Declarative, Sliding Window Aggregations for Computations at the Edge

Christopher Meiklejohn
Machine Zone, Inc.
Palo Alto, CA
Email: cmeiklejohn@machinezone.com

Seyed H. Haeri (Hossein), Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium
Email: {hossein.haeri, peter.vanroy}@uclouvain.be

***Abstract*—We present a work in progress report on a new programming model that supports declarative, functional style aggregation operations over devices at the edge. This programming model bridges the gap between the two competing approaches for large-scale aggregations, streaming all data back to a central coordinator versus designing an optimized, distributed algorithm, by leveraging convergent data structures, dynamic scoping, and a declarative functional semantics implemented by a distributed runtime. We motivate our design with an industrial application susceptible to message reordering and arbitrary message delays on an unreliable network.**

I. Introduction

Large-scale computation continues to increase in importance as the reach of mobile Internet applications grows and more devices become connected to the Internet. This is especially true with the growth in popularity of "Internet of Things"-style applications, specifically large-scale sensor networks that are widely deployed, with both limited connectivity and limited power.

Supporting these types of applications highlights two challenges: a technique for reliably distributing computations across these devices, and a technique for reliably collecting the results of these computations at an aggregation point.

We motivate our work with the following industrial example [1], as depicted in Figure 1. We assume a sensor network, composed of sensors that record events at a given interval. The sensors are sufficiently powerful: each contains persistent storage with a processor and working memory capable of running a programming language runtime system.

Now, consider the case where the operator of this sensor network wants to compute the network-wide aggregate over a sliding window. Historically, solving this problem has been done in two ways, in both academia and industry.

A. Previous Approaches

The first approach relies on the central aggregation of edge-generated data, typically performed at a data center of an application service provider (ASP) [2]. This allows existing off-the-shelf stream processing systems to be used to operate over the data, at any time after the data has been collected [3].

This approach makes three assumptions that may not necessarily to hold with the growth of edge-generated data. First, this approach assumes that ASP's will have the ability to store all edge-generated data. Second, this approach assumes that clients will have enough available storage to buffer events during periods without connectivity. Finally, this approach assumes that clients have enough power on their devices to support the transmission of this data.

The second approach typically involves choosing a distributed algorithm for data aggregation and adapting an arbitrary computation to fit this algorithm. One example of this approach is *digest diffusion* [4]. This algorithm combines the efficient *directed diffusion* distribution model with an approach to computing monotonic aggregates. The major drawback with this approach is that the author must ensure that the original safety and liveness guarantees of each algorithm are preserved under this composition.

Returning to our example in Section I, we have two possible ways to approach the problem:

1) Stream all computation to a central data center while sensors are online; this requires unbounded storage for indefinite network partitions and enough energy to continuously transmit samples on the network, but provides the most generality in program design.
2) Design an efficient algorithm for aggregating information throughout the network for the given computation being run; this requires designing an efficient way of propagating results for the type of computation being performed, but limits program reuse.

Given this, it is desirable to design computations that can do a majority of the processing at the edge, with an efficient way of distributing the results of the computation, in a manner where out-of-order delivery, and delayed messages do not cause anomalies in the computation.

We depict the ideal design in Figure 1. In this example, each sensor maintains a sliding window of events referred to as its samples and a fold operation is used to fold the samples into a local average using the binary operation "avg"[1].

B. A New Hope

Is it possible to combine these approaches where developers can write applications that operate on the entire data stream, but take advantage of the optimized, fault-tolerant algorithms for aggregating the results of these computations?

[1] In this example, we focus on the binary relation "avg", however, we believe that our solution is generic enough for any binary operation that is associative, commutative and invertible.

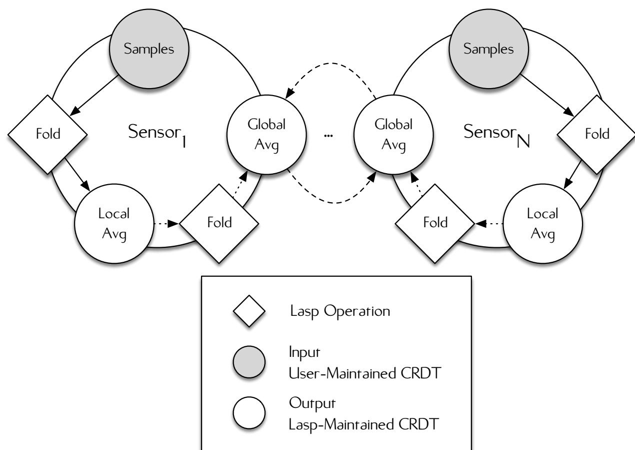

Figure 1: *Simplified diagram of the code presented in Figure 3. This application computes both local and global averages from a set of samples recorded by sensors in a sensor network.*

We believe so. We propose a system where distributed computations can be written using functional programming techniques, but using conflict-free replicated data types (CRDTs) that ensure that computations converge to the correct result; this happens regardless of delays in propagation, out-of-order message delivery, and limited connectivity.

In our previous work, we proposed a solution to the problem of large-scale distributed programming without coordination, named Lasp [5]. Lasp uses functional programming techniques to deterministically compose conflict-free replicated data types (CRDTs), which model sequential data structures that when distributed, guarantee convergence under concurrent mutations. This gives applications developed in Lasp a strong convergence property: given replicated state that is concurrently edited and eventually communicated to every node in a distributed system, regardless of ordering, distributed applications will converge to the correct result.

We subsequently extended Lasp by introducing a distributed, epidemic-based distribution model, named *Selective Hearing* [6]. This distribution model uses an efficient epidemic broadcast protocol [7] to support a large number of nodes, with out-of-order message delivery and pairwise repair between nodes. Lasp exploits this distribution model for all application state, given that its use of CRDTs guarantees convergence under message replay and reordering, fundamental properties of unreliable, asynchronous networks.

However, the initial version of this model introduced in [6] needs to be extended: it assumes that all application state will be used by, and should be shared by, all nodes in the system. This is inefficient in our example application for two reasons.

- Given that all state is shared by all nodes in the system, any intermediate state or computations performed at nodes that does not need to be shared, will be.
- Given that state created at each node is considered unique, we require a mechanism for explicitly identifying state that should be merged across replicas.

C. Contributions

We propose a programming model and implementation for aggregate computations on a dynamic edge network that supports out-of-order delivery, network churn, and limited connectivity, all while looking to the programmer like a declarative functional language that supports the programming and reasoning techniques of functional programming.

The specific contributions of this paper are as follows:

- We introduce a "dynamic" declare operation. This version of the declare operation creates a dynamically scoped variable that is globally known, but has an independent value per node.
- We introduce a "dynamic" fold operation. This fold operation reduces the independent, per node values of a dynamically scoped variable into a global variable.
- We introduce a "by-identifier" declare operation that allows a Lasp variable to be declared with a globally unique identifier that can be referenced by several applications. This allows different applications in a Lasp cluster to operate with the same data.
- We introduce the "Bounded-LWW-Set" CRDT and provide its formal semantics. This set supports the addition of arbitrary items at a given time, and ensures that a bound is enforced when new items are added, or pairwise merge operations occur due to replica-to-replica communication.

II. Background

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

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 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 in [8].

To demonstrate, 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.

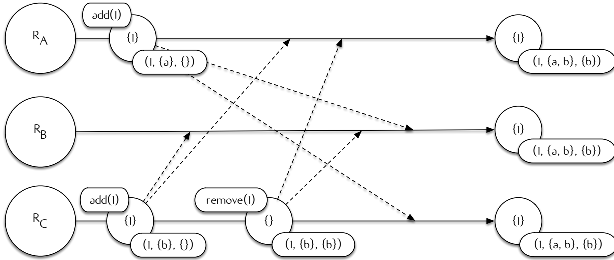

Figure 2: *Example of resolving concurrent operations with an Observed-Remove Set. In this example, concurrent operations are represented via unique identifiers at each replica.*

Figure 2 is an example of the Observed-Remove 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 [5]. Lasp allows programmers to build applications using CRDTs while ensuring that the composition of the CRDTs also observes the same strong convergence properties 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.

III. Example Code

We present a walkthrough the example code in Figure 3. This code illustrates a declarative approach to defining efficient, correct data transformations across distributed data through a familiar functional programming approach.

```
1  %% Define a pair of counters to store the global
2  %% average.
3  GlobalAverage = lasp:declare(
4     {counter, counter}, global_average),
5
6  %% Declare a dynamic variable.
7  Samples = lasp:declare_dynamic(
8     {bounded_lww_set, 100}),
9
10 %% Define a local average; this will be
11 %% computed from the local Bounded-LWW set.
12 LocalAverage = lasp:declare_dynamic(
13    {counter, counter}),
14
15 %% Register an event handler with the sensor
16 %% that is triggered each time an event X is
17 %% triggered at a given timestamp T.
18 EventHandler = fun({X, T} ->
19    lasp:update(Samples, {add, x, t}, Actor)
20 end,
21 register_event_handler(EventHandler),
22
23 %% Fold the samples using the binary function
24 %% 'avg' into a local average.
25 lasp:fold(Samples, fun avg/2, LocalAverage)
26
27 %% Fold the local average using the binary
28 %% function 'avg' into a global average.
29 lasp:fold_dynamic(LocalAverage,
30                   fun sum_pairs/2,
31                   GlobalAverage)
```

Figure 3: *Example application that computes both the local and global average from a sliding set of samples recorded by sensors in a sensor network.*

1) We begin by declaring a pair of counters to store the result of the global average computation. The average is a pair of CRDT counters: sum and count.
2) We declare a local bounded "Last-Writer-Wins" set. Each node executing this code will have its own local set that it will operate on and this set will not be replicated. We specify that this instance should be bounded at 100 elements.
3) We declare a local pair of counters to store the result of the local average computation.
4) We declare an event handler that is used to register events into the Lasp system.
5) We use the *fold*[3] operation with the binary function *avg/2* to compute the local average from the samples.
6) We use the *fold_dynamic* operation with the binary function *sum_pairs/2* to compute the global average from the local averages.

IV. System Model

We assume a dynamic set of nodes running the Lasp runtime with the Selective Hearing distribution model [5], [6]. We assume reliable broadcast implemented by an epidemic broadcast protocol. We assume nodes fail by crashing and each node can be uniquely identified in the network. We assume that

[2]For more information about how this transformation is performed and maintained, the reader is referred to [5].

[3]Fold is a process that does not terminate and continues to update the local average as the samples change. See [5] for more information on how this transformation is performed.

the Lasp runtime has a way to register a function with the edge device: this function will be used to ingest events into the Lasp system from the device as events are generated. We assume that the device executes sequentially, and events arrive to the Lasp runtime in order.

V. SEMANTICS

We give the formal semantics of our contributions to the Lasp programming model.

A. Preliminaries

This section develops a number of definitions to set up the stage for the next sections. We begin by the definition of all the n-sized subsets of an arbitrary set.

Definition 5.1. For a set S and a natural number n such that $n \leq |S|$, define $\wp_n(S) = \{S' \in \wp(S) \mid |S'| = n\}$.

Next, we show how to lift a total order on elements to an induced total order on the subsets of a totally ordered set.

Definition 5.2. Let $(T, \leq)$ be a total order. For $S, S' \in \wp(T)$, write $S \leq S'$ when $\exists a' \in S' \forall a \in S.\ a \leq a'$.

Armed with those definitions, one can get to *a* smallest n-sized subsets of a totally ordered set:

Definition 5.3. Let $(T, \leq)$ be a total order. For a natural number n, fix $F_n(T) = \min \wp_n(T)$.

Note that the notation "min" above denotes minim*al* rather than minim*um*. As such, $F_n(T)$ is not a unique set. It can well be more than one set – depending on whether the original element-level $\leq$ has a unique minimum or multiple minimals. Note also that the above "min" functions using the induced $\leq$, i.e., at the level of subsets of the totally ordered set.

B. Lasp

We refer to our previous notation for CRDTs, lattices, processes, and variables in Lasp. [5]

A replicated triple (S, M, Q) where S is a bounded join-semilattice representing the state of each replica, M is a set of functions for mutating the state, and Q is a set of functions for querying the state, is one type of CRDT.

A stream s is an infinite sequence of states of which only a finite prefix of length n is known at any given time.

$$s = [s_i \mid i \in \mathbb{N}] \tag{1}$$

The execution of one CRDT is represented by a stream of states s_i, each of which is an element of the lattice S. Each subsequent value of a stream performs a *join* with the previous value to compute the next value.

A Lasp process tracks the monotonic growth of the internal state of each CRDT and maintains a functional semantics between the state of the input and output instances. Each process correctly transforms the internal metadata of the input CRDTs to compute the correct mapping of value and metadata for the output CRDT.

C. Bounded LWW-Set

We define the bounded "Last-Writer-Wins" set. The **Bounded LWW-Set**, given a bound b, is a state-based CRDT whose bounded join-semilattice is defined by a list of triples, each representing a value at a unique timestamp and whether or not that value has been deleted.

$$s_i = \{(t_0, v_0, r_0), \ldots, (t_n, v_n, r_n)\} \tag{2}$$

We induce a total order over t.

$$\leq_t = t_0 \leq t_1 \leq \ldots \leq t_{n-1} \leq t_n \tag{3}$$

We induce a total order over s_i, ordered by t.

$$\leq_{s_i} = (t_0, v_0, r_0) \leq \ldots \leq (t_n, v_n, r_n) \tag{4}$$

We define a function α that returns the list of elements that have not been marked as removed.

$$\alpha(s_i) = \{(t, v, \bot) \mid \forall (t, v, \bot) \in s_i\} \tag{5}$$

We define one mutation on the Bounded LWW-Set, *add*. Add inserts a new element v, observed at timestamp t. If the bound is exceeded during the add, the oldest element as defined by $\leq_{s_i}$ is marked as removed.

$$add(t, v) = \begin{cases} s_i \cup \{(t, v, \bot)\} & \text{if } |\alpha(s_i)| < b, \\ s_i \cup \{(t, v, \bot)\} \cup & \\ \quad \{(t, v, \top) \mid & \text{otherwise} \\ \quad\quad (t, v, \bot) = m\} \setminus \{m\} & \end{cases} \tag{6}$$

where $m = \min s_i$.

With all that, one can get to *a* first n active triples of an Bounded LWW-Set. Note that not necessarily there is a unique first n active triples. The definition below simply picks one possible first n. Of course, the definition works just as fine when there is indeed a unique first n.

Definition 5.4. Let s_i be the ith state of an LWW-Set. For a bound b such that $b < |s_i|$, define $r_\alpha(s_i, b) = f$, for some $f \in F_b(\alpha(S_i))$.

For the ith state of an Bounded LWW-Set, define the following shorthand for the tombstoned triples:

$$\tau(s_i) = s_i \setminus \alpha(s_i). \tag{7}$$

We define the merge process for the Bounded LWW-Set as a two phase process: first, we compute the merge of two objects, then we enforce that the number of elements present in the set are within the bound b.

$$\begin{aligned} m(s_i, s_i') = \{&(t, v, r \sqcup r') \mid (t, v, r) \in s_i \wedge (t, v, r') \in s_i'\} \cup \\ \{&(t, v, r) \mid (t, v, r) \in s_i \wedge (t, v, _) \notin s_i'\} \cup \\ \{&(t, v, r) \mid (t, v, r) \in s_i' \wedge (t, v, _) \notin s_i\} \end{aligned} \tag{8}$$

$$\begin{aligned} merge(s_i, s_i', b) = \alpha(m) \setminus & \\ & r \cup \{(ts, v, \top) \mid (ts, v, _) \in r \cup \tau(m)\} \end{aligned} \tag{9}$$

where $m = m(s_i, s_i')$ and $r = r_\alpha(m, b)$.

We define the fold operation over the Bounded-LWW-Set. The *fold* function defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. Given $query(s_i) = V = \{v_0, ..., v_{n-1}\}$ and an operation op of t's type with neutral element e, this should return the state $t_i = e \ \mathrm{op}\ v_0 \ \mathrm{op}\ v_1 \cdots \mathrm{op}\ v_{n-1}$. If $remove(v_k)$ is done on s_i, then v_k is removed from V, so v_k must be removed from this expression in order to calculate t_{i+1}. The difficulty is that this must be done through a monotonic update of t_i's metadata. We present a correct but inefficient solution below. This solution assumes that op is associative, commutative, and has an inverse denoted by op'.

$$fold'(s_i, \mathrm{op}) = \mathrm{Op}_{(t,v,r) \in s_i} \begin{cases} \mathrm{Op}\ v \ \mathrm{op}\ \mathrm{Op}'\ v & \text{if } r = \top \\ \mathrm{Op}\ v & \text{otherwise.} \end{cases}$$
$$fold(s, \mathrm{op}) = t = [fold'(s_i, f) \mid s_i \in s] \tag{10}$$

The *dynamically scoped fold*, as seen in Figure 4, defines a process that reads two input streams s and t, and creates elements in the output stream t. As both the input stream s and the accumulator stream t grow monotonically, the *fold* operation is re-executed with the accumulator and the input stream s until a fixed point is reached[4].

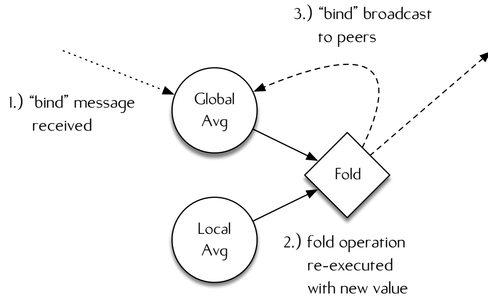

Figure 4: *Example of the dynamically scoped fold. As the the value of the accumulator changes, the fold operation is re-executed with the local average and the result broadcast until a fixed point is computed in the network.*

D. Lasp API

We extend the Lasp gossip-based distribution model, as presented in [6]. We present the extended API below:

- $i = declare(t, u)$: Given type information t and an unique constant u, declare a new identifier i that contains the type information and broadcast this identifier to all nodes.
- $i = declare_dynamic(t, u)$: Given type information t, return and a unique constant u, declare a new identifier i that contains the type information and broadcast this identifier to all nodes. This variable is "dynamically scoped" per node; that is, its value is node specific and not replicated across nodes.
- $fold_dynamic(s, op, t)$: Fold values from s into t using op. This is dynamically scoped: values of s across all nodes in the cluster will be folded into t.

[4]It is important to note that the *bind* operation always performs a merge with the current state when received, so a *fold* over partial state still guarantees monotonicity.

E. Node State

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

$$(\sigma^{(0)}, \delta_i^{(0)}, \delta_v^{(0)}, \delta_d^{(0)}) = (\{\}, \{\}, \{\}, \{\}) \to \cdots \to (\sigma^{(k)}, \delta_i^{(k)}, \delta_v^{(k)}, \delta_d^{(k)}) \to \cdots \tag{11}$$

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:

$$\sigma = \{i_0, i_1, \ldots\} \tag{12}$$

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.

$$\delta_i = \{(i_0, \{(p_0, c_0), \ldots\}),\ (i_1, \{(p_1, c_1), \ldots\}), \ldots\} \tag{13}$$

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:

$$\delta_v = \{(i_0, v_0), (i_1, v_1), \ldots\} \tag{14}$$

The *dynamic variables set* δ_d contains the unique variable identifiers declared at the node:

$$\delta_d = \{i_0, i_1, \ldots\} \tag{15}$$

F. 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 $(\sigma, \delta_i, \delta_v, \delta_d)$ to compute the subsequent state $(\sigma', \delta_i', \delta_v', \delta_d')$; 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 redefine the local semantics of **bind** as presented in [6] to use the dynamic variable set (δ_d), and introduce the new **declare**, **declare_dynamic** and **fold_dynamic** operations.

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

$$bind_k(i, v) : \mathrm{true} \tag{16}$$

The operation is then only broadcast from k to all nodes j if the variable is not contained by the dynamic variable set δ_d. If it is contained, bind the variables locally. When this message

is broadcast, the message follows the semantics as presented in [6].

$$(\nexists i \in \delta_i \wedge \exists i \in \delta_v) \Rightarrow \ \forall j.\ bind_k^j(i,v)\ ;\ bind_k^k(i,v) \quad (17)$$

declare The operation $i = declare(t,u)$ returns the variable identifier i.

$$i = declare_k(t,u) : i = (u,t) \quad (18)$$

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.

$$declare_k^j(i) : \sigma' = \sigma \cup \{i\} \quad (19)$$

declare_dynamic The operation $i = declare_dynamic(t,u)$ returns the variable identifier i and adds the variable identifier to the known variable set σ.

$$i = declare_dynamic_k(t,u) : i = (u,t) \quad (20)$$

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 $declare_dynamic_k^j(i)$ means that node k broadcasts to node j). This adds the variable identifier to the known variables σ.

$$\begin{aligned} declare_dynamic_k^j(i) :\ & \wedge i = (u,t) \wedge \sigma' = \sigma \cup \{i\} \\ & \wedge \delta_d' = \delta_d \cup \{i\} \end{aligned} \quad (21)$$

fold_dynamic The operation $fold_dynamic(s, op, t)$ defines a process that never terminates, which reads elements from input streams s and t and produces elements in the output stream t when either stream increases. $fold_dynamic(s, op, t)$ operates similarly to the $fold$ operation defined in [5], with one exception, detailed below.

In this example, s is assumed to be a *dynamically scoped variable* and t is assumed to be a global variable. Because of this, s_i represents a single element in a collection existing across nodes. The *fold_dynamic* operation executes with the singleton set of s_i and the result of the computation t_i is then broadcast to all nodes. Upon receipt of a new element of t, or a change in s, the operation is re-executed until a fixed point is reached.

$$\begin{aligned} f'(s_i, \mathrm{op}) &= \mathrm{Op}_{(v,a,r)\in s_i}(\mathrm{Op}_{u\in a} v \quad \mathrm{op} \quad \mathrm{Op'}_{u\in r} v) \\ fold_dynamic(s, \mathrm{op}, t) &= t = [f'(\{s_i\}, f) \mid s_i \in s] \end{aligned} \quad (22)$$

The operation then follows the broadcast specification for the *bind* operation as described in Equation 17.

VI. Related Work

Digest diffusion [4], building on directed diffusion [9], presents a energy-efficient model of distributed computation through the use of function decomposition on monotonic functions that compute aggregate values. Digest diffusion can produce incorrect results if functions are not idempotent, given the state dissemination mechanism used does not guarantee at-most or exactly-once delivery.

Tiny AGgregation (TAG) [10] provides a declarative interface for data collection and aggregation across sensor networks, inspired by modern database query languages. TAG performs aggregations as data is propagated through the routing tree defined by the network. TAG does not provide a general programming model: aggregations are performed against a sensor table over user-defined epochs.

Finally, *Lisp, developed for the Connection Machine [11] presented the idea of PVARS: parallel variables that had a different value per virtual processor. Parallel operations are used to modify the values for all instances of a variable across processors with a single operation, or combine the values across different parallel variables.

VII. Conclusion

This paper presents a work in progress report on a declarative method for performing aggregations over devices at the edge by extending the Lasp programming model with the notion of dynamic scoping. We plan to continue this work by implementing these extensions and evaluating how general our approach is for computing aggregates.

Acknowledgment

This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n° 609551.

References

[1] M. Nijdam, private communication, 2015.

[2] S. Khare, K. An, A. Gokhale, and S. Tambe, "Functional Reactive Stream Processing for Data-centric Publish/Subscribe Systems."

[3] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[4] 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.

[5] C. Meiklejohn and P. Van Roy, "Lasp: A Language for Distributed, Coordination-Free Programming," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2015, pp. 184–195.

[6] ——, "Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation," in *Workshop on Planetary-Scale Distributed Systems collocated with SRDS 2015*. IEEE, 2015.

[7] 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.

[8] 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.

[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] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 131–146, 2002.

[11] N. H. Brown Jr, "Neural network implementation approaches for the connection machine," in *Neural Information Processing Systems*, 1988, pp. 127–136.