Monadic Logs for Collaborative Web Applications

Sumit Agarwal, Daniel Bellinger, Oliver Kennedy, Ankur Upadhyay, Lukasz Ziarek
SUNY Buffalo
{sumitaga, danielbe, okennedy, ankurupa, lziarek}@buffalo.edu

Abstract

Cloud based web-applications are quickly becoming common in modern society. A new class of such applications, collaborative cloud applications, are gaining in popularity as they greatly improve remote collaboration. Most of these applications use a log structure as a coordination mechanism for shared application state. Such structures typically store the entire application state as well as deltas (changes sets) while the application runs. In this paper we propose a monadic, dependency-aware, self-cleaning log structure for collaborative cloud applications, which we refer to as a *monadic log*. This structure provides a rich set of analytical tools to support a variety of log transformations and rewrites. For example, the garbage collection mechanisms already present in any managed language will automatically bound the memory footprint of a monadic log. Moreover, a monadic log substantially eases the computational and bandwidth burdens of a server infrastructure when compared with traditional log structures.

1. INTRODUCTION

Over the past several years, many web and cloud based versions of classic desktop applications have been released (*e.g.* Google Docs). A natural consequence of this migration to the cloud is that applications have become collaborative, with multiple clients simultaneously accessing the same application state. Although these *collaborative cloud applications* are structured using a client/server model, the core functionality of the application is typically built into the client itself. The server's roles are to relay state updates, provide coordination, and persist state.

Although such features (pub/sub, atomic broadcast, and databases respectively) are independently straightforward, and well known solutions exist for each, developers still actively expend substantial effort to build scalable, robust, and efficient server infrastructures for their collaborative applications. Much of this effort goes towards efficiently supporting specialized persistence-layer functionality. Common examples include *recover*ing from transient link failures, such as when the host platform wakes from sleep mode, and *revert*ing undesired updates made by a peer. Supporting such features requires a tight coupling between the persistence infrastructure and the application's semantics.

In this paper we introduce the Laasie[1] data management system, a prototype persistence, communication, and coordination infrastructure for collaborative applications. Laasie clients are application front-ends, each maintaining and mediating user interactions with a local replica of the global application state. Clients post state updates to the Laasie infrastructure, which defines a canonical order over updates that it receives, and ensures delivery updates to all participating clients.

A key feature of Laasie's communications and coordination layer is that it is not stateful. In lieu of connection state, clients actively request *update monads* from the infrastructure. Update monads are small program fragments that express the *delta* between a client's local state replica and the global state; An update monad computes the current global state from the client's local state replica. Laasie is able to generate update monads efficiently, using a novel datastructure that we call a *monadic log*.

Monadic logs encode application state updates in a functional form, thereby storing *computation* rather than effects resulting from the update. By encoding updates as functions in a general purpose composable data manipulation language that is amenable to static analysis, monadic logs enable a broad new class of queries and manipulations over application state. For example, an application-specific policy might decide whether the infrastructure should choose to fuse multiple updates together in one bulk state update (*e.g.*, sending an entire image file), or to send smaller, domain-specific transformations (*e.g.*, brighten all pixels by 10%). As another example, individual updates can be deleted without altering the semantics of subsequent updates, effectively rewriting the history of an application.

Through the functional state representation employed by the monadic log, Laasie gains several other important benefits: (1) Applications are entirely encapsulated in the client, allowing developers to focus their efforts on a single codebase. (2) Application semantics are explicit, improving the system's ability to automate reconciliation of conflicting updates. (3) No code evaluation or query processing is actually required from the infrastructure, lowering overheads and limiting security risks.

Sixteenth International Workshop on the Web and Databases (WebDB 2013),
June 23, 2013 - New York, NY, USA.

[1]Log-As-A-Service InfrastructurE

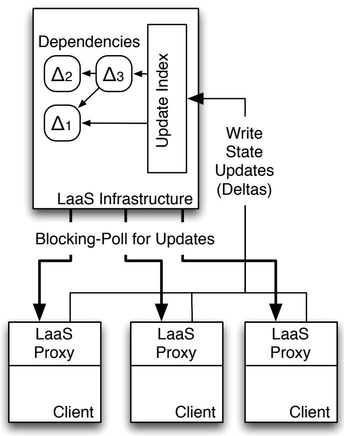

Figure 1: An application deployed in the Laasie ecosystem: The Laasie infrastructure maintains a canonical replica of the state as a DAG of delta procedures. A proxy library in each application frontend client maintains state replicas.

Concretely, we make the following contributions.

1. We present the design of the Laasie data management system for collaborative cloud applications.

2. We introduce the monadic log datastructure, a purely functional representation of an application's state, and discuss its benefits.

3. We describe BarQL, a monad algebra-based state manipulation language designed to synergize with monadic logs.

4. We show preliminary experiments that demonstrate the feasibility of infrastructures based on monadic logs.

2. SYSTEM OVERVIEW

The Laasie data management system is designed to provide communication, coordination, persistence, and logging functionality for collaborative applications. An application in Laasie's ecosystem consists of multiple application clients connected to a central Laasie server infrastructure. The infrastructure maintains a canonical instance of the application's state. The infrastructure provides each client with a *view* of this canonical state, and the ability to write *updates* to that state. This architecture is presented in Figure 1, and can be summarized in the following two methods that it exposes to clients:

```
typedef Value (Monad)(Value v);
int write(Monad update);
Pair<Monad,int>  poll(View v, int ts);
```

Applications *write* state updates functionally, as *monads* that transform the prior version of the application state into a new version. To allow the infrastructure to remain stateless, clients *poll* for updates since their last request.

Laasie assigns a monotonically increasing timestamp to each update written (this timestamp is returned by the write operation). Our initial implementation assigns timestamps using a centralized service. Decentralized approaches such as those of the Isis [3] or Percolator [17] systems complement Laasie nicely, and can be adopted to improve scalability.

Each state version is identified by the timestamp of the most recent update applied to that version. When reading, the client obtains an update function that transforms the client's local state version, identified by the timestamp provided as an argument to poll, to the most recent version of the state, identified by the timestamp returned by poll. As an optimization, if no updates are available since the client's last request, the poll request blocks until the next relevant update occurs.

The primary purpose of the read operation is to provide the client with an update monad that can be applied to the client's current view to update it to be consistent with the most recent state. The use of a monotonically increasing timestamp for updates brings to mind log-based structures common in this class of applications. This similarity is intentional. A write operation is effectively a log-append, timestamps are pointers into the log, and read operations corresponds to scans through the log.

The resulting log provides an abstraction, not just for efficient updates, but also for meta-analysis of the application state. As we will soon see, Laasie's log abstraction is a simple, but extremely powerful primitive for querying, monitoring, and dynamically rewriting application state.

3. LOGS

Logging and related techniques frequently appear in server-side persistence layers for cloud-based web-applications. Keeping prior versions of application state makes it feasible to revert unwanted changes (*e.g.*, [22]), recover from transient link failures (*e.g.*, [11]), or to provide an official record of how state changes occur over time (*e.g.,* [18]). The structure and design of optimal log for each application varies with the application's semantics, feature requirements, and even its expected workload. Decisions related to checkpointing, garbage collection, and strategies for indexing the log affect server/client performance, memory usage, throughput, bandwidth, and are crucial to the design of a performant application.

As a simple example, consider an application that needs infrastructure support for recovery from transient link failures. If the application state is small and/or changes rapidly (*e.g.*, a chat client with a short history), the client can be sent the full application state. If the application state is large and/or changes infrequently (*e.g.*, a text document), it is more efficient to send the client a set of changes, or *deltas.*

These two application styles lean towards opposite extremes in the space of log design. (1) Flat logs, logs which store the most recent state of the application in a monolithic fashion, trivially support recoverability by simply sending the most up-to-date version of the state to a reconnecting client. This approach has a high bandwidth cost, and requires the server to be able to process state updates. (2) Write logs, logs that store singleton or batch updates, have a lower bandwidth cost for recovery, as clients only need a set of deltas, but have substantially higher costs for fresh reads, where the client requires all deltas.

In both of these log designs however, state is a conceptual mapping between objects and their concrete values; The logs store the *results* of the computation that the clients perform via the cloud application.

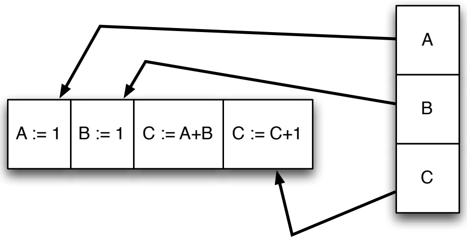

Figure 2: The figure depicts a classic write log structure whose entries are computations with a common optimization: an index from variables to entries.

We propose a more radical log structure that instead stores state as a sequence of *update monads*. In a monadic log, the server does not materialize state, but instead stores state updates as functions that encode deltas on state. Storing state as a computation instead of explicitly makes the log aware of update semantics. Static analysis on update monads leads to efficient support for semantics driven application features. For example, tracking the read and write sets of each monad allows log entries unnecessary for recovery to be quickly identified and discarded.

3.1 Monadic Logs

Unlike traditional logs, a monadic log stores *computation* instead of values. Figure 2 shows a monadic write log. Like a traditional log, the primary mechanism by which clients mutate a monadic log is to append updates to the tail of the log, the `write` operation presented in Section 2.

The first entry conceptually adds a variable A to the shared state and sets that variable to the value 1. Similarly, the second entry in the log encodes the addition of a variable B to the shared state and sets that variable to value 1. The third entry is a computation that reads variables A and B from the state and assigns the results of the addition of those variables to a variable C. The last entry updates the values stored in C by incrementing the current value by one.

To reify this log into a materialized application state, the infrastructure or client need simply to execute the computation stored in the entries. The client makes a read request, as per the `poll` operation presented in Section 2. The infrastructure identifies all log updates relevant to the client's view and produces a single update monad by composing all updates in the sequence. Depending on the desired tradeoff between compute and bandwidth costs, the infrastructure may optimize the resulting monad, effectively reifying portions of the updated state as necessary. Traditional log optimizations such as skip-lists, indexes, and garbage collection can also be applied, as shown in Figures 2 and 3.

3.2 Analysis of Monadic Logs

Storing raw computations in the log rather than the effects of those computations gives monadic logs a great deal of flexibility. The collection of monads can be executed to reify the log into a materialized state. In general, monadic logs provide an expressive representation for dynamically inferring security properties, monitoring, and self-healing. By querying or rewriting individual log entries, both the server and clients can assert and/or effect a broad range of policies or optimizations.

The advantage of expressing the log as a collection of computations is that the structure itself can encode computational dependencies, dependencies between state fragments, or variables, and the computations necessary to execute to calculate the value stored in that variable. By exploiting inherent features of monadic logs we can achieve:

Dynamic Honey Potting. In today's collaborative world, a malicious user can gain access to an authorized session of an application very easily. Monadic logs can revert and flag the malicious changes as well as transitive changes dependent upon them, once the malicious activities are detected through application-agnostic mechanisms. Monadic logs also allow replication of session state in an on-demand fashion (*i.e.* when a honeypot is required), and seamless reintegration of session replicas *e.g.* when a honey potted user is cleared of wrongdoing). Information that the malicious user receives through the honeypot session is limited and/or falsified. More importantly the user's actions are applied only to the honeypot environment, thus maintaining the integrity of the original authorized session and resources.

Obfuscated Collaboration. The log structure of the monadic logs provides the capability to present dynamic views of a document to different users as predicted on a security policy. Different users can edit the same document simultaneously, however, each user has a unique view of the document and/or data depending on their clearance level. Thus, each low-clearance user will have certain fragments of the document obscured, obfuscated, or removed entirely. This not only allows for safe collaboration without duplication, but inherently provides a *provenance* mechanism.

Tracking Provenance and Reconstructing Data. A monadic log natively provides provenance information without requiring any additional metadata, or additional overheads. This information can be used to assert properties about the log and/or improve security. Furthermore, this information can be used to reconstruct the effects of the computation (e.g., in hypothetical scenarios where the initial values are different).

Retroactive Policy Enforcement. In a scenario where access policies needs to be enforced by deleting log entries that violate the access policy, out of order undo's can be done very easily on monadic log structure. As each log entry encodes the full semantics of its update; effects of deleting or modifying an update are immediately propagated through all subsequent updates.

Fluid Computation. Monadic logs make it easy to offload compute-intensive tasks to the client to reduce server load, or move such tasks from client to server if the client's capabilities are limited. The server itself, may be distributed, with portions of the monadic log spread amongst many machines. Techniques like memoization and self-adjusting computation can be leveraged to expedite state reification for distributed servers.

4. BARQL

To provide rewriting, monitoring, and querying capabilities over the BarQL log, the language underpinning the log must be amenable to static analysis. As an example of a language suitable for this purpose, we now provide a high-level overview of Bar_{QL} [12], a state update language built around an algebra of composition. Bar_{QL}'s composition algebra makes it possible to efficiently analyze sequences of

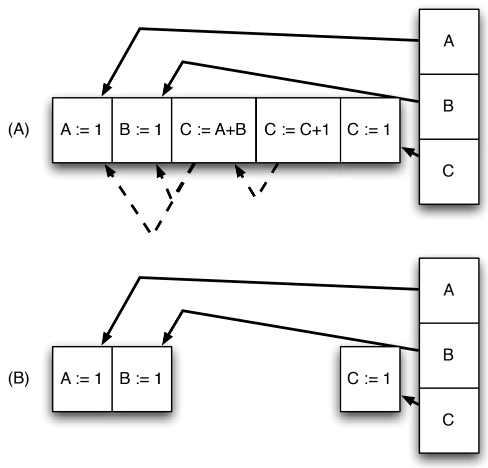

Figure 3: (A) The Laasie log structure, containing an index as well as a dependency chain. Index pointers are shown with solid arrows and dependencies are shown with dotted arrows. (B) Since the entries are not stored in a collection, if dependencies are overwritten, the log will be automatically cleaned.

$$\begin{array}{rcl} Q & := & Q.k \mid \{k := Q\} \mid Q \Leftarrow Q \mid \textbf{map } Q \textbf{ using } Q \\ & \mid & Q \ \textbf{op}_{[\theta]}\ Q \mid \textbf{agg}_{[\theta]}(Q) \mid \textbf{agg}_{[\Leftarrow]}(Q) \mid Q \circ Q \\ & \mid & \textbf{filter } Q \textbf{ using } Q \mid \textbf{if } Q \textbf{ then } Q \textbf{ else } Q \\ & \mid & \textbf{c} \mid \textbf{null} \mid \emptyset \end{array}$$

Figure 4: The grammar for Bar_{QL} queries. In this grammar, k represents values of key type, c represents constant primitive values, and θ represents any binary operation on primitive values.

state updates, making it an ideal candidate for reasoning about updates in a monadic log.

Bar_{QL} is loosely based on the Monad Algebra [13], although unlike the latter, which uses sets as its base collection type, Bar_{QL} uses maps[2] and has weaker type semantics – maps need not consist of uniform types, allowing a map to fill the role of both a collection and a tuple type. Bar_{QL} is intentionally limited to operations with linear computational complexity in the size of the input data; neither the pair with nor cross-product operations of Monad Algebra are included. Note that this is not a limitation for Laasie, as the computation of cross products and joins can be pushed to client front-ends.

The grammar for Bar_{QL} is given in Figure 4. Bar_{QL} uses an unstructured hierarchical type-system, analogous to unstructured XML or JSON. Values in Bar_{QL} are either of primitive type, null, or collections (mappings from keys of type k to values). Collections are total mappings, although we say that key k is *defined* in a collection v if k maps to a non-null value. When specifying a collection, we will enumerate only defined keys.

In Bar_{QL}, queries are monads, structures that represent computation. Reducing the query corresponds to evaluating the computation expressed by that query. The constant operations: **c**, *null*, $\emptyset$, are all defined as operations that produce their respective constant value as output, regardless of input. For example: $\mathbf{5}(null) = 5$

The empty collection operator $\emptyset$ creates a collection for which all keys map to *null*.

The identity and singleton operators (**id** and $\{k := Q\}$, respectively) retain their behaviors from the monad algebra: The identity operation returns its input unchanged, while the singleton constructor creates a collection with a single key k defined by the value returned by Q. For example: $\{A := \mathbf{id}\}(5) = \{A \to 5\}$

Note that the singleton constructor functions as both a collection singleton, as well as a tuple constructor. Because collection elements are identified by keys, we can reference specific elements of the collection as we would reference fields of a tuple.

The most significant way in which Bar_{QL} differs from Monad Algebra is its use of the *Merge* operation ($\Leftarrow$) instead of set union ($\cup$). $\Leftarrow$ combines two sets, overwriting undefined entries (keys for which the collection maps to *null*) with their values from the other collection.

$$(\{A := 1\} \Leftarrow \{B := 2\})(null) = \{A \to 1, B \to 2\}$$

If a key is defined in both collections, the right collection takes precedence.

$$(\{A := 1\} \Leftarrow \{A := 2\})(null) = \{A \to 2\}$$

The merge operator can be combined with singleton and identity to define updates to collections:

$$(\mathbf{id} \Leftarrow \{A := 3\})(\{A \to 1, B \to 2\}) = \{A \to 3, B \to 2\}$$

Subscripting $Q.k$ dereferences key k on the collection returned by Q, and can be combined with other operators to define point modifications to collections.

$$\begin{aligned}(\mathbf{id} \Leftarrow \{A := (\mathbf{id}.A \Leftarrow \{B := 2\})\})(\{A \to \{C \to 1\}\}) \\ = \{A \to \{B \to 2, C \to 1\}\}\end{aligned}$$

Primitive binary operators are defined monadically ($\mathbf{op}_{[\theta]}$), and include basic arithmetic, comparisons, and boolean operations. These operations can be combined with identity, singleton, and merge to define updates. For example, to increment A by 1, we write

$$\{\mathbf{id} \Leftarrow \{A := (\mathbf{id}.A)\ \mathbf{op}_{[+]}\ (1)\}\}(\{A \to 2\}) = \{A \to 3\}$$

Bar_{QL} provides constructs for mapping, flattening and aggregation. The **map** operation, analogous to its behavior in the Monad Algebra applies a transformation (the **using** clause) to all values in a collection, preserving key names. The flatten ($\mathbf{agg}_{[\Leftarrow]}(Q)$) operation is also similar, except that it uses $\Leftarrow$, instead of $\cup$ as in Monad Algebra. Aggregation is defined analogously to flatten using any closed binary operator θ operating over values of primitive type. For example, to increment all children of the root we write:

$$\begin{aligned}(\textbf{map id using } (\mathbf{id} + 1))(\{A \to 1, B \to 2\}) = \\ \{A \to 2, B \to 3\}\end{aligned}$$

To increment the child C of each child of the root by 1, we write

$$\begin{aligned}(\textbf{map id using } (\mathbf{id} \Leftarrow \{C := \mathbf{id}.C + 1\}))(\\ \{A \to \{C \to 1\}, B \to \{C \to 2, D \to 1\}\} \\ ) = \{A \to \{C \to 2\}, B \to \{C \to 3, D \to 1\}\}\end{aligned}$$

Finally, Bar_{QL} supports standard function conditionals, conditional filtering of collections, as well as composition ($\circ$) of queries.

[2]Maps are commonly referred to as hashes or dictionaries.

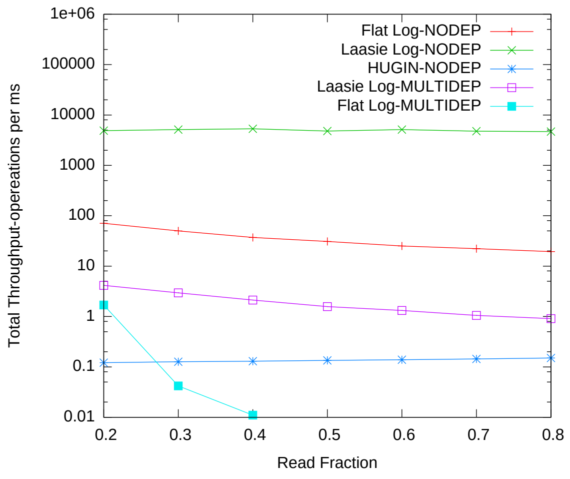

Figure 5: The graph compares the total throughput of all the three log structures (Laasie, Flat Log and Hugin) by varying the number of read operations and keeping the constant number of write operations.

5. PRELIMINARY RESULTS

We tested a Java-based implementation of the Laasie Monadic Log against two other log structures. Our first comparison is to a Flat Monadic Log which doesn't make use of an indexing system and is not capable of storing entry dependencies. In contrast to Laasie a read operation iterates over the entire log until it finds the desired entry. Upon finding this entry a continued iteration will be performed for all dependent entries.

As a second comparison point, we consider a real-world application – the Hugin Open Source Mapping System [14]. Hugin uses a general purpose state replication system called ICON to provide persistence and coordination functionality. ICON is implemented in 500 lines of PHP, and supports recovery through a *non-monadic* log. The log is backed by a single Postgres table with indexes on the columns corresponding to update timestamp and the name of the variable being updated. Update data is stored in a Postgres TEXT blob.

Tests were uniformly run on a 2x6 core 2.5 GHz Intel Xeon server with 64 GB of RAM; All tests were run single-threaded. Unless otherwise specified, all results presented are the average performance over 10 trials.

Each log consists of a fixed number of randomly generated log entries operating over an application state consisting of 10,000 values. We assign the generated entry a randomly selected number of read dependencies. Unless otherwise stated, the set of dependencies is randomly selected from the set of all values in the application state. For some tests however, we limit the number of dependencies to a predefined maximum. We generate two classes of workload: NODEP where all entries have an empty dependency list and MULTIDEP where all entries have a dependency list of size greater than zero.

The first workload consists of two test cases. In the first case the Laasie Log and Flat Log structures were tested with NODEP for 1,000,000 operations and Hugin NODEP for 10,000 operations. In the second case the Laasie and Flat Log structures were tested with MULTIDEP for 1,000 operations. For this workload we measured the total throughput (read throughput and write throughput) by varying the number of read operations while keeping the number of write operations constant. The number of write operations on the Lassie and Flat Log was increased from 10,000 to 1,000,000. For the NODEP case, throughputs for both log sizes are comparable; The larger log was necessary to avoid sub-second runtimes for the Laasie log and produce more stable performance results. Our results for the first workload are shown in Figure 5.

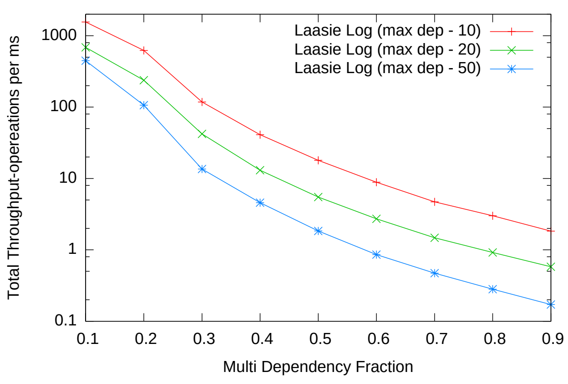

Figure 6: The graph compares the total throughput of Laasie log versions having different maximum size of dependency list by varying the number of log entries having multiple dependencies.

The NODEP Laasie Log performs much better than its counterparts (NODEP Flat Log and NODEP Hugin Log). The total throughput of the NODEP Laasie Log remains nearly constant regardless of the read/write ratio. This is in contrast to the NODEP Flat Log of which total throughput decreases for high-read workloads. The MULTIDEP Laasie Log performs better than the MULTIDEP Flat Log, and remains minimally affected by the read/write ratio. Unlike our monadic structures, write operations in the Hugin Log are more expensive than reads, because Hugin performs an explicit garbage collection step on every write. Thus, unlike the monadic log structures, Hugin's performance actually improves for high-read workloads.

The second workload compared the total throughput of Laasie Log instances for maximum dependency list sizes of 10,20,and 50. The workload was a mixture of NODEP and MULTIDEP operations, ranging from a 10% NODEP/90% MULTIDEP split to 90%/10%. The workload consisted of 10,000 write operations. Our results for the second workload are shown in Figure 6.

As the percentage of MULTIDEP operations increases and the size of the dependency lists increases, the total throughput decreases significantly. This is a result of an increased number of operations depending on a large number of previous operations. These dependencies prevent old values from being discarded by overwrites. The most drastic decrease in performance can be seen between 20 and 30 percent of all operations being MULTIDEP. This behavior is characteristic of large dependency chains being developed as a result of a lack of deletes from NODEP operations. With fewer NODEP operations larger portions of the log are being returned for read operations decreasing performance. More realistic workloads will have a more structured dependency sets, which are likely to result in smaller dependency chains.

6. RELATED WORK

The Laasie project brings together databases, programming languages, and distributed systems. Our problem space has been explored by each area individually, but to the best of our knowledge, Laasie is the first solution to bring together all three.

Update Sequencing and Logs. A core challenge in many distributed systems is applying canonical orders to update sets. Solutions based on a distributed log have been developed for distributed concurrency control [6, 2], distributed collection types [7], and several domain-specific systems such as Antiquity [20], Aurora [1] and LiveObjects [16].

LiveObjects in particular targets an application domain similar to Laasie's. However, the core focus of LiveObjects is implementing a fully peer-to-peer distributed log – the coordination layer, while Laasie focuses on persistence and recovery – the data management layer. The approaches taken by these two systems are orthogonal and may be combined.

Algebraic Properties of State Updates. Algebras built over sequences of updates have been used to optimize distributed systems. Commutativity [21, 19] in particular, plays a crucial role in detecting conflicting updates that must be resolved. Unfortunately, many of the systems results in this area have been in the context of domain-specific applications [19, 15], such as edits to textual data.

Laasie employs an authoritative ordering over updates to prevent conflicts. In the absence of such an ordering, techniques must be applied to resolve conflicts as they occur. The Operational Transform [8] and Edit Lenses [10] are two approaches to resolution. These techniques can relevant to Laasie, as interactivity requirements can necessitate the use of view-based optimistic concurrency control protocols [9].

Functional Updates. Functional state updates are used frequently by the database community, especially in the context of distributed databases. Two concrete examples are Starburst [4], and BigTable [5].

Functional updates have appeared previously in the context of log-based recovery [11]. However, in such systems, the server is responsible for reifying the updates before committing them to the log. For a sufficiently complex state manipulation language, this is not scalable.

7. CONCLUSION

In this paper we introduced the design of the Laasie data management system for collaborative cloud applications. Laasie, at its heart, is based on a low-level monadic, self-cleaning log structure. By dramatically changing the way in which data is represented within the log, we believe Laasie will be able to support *semantics* driven, dynamic restructurings of the log. Preliminary experiments on our log infrastructure indicate the monadic logs are a viable implementation structure.

8. REFERENCES

[1] Balazinska, M., Balakrishnan, H., Madden, S., and Stonebraker, M. Availability-Consistency Trade-Offs in a Fault-Tolerant Stream Processing System. Tech. rep., MIT, 2004.

[2] Bernstein, P. A., Reid, C., and Das, S. Hyder–A Transactional Record Manager for Shared Flash. *CIDR* (2011).

[3] Birman, K., and Cooper, R. The ISIS project: Real experience with a fault tolerant programming system. In *SIGOPS* (1990), pp. 1–5.

[4] Ceri, S., and Widom, J. Production rules in parallel and distributed database environments. *PVLDB* (1992).

[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A distributed storage system for structured data. *TOCS 26*, 2 (2008), 4.

[6] Ellis, C. A., and Gibbs, S. J. Concurrency control in groupware systems. *SIGMOD* (1989).

[7] Eugster, P. T., and Guerraoui, R. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. *ECOOP* (2000).

[8] Feldman, A. J., Zeller, W. P., Freedman, M. J., and Felten, E. W. SPORC: Group Collaboration using Untrusted Cloud Resources. In *OSDI* (2010).

[9] Gupta, N., Demers, A., Gehrke, J., Unterbrunner, P., and White, W. Scalability for virtual worlds. In *ICDE* (2009), pp. 1311–1314.

[10] Hofmann, M., Pierce, B., and Wagner, D. Edit lenses. In *SIGPLAN-SIGACT* (2012), pp. 495–508.

[11] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC* (2010), vol. 10.

[12] Kennedy, O., and Ziarek, L. Barql: Collaborating through change. Tech. rep., CORR, arXiv:1303.4471, 2013.

[13] Lellahi, K., and Tannen, V. A calculus for collections and aggregates. In *Category Theory and Computer Science* (1997), Springer, pp. 261–280.

[14] Medieval Software. The hugin mapper. http://hugin-mapper.sourceforge.net.

[15] Oster, G., Urso, P., Molli, P., and Imine, A. Data Consistency for P2P Collaborative Editing. In *CSCW* (2006), p. 259.

[16] Ostrowski, K., and Birman, K. Storing and accessing live mashup content in the cloud. *SIGOPS Review 44*, 2 (Apr. 2010).

[17] Peng, D., and Dabek, F. Large-scale incremental processing using distributed transactions and notifications. In *OSDI* (2010), pp. 1–15.

[18] Piazza Technologies, Inc. Piazza. http://www.piazza.com.

[19] Shapiro, M., and Preguiça, N. Designing a commutative replicated data type. Tech. Rep. arXiv:0710.1784, CORR, 2007.

[20] Weatherspoon, H., Eaton, P., Chun, B.-G., and Kubiatowicz, J. Antiquity: exploiting a secure log for wide-area distributed storage. In *EuroSys* (2007).

[21] Weihl, W. E. Commutativity-based concurrency control for abstract data types. *IEEE TC 37*, 12 (1988), 1488–1505.

[22] Wikimedia Commons. Wikipedia. http://www.wikipedia.org.