Consistency Analysis in Bloom: a CALM and Collected Approach

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

{palvaro, nrc, hellerstein, wrm}@cs.berkeley.edu
University of California, Berkeley

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 program analysis technique that identifies *points of order* in Bloom programs: code locations where programmers may need to inject coordination logic to ensure consistency. We illustrate these ideas with two case studies: a simple key-value store and a distributed shopping cart service.

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., [6, 12, 13, 18, 24]). This approach enables machines to continue operating in the face of 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 connects the theory of non-monotonic logic to 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 present two case studies: a replicated key-value store and a fault-tolerant shopping cart service.

2. CONSISTENCY AND LOGICAL MONOTONICITY (CALM)

In this section we present the 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 the consistent behavior of a program 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 [26]. A sufficient condition for eventual consistency is *order independence*:

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2011.
5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9–12, 2011, Asilomar, California, USA.
.

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 and web search infrastructure. 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 (even with recursion)—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 negation 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 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 counting messages to establish that a majority of the members have agreed to a proposal; Two-Phase Commit requires 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 dataflow dependencies in the code. As students of the logic programming literature will recognize [19, 20, 21], 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. Many refinements along these lines exist, 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 a non-monotonicity 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 a code module with coordination logic actually increases the number of syntactic points of order in a program. To avoid this problem, the coordination module itself must be verified for order independence, either manually or via a refined monotonicity test during analysis. When the verification is done by hand, annotations can inform the analysis tool to skip the module in its analysis, and hence avoid attempts to coordinate the coordination logic.

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 our discussion of shopping carts (Section 5), logic languages and the analysis of points of order can help programmers redesign code to reduce coordination requirements.

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 ordering assumptions implicit in 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 sets. Computation is expressed in logic: an unordered set of declarative rules, each consisting of an unordered conjunction 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.

Unlike earlier efforts such as Prolog, active database languages, and our own Overlog language for distributed systems [16], 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 [3].

The prototype version of Bloom we describe here is embodied in an implementation we call *Bud* (Bloom Under Development). Bud

[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 network address stored in their location specifier.
periodic	A scratch collection of key-value pairs (`id`, `timestamp`). The definition of a periodic collection 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.
interface	A scratch collection specially designated as an interface point between modules.

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**	lhs will include the content of the rhs in the next timestep.
<-	**table**	tuples in the rhs will be absent from the lhs at the start of the next timestep.
<~	**channel**	tuples in the rhs will appear in the (remote) lhs at some non-deterministic future time.

Figure 1: Bloom collection types and operators.

```
 0  module DeliveryProtocol
 1    def state
 2      interface input, :pipe_in,
 3        ['dst', 'src', 'ident'], ['payload']
 4      interface output, :pipe_sent,
 5        ['dst', 'src', 'ident'], ['payload']
 6    end
 7  end

 9  module ReliableDelivery
10    include DeliveryProtocol

12    def state
13      channel :data_chan, ['@dst', 'src', 'ident'], ['payload']
14      channel :ack_chan, ['@src', 'dst', 'ident']
15      table :send_buf, ['dst', 'src', 'ident'], ['payload']
16      periodic :timer, 10
17    end

19    declare
20    def send_packet
21      send_buf <= pipe_in
22      data_chan <~ pipe_in
23    end

25    declare
26    def timer_retry
27      data_chan <~ join([send_buf, timer]).map{|p, t| p}
28    end

30    declare
31    def send_ack
32      ack_chan <~ data_chan.map{|p| [p.src, p.dst, p.ident]}
33    end

35    declare
36    def recv_ack
37      got_ack = join [ack_chan, send_buf],
38                     [ack_chan.ident, send_buf.ident]
39      pipe_sent <= got_ack.map{|a, sb| sb}
40      send_buf <- got_ack.map{|a, sb| sb}
41    end
42  end
```

Figure 2: Reliable unicast messaging in Bloom.

is a domain-specific 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 relatively flat for programmers familiar with modern scripting languages. Bud uses a Ruby-flavored syntax, but this is not fundamental; we have experimented with analogous Bloom embeddings in other languages including Python, Erlang and Scala, and they look similar in structure.

3.1 Bloom Basics

Bloom programs are bundles of declarative statements about collections of "facts" or tuples, similar to SQL views or Datalog rules. A statement can only reference data that is local to a node. Bloom statements 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 non-deterministic 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 [3].

3.2 State in Bloom

Bloom programs manage state using five collection types described in the top of Figure 1. A collection is defined with a relational-style schema of named columns, including an optional subset of those columns that forms a primary key. Line 15 in Figure 2 defines a collection named `send_buf` with four columns `dst`, `src`, `ident`, and `payload`; the primary key is `(dst, src, ident)`. The type system for 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 (including nested Bud collections). In Bud, a tuple in a 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 [23], column values can be manipulated using their own (non-destructive) methods. Bloom also provides for nesting and unnesting of collections using standard Ruby constructs like `reduce` and `flat_map`. Note that collections in Bloom provide set semantics—collections do not contain duplicates.

The persistence of a tuple is determined by the type of the collection that contains the tuple. **scratch** collections are useful for transient data like intermediate results and "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 [3], it is convenient to think operationally as follows: scratch collections are "emptied" before each timestep begins, tables are "stored" collections (similar to tables in SQL), 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

Method	Description
`bc.map`	Takes a code block and returns the collection formed by applying the code block to each element of `bc`.
`bc.flat_map`	Equivalent to `map`, except that any nested collections in the result are flattened.
`bc.reduce`	Takes a `memo` variable and code block, and applies the block to `memo` and each element of `bc` in turn.
`bc.empty?`	Returns true if `bc` is empty.
`bc.include?`	Takes an object and returns true if that object is equal to any element of `bc`.
`bc.group`	Takes a list of grouping columns, a list of aggregate expressions and a code block. For each group, computes the aggregates and then applies the code block to the group/aggregation result.
`join`, `leftjoin`, `outerjoin`, `natjoin`	Methods of the Bud class to compute join variants over BudCollections. `join`, `leftjoin` and `outerjoin` take an array of collections to join, as well as a variable-length list of arrays of join conditions. The natural join `natjoin` takes only the array of BudCollection objects as an argument.

Figure 3: Commonly used methods of the BudCollection class.

into the logical semantics of this non-determinism [3]. 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: the facts in a channel are sent to a remote node via an unreliable transport protocol like UDP; the address of the remote node is indicated by a distinguished column in the channel called the *location specifier* (denoted by the symbol `@`). The definition of a periodic collection instructs the runtime to "inject" facts at regular wall-clock intervals to "drive" further derivations. Lines 13 and 16 in Figure 2 contain examples of channel and periodic definitions, respectively.

The final type of collection is an **interface**, which specifies a connection point between Bloom modules. Interfaces are described in Section 3.4.

3.3 Bloom Statements

Bloom statements are declarative relational expressions that define the contents of derived collections. They can be viewed operationally as specifying the insertion or accumulation of expression results into collections. The syntax is:

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

The bottom of Figure 1 describes the five 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, the lhs of a statement may be referenced recursively in its rhs, or recursion can be defined mutually across statements.

In the Bud prototype, both the lhs and rhs are instances of (a descendant of) a Ruby class called `BudCollection`, which supports several useful methods for manipulating collections (Figure 3).[2] The rhs of a statement typically invokes `BudCollection` methods on one or more collection objects to produce a derived collection. The most commonly used method is `map`, which applies a scalar operation to every tuple in a collection; this can be used to implement relational selection and projection. For example, line 32 of Figure 2 projects the `data_chan` collection to its `src`, `dst`, and `ident` fields. Multiway joins are specified using the `join` method, which takes a list of input collections and an optional list of join conditions. Lines 37–38 of Figure 2 show a join between `ack_chan` and `send_buf`. Syntax sugar for natural joins and outer joins is also provided. `BudCollection` also defines a `group` method similar to SQL's `GROUP BY`, supporting the standard SQL aggregates; for example, lines 15–17 of Figure 14 compute the count of unique `reqid` values for every combination of values for `session`, `item` and `action`.

Bloom statements are specified within method definitions that are flagged with the `declare` keyword (e.g., line 20 of Figure 2). The semantics of a Bloom program are defined by the union of its `declare` methods; the order of statements is immaterial. Dividing statements into multiple methods improves the readability of the program and also allows the use of Ruby's method overriding and inheritance features: because a Bloom class is just a stylized Ruby class, any of the methods in a Bloom class can be overridden by a subclass. We expand upon this idea next.

3.4 Modules and Interfaces

Conventional wisdom in certain quarters says that rule-based languages are untenable for large programs that evolve over time, since the interactions among rules become too difficult to understand. Bloom addresses this concern in two different ways. First, unlike many prior rule-based languages, Bloom is purely declarative; this avoids forcing the programmer to reason about the interaction between declarative statements and imperative constructs. Second, Bloom borrows object-oriented features from Ruby to enable programs to be broken into small modules and to allow modules to interact with one another by exposing narrow interfaces. This aids program comprehension, because it reduces the amount of code a programmer needs to read to understand the behavior of a module.

A Bloom module is a bundle of collections and statements. Like modules in Ruby, a Bloom module can "mixin" one or more other modules via the `include` statement; mixing-in a module imports its collections and statements. A common pattern is to specify an abstract interface in one module and then use the mixin feature to specify several concrete realizations in separate modules. To support this idiom, Bloom provides a special type of collection called an **interface**. An input interface defines a place where a module accepts stimuli from the outside world (e.g., other Bloom modules). Typically, inserting a fact into an input interface results in a corresponding fact appearing (perhaps after a delay) in one of the module's output interfaces.

For example, the DeliveryProtocol module in Figure 2 defines an abstract interface for sending messages to a remote address. Clients use an implementation of this interface by inserting a fact into `pipe_in`; this represents a new message to be delivered. A corresponding fact will eventually appear in the `pipe_sent` output interface; this indicates that the delivery operation has been completed. The ReliableDelivery module of Figure 2 is one possible implementation of the abstract DeliveryProtocol interface—it uses a buffer and acknowledgment messages to delay emitting a `pipe_sent` fact until the corresponding message has been acknowledged by the remote node. Figure 18 in Appendix A contains a different implementation of the abstract DeliveryProtocol. A client program that is indifferent to the details of message delivery can simply interact with the abstract DeliveryProtocol; the particular implementation of this protocol can be chosen independently.

A common requirement is for one module to "override" some of the statements in a module that it mixes in. For example, an OrderedDelivery module might want to reuse the functionality provided by ReliableDelivery but prevent a message with sequence number x from being delivered until all messages with sequence numbers $< x$ have been acknowledged. To support this pattern, Bloom allows an interface defined in another module to be overridden simply by redeclaring it. Internally, both of these redundantly-named interfaces exist in the namespace of the module that declared them, but they

[2]Note that many of these methods are provided by the standard Ruby `Enumerable` module, which `BudCollection` imports.

only need to be referenced by a fully qualified name if their use is otherwise ambiguous. If an input interface appears in the lhs of a statement in a module that declared the interface, it is rewritten to reference the interface with the same name in a mixed-in class, because a module cannot insert into its own input interface. The same is the case for output interfaces appearing in the rhs of statements. This feature allows programmers to reuse existing modules and interpose additional logic in a style reminiscent of superclass invocation in object-oriented languages. We provide an example of interface overriding in Section 4.3.

3.5 Bud Implementation

Bud is 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 2400 lines of Ruby code, developed as a part-time effort over the course of a semester.

A Bud program is just a Ruby class definition. To make it operational, a small amount of imperative Ruby code is needed to create an instance of the class and invoke the Bud `run` method. This imperative code can then be launched on as many nodes as desired (e.g., via the popular Capistrano package for Ruby deployments). As an alternative to the `run` method, the Bud class also provides a `tick` method that can be used to force evaluation of a single timestep; this is useful for debugging Bloom code with standard Ruby debugging tools or for executing a Bud program that is intended as a "one-shot" query.

Because Bud is pure Ruby, some programmers may choose to embed it as a domain-specific language (DSL) within traditional imperative Ruby code. In fact, nothing prevents a subclass of Bud from having both Bloom code in `declare` methods and imperative code in traditional Ruby methods. This is a fairly common usage model for many DSLs. A mixture of declarative Bloom methods and imperative Ruby allows the full range of existing Ruby code—including the extensive RubyGems repositories—to be combined with checkable distributed Bloom programs. The analyses we describe in the remaining sections still apply in these cases; the imperative Ruby code interacts with the Bloom logic in the same way as any external agent sending and receiving network messages.

4. CASE STUDY: KEY-VALUE STORE

In this section, we present two variants of a key-value store (KVS) implemented using Bloom.[3] We begin with an abstract protocol that any key-value store will satisfy, and then provide both single-node and replicated implementations of this protocol. We then introduce a graphical visualization of the dataflow in a Bloom program and use this visualization to reason about the *points of order* in our programs: places where additional coordination may be required to guarantee consistent results.

4.1 Abstract Key-Value Store Protocol

Figure 4 specifies a protocol for interacting with an abstract key-value store. The protocol comprises two input interfaces (representing attempts to insert and fetch items from the store) and a single output interface (which represents the outcome of a fetch operation). To use an implementation of this protocol, a Bloom program can store key-value pairs by inserting facts into `kvput`. To retrieve the value associated with a key, the client program inserts a fact into `kvget` and looks for a corresponding response tuple in `kvget_response`. For both put and get operations, the client must supply a unique request identifier (`reqid`) to differentiate tuples in the event of multiple concurrent requests.

[3]The complete source code for both of the case studies presented in this paper can be found at `http://boom.cs.berkeley.edu/cidr11/`.

```
0  module KVSProtocol
1    def state
2      interface input, :kvput,
3        ['client', 'key', 'reqid'], ['value']
4      interface input, :kvget, ['reqid'], ['key']
5      interface output, :kvget_response,
6        ['reqid'], ['key', 'value']
7    end
8  end
```

Figure 4: Abstract key-value store protocol.

```
0  module BasicKVS
1    include KVSProtocol

3    def state
4      table :kvstate, ['key'], ['value']
5    end

7    declare
8    def do_put
9      kvstate <+ kvput.map{|p| [p.key, p.value]}
10     prev = join [kvstate, kvput], [kvstate.key, kvput.key]
11     kvstate <- prev.map{|b, p| b}
12   end

14   declare
15   def do_get
16     getj = join [kvget, kvstate], [kvget.key, kvstate.key]
17     kvget_response <= getj.map do |g, t|
18       [g.reqid, t.key, t.value]
19     end
20   end
21 end
```

Figure 5: Single-node key-value store implementation.

A module which uses a key-value store but is indifferent to the specifics of the implementation may simply mixin the abstract protocol and postpone committing to a particular implementation until runtime. As we will see shortly, an implementation of the KVSProtocol is a collection of Bloom statements that read tuples from the protocol's input interfaces and send results to the output interface.

4.2 Single-Node Key-Value Store

Figure 5 contains a single-node implementation of the abstract key-value store protocol. Key-value pairs are stored in a persistent table called `kvstate` (line 4). When a `kvput` tuple is received, its key-value pair is stored in `kvstate` at the next timestep (line 9). If the given key already exists in `kvstate`, we want to replace the key's old value. This is done by joining `kvput` against the current version of `kvstate` (line 10). If a matching tuple is found, the old key-value pair is removed from `kvstate` at the beginning of the next timestep (line 11). Note that we also insert the new key-value pair into `kvstate` in the next timestep (line 9); hence, an overwriting update is implemented as an atomic deletion and insertion.

4.3 Replicated Key-Value Store

Next, we extend the basic key-value store implementation to support replication (Figure 6). To communicate between replicas, we use a simple multicast library implemented in Bloom; the source code for this library can be found in Appendix A. To send a multicast, a program inserts a fact into `send_mcast`; a corresponding fact appears in `mcast_done` when the multicast is complete. The multicast library also exports the membership of the multicast group in a table called `members`.

Our replicated key-value store is implemented on top of the single-node key-value store described in the previous section. When a new key is inserted by a client, we multicast the insertion to the other

```
0   module ReplicatedKVS
1     include BasicKVS
2     include MulticastProtocol

4     def state
5       interface input, :kvput,
6         ['client', 'key', 'reqid'], ['value']
7     end

9     declare
10    def replicate
11      send_mcast <= kvput.map do |k|
12        unless members.include? [k.client]
13          [k.reqid, [@local_addr, k.key, k.reqid, k.value]]
14        end
15      end
16    end

18    declare
19    def apply_put
20      kvput <= mcast_done.map{|m| m.payload}

22      kvput <= pipe_chan.map do |d|
23        if d.payload.fetch(1) != @local_addr
24          d.payload
25        end
26      end
27    end
28  end
```

Figure 6: Replicated key-value store implementation.

```
0  class RealizedReplicatedKVS < Bud
1    include ReplicatedKVS
2    include SimpleMulticast
3    include BestEffortDelivery
4  end

6  kvs = RealizedReplicatedKVS.new("localhost", 12345)
7  kvs.run
```

Figure 7: A fully specified key-value store program.

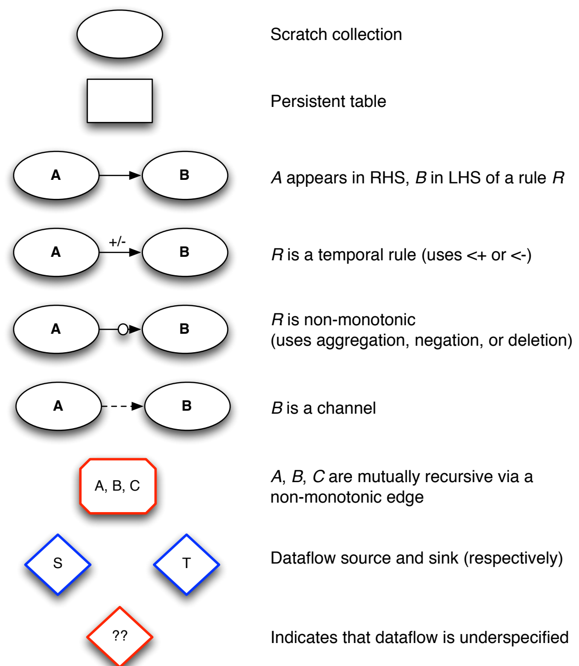

Figure 8: Visual analysis legend.

replicas (lines 11–15). To avoid repeated multicasts of the same inserted key, we avoid multicasting updates we receive from another replica (line 12). We apply an update to our local `kvstate` table in two cases: (1) if a multicast succeeds at the node that originated it (line 20) (2) whenever a multicast is received by a peer replica (lines 22–26). Note that `@local_addr` is a Ruby instance variable defined by Bud that contains the network address of the current Bud instance.

In Figure 6 ReplicatedKVS wants to "intercept" `kvput` events from clients, and only apply them to the underlying BasicKVS module when certain conditions are met. To achieve this, we "override" the declaration of the `kvput` input interface as discussed in Section 3.4 (lines 5–6). In ReplicatedKVS, references to `kvput` appearing in the lhs of statements are resolved to the `kvput` provided by BasicKVS, while references in the rhs of statements resolve to the local `kvput`. As described in Section 3.4, this is unambiguous because a module cannot insert into its own input or read from its own output interfaces.

Figure 7 combines ReplicatedKVS with a concrete implementation of MulticastProtocol and DeliveryProtocol. The resulting class, a subclass of Bud, may be instantiated and run as shown in lines 6 and 7.

4.4 Predicate Dependency Graphs

Now that we have introduced two concrete implementations of the abstract key-value store protocol, we turn to analyzing the properties of these programs. We begin by describing the graphical dataflow representation used by our analysis. In the following section, we discuss the dataflow graphs generated for the two key-value store implementations.

A Bloom program may be viewed as a dataflow graph with external input interfaces as sources, external output interfaces as sinks, collections as internal nodes, and rules as edges. This graph represents the dependencies between the collections in a program and is generated automatically by the Bud interpreter. Figure 8 contains a list of the different symbols and annotations in the graphical visualization; we provide a brief summary below.

Each node in the graph is either a collection or a cluster of collections; tables are shown as rectangles, ephemeral collections (scratch, periodic 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 statement that references A in the rhs, either directly or through a join expression. An edge is annotated based on the operator symbol in the statement. If the statement uses the <+ or <- operators, the edge is marked with "+/−". This indicates that facts traversing the edge "spend" a timestep to move from the rhs to the lhs. Similarly, if the statement uses the <~ operator, the edge is a dashed line—this indicates that facts from the rhs appear at the lhs at a non-deterministic future time. If the statement involves a non-monotonic operation (aggregation, negation, or deletion via the <- operator), then the edge is marked with a white circle. To make the visualizations more readable, any strongly connected component marked with both a circle and a +/− edge is collapsed into an octagonal "temporal cluster," which can be viewed abstractly as a single, non-monotonic node in the dataflow. Any non-monotonic edge in the graph is a *point of order*, as are all edges incident to a temporal cluster, including their implicit self-edge.

4.5 Analysis

Figure 9 presents a visual representation of the abstract key-value store protocol. Naturally, the abstract protocol does not specify a connection between the input and output events; this is indicated in the diagram by the red diamond labeled with "??", denoting an underspecified dataflow. A concrete realization of the key-value

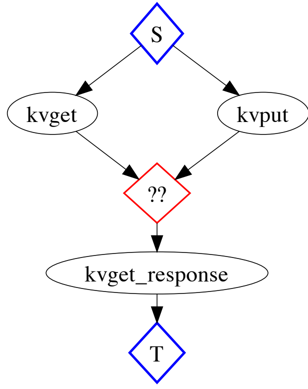

Figure 9: Visualization of the abstract key-value store protocol.

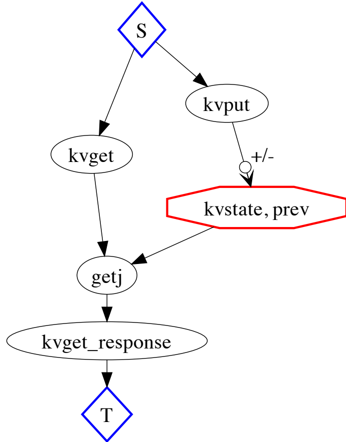

Figure 10: Visualization of the single-node key-value store.

store protocol must, at minimum, supply a dataflow that connects an input interface to an output interface.

Figure 10 shows the visual analysis of the single-node KVS implementation, which supplies a concrete dataflow for the unspecified component in the previous graph. `kvstate` and `prev` are collapsed into a red octagon because they are part of a strongly connected component in the graph with both negative and temporal edges. Any data flowing from `kvput` to the sink must cross at least one non-monotonic point of order (at ingress to the octagon) and possibly an arbitrary number of them (by traversing the dependency cycle collapsed into the octagon), and any path from `kvget` to the sink must join state potentially affected by non-monotonicity (because `kvstate` is used to derive `kvget_response`).

Reviewing the code in Figure 5, we see the source of the non-monotonicity. The contents of `kvstate` may be defined via a "destructive" update that combines the previous state and the current input from `kvput` (lines 9–11 of Figure 5). Hence the contents of `kvstate` may depend on the order of arrival of `kvput` tuples.

5. CASE STUDY: SHOPPING CART

In this section, we develop two different designs for a distributed shopping-cart service in Bloom. In a shopping cart system, clients add and remove items from their shopping cart. To provide fault tolerance and persistence, the content of the cart is stored by a

```
0   module CartProtocol
1     def state
2       channel :action_msg,
3         ['@server', 'client', 'session', 'reqid'],
4         ['item', 'action']
5       channel :checkout_msg,
6         ['@server', 'client', 'session', 'reqid']
7       channel :response_msg,
8         ['@client', 'server', 'session'], ['contents']
9     end
10  end

12  module CartClientProtocol
13    def state
14      interface input, :client_action,
15        ['server', 'session', 'reqid'], ['item', 'action']
16      interface input, :client_checkout,
17        ['server', 'session', 'reqid']
18      interface output, :client_response,
19        ['client', 'server', 'session'], ['contents']
20    end
21  end
```

Figure 11: Abstract shopping cart protocol.

```
0   module CartClient
1     include CartProtocol
2     include CartClientProtocol

4     declare
5     def client
6       action_msg <~ client_action.map do |a|
7         [a.server, @local_addr, a.session, a.reqid, a.item, a.action]
8       end
9       checkout_msg <~ client_checkout.map do |a|
10        [a.server, @local_addr, a.session, a.reqid]
11      end
12      client_response <= response_msg
13    end
14  end
```

Figure 12: Shopping cart client implementation.

collection of server replicas. Once a client has finished shopping, they perform a "checkout" request, which returns the final state of their cart.

After presenting the abstract shopping cart protocol and a simple client program, we implement a "destructive," state-modifying shopping cart service that uses the key-value store introduced in Section 4. Second, we illustrate a "disorderly" cart that accumulates updates in a set-wise fashion, summarizing 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.

5.1 Shopping Cart Client

An abstract shopping cart protocol is presented in Figure 11. Figure 12 contains a simple shopping cart client program: it takes client operations (represented as `client_action` and `client_checkout` facts) and sends them to the shopping cart service using the CartProtocol. We omit logic for clients to choose a cart server replica; this can be based on simple policies like round-robin or random selection, or via more explicit load balancing.

5.2 "Destructive" Shopping Cart Service

We begin with a shopping cart service built on a key-value store. 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 currently 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

```
0  module DestructiveCart
1    include CartProtocol
2    include KVSProtocol

4    declare
5    def do_action
6      kvget <= action_msg.map{|a| [a.reqid, a.key]}

8      kvput <= action_msg.map do |a|
9        if a.action == "A"
10         unless kvget_response.map{|b| b.key}.include? a.session
11           [a.server, a.client, a.session, a.reqid, [a.item]]
12         end
13       end
14     end

16     old_state = join [kvget_response, action_msg],
17                      [kvget_response.key, action_msg.session]
18     kvput <= old_state.map do |b, a|
19       if a.action == "A"
20         [a.server, a.client, a.session,
21          a.reqid, b.value.push(a.item)]
22       elsif a.action == "D"
23         [a.server, a.client, a.session,
24          a.reqid, delete_one(b.value, a.item)]
25       end
26     end
27   end

29   declare
30   def do_checkout
31     kvget <= checkout_msg.map{|c| [c.reqid, c.session]}
32     lookup = join [kvget_response, checkout_msg],
33                   [kvget_response.key, checkout_msg.session]
34     response_msg <~ lookup.map do |r, c|
35       [c.client, c.server, c.session, r.value]
36     end
37   end
38 end
```

Figure 13: Destructive cart implementation.

```
0  module DisorderlyCart
1    include CartProtocol

3    def state
4      table :cart_action, ['session', 'item', 'action', 'reqid']
5      table :action_cnt, ['session', 'item', 'action'], ['cnt']
6      scratch :status, ['server', 'client', 'session', 'item'],
7                       ['cnt']
8    end

10   declare
11   def do_action
12     cart_action <= action_msg.map do |c|
13       [c.session, c.item, c.action, c.reqid]
14     end
15     action_cnt <= cart_action.group(
16       [cart_action.session, cart_action.item, cart_action.action],
17       count(cart_action.reqid))
18   end

20   declare
21   def do_checkout
22     del_items = action_cnt.map{|a| a.item if a.action == "Del"}
23     status <= join([action_cnt, checkout_msg]).map do |a, c|
24       if a.action == "Add" and not del_items.include? a.item
25         [c.client, c.server, a.session, a.item, a.cnt]
26       end
27     end

29     status <= join([action_cnt, action_cnt,
30                     checkout_msg]).map do |a1, a2, c|
31       if a1.session == a2.session and a1.item == a2.item and
32          a1.session == c.session and
33          a1.action == "A" and a2.action == "D"
34         [c.client, c.server, c.session, a1.item, a1.cnt - a2.cnt]
35       end
36     end

38    response_msg <~ status.group(
39       [status.client, status.server, status.session],
40       accum(status.cnt.times.map{status.item}))
41   end
42 end
```

Figure 14: Disorderly cart implementation.

that reflects the effect of the update. Deletion requests are ignored if the item they refer to does not exist in the cart.

Figure 13 shows the Bloom code for this design. The `kvput` collection is provided by the abstract KVSProtocol described in Section 4. Our shopping cart service would work with any concrete realization of the KVSProtocol; we will choose to use the replicated key-value store (Section 4.3) to provide fault-tolerance.

When client actions arrive from the CartClient, the cart service checks to see if there is a record in the key-value store associated with the client's session. If no record is found (i.e., this is the first operation for a new session), then lines 9–13 generate an entry for the new session in `kvstate`. Otherwise, the join conditions in line 17 are satisfied and lines 19–25 "replace" the value in the key-value store with an updated set of items for this session; this uses the built-in overwriting capability provided by the key-value store. When a `checkout_msg` appears at a server replica, the key-value store is queried to retrieve the cart state associated with the given session (lines 31–35), and the results are returned to the client.

5.3 "Disorderly" Shopping Cart Service

Figure 14 shows an alternative shopping cart implementation, in which updates are monotonically accumulated in a set, and summed up only at checkout. Lines 12–14 insert client updates into the persistent table `cart_action`. Lines 15–17 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 22–27 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 29–36 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 34), or simply the number of additions if there are no deletions (line 25). Upon the appearance of a `checkout_msg`, the replica returns a `response_msg` to the client containing the final quantity (lines 38–40). Because the CartClient expects the cart to be returned as an array of items on checkout, we use the `accum` aggregate function to nest the set of items into an array.

5.4 Analysis

Figure 15 presents the analysis of the "destructive" shopping cart variant. Note that because all dependencies are analyzed, collections defined in mixins but not referenced in the code sample (e.g., `pipe_chan`, `member`) also appear in the graph. Although there is no syntactic non-monotonicity in Figure 13, the underlying key-value store uses the non-monotonic `<-` operator to model updateable state. Thus, while the details of the implementation are encapsulated by the key-value store's abstract interface, its points of order resurface in the full-program analysis. Figure 15 indicates that there are points of order between `action_msg`, `member`, and the temporal cluster. This figure also 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*. The programmer can achieve this coordination by supplying a "reliable"

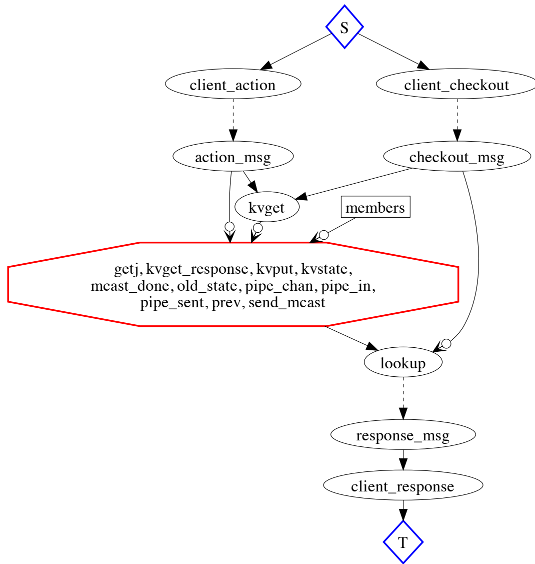

Figure 15: Visualization of the destructive cart program.

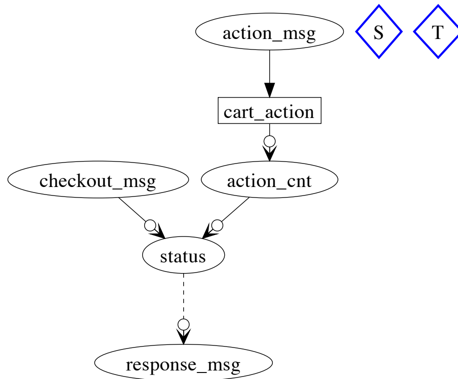

Figure 16: Visualization of the core logic for the disorderly cart.

implementation of multicast that awaits acknowledgements from all replicas before reporting completion: this fine-grained coordination is akin to "eager replication" [9]. Unfortunately, it would incur the latency of a round of messages per server per client update, decrease throughput, and reduce availability 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 from our analysis of the disorderly cart service. Figure 16 shows a visualization of the core logic of the disorderly cart module presented in Figure 14. This program is not complete: its inputs and outputs are channels rather than interfaces, so the dataflow from source to sink is not completed. To complete this program, we must mixin code that connects input and output interfaces to `action_msg`, `checkout_msg`, and `response_msg`, as

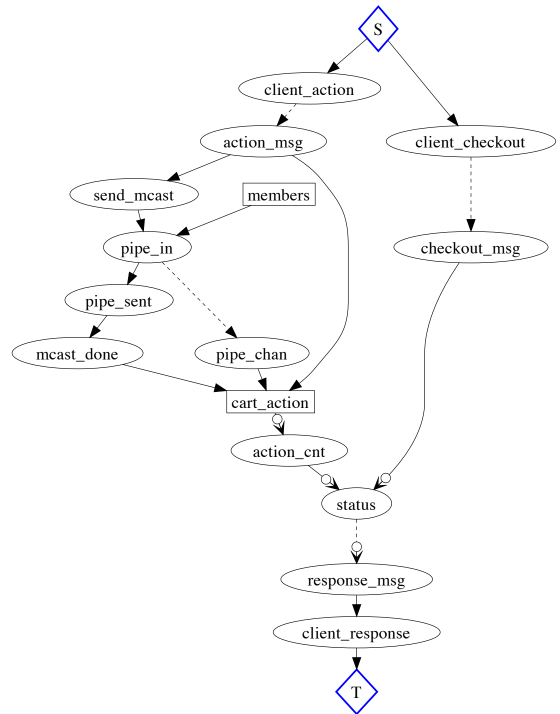

Figure 17: Visualization of the complete disorderly cart program.

the CartClient does (Figure 12). Note that the disorderly cart has points of order on all paths but there are no cycles.

Figure 17 shows the analysis for a complete implementation that mixes in both the client code and logic to replicate the `cart_action` table via best-effort multicast (see Figure 20 in Appendix A for the corresponding source code). Note 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. Additionally, using the `accum` aggregate adds a point of order to the end of the dataflow, between `status` and `response_msg`. Although the accumulation of shopping actions is monotonic, summarization of the cart state requires us to ensure that there will be no further cart actions.

Comparing Figure 15 and Figure 17, we can see that the disorderly cart requires less coordination than the destructive cart: 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 [13].

5.5 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 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 the destructive cart program, 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 dataflow analysis can help a developer 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.

6. TOLERATING INCONSISTENCY

In the previous section we showed how to identify points of order: code locations that are sensitive to non-deterministic input ordering. We then demonstrated how to resolve the non-determinism by introducing coordination. However, in many cases adding additional coordination is undesirable due to concerns like latency and availability. In these cases, Bloom's point-of-order analysis can assist programmers with the task of *tolerating inconsistency*, rather than resolving it via coordination. A notable example of how to manage inconsistency is presented by Helland and Campbell, who reflect on their experience programming with patterns of "memories, guesses and apologies" [13]. We provide a sketch here of ideas for converting these patterns into developer tools in Bloom.

"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 [22, 25]. 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.

Most of these patterns can be implemented as automatic program rewrites. We envision building a system that facilitates running low-latency, "guess"-driven decision making in the foreground, and expensive but consistent logic as a background process. When the background process detects an inconsistency in the results produced by the foreground system (e.g., because a "guess" turns out to be mistaken), it can then take corrective action by generating an "apology." Importantly, both of these subsystems are implementations of the same high-level design, except with different consistency and coordination requirements; hence, it should be possible to synthesize both variants of the program from the same source code. Throughout this process—making calculated "guesses," storing appropriate "memories," and generating the necessary "apologies"—we see significant opportunities to build scaffolding and tool support to lighten the burden on the programmer.

Finally, 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.

7. RELATED WORK

Systems with loose consistency requirements have been explored in depth by both the systems and database management communities (e.g., [6, 8, 9, 24]); we do not attempt to provide an exhaustive survey here. The shopping cart case study in Section 5 was motivated by the Amazon Dynamo paper [11], as well as the related discussion by Helland and Campbell [13].

The Bloom language is inspired by earlier work that attempts to integrate databases and programming languages. This includes early research such as Gem [27] and more recent object-relational mapping layers such as Ruby on Rails. Unlike these efforts, Bloom is targeted at the development of both distributed infrastructure and distributed applications, so it does not make any assumptions about the presence of a database system "underneath". Given our prototype implementation in Ruby, it is tempting to integrate Bud with Rails; we have left this for future work.

There is a long history of attempts to design programming languages more suitable to parallel and distributed systems; for example, Argus [15] and Linda [7]. Again, we do not hope to survey that literature here. More pragmatically, Erlang is an oft-cited choice for distributed programming in recent years. Erlang's features and design style encourage the use of asynchronous lightweight "actors." As mentioned previously, we did a simple Bloom prototype DSL in Erlang (which we cannot help but call "Bloomerlang"), and there is a natural correspondence between Bloom-style distributed rules and Erlang actors. However there is no requirement for Erlang programs to be written in the disorderly style of Bloom. It is not obvious that typical Erlang programs are significantly more amenable to a useful points-of-order analysis than programs written in any other functional language. For example, ordered lists are basic constructs in functional languages, and without program annotation or deeper analysis than we need to do in Bloom, any code that modifies lists would need be marked as a point of order, much like our destructive shopping cart. We believe that Bloom's "disorderly by default" style encourages order-independent programming, and we know that its roots in database theory helped produce a simple but useful program analysis technique. While we would be happy to see the analysis "ported" to other distributed programming environments, it may be that design patterns using Bloom-esque disorderly programming are the natural way to achieve this.

Our work on Bloom bears a resemblance to the Reactor language [5]. Both languages target distributed programming and are grounded in Datalog. Like many other rule languages including our earlier work on Overlog, Reactor updates mutable state in an operational step "outside Datalog" after each fixpoint computation. By contrast, Bloom is purely declarative: following Dedalus, it models updates as the logical derivation of immutable "versions" of collections over time. While Bloom uses a syntax inspired by object-oriented languages, Reactor takes a more explicitly agent-oriented approach. Reactor also includes synchronous coupling between agents as a primitive; we have opted to only include asynchronous communication as a language primitive and to provide synchronous coordination between nodes as a library.

Another recent language related to our work is Coherence [4], which also embraces "disorderly" programming. Unlike Bloom, Coherence is not targeted at distributed computing and is not based on logic programming.

8. CONCLUSION AND FUTURE WORK

In this paper we make three main contributions. First, we present the CALM principle, which connects the notion of eventual consistency in distributed programming to theoretical foundations in

database theory. Second, we show that we can bring that theory to bear on the practice of software development via "disorderly" programming patterns, complemented with automatic analysis techniques for identifying and managing a program's points of order in a principled way. Finally, we present our Bloom prototype as an example of a practically-minded disorderly and declarative programming language, with an initial implementation as a domain-specific language within Ruby.

We plan to extend the work described in this paper in several directions. First, we are building a more mature Bloom language environment, including a library of modules for distributed computing. We intend to compose those modules to implement a number of variants of distributed systems. The design of Bloom itself was motivated by our experience implementing scalable services and protocols in Overlog [1, 2], and this practice of system/language co-design continues to be part of our approach. Second, we hope to expand our suite of analysis techniques to address additional important properties in distributed systems, including idempotency and invertability of interfaces. Third, we are hopeful that the logic foundation of Bloom will enable us to develop better tools and techniques for the debugging and systematic testing of distributed systems under failure and security attacks, perhaps drawing on recent work on this topic [10, 17]. Finally, we are working to formally tighten our ideas connecting non-monotonic logic, distributed coordination, and consistency of distributed programs [14].

Acknowledgments

We would like to thank Ras Bodík, Kuang Chen, Haryadi Gunawi, Dmitriy Ryaboy, Russell Sears, and the anonymous reviewers for their helpful comments. This work was supported by NSF grants 0917349, 0803690, 0722077, 0713661 and 0435496, Air Force Office of Scientific Research award 22178970-41070-F, the Natural Sciences and Engineering Research Council of Canada, and gifts from Yahoo Research, IBM Research and Microsoft Research.

9. REFERENCES

[1] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *EuroSys*, 2010.

[2] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I Do Declare: Consensus in a Logic Language. *SIGOPS Oper. Syst. Rev.*, 43:25–30, January 2010.

[3] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in Time and Space. In *Proc. Datalog 2.0 Workshop (to appear)*, 2011.

[4] J. Edwards. Coherent Reaction. In *OOPSLA*, 2009.

[5] J. Field, M.-C. Marinescu, and C. Stefansen. Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications. *Theoretical Computer Science*, 410(2-3), February 2009.

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

[7] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.

[8] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.

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

[10] H. S. Gunawi et al. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI (to appear)*, 2011.

[11] D. Hastorun et al. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, 2007.

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

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

[14] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[15] B. Liskov. Distributed programming in Argus. *CACM*, 31:300–312, 1988.

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

[17] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. Secureblox: customizable secure distributed data processing. In *SIGMOD*, 2010.

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

[19] T. C. Przymusinski. *On the Declarative Semantics of Deductive Databases and Logic Programs*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[20] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *PODS*, 1990.

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

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

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

[24] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

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

[26] W. Vogels. Eventually Consistent. *CACM*, 52(1):40–44, 2009.

[27] C. Zaniolo. The database language GEM. In *SIGMOD*, 1983.

APPENDIX

A. ADDITIONAL SOURCE CODE

Figures 18, 19 and 20 contain the remainder of the Bloom code used in this paper: best-effort protocols for unicast and multicast messaging, and the complete program for the replicated disorderly cart described in Section 5.

```
0  module BestEffortDelivery
1    include DeliveryProtocol

3    def state
4      channel :pipe_chan,
5        ['@dst', 'src', 'ident'], ['payload']
6    end

8    declare
9    def snd
10     pipe_chan <~ pipe_in
11   end

13   declare
14   def done
15     pipe_sent <= pipe_in
16   end
17 end
```

Figure 18: Best-effort unicast messaging in Bloom.

```
0  module MulticastProtocol
1    def state
2      table :members, ['peer']
3      interface input, :send_mcast, ['ident'], ['payload']
4      interface output, :mcast_done, ['ident'], ['payload']
5    end
6  end

8  module SimpleMulticast
9    include MulticastProtocol
10   include DeliveryProtocol

12   declare
13   def snd_mcast
14     pipe_in <= join([send_mcast, members]).map do |s, m|
15       [m.peer, @local_addr, s.ident, s.payload]
16     end
17   end

19   declare
20   def done_mcast
21     mcast_done <= pipe_sent.map{|p| [p.ident, p.payload]}
22   end
23 end
```

Figure 19: A simple unreliable multicast library in Bloom.

```
0  class ReplicatedDisorderlyCart < Bud
1    include DisorderlyCart
2    include SimpleMulticast
3    include BestEffortDelivery

5    declare
6    def replicate
7      send_mcast <= action_msg.map do |a|
8        [a.reqid, [a.session, a.item, a.action, a.reqid]]
9      end
10     cart_action <= pipe_chan.map{|c| c.payload }
11   end
12 end
```

Figure 20: The complete disorderly cart program.