Consistency Analysis in Bloom: a CALM and Collected Approach

Peter Alvaro Neil Conway Joseph M. Hellerstein
William R. Marczak

ABSTRACT

Distributed programming has become a topic of widespread interest, and many programmers now wrestle with tradeoffs between data consistency, availability and latency. Distributed transactions are often rejected as an undesirable tradeoff today, but in the absence of transactions there are few concrete principles or tools to help programmers design and verify the correctness of their applications.

We address this situation with the *CALM* principle, which connects the idea of distributed consistency to program tests for logical monotonicity. We then introduce *Bloom*, a distributed programming language that is amenable to high-level consistency analysis and encourages order-insensitive programming. We present a prototype implementation of Bloom as a domain-specific language in Ruby. We also propose a static analysis technique that identifies *points of order* in Bloom programs: code locations where programmers need to inject coordination logic to ensure consistency. We illustrate these ideas in the context of two variants of a distributed "shopping cart" application in Bloom. We also sketch the feasibility of code rewrites to support runtime annotation of data consistency, a topic for a longer paper.

1. INTRODUCTION

Until fairly recently, distributed programming was the domain of a small group of experts. But recent technology trends have brought distributed programming to the mainstream of open source and commercial software. The challenges of distribution—concurrency and asynchrony, performance variability, and partial failure—often translate into tricky data management challenges regarding task coordination and data consistency. Given the growing need to wrestle with these challenges, there is increasing pressure on the data management community to help find solutions to the difficulty of distributed programming.

There are two main bodies of work to guide programmers through these issues. The first is the "ACID" foundation of distributed transactions, grounded in the theory of serializable read/write schedules and consensus protocols like Paxos and Two-Phase Commit. These techniques provide strong consistency guarantees, and can help shield programmers from much of the complexity of distributed programming. However, there is a widespread belief that the costs of these mechanisms are too high in many important scenarios where availability and/or low-latency response is critical. As a result, there is a great deal of interest in building distributed software that avoids using these mechanisms.

The second point of reference is a long tradition of research and system development that uses application-specific reasoning to tolerate "loose" consistency arising from flexible ordering of reads, writes and messages (e.g., [2, 4, 5, 7]). This approach enables machines to more easily tolerate temporary delays, message reordering, and component failures. The challenge with this design style is to ensure that the resulting software tolerates the inconsistencies in a meaningful way, producing acceptable results in all cases. Although there is a body of wisdom and best practices that informs this approach, there are few concrete software development tools that codify these ideas. Hence it is typically unclear what guarantees are provided by systems built in this style, and the resulting code is hard to test and hard to trust.

Merging the best of these traditions, it would be ideal to have a robust theory and practical tools to help programmers reason about and manage high-level program properties in the face of loosely-coordinated consistency. In this paper we demonstrate significant progress in this direction. Our approach is based on the use of a declarative language and program analysis techniques that enable both static analyses and runtime annotations of consistency. We begin by introducing the *CALM* principle, which makes a formal connection between the theory of monotonic logic and the need for distributed coordination to achieve consistency. We present an initial version of our *Bloom* declarative language, and translate concepts of monotonicity into a practical program analysis technique that detects potential consistency anomalies in distributed Bloom programs. We then show how such anomalies can be handled by a programmer during the development process, either by introducing coordination mechanisms to ensure consistency or by applying program rewrites that can track inconsistency "taint" as it propagates through code. To illustrate the Bloom language and the utility of our analysis, we study two implementations of a popular example of distributed consistency: a fault-tolerant replicated shopping cart service.

2. CONSISTENCY AND LOGICAL MONOTONICITY (CALM)

In this section we present a strong connection between distributed consistency and logical monotonicity. This discussion informs the language and analysis tools we develop in subsequent sections.

A key problem in distributed programming is reasoning about consistency in the face of *temporal nondeterminism*: the delay and re-ordering of messages and data across nodes. Because delays can be unbounded, analysis typically focuses on "eventual consistency" after all messages have been delivered [13]. A sufficient condition for eventual consistency is *order independence*: the independence of program execution from temporal nondeterminism.

Order independence is a key attribute of declarative languages based on sets, which has led most notably to the success of parallel databases. But even set-oriented languages can require a degree of ordering in their execution if they are sufficiently expressive. The theory of relational databases and logic programming provides a framework to reason about these issues. *Monotonic* programs—e.g., programs expressible via selection, projection and join—can be implemented by streaming algorithms that incrementally produce output elements as they receive input elements; the final order or contents of the input will never cause any earlier output to be "revoked" once it has been generated.[1] *Non-monotonic* programs—e.g., those that contain aggregation or anti-join operations—can only be implemented correctly via blocking algorithms that do not produce any output until they have received all tuples in logical partitions of an input set. For example, aggregation queries need to receive entire "groups" before producing aggregates, which in general requires receiving the entire input set.

The implications for distributed programming and delayed messaging are clear. Monotonic programs are easy to distribute: they can be implemented via streaming set-based algorithms that produce actionable outputs to consumers while tolerating message reordering and delay from producers. By contrast, even simple non-monotonic tasks like counting are difficult in distributed systems. As a mnemonic, we say that *counting requires waiting* in a distributed system: in general, a complete count of distributed data must wait for all its inputs, including stragglers, before producing the correct output.

"Waiting" is specified in a program via *coordination logic*: code that (a) computes and transmits auxiliary information from producers to enable the recipient to determine when a set has completely arrived across the network, and (b) postpones production of results for consumers until after that determination is made. Typical coordination mechanisms include sequence numbers, counters, and consensus protocols like Paxos or Two-Phase Commit.

Interestingly, these coordination mechanisms themselves typically involve counting. For example, Paxos requires message counting to establish that a majority of the members have agreed to a proposal; Two-Phase Commit requires message counting to establish that all members have agreed. Hence we also say that *waiting requires counting*, the converse of our earlier mnemonic.

Our observations about waiting and counting illustrate the crux of what we call the *CALM* principle: the tight relationship between Consistency And Logical Monotonicity. Monotonic programs *guarantee* eventual consistency under any interleaving of delivery and computation. By contrast, non-monotonicity—the property that adding an element to an input set may revoke a previously-valid element of an output set—requires coordination schemes that "wait" until inputs can be guaranteed to be complete.

We typically wish to minimize the use of coordination, because of well-known concerns about latency and availability in the face of message delays and network partitions. We can use the CALM principle to develop checks for distributed consistency in logic languages, where conservative tests for monotonicity are well understood. A simple syntactic check is often sufficient: if the program does not contain any of the symbols in the language that correspond to non-monotonic operators (e.g., `NOT IN` or aggregate symbols), then it is monotonic and can be implemented without coordination, regardless of any read-write dependencies in the code. These conservative checks can be refined further to consider semantics of predicates in the language. For example, the expression "`MIN(x)` < 100" is monotonic despite containing an aggregate, by virtue of the semantics of `MIN` and <: once a subset S satisfies this test, any superset of S will also satisfy it. Further refinements along these lines exist (e.g., [8, 9]), increasing the ability of program analyses to verify monotonicity.

In cases where an analysis cannot guarantee monotonicity of a whole program, it can instead provide a conservative assessment of the points in the program where coordination may be required to ensure consistency. For example, a shallow syntactic analysis could flag all non-monotonic predicates in a program (e.g., `NOT IN` tests or predicates with aggregate values as input). The loci produced by this analysis are the program's *points of order*. A program with non-monotonicity can be made consistent by including coordination logic at its points of order.

The reader may observe that because "waiting requires counting," adding coordination logic actually increases the number of points of order in a program. To avoid this problem, the coordination logic must be hand-verified for consistency, after which annotations on the coordination logic can inform the analysis tool to (a) skip the coordination logic in its analysis, and (b) skip the point of order that the coordination logic handles.

Because analyses based on the CALM principle operate with information about program semantics, they can avoid coordination logic in cases where traditional read/write analysis would require it. Perhaps more importantly, as we will see in the next sections, logic languages and the analysis of points of order can help programmers redesign code to achieve goals of minimizing coordination.

3. BUD: BLOOM UNDER DEVELOPMENT

Bloom is based on the conjecture that many of the fundamental problems with parallel programming come from a legacy of assumptions regarding classical von Neumann architectures. In the von Neumann model, state is captured in an ordered array of addresses, and computation is expressed via an ordered list of instructions. Traditional imperative programming grew out of these pervasive assumptions about order. Therefore, it is no surprise that popular imperative languages are a bad match to parallel and distributed platforms, which make few guarantees about order of execution and communication. By contrast, set-oriented approaches like SQL and batch dataflow approaches like MapReduce translate better to architectures with loose control over ordering.

Bloom is designed in the tradition of programming styles that are "disorderly" by nature. State is captured in unordered relations. Computation is expressed in logic: an unordered set of declarative rules, each consisting of an unordered set of predicates. As we discuss below, mechanisms for imposing order are available when needed, but the programmer is provided with tools to evaluate the need for these mechanisms as special-case behaviors, rather than a default model. The result is code that runs naturally on distributed machines with a minimum of coordination overhead.

[1]Formally, in a monotonic logic program, any true statement continues to be true as new axioms—including new facts—are added to the program.

Type	Behavior
table	A collection whose contents persist across timesteps.
scratch	A collection whose contents persist for only one timestep.
channel	A scratch collection with one attribute designated as the *location specifier*. Tuples "appear" at the address stored in their location specifier.
periodic	A scratch collection of key-value pairs (`id`, `timestamp`). The spec for a **periodic** is parameterized by a `period` in seconds; the runtime system arranges (in a best-effort manner) for tuples to "appear" in this collection approximately every `period` seconds, with a unique `id` and the current wall-clock time.

Op	Valid lhs types	Meaning
=	**scratch**	rhs defines the contents of the lhs for the current timestep. lhs must not appear in lhs of any other statement.
<=	**table**, **scratch**	lhs includes the content of the rhs in the current timestep.
<+	**table**, **scratch**, **channel**	lhs will include the content of the rhs in the next timestep; remote **channel** tuples subject to delay.
<-	**table**	tuples in the rhs will be missing from the lhs at the start of the next timestep.

Figure 1: Bloom collection types and operators.

Unlike earlier efforts such as Prolog, active database languages, and our own Overlog language for distributed systems [6], Bloom is *purely declarative*: the syntax of a program contains the full specification of its semantics, and there is no need for the programmer to understand or reason about the behavior of the evaluation engine. Bloom is based on a formal temporal logic called Dedalus [1].

The prototype version of Bloom we describe here is embodied in an implementation we call *Bud* (Bloom Under Development). Bud is a subset of the popular Ruby scripting language and is evaluated by a stock Ruby interpreter via a **Bud** Ruby class. Compared to other logic languages, we feel it has a familiar and programmer-friendly flavor, and we believe that its learning curve will be quite flat for programmers familiar with modern scripting languages.

Bloom Basics

Bloom programs are bundles of declarative statements about collections of "facts" or tuples, akin to SQL views or Datalog rules. Bloom statements can only reference data that is local to a node. Bloom rules are defined with respect to atomic "timesteps," which can be implemented via successive rounds of evaluation. In each timestep, certain "ground facts" exist in collections due to persistence or the arrival of messages from outside agents (e.g., the network or system clock). The statements in a Bloom program specify the derivation of additional facts, which can be declared to exist either in the current timestep, at the very next timestep, or at some time in the future at a remote node. A Bloom program also specifies the way that facts persist (or do not persist) across consecutive timesteps on a single node. Bloom is a side-effect free language with no "mutable state": if a fact is defined at a given timestep, its existence at that timestep cannot be refuted by any expression in the language. This technicality is key to avoiding many of the complexities involved in reasoning about earlier "stateful" rule languages. The paper on Dedalus discusses these points in more detail [1].

State in Bloom

Bloom programs manage state using four collection types described in the top of Figure 1. Each object of these types is defined with a relational-style schema of named columns, including an optional subset of those columns that forms a primary key. Line 2 of Figure 2 shows the definition of a **table** with 4 columns `session`, `reqid`, `item` and `action`; the primary key is `(session, reqid)`. In Bud, the type system for the columns is taken from Ruby, so it is possible to have a column based on any Ruby class the programmer cares to define or import. In Bud, a tuple in a Bloom collection is simply a Ruby array containing as many elements as the columns of the collection's schema. As in other object-relational ADT schemes like Postgres [11], column values can be manipulated using their own (non-destructive) methods, but the Bloom language has no knowledge of those types and can only reference the column values as opaque objects.

```
0  class BasicCartServer < Bud
1    def state
2      table :cart_action, ['session', 'reqid'], ['item', 'action']
3      scratch :status, ['server', 'client', 'session', 'item'],
4                        ['cnt']
5      channel :action_msg, 0, ['server', 'client', 'session',
6                                'item', 'action', 'reqid']
7    end
8    [...]
9  end
```

Figure 2: Example Bloom collection declarations.

Persistence in Bloom is determined by the type of the collection. **scratch** collections are handy for transient data like network messages and for "macro" definitions that enable code reuse. The contents of a **table** persist across consecutive timesteps (until that persistence is interrupted via a Bloom statement containing the `<-` operator described below). Although there are precise declarative semantics for this persistence [1], it is convenient to think operationally as follows: **scratch** collections are "emptied" before each timestep, **table**s are "stored" collections, and the `<-` operator represents batch "deletion" before the beginning of the next timestep.

The facts of the "real world," including network messages and the passage of wall-clock time, are captured via **channel** and **periodic** collections; these are **scratch** collections whose contents "appear" at non-deterministic timesteps. The paper on Dedalus delves deeper into the logical semantics of this non-determinism [1]. Note that failure of nodes or communication is captured here: it can be thought of as the repeated "non-appearance" of a fact at every timestep. Again, it is convenient to think operationally as follows: facts in a **channel** are delivered to the address in their location specifier via a best-effort unordered network protocol like UDP, and the definition of a **periodic** collection instructs the runtime to "inject" facts at regular wall-clock intervals to "drive" further derivations.

Bloom Statements

Statements in Bloom are akin to rules in Datalog or views in SQL. They consist of declarative relational statements that define the contents of derived collections. The syntax is:

<collection-variable> <op> <collection-expression>

In the Bud prototype, both sides of the operator are instances of (a subclass of) a Ruby class called `BudCollection`, which inherits Ruby's built-in `Enumerable` module supporting typical collection methods. Figure 1 describes the four operators that can be used to define the contents of the left-hand side (lhs) in terms of the right-hand side (rhs).

As in Datalog or SQL, the lhs of a statement may be referenced recursively in its rhs, or recursion can be defined mutually across statements. The rhs typically includes methods of `BudCollection` objects. Most common is the `map` method of Ruby's `Enumerable`

```
0  | declare
1  | def store
2  |   kvput <= action_msg.map do |a|
3  |     if not bigtable.map{|b| b.key}.include? a.session
4  |       if a.action == "Add"
5  |         [a.server, a.client, a.session, a.reqid, [a.item]]
6  |       elsif a.action == "Del"
7  |         [a.server, a.client, a.session, a.reqid, []]
8  |       end
9  |     end
10 |   end

12 |   kvput <= join([bigtable, action_msg]).map do |b, a|
13 |     if b.key == a.session
14 |       if a.action == "Add"
15 |         [a.server, a.client, a.session,
16 |          a.reqid, b.value.push(a.item)]
17 |       elsif a.action == "Del"
18 |         [a.server, a.client, a.session,
19 |          a.reqid, b.value.reject{|bv| bv == a.item}]
20 |       end
21 |     end
22 |   end
23 | end

25 | declare
26 | def communicate
27 |   action_msg <+ client_action.map{|a| a}

29 |   response_msg <+ join([bigtable, checkout_msg]).map do |s, c|
30 |     if s.key == c.session
31 |       [c.client, c.server, s.key, s.value]
32 |     end
33 |   end
34 | end
```

Figure 3: Destructive cart implementation.

module, which is used to specify scalar operations on all tuples of a `BudCollection`, including relational selection and projection. For example, lines 2–4 of Figure 4 project the `action_msg` collection to its `session`, `item`, `action` and `reqid` fields. `BudCollection` defines a `group` method akin to SQL's `GROUP BY`, supporting the standard SQL aggregates; for example, lines 6–8 of Figure 4 compute the count of unique `reqid` values for every combination of values for `session`, `item` and `action`. Multiway joins are specified using the `join` method, which produces an anonymous **scratch** collection that can be used in the rhs of a statement.[2] Line 38 of Figure 4 shows a join between `action_msg` and `member`.

Programmers declare Bloom statements within methods of a Bud subclass definition that are flagged with the `declare` modifier (e.g., line 0 of Figure 3). The semantics of a Bloom program are defined by the union of all the `declare` methods; the order of statements is immaterial. Dividing statements into multiple methods improves the readability of the program and allows use of Ruby's method overriding and inheritance features, as described below.

The constructs above form the core of the Bloom language. Bud also includes some additional convenience methods that provide macros over these methods, and admits the use of simple side-effect-free Ruby expressions within statements. More importantly, Bud takes advantage of the features of Ruby to enrich its declarative constructs with familiar programming metaphors that are popular with software developers. Like any Ruby class, a Bud class can be specialized via subclassing. In particular, `declare` methods can be overridden in subclasses or in specific instances, allowing for selective rewriting of encapsulated bundles of statements.

Bud Implementation

[2]For clarity, we write out the join condition explicitly. Bloom provides syntax sugar for common join types (e.g., natural join).

```
0  | declare
1  | def accumulate
2  |   cart_action <= action_msg.map do |c|
3  |     [c.session, c.item, c.action, c.reqid]
4  |   end

6  |   action_cnt <= cart_action.group(
7  |         [cart_action.session, cart_action.item, cart_action.action],
8  |         count(cart_action.reqid))
9  | end

11 | declare
12 | def summarize
13 |   status <= join([action_cnt, checkout_msg]).map do |a, c|
14 |     if a.action == "Add" and not
15 |        action_cnt.map{|d| d.id if d.action == "Del"}.include? a.id
16 |       [a.session, a.item, a.cnt]
17 |     end
18 |   end

20 |   status <= join([action_cnt, action_cnt,
21 |                   checkout_msg]).map do |a1, a2, c|
22 |     if a1.session == a2.session and a1.item == a2.item and
23 |        a1.session == c.session and a1.action == "Add" and
24 |        a2.action == "Del"
25 |       [a1.session, a1.item, a1.cnt - a2.cnt]
26 |     end
27 |   end
28 | end

30 | declare
31 | def communicate
32 |   response_msg <+ join([status, checkout_msg]).map do |s, c|
33 |     if s.session == c.session
34 |       [c.client, c.server, s.session, s.item, s.cnt]
35 |     end
36 |   end

38 |   action_msg <+ join([action_msg, member]).map do |a, m|
39 |     unless member.map{|nm| nm.player}.include? a.client
40 |       [m.player, a.server, a.session, a.item, a.action, a.reqid]
41 |     end
42 |   end
43 | end
```

Figure 4: Disorderly cart implementation.

Bud was intended to be a lightweight rapid prototype of Bloom: a first effort at embodying the Dedalus logic in a syntax familiar to programmers. Bud consists of less than 1400 lines of Ruby code, developed in approximately two person-months of part-time effort.

4. CASE STUDY

In this section, we develop two different designs for a distributed shopping-cart application in Bloom.[3] First, we implement a "destructive," state-modifying shopping cart application using a simple key-value store (also written in Bloom). Second, we implement a "disorderly" cart that accumulates updates in a set-wise fashion, summing up the updates at checkout into a final result. These two different designs illustrate our analysis tools and the way they inform design decisions for distributed programming.

We begin with a shopping cart built on a key-value storage abstraction. Each cart is a `(key,value)` pair, where `key` is a unique session identifier and `value` is an object containing the session's state, including a Ruby array that holds the items in the cart. Adding or deleting items from the cart result in "destructive" updates: the value associated with the key is replaced by a new value that reflects the effect of the update. Deletion requests are ignored if the item they refer to does not exist in the cart.

Figure 3 shows the Bloom code for this design. The **scratch** `kvput`

[3]The complete source code for both implementations can be found at `https://trac.declarativity.net/browser/bud/test/cart`.

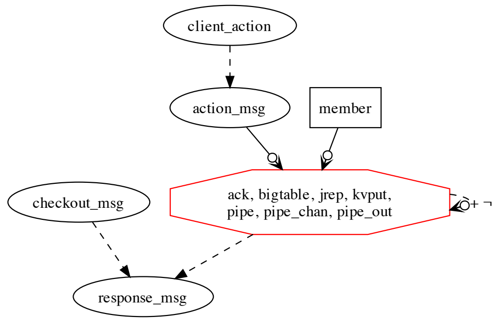

Figure 5: Destructive cart analysis.

is defined by the `KeyValueStore` module (10 lines of Bloom not shown here), which the shopping cart extends via inheritance; it represents the input interface to the key-value store. The set of shopping carts is represented by the persistent **table** `bigtable`, which is replicated by shipping `kvput` tuples between replicas upon updates.

In line 27, the client transmits `client_action` tuples that correspond to cart updates over the `action_msg` **channel** to an individually chosen server replica. We assume that the client has already chosen a server based on load balancing. For each such arriving tuple, line 3 checks `bigtable` to see if a record exists for the session associated with the `action_msg`. If none is there (i.e., this is the first update for a new session), then lines 4–8 generate an entry for the new session in `bigtable`. Otherwise, the join conditions in lines 12–13 are satisfied and lines 14–20 "replace" the item array at the next timestep with a new version. Whenever a `checkout_msg` appears in a server replica, the `bigtable` tuple associated with the given session is identified (via the join on lines 29–33), and the item array in its value field is sent back to the client.

Figure 4 shows an alternative shopping cart implementation, in which updates are monotonically accumulated during shopping in a disorderly set, and summed up only at checkout. Lines 2–4 insert client updates into the persistent table `cart_action`. Lines 6–8 define `action_cnt` as an aggregate over `cart_action`, in the style of an SQL `GROUP BY` statement: for each item associated with a cart, we separately count the number of times it was added and the number of times it was deleted. Lines 13–18 ensure that when a `checkout_msg` tuple arrives, `status` contains a record for every added item for which there was no corresponding deletion in the session. Lines 21–27 additionally define `status` as the 3-way join of the `checkout_msg` message and two copies of `action_cnt`—one corresponding to additions and one to deletions. Thus, for each item, `status` contains its final quantity: the difference between the number of additions and deletions (line 25), or simply the number of additions if there are no deletions (line 16). Upon the appearance of a `checkout_msg`, the replica returns a `response` message to the client containing the final quantity (lines 32–36). Because the disorderly implementation does not use a separate storage system, it includes its own logic for state replication (lines 38–42). Line 40 ensures that only the replica contacted by the client performs a multicast, by probing the **table** `member` to see if the `action_msg` originated inside the quorum.

Analysis
The Bud interpreter automatically generates a graphical representation of dependencies between collections in a program (Figures 5 and 6). Each node in the graph is either a collection or a cluster of collections; **table**s are shown as rectangles, ephemeral collections (**scratch** and **channel**) are depicted as ovals, and clusters (described below) as octagons. A directed edge from node A to node B indicates that B appears in the lhs of a Bloom rule with A referenced directly, or through a join expression, in the rhs. An edge is annotated based on the operator symbol in the rule and the type of B. If the rule is "inductive"—i.e., has the <+ operator and a **table** lhs—then the edge is marked with a +. If the rule is "asynchronous"—i.e., a rule with the <+ operator and a **channel** lhs—then the edge is a dashed line. If the rule involves non-monotonicity (aggregation or negation) over A, then the edge is marked with a ¬. To make the visualizations more readable, any strongly connected component with both a ¬ and a + edge is collapsed into a octagonal "temporal cluster," which can be viewed abstractly as a single, nonmonotonic node in the dataflow.[4] Points of order are indicated in the graph by an edge with a white circle. Any negated edge in the graph is a point of order, as are all edges incident to a temporal cluster, including any self-edges.

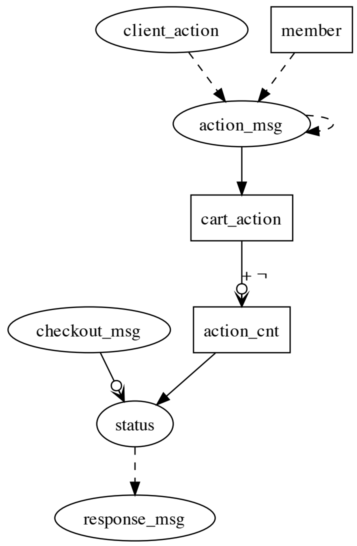

Figure 6: Disorderly cart analysis.

Figure 5 presents the analysis of the "destructive" shopping cart variant. Note that because all dependencies are analyzed, collections defined in the superclass but not referenced in the code sample (e.g., `pipe_chan`, `member`) also appear in the graph. Although there is no non-monotonicity in Figure 3, the underlying key-value store uses the non-monotonic `<-` operator to model updateable state.[5] The full-program analysis shown in Figure 5 indicates that there are points of order between `action_msg`, `member` and the temporal cluster, and between the temporal cluster and itself. This figure tells the (sad!) story of how we could ensure consistency of the destructive cart implementation: introduce coordination between client and server—and between the chosen server and all its replicas—for *every client action or kvput update*. This fine-grained coordination is akin to

[4]Datalog aficionados may be troubled by our mention of cycles involving negation in the dependency graph. In [1] we describe why the inclusion of a + edge in such a cycle ensures stratifiability.
[5]Our full-length paper will discuss the non-monotonic nature of <-, and will expand the SCC to analyze the key-value store in detail.

"eager replication" [3]. It would incur the latency of a round of messages per server per client update, decrease system throughput, and make the system fragile in the face of replica failures.

Because we only care about the *set* of elements contained in the value array and not its order, we might be tempted to argue that the shopping cart application is eventually consistent when asynchronously updated, and forego the coordination logic. Unfortunately, such informal reasoning can hide serious bugs. For example, consider what would happen if a delete action for an item arrived at some replica before any addition of that item: the delete would be ignored, leading to inconsistencies between replicas.

A happier story emerges via the analysis of the disorderly implementation shown in Figure 6. Here we see that communication (via `action_msg`) between client and server—and among server replicas—crosses no points of order, so all the communication related to shopping actions converges to the same final state without coordination. However, there are points of order upon the appearance of `checkout_msg` messages, which must be joined with an `action_cnt` aggregate over the set of updates. Although the accumulation of shopping actions is monotonic, summarization of the cart state requires us to assume (or rather, prove) that there will be no further cart actions. The story of coordination in this graph is much happier: to ensure that the response to the client is deterministic and consistently replicated, we need to coordinate once per *session* (at checkout), rather than once per shopping action. This is analogous to the desired behavior in practice [5].

Discussion

Strictly monotonic programs are rare in practice, so adding some amount of coordination is often required to ensure consistency. In this running example we studied two candidate implementations of a simple distributed application with the aid of our language and program analysis. Both programs have points of order, but the analysis tool helped us reason about their relative coordination costs. Deciding that the disorderly approach is "better" required us to apply domain knowledge: checkout is a coarser-grained coordination point than cart actions and their replication.

By providing the programmer with a set of abstractions that are predominantly order-independent, Bloom encourages a style of programming that minimizes coordination requirements. But as we see in our destructive cart implementation, it is nonetheless possible to use Bloom to write code in an imperative, order-sensitive style. Our analysis tools provide assistance in this regard. Given a particular implementation with points of order, Bloom's static analysis can help a programmer iteratively refine their program: either to "push back" the points to as late as possible in the dataflow, as we did in this example, or to "localize" points of order by moving them to locations in the program's dataflow where the coordination can be implemented on individual nodes without communication.

5. CONCLUSION AND DISCUSSION

In this short submission, we make three main contributions. First, we present the CALM principle, which connects the common practice of eventual consistency in distributed programming to a strong theoretical foundation in database theory. Second, we show that we can bring that theory to bear on the practice of software development, via "disorderly" programming patterns and automatic analysis techniques for identifying the points of order in a program. Finally, we present our Bloom prototype as an example of a practically-minded declarative programming language, with an initial implementation as a Domain-Specific Language within Ruby.

We have more ideas about codifying best practices of distributed programming in software tools, which we hope to present in a full paper. Chief among these is to provide programmer tools for *managing inconsistency*, rather than resolving it via coordination. Helland and Campbell reflect on their experience programming with patterns of "memories, guesses and apologies" [5]. We provide a sketch here of ideas for converting these patterns into developer tools.

"Guesses"—facts that may not be true—may arise at the inputs to a program, e.g., from noisy sensors or untrusted software or users. But Helland and Campbell's use of the term corresponds in our analysis to unresolved points of order: non-monotonic logic that makes decisions without full knowledge of its input sets. We can rewrite the schemas of Bloom collections to include an additional attribute marking each fact as a "guarantee" or "guess," and automatically augment user code to propagate those labels through program logic in the manner of "taint checking" in program security [10, 12]. Moreover, by identifying unresolved points of order, we can identify when program logic derives "guesses" from "guarantees," and rewrite user code to label data appropriately.

By rewriting programs to log guesses that cross interface boundaries, we can also implement Helland and Campbell's idea of "memories": a log of guesses that were sent outside the system. If at a coarser time-scale, the logic of a program can establish the veracity of previous guesses (e.g., via periodic background coordination or external inputs), exception-handling rules can be invoked to send confirmations or "make apologies": contact customers, issue refunds, etc.

Most of these patterns can be implemented as automatic program rewrites. We cannot fully automate the generation of "apologies," since they are typically application-specific. But we believe we can help there too, by generating code scaffolds for apology generation and issuance. These could include background process scaffolds for resolving guesses via long-timescale coordination, and apology scaffolds for joining guesses that were incorrect with the history of messages that require apologies. More ambitiously, we hope to provide analysis techniques that can prove the consistency of the high-level workflow: i.e., prove that any combination of user behavior, background guess resolution, and apology logic will eventually lead to a consistent resolution of the business rules at both the user and system sides.

6. REFERENCES

[1] P. Alvaro et al. Dedalus: Datalog in Time and Space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, 2009.

[2] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.

[3] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, pages 173–182, 1996.

[4] P. Helland. Life Beyond Distributed Transactions: an Apostate's Opinion. In *CIDR*, 2007.

[5] P. Helland and D. Campbell. Building on Quicksand. In *CIDR*, 2009.

[6] B. T. Loo et al. Implementing Declarative Overlays. In *SOSP*, 2005.

[7] D. Pritchett. BASE: An Acid Alternative. *Queue*, 6(3):48–55, 2008.

[8] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *PODS*, pages 161–171, 1990.

[9] K. A. Ross. A syntactic stratification condition using constraints. In *International Symposium on Logic Programming*, pages 76–90, 1994.

[10] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *Selected Areas in Communications*, 21(1):5–19, 2003.

[11] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *ICDE*, 1986.

[12] S. Vandebogart et al. Labels and Event Processes in the Asbestos Operating System. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[13] W. Vogels. Eventually Consistent. *Communications of the ACM*, 52(1):40–44, 2009.

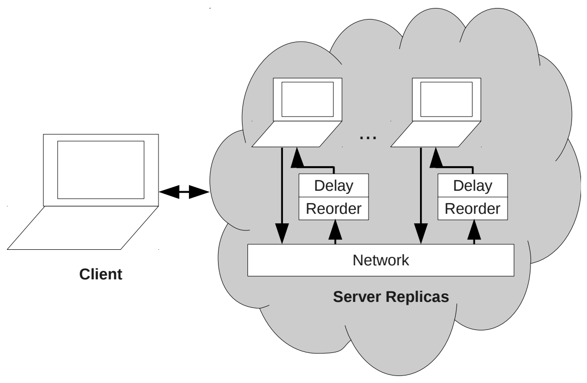

Figure 7: Demo architecture.

7. DEMO PROPOSAL

The goal of our demonstration is to illustrate the basic concepts of the Bloom programming language and the program analysis techniques it supports for reasoning about consistency and coordination in distributed programs. We will demonstrate how to use Bloom to develop several variants of a distributed shopping cart system, similar to the case study described in Section 4. The demo will involve on-the-fly coding in Bloom, graphical representations of our program analysis techniques, and graphical visualizations of the communication behavior in live distributed services.

Although our Bloom code is currently running on Amazon EC2, we are concerned about connectivity at Asilomar and hence plan to run our demo on a local-area cluster of laptops.

We will begin by developing the destructive and disorderly variants of the shopping cart example, initially without any coordination logic. We will run these programs on our cluster, with one laptop as the client—submitting requests to add or remove items from the cart—and the rest of the laptops as server replicas (Figure 7). The client will dispatch each request to a random server replica. For demonstration purposes, we will interpose a *demonic* message delivery module, which operates at each server replica and delivers messages in a random "bad" order (e.g., deletions before insertions, checkouts before all insertions and deletions).

We will display a trace of the cart's execution in real time by visualizing the messages sent by the client and received by each server replica, in the order they are sent or received, and explain the potential hazards associated with different orders. We will then apply our points-of-order program analysis in a graphical form, and show how this technique identifies the possible message ordering inconsistencies we have already demonstrated experimentally.

Finally, following the recommendations of the program analysis tool, we will add coordination logic to the two shopping cart variants, verify consistency of the result in our analysis tool, and run the modified code on the cluster of laptops. We will benchmark the performance of the modified shopping cart programs and visualize the results, demonstrating how the increased coordination in the destructive shopping cart implementation results in reduced performance. To ensure that this effect is visible, as it would be in a distributed cloud environment, we need to ensure realistic latencies between the laptops. The wireless spectrum in Asilomar's conference room may be sufficiently congested to serve this purpose. Alternatively, we may interpose a message delivery module to introduce delay.