G-Set) CRDT and applies the filter operation with predicate $\lambda x.odd(x)$ from A to B. This Lasp program has four instructions, each of which executes on a different node. The executing node is written to the left of each instruction.

Subfigure (b) is a Lasp program that defines the *filter* operation: a recursive function that repeatedly reads new values of A and computes new values of B. Each iteration executes a read on A that waits for predicate P to be satisfied, at which time the continuation C is executed.

Subfigure (c) shows the operations executed and the state at each node.

This example executes as follows:

1) The $declare_1$ operation is executed on node 1 which locally generates the unique identifier A. This operation results in a $declare_1^j(A)$ message broadcast to all members of the cluster.
2) The $declare_2$ operation is executed on node 2 which locally generates the unique identifier B. This operation results in a $declare_2^j(B)$ message broadcast to all members of the cluster.
3) The *filter* operation is executed on node 3. This operation results in a $read_3(A, P, C)$.
4) The $bind_4(A, \{1, 2, 3\})$ operation is issued on node 4. This operation results in a $bind_4^j(A, \{1, 2, 3\})$ message broadcast to all members of the cluster; however, only node 3 is waiting for a value of A. Given the predicate is satisfied, the continuation is invoked on node 3, trigging a local $bind_3(B, \{1, 3\})$ operation and a broadcast of a $bind_3^j(B, \{1, 3\})$ message. Given no nodes are waiting for a value of B, the message is not processed.

VI. Related Work

We examine related work in programming languages and sensor networks.

A. Process Groups and Programming Languages

The earliest known use of a publish/subscribe model was by Birman and Thomas in 1987 with the ISIS Toolkit [7]. In ISIS, process groups were used to handle the replication of an object using reliable broadcast.

The Quicksilver system described by Ostrowski et al. [8] presents the design of a "live" distributed objects system that has pluggable communication substrates. This system places the onus on the communication layer for managing

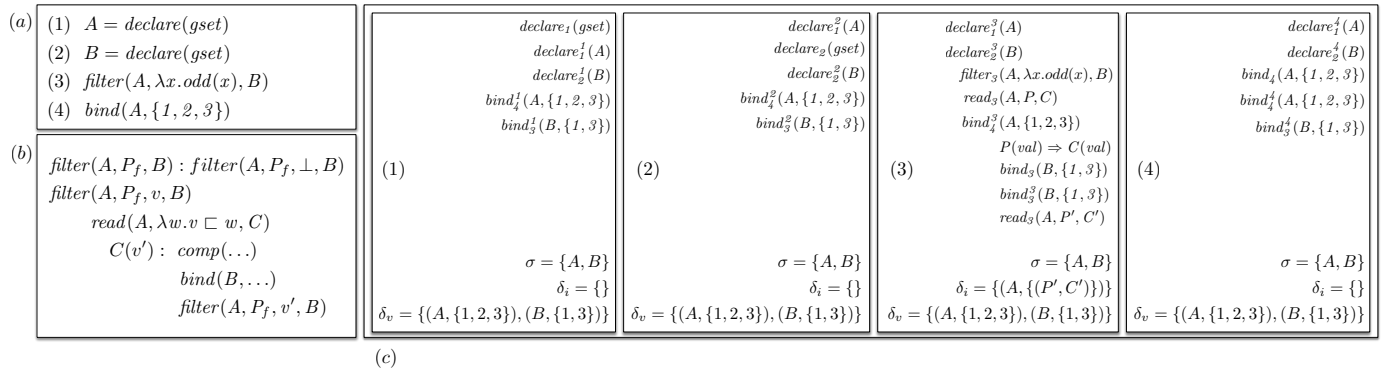

Figure 3: *Execution of a Lasp process over gossip with four nodes. Subfigure (a) shows an example program with the nodes selected for the execution of each Lasp operation; subfigure (b) shows the definition of a Lasp filter process; subfigure (c) shows where each operation executes and where each broadcast message arrives along with the final state at each node at the end of the execution.*

consistency, replication, and propagation of events. While a gossip protocol version is mentioned as future work, no further details of how it would manage object state are provided.

B. Directed Diffusion and Digest Diffusion

Directed diffusion [9] is an efficient protocol for performing realtime dissemination of "interests", metadata describing information that should be collected, and "samples", collections of information coming from the sensors in the network.

Directed diffusion shares many common traits with Selective Hearing: a publish/subscribe paradigm, use of a broadcast protocol, and a API that supports aggregation of information. However, directed diffusion's primary focus is on capturing immutable samples in a large-scale sensor network, whereas Selective Hearing focuses on general computations over shared state using a declarative, functional programming approach.

Digest diffusion [10] presents a energy-efficient model of computation, where operations that are idempotent can be decomposed and distributed in a network so that nodes will converge to a correct value. This work outlines the problems of performing operations that are not idempotent, but can be decomposed: for example, even under ideal network conditions a *count* operation can exhibit anomalies from message duplication.

Our use of the Lasp programming model yields computations that can be decomposed and distributed by design; all of the variables in the language are CRDTs: data structures designed to be resilident to replay and re-ordering of messages.

VII. Conclusion

This paper presents a new distribution model for Lasp based on combining a Lasp execution layer with an epidemic broadcast communication layer. Since the Lasp semantics obeys strong eventual consistency, it can use a communication layer that does not provide ordering guarantees. The model is designed to operate with two new classes of applications: mobile gaming with shared state, and "Internet of Things" style applications. We believe this is the first programming model to use an epidemic broadcast protocol as a core component of its runtime system. We are in the process of implementing this model and evaluating it. We plan to continue improving the design and distribution of Lasp by modeling several industrial applications, including the industrial use cases of SyncFree partner Rovio Entertainment, a mobile gaming provider [11].

Acknowledgment

Thanks to Larry Marburger and our anonymous reviewers for their feedback. This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement nº 609551.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[2] C. Meiklejohn and P. Van Roy, "Lasp: a language for distributed, eventually consistent computations with CRDTs," in *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, 2015, p. 7.

[3] ——, "Lasp: A language for distributed, coordination-free programming," in *Proceedings of the 17th Symposium on Principles and Practice of Declarative Programming (PPDP 2015)*. ACM, Jul. 2015.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," INRIA, Tech. Rep. RR-7506, 01 2011.

[5] J. Leitao, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," in *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, 2007, pp. 301–310.

[6] "Lasp source code repository," https://github.com/lasp-lang/lasp, accessed: 2015-06-14.

[7] K. Birman and T. Joseph, *Exploiting virtual synchrony in distributed systems*. ACM, 1987, vol. 21, no. 5.

[8] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn, "Programming with live distributed objects," in *ECOOP 2008–Object-Oriented Programming*. Springer, 2008, pp. 463–489.

[9] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM, 2000, pp. 56–67.

[10] J. Zhao, R. Govindan, and D. Estrin, "Computing aggregates for monitoring wireless sensor networks," in *Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*. IEEE, 2003, pp. 139–148.

[11] "SyncFree: Large-scale computation without synchronisation," https://syncfree.lip6.fr, accessed: 2015-02-13.