Computing Weak Consistency in Polynomial Time

[Extended Abstract]

Wojciech Golab[*] [†]
Dept. of Electrical and Computer Engineering
University of Waterloo, Canada
wgolab@uwaterloo.ca

Xiaozhou (Steve) Li[‡]
Google Inc.
Mountain View, CA, USA
xzli@google.com

Alejandro López-Ortiz[†]
School of Computer Science
University of Waterloo, Canada
alopez-o@uwaterloo.ca

Naomi Nishimura[†]
School of Computer Science
University of Waterloo, Canada
nishi@uwaterloo.ca

ABSTRACT

The k-atomicity property can be used to describe the consistency of data operations in large distributed storage systems. The weak consistency guarantees offered by such systems are seen as a necessary compromise in view of Brewer's CAP principle. The k-atomicity property requires that every read operation obtains a value that is at most k updates (writes) old, and becomes a useful way to quantify weak consistency if k is treated as a variable that can be computed from a history of operations. Specifically, the value of k quantifies how far the history deviates from Lamport's atomicity property for read/write registers. We address the problem of computing k indirectly by solving the k-atomicity verification problem (k-AV): given a history of read/write operations and a positive integer k, decide whether the history is k-atomic. Gibbons and Korach showed that in general this problem is NP-complete when $k = 1$, and hence not solvable in polynomial time unless P = NP. In this paper we present two algorithms that solve the k-AV problem for any $k \geq 2$ in special cases. Similarly to known solutions for $k = 1$ and $k = 2$, both algorithms assume that all the values written to a given object are distinct. The first algorithm places an additional restriction on the structure of the input history and solves k-AV in $O(n^2 + n \cdot k \log k)$ time. The second algorithm does not place any additional restrictions on the input but is efficient only when k is small and when concurrency among write operations is limited. Its time complexity is $O(n^2)$ if both k and our particular measure of write concurrency are bounded by constants.

Categories and Subject Descriptors

F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; D.2.4 [**Software**]: Software Engineering—*Software/Program Verification*

General Terms

Algorithms, theory, verification

Keywords

Consistency; atomicity; verification; distributed storage

1. INTRODUCTION

Distributed storage and data management systems empower a broad range of data-intensive services today including social networking, web search, e-mail, calendars, and online auctions. In an effort to cope with web-scale workloads in the face of immense competitive pressures, the designs of such services have shifted away from conventional relational databases and towards simpler but more scalable solutions. As a result, many practical systems offer simple key-value operations and BASE (Basically Available, Soft state, Eventual consistency) properties either instead of or side-by-side with conventional transactions and more powerful ACID (Atomicity/Consistency/Isolation/Durability) properties.

Eventually consistent key-value stores are an important breed of distributed storage systems that provide BASE properties [7, 25, 27]. Their distinguishing characteristic is the ability to remain available in the face of network partitions, in which disjoint subsets of hosts become isolated from one another and are unable to communicate. To ensure that storage operations can proceed despite such partitions, specialized replication techniques such as *sloppy quorums* [7, 17, 23] are used. As a side effect, eventually-consistent systems can only provide weak consistency guarantees—a necessary compromise according to Brewer's CAP principle [5], which states that a distributed storage system cannot simultaneously provide strong consistency (C), high availabil-

[*]Author supported in part by the Google Faculty Research Awards Program.

[†]Author supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

[‡]Part of this research was conducted while this author was at Hewlett-Packard Labs.

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.
ACM 978-1-4503-3617-8/15/07.
http://dx.doi.org/10.1145/2767386.2767407.

ity (A), and partition-tolerance (P). This observation has been formalized by Lynch and Gilbert as the impossibility of implementing an atomic read/write register object in an asynchronous message passing system with unreliable communication channels [11]. A read/write register object in this context corresponds to a key-value pair as follows: the key is the object's name and is immutable, whereas the value is the object's state and can be accessed using read and write operations.

As defined by Vogels [27], eventual consistency means that if an object is not updated and no failures occur for a sufficiently long period of time, then eventually all read operations on the object return the last updated value of that object. Although this definition captures nicely the essence of eventual consistency, it also leaves open important questions regarding the behavior of key-value stores in practice: how long is "eventual", and how consistent are reads when objects are updated continuously? In an attempt to answer the latter question, various notions of *data staleness* have been defined and analyzed. In this context, a value written to a key is considered stale once it is overwritten by another value, and so the staleness of a particular read is a measure of the "distance" between two write operations: the write that assigned the value read (called the *dictating write* of the read), and the write that assigned the latest value to the object under consideration.

In this paper, we focus on a formal notion of *version-based staleness*, which is defined by counting write operations. Specifically, we consider the *k-atomicity* property of Aiyer et al. [1], which is a generalization of Lamport's atomicity property for read/write registers [18]. Both properties are defined over an *execution history*, which is a collection of read/write operations tagged with distinct start and finish times. An operation in such a history "happens before" another operation if the former finishes before the latter starts, and a history is called k-atomic if there exists a total order over the operations that extends the "happens before" partial order, and in which every read returns a value written by one of the last k writes preceding the read in the total order.

The k-atomicity verification (k-AV) problem is to decide for a given execution history and integer $k \geq 1$ whether the history is k-atomic. An efficient solution to this problem makes it possible to compute the smallest k for which a given history is k-atomic, which quantifies the maximum degree of staleness observed by clients accessing an eventually consistent storage system. Thus, solving k-AV makes it possible to analyze the actual consistency provided by such systems in arbitrary workloads, including ones where writes occur continuously and concurrently with each other.

Our main contribution in this paper is a pair of algorithms that solve the k-AV problem for arbitrary $k \geq 2$ and address the previously unsolved case $k \geq 3$. Both algorithms assume that all the values written to a given object in a history are distinct, which makes 1-AV solvable in $O(n \log n)$ time for a history of n operations. The first algorithm, called GPO ("greedy plus obligations"), places an additional restriction on the structure of the input history and solves k-AV for arbitrary k in $O(n^2 + n \cdot k \log k)$ time. The second algorithm, called CGS ("configuration graph search"), does not place any additional restrictions on the input but is efficient only when k is small and when concurrency among write operations is restricted. Its time complexity is $O(n^2)$ if both k and our specific measure of write concurrency are bounded by constants. As explained in Section 7, we expect these algorithms to perform efficiently in the vast majority of cases encountered in practice.

2. RELATED WORK

The fundamental problem of defining consistency properties for shared objects was first addressed by Misra [20] and Lamport [18]. Lamport defined a number of key concepts in this area, including the *happens before* and $\longrightarrow$ (i.e., precedes) relations, and the atomicity property for registers [18]. The "happens before" relation is an irreflexive partial order over instantaneous events that captures the following notion of causality: given two events e_1, e_2 in Einstein's four-dimensional space-time, we say that e_1 happens before e_2 if it is possible for a pulse of light emitted at A to arrive at B, and hence for A to physically cause B. Operations on objects are modeled as collections of events, and given two operations op_1, op_2 we say that $op_1 \longrightarrow op_2$ if every event of op_1 happens before every event of op_2. In the presence of a global clock events can be totally ordered according to time, in which case $op_1 \longrightarrow op_2$ means that op_1 finishes before op_2 starts. Lamport's atomic read/write register satisfies the property that any collection of operations applied to the object can be arranged (conceptually) in some total order that extends $\longrightarrow$ and in which every read returns the value assigned by the latest write [18]. Misra's axioms provide an alternative definition of such a register, and explicitly address the case of multiple concurrent writers [20], whereas Lamport's definitions are given for the single-writer case but generalize easily to multiple writers. Herlihy and Wing's *linearizability* property is a generalization of Lamport's atomicity property to arbitrary typed shared objects, and to scenarios that include pending (i.e., incomplete) operations that may appear to take effect before they finish [15]. Herlihy and Wing's *happens before* relation over pairs of operations is a special case of Lamport's $\longrightarrow$ relation when events are totally ordered.

Several relaxed consistency properties have been defined in an attempt to describe the behavior of weakly consistent distributed storage systems. In general these properties capture the staleness of the values returned by read operations, which we can think of more precisely as the "distance" between operations, as explained in Section 1. Two notions of staleness have been defined in literature: *time-based* and *version-based*. A time-based staleness of t time units means that a read returned a value that was considered fresh at most t time units earlier. Torres-Rojas et al. formalize this notion by defining *timed consistency* properties in distributed message passing systems as generalizations of sequential consistency and causal consistency [26]. Similarly, Golab et al. generalize Lamport's atomicity property for read/write registers [18] by defining Δ-atomicity [13] and Γ-atomicity [14]. On the other hand a version-based staleness of k versions means that a read returned the value assigned by one of the last k writes in some total ordering of the write operations. Aiyer et al. formalize this notion as the *k-atomicity* property, which is also inspired by Lamport's atomic register. A collection of read/write operations is k-atomic if the operations can be arranged in some total order that extends $\longrightarrow$ and in which every read returns the value assigned by one of the k latest writes.

Several storage systems and modeling frameworks incorporate relaxed consistency properties. The TACT (Tunable Availability/Consistency Tradeoffs) framework of Yu and Vahdat defines version-based and time-based staleness informally as *order error* and *staleness* [29]. Lee and Welch describe a probabilistic consistency framework in the context of *randomized registers* implemented using probabilistic quorums [19]. Their analysis refers implicitly to version-based staleness by deriving bounds on the probability that a stale replica exists for a data item that has been overwritten a certain number of times. The Probabilistically Bounded Staleness (PBS) framework of Bailis et al. uses similar principles to derive bounds on time-based and version-based staleness [3]. They formalize both notions of staleness in the simplified model where write operations do not overlap in time with other read or write operations. Ardekani and Terry as well as Terry et al. propose storage systems that can be configured using service level agreements (SLAs) to provide bounds on time-based staleness [2, 24]. Our contributions complement this body of work in the following way: our techniques for computing version-based staleness from observed histories could be used to validate frameworks like TACT and PBS, and verify the correctness of storage systems that support staleness-based SLAs, provided that technical differences among various definitions of staleness can be reconciled. This approach stands in contrast to the technique used by Wada et al., and by Bermbach and Tai, which measures the convergence time of the replication protocol of a distributed system rather than capturing the staleness actually observed by clients [4, 28].

Decision problems pertaining to the consistency properties discussed herein were first studied in the context of verifying shared memories. The input to the decision problem is an *execution history*—a sequence of invocation and response events corresponding to operations applied to a collection of shared objects. The events in a history define the invocations and responses of operations on objects. Gibbons and Korach showed that deciding whether such a history is atomic is NP-complete in the general case [10]. However, in the special case when each write operation on a given object assigns a distinct value, polynomial-time decision algorithms for atomicity have been developed. Under this assumption, Misra's axioms can be used to characterize atomicity as the absence of cycles in a *conflict graph* whose vertices represent values and whose edges represent precedence constraints imposed by the ordering of reads and writes in the history [20]. For a history with n operations the time complexity of this algorithm is $O(n^2)$. Gibbons and Korach instead followed the approach of clustering operations according to the value read or written, and characterized atomicity as the absence of conflicts between pairs of such clusters [10]. Their algorithm has time complexity $O(n \log n)$.

The decision problem for the k-atomicity property is called the k-atomicity verification (k-AV) problem. Given a history and an integer $k \geq 1$, the objective is to decide whether the history is k-atomic. The *locality* property of k-atomicity states that the input history is k-atomic if and only if for each object accessed, the subhistory of operations on that specific object is k-atomic [12, 15]. Accordingly, solutions to k-AV assume without loss of generality that all operations are applied to the same object. A solution to k-AV can be used to compute the *k-value* of a history: the smallest k for which the history is k-atomic. The k-value equals 1 if and only if the history is atomic in Lamport's sense [18], and larger k-values indicate deviation from atomicity and hence from linearizability; the higher the k the greater the deviation.

The k-AV problem was first considered by Golab et al. [13]. Like the problem of deciding atomicity, k-AV is NP-complete in the general case, assuming that k is part of the input. Golab et al. present the first polynomial-time solutions to 2-AV under the assumption that each write operation on a given object assigns a distinct value, which is the same assumption that makes 1-AV tractable [12]. One algorithm, called LBT, runs in $O(n^2)$ time for a history with n operations. It leverages ideas inspired by Misra [20], as well as the technique of limited backtracking [8]. The other algorithm, called FZF, runs in $O(n \log n)$ time, and borrows the clustering idea of Gibbons and Korach [10].

The k-AV problem is somewhat similar to the graph bandwidth problem (GBW). Given a graph G and a positive integer k, the GBW problem is to decide whether it is possible to arrange the vertices of G at distinct positions on a line such that any two vertices that are adjacent in G are separated by at most $k-1$ other vertices on the line. GBW is NP-complete when k is part of the input [21], and is solvable in polynomial time when k is fixed [9, 22]. The algorithm of Saxe uses dynamic programming and runs in $O(n^{k+1})$ time where n is the number of vertices [22]. Kleitman and Vohra [16] present a GBW algorithm that runs in $O(n \log n)$ time, but is correct only for interval graphs.

Our graph-theoretic characterization of k-AV presented in Section 2 (see Lemma 2) has a similar flavor to GBW in the sense that it refers to a linear arrangement of vertices that is constrained by edges. The main difference is that we consider constraints with respect to two different but related sets of edges, with one set imposing "hard" constraints similarly to edges in a topological ordering, and the other set imposing "soft" constraints similarly to edges in GBW. Furthermore, our edges are directed and the union of the two edge sets has a structure similar to an interval graph, whereas GBW is defined over arbitrary undirected graphs. As regards time complexity, we are not aware of a polynomial time Turing reduction from GBW to k-AV or vice versa. Such a reduction from k-AV to GBW would imply that k-AV is solvable in polynomial time for fixed k.

3. PRELIMINARIES

Following the model of Herlihy and Wing [15], we define an *execution history* (or *history* for short) as a sequence of *events* where each event is either the invocation or the response of an operation on a read/write register. Each operation in the history has both an invocation and a response event, and all operations are applied to the same object (see discussion of locality in Section 2). Each event is tagged with a unique time, and events appear in the history in increasing order of their times. The exact values of these times are not important, only their relative order, and so we assume without loss of generality that the ith event (counting from $i = 1$) occurs at time i. Each event also records the operation type (i.e., read vs. write), its argument (for writes only) and its return value (for reads only). The *value* of an operation *op* is its argument if *op* is a write, and its response if *op* is a read. Given a read r and a write w we call w a *dictating write* of r if r and w share the same value. In that

case we call r a *dictated read* of w. In the remainder of the paper, the number of operations in H and the number of writes in H are denoted by n and n_w, respectively.

Given an operation op, we denote by $s(op)$ and $f(op)$ the start time and finish time of op, respectively. Given a pair of operations op_1, op_2 in a history H we denote Herlihy and Wing's *happens before* relation by $<_H$ (i.e., $op_1 <_H op_2$ if and only if $f(op_1) < s(op_2)$). We say that op_1 and op_2 are *concurrent* if neither $op_1 <_H op_2$ nor $op_2 <_H op_1$. A total order $<_T$ over the operations in a history H *extends* the partial order $<_H$, denoted $<_T \supseteq <_H$, if every pair of operations op_1, op_2 in H satisfies $op_1 <_H op_2 \Rightarrow op_1 <_T op_2$. We call $<_T$ a *k-atomic total order for H* if $<_T \supseteq <_H$ and every read operation returns the value assigned by one of the last k writes that precede it in $<_T$. A history H is k-atomic if and only if there exists a k-atomic total order $<_T$ for H. Mathematically a total order $<_T$ over the values in H is a relation, but for convenience we often treat it as the corresponding sequence of values and refer to it as T.

In the remainder of the paper, we make the following assumption to circumvent the NP-completeness of k-AV:

Assumption 1. *For any history H, every read operation in H has exactly one dictating write.*

Without Assumption 1 the k-AV problem is NP-complete, assuming that k is part of the input, because 1-AV is NP-complete [10]. Assumption 1 can be enforced in practical storage systems by embedding a unique identifier, for example based on the current time and client ID, in each value.

Under Assumption 1 we adopt the following notational convention: for any history H and any value v read or written in H, $w(v)$ denotes the unique dictating write of v in H. Similarly we use $r(v)$ to denote a read of v in H. If more than one read of v exists in H then the particular operation corresponding to $r(v)$ is specified in the context.

We make two additional assumptions to simplify our algorithms and their analysis. First, we assume that a value cannot be read before it is written:

Assumption 2. *For any history H, if r is a read in H and w is its dictating write then $r <_H w$ is false.*

Any history H that violates Assumption 2 automatically fails to satisfy k-atomicity for any $k \geq 1$. Assumption 2 can be tested in $O(n \log n)$ steps by clustering operations by their value, and checking for each value that no read precedes its dictating write.

Second, we assume that a read never finishes before its dictating write:

Assumption 3. *For any history H, if r is a read and w is its dictating write then $f(w) < f(r)$.*

Assumption 3 implies no loss of generality because any history H that satisfies Assumptions 1 and 2 can be transformed efficiently into a history H' that in addition satisfies Assumption 3 in a manner that preserves k-atomicity. This is done using the normalization procedure presented as Algorithm 1. Theorem 1 asserts the correctness of this procedure, and its proof is omitted due to lack of space.

Theorem 1. *If the input history H of the normalization procedure (Algorithm 1) satisfies the pre-conditions (Assumptions 1–2) then the output history H' satisfies the post-conditions (Assumptions 1–3, H is k-atomic if and only if H' is k-atomic).*

Input: history H that satisfies Assumptions 1–2.
Output: history H' that satisfies Assumptions 1–3, and is k-atomic if and only if H is.

```
1 H' := H
2 for every value v in H do
3 |   r(v) := read of v with earliest finish time in H
4 |   if f(r(v)) < f(w(v)) then
5 |   |   shift the response event for w(v)'s
  |   |   counterpart in H' to the position
  |   |   immediately preceding the response event of
  |   |   r(v)'s counterpart in H'
6 |   end
7 end
```

Algorithm 1: Normalization procedure for histories.

In general our algorithms assume that $2 \leq k \leq n$. For $k = 1$ the algorithm of Gibbons and Korach can be used instead [10], and its time complexity is $O(n \log n)$. For $k > n$ one can decide k-atomicity efficiently by verifying that each read has a dictating write and that no read precedes any of its dictating writes. These conditions are implied by Assumptions 1 and 2, and can be verified in $O(n \log n)$ time.

To simplify the presentation and analysis of our k-AV algorithms we first describe two graph-theoretic representations of the input history. These are given in Definition 1, which is inspired by Misra's *before* relation on values [20], and in Definition 2. We use both definitions to characterize k-atomicity in Lemma 2, which we use extensively later in Sections 4 and 5. The proof of Lemma 2 is omitted due to lack of space.

Definition 1. *The* read value graph *for a history H, denoted $G_R(V, E_R)$, is a directed graph where V is the set of values written in H, and E_R contains an edge (v, v') if and only if $v \neq v'$ and there exists a read $r(v')$ in H such that $w(v) <_H r(v')$.*

Definition 2. *The* write graph *for a history H, denoted $G_W(V, E_W)$, is a directed graph where V is the set of values written in H, and E_W contains an edge (v, v') if and only if $w(v) <_H w(v')$.*

Lemma 2. *A history H is k-atomic if and only if its corresponding write graph $G_W(V, E_W)$ has a topological ordering $T = \langle v_1, v_2, \ldots, v_{|V|} \rangle$, such that for any two vertices v_i and v_j in T, if the read value graph $G_R(V, E_R)$ for H contains an edge (v_i, v_j) then $j > i - k$.*

Informally speaking, G_R and G_W both capture constraints on the order in which writes may appear to take effect. Lemma 2 then asserts that H is k-atomic if and only if there is an ordering of the writes that is consistent with G_W and in which the edges of G_R never point "backward" by more than $k - 1$ positions.

Examples of G_R and G_W are presented in Figure 1 with respect to a small input history. Operations in the history are illustrated as intervals of time following the style of diagrams used by Herlihy and Wing [15]. The horizontal dimension represents time and the vertical dimension is used only to separate the intervals visually. The history has two 3-atomic total orders, namely $\langle 5, 2, 1, 3, 4 \rangle$ and $\langle 5, 2, 3, 1, 4 \rangle$,

but fails to satisfy 2-atomicity because $w(1)$ and $w(3)$ both separate $w(2)$ from $r(2)$. Consequently, the history is not 1-atomic either.

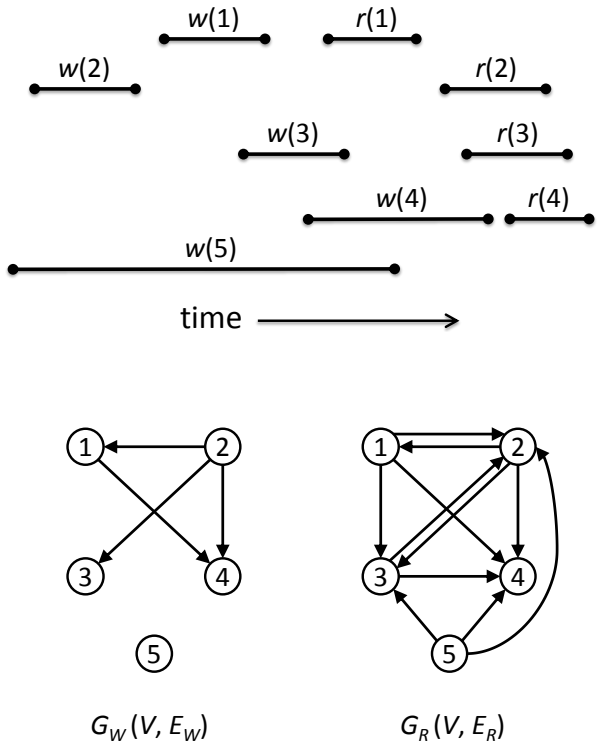

Figure 1: Example execution history (top) and corresponding graph representation (bottom).

4. AN EFFICIENT ALGORITHM FOR A RESTRICTED CLASS OF HISTORIES

In this section, we present an efficient algorithm called GPO ("greedy plus obligations") that solves k-AV for arbitrary k in the special case when the input history satisfies Assumptions 1–3 as well as the following:

ASSUMPTION 4. *For any history H and for any value v written in H, the write $w(v)$ has a dictated read $r(v)$ such that $w(v) <_H r(v)$.*

Using the terminology of Gibbons and Korach, Assumption 4 implies that every zone in the history is a forward zone [10]. Intuitively, this requirement constrains the k-AV problem by ensuring the presence of certain edges in the representation of a history H as a write graph G_W and a read value graph G_R. Specifically, for every pair of values v and v', Assumption 4 ensures that G_R contains the edge (v, v') if $f(w(v)) < f(w(v'))$, or the edge (v', v) if $f(w(v')) < f(w(v))$. In contrast, Assumptions 1–3 alone do not imply that v and v' are ordered by G_W or G_R in the case when $w(v)$ and $w(v')$ are concurrent.

4.1 Algorithm description

The main idea behind GPO is "greedy plus obligations;" hence the name. Like the LBT algorithm for 2-AV [12], GPO constructs a k-atomic total order in reverse. For each value placed in the k-atomic total order the algorithm evaluates any constraints this placement imposes on the choice of subsequent values, and records these constraints using internal data structures.

Given multiple candidates for the next value to be placed, GPO greedily picks the one whose dictating write has the largest finish time. In some cases this choice is narrowed down to a subset of values that must be placed within the next $k-1$ (or fewer) positions in the k-atomic total order due to constraints imposed on them by values that were placed earlier. Once GPO chooses the next value v, it identifies two types of constraints imposed by v: (i) if the write of v happens before the read of some value v' that has not yet been placed then v' must be placed in one of the next $k-1$ positions; and (ii) if the write of such a value v' happens before the write of another value v'' that has not yet been placed then v'' must also be placed in the next $k-1$ positions.

The constraints are tracked using $k-1$ sets of values called *obligation sets*, denoted by $B[1]$ to $B[k-1]$, initially all empty. The purpose of $B[i]$ is to record the values that must be placed in the next i positions of the k-atomic total order, and so naturally GPO must maintain the invariant $B[1] \subseteq B[2] \subseteq \cdots \subseteq B[k-1]$. An obligation set $B[i]$ is called *overfull*, *full*, or *underfull* if $|B[i]| > i$, $|B[i]| = i$, or $|B[i]| < i$, respectively. As GPO decides the next value to be placed it faces three possibilities. If there exists an overfull obligation set then GPO simply outputs NO because it is impossible to place more than i values in the next i positions. If no obligation sets are overfull but at least one of them is full then GPO first identifies the smallest full set, and then among the values in that set it greedily picks the one whose dictating write has the largest finish time. Otherwise all obligation sets are underfull, and GPO greedily chooses a value that has not yet been placed and whose dictating write has the largest finish time.

Having chosen the next value v, GPO updates the obligation sets by removing the value that was just placed, and then copying the remaining values from $B[i+1]$ to $B[i]$. Intuitively, if a value was supposed to be placed in the next $i+1$ positions at the beginning of a given iteration and is not chosen in the same iteration then it must be placed in the next i positions at the beginning of the next iteration. In addition, GPO updates $B[k-1]$ by adding values determined according to the two types of constraints imposed by v, as described earlier. As long as no overfull obligation set is encountered the algorithm continues to choose values and update the obligation sets until it has constructed a k-atomic total order for the input history. At that point GPO returns YES. The full procedure is presented as Algorithm 2.

The intuition underling lines 7 and 9 of GPO, which choose a value v with maximum finish time over any other candidate value v', follows from two observations. First, $w(v)$ happens before a subset of the operations that $w(v')$ happens before, and hence the choice of v as the next value in the reverse k-atomic order is the least constraining with respect to other values that have not yet been chosen. Second, choosing v in an earlier iteration than v' positions v' before v in the k-atomic total order, which tends to be consistent with the direction of edges in G_R because $f(w(v')) < f(w(v))$ implies that G_R contains the edge (v', v) under Assumption 4. Given that this edge exists, choosing v' instead in an earlier iteration than v would open the possibility that v precedes

v' by more than $k-1$ positions in the chosen total order over values, which precludes such an order being k-atomic (see Lemma 2).

4.2 Examples

With respect to the example history presented in Figure 1, GPO works correctly only if we remove $w(5)$, which lacks a dictated read and hence violates Assumption 4. Consider the case $k = 3$ with $w(5)$ removed. The algorithm chooses 4 at line 9 in the first iteration, which imposes no additional constraints. It then chooses 3 at line 9 in the second iteration, and adds the values $\{1, 2\}$ to $B[2]$. In the third iteration it chooses 1 from $B[2]$ at line 7, and then copies 2 to $B[1]$. In the fourth iteration it chooses 2 from $B[1]$ at line 7. Thus, GPO computes the total order $\langle 2, 1, 3, 4\rangle$ and returns YES. This response is correct because the input history without $w(5)$ is 3-atomic.

Next, consider the case $k = 2$ with $w(5)$ removed. As before the algorithm chooses 4 in the first iteration. In the second iteration it chooses 3 at line 9, and adds the values $\{1, 2\}$ to $B[1]$, causing a return of NO at line 19 because $|B[1]| = 2 > 1$. This response is correct because the input history is not 2-atomic, even without $w(5)$.

Finally, consider the full history in Figure 1 with $w(5)$ included. The algorithm chooses 4 at line 9 in the first iteration, which imposes no additional constraints. It then chooses 5 at line 9 in the second iteration, and adds the values $\{1, 2, 3\}$ to $B[2]$, causing a return of NO at line 19 because $|B[2]| = 3 > 2$. This response is incorrect because the input history is 3-atomic. Thus, GPO breaks because the input history fails to satisfy Assumption 4.

4.3 Analysis

References to $G_R(V, E_R)$ and $G_W(V, E_W)$ in our analysis pertain to the read value graph and the write graph for the input history H (Definitions 1 and 2). These structures are needed only for analysis and therefore do not appear in the pseudo-code of Algorithm 2. The symbols T_x, U_x, and $B_x[i]$ denote the values of T, U, and $B[i]$ at the end of iteration x of the outer loop. Detailed proofs of correctness are omitted due to lack of space.

LEMMA 3. *The outer loop of the algorithm provides the following invariant:*

(a) for every value v accessed in H, v is either in T_x or in U_x (but not in both); and

(b) for every i, $1 \le i \le k-1$, if there exists a value $v \in B_x[i]$ and a value $v' \in U_x$ such that $w(v) <_H w(v')$ then $v' \in B_x[i]$.

LEMMA 4. *Suppose that the algorithm outputs YES. Then the sequence of values T obtained when line 22 is reached is a topological ordering of G_W.*

LEMMA 5. *If the algorithm outputs YES then H is k-atomic.*

LEMMA 6. *If H is k-atomic then the algorithm outputs YES.*

PROOF SKETCH. Suppose that H is k-atomic. We show that the algorithm outputs YES by proving the following predicate $P(x)$ for all $x \le n_w$, where n_w denotes the number of writes in H:

```
   Input: history H and integer k, 2 ≤ k ≤ n.
   Output: YES if H is k-atomic, NO otherwise.
   // suffix of k-atomic total order for H
 1 T := empty sequence
   // set of values that have not yet been added
      to T
 2 U := set of values written in H
 3 for i := 1 to k − 1 do B[i] := ∅
 4 while U ≠ ∅ do
       // apply greedy heuristic
 5     if ∃ i : |B[i]| = i then
 6         ℓ := smallest such i
 7         v := value in B[ℓ] such that w(v) has the
           largest finish time
 8     else
 9         v := value in U such that w(v) has the
           largest finish time
10     end
11     remove v from U and pre-pend it to T
       // update obligation sets
12     W := subset of values v' ∈ U such that some
       read r(v') satisfies w(v) <_H r(v')
13     W' := subset of values v' ∈ U such that some
       value v'' ∈ W satisfies w(v'') <_H w(v')
14     for j := 1 to k − 2 do
15         B[j] := B[j + 1] \ {v}
16     end
17     B[k − 1] := (B[k − 1] ∪ W ∪ W') \ {v}
18     if ∃ i : |B[i]| > i then
19         return NO
20     end
21 end
22 return YES
```

Algorithm 2: The GPO algorithm.

(a) the algorithm executes line 11 at least x times and computes a sequence $T_x = \langle v_1, v_2, \ldots, v_x\rangle$ that is the suffix of some k-atomic total order for H;

(b) for any i such that $1 \le i \le k-1$ and for any value v, if $v \in B_x[i]$ then v is one of the i immediate predecessors of v_1 in any k-atomic total order for which T_x is a suffix, and hence $|B_x[i]| \le i$; and

(c) for any value v, if G_R contains an edge (a, b) such that a is $i \ge 0$ positions to the right of v_1 in T_x and b is not in T_x, then $b \in B_x[k-1-j]$ for all j such that $0 \le j \le i$.

Part (b) of the predicate $P(x)$ ensures that the algorithm never exits at line 19, and therefore outputs YES at line 22, as wanted. The predicate is proved by induction on x, and the most technically challenging aspect of the induction step is to establish $P(x+1)$-(a). To that end, we show that no matter which value v the algorithm chooses during iteration $x+1$ (at line 7 or 9), there is some k-atomic total order T for H such that T_x is a suffix of T. □

THEOREM 7. *The algorithm outputs YES if and only if H is k-atomic. Furthermore, the algorithm has time complexity $O(n^2 + n \cdot k \log k)$ where n is the number of operations in the input history H.*

PROOF SKETCH. Correctness of the return value follows from Lemmas 5 and 6. Assuming that U and $B[1]$ to $B[k-1]$ are represented using balanced trees, each iteration can be completed in $O(n + k \log k)$ steps: $O(n)$ to remove v from U as well as to compute W and W', and $O(k \log k)$ to update $B[i]$ for all i. We assume that each set $B[i]$ is implemented using a balanced tree, and that Algorithm 2 is optimized to return NO at line 15 or 17 if the size of $B[i]$ exceeds i, in accordance with lines 18–19. This optimization ensures that $B[\ell]$ at line 7 can be scanned in $O(k)$ time, that deletions from $B[i]$ at line 15 or 17 complete in $O(\log k)$ time, and that the set unions at line 17 complete in $O(k \log k)$ time. The total number of iterations is bounded by n and hence the overall time complexity is $O(n^2 + n \cdot k \log k)$. □

5. A GENERAL ALGORITHM

In this section we present a k-AV algorithm called CGS ("configuration graph search") that is correct for any history satisfying Assumptions 1–3, and whose worst case time complexity is polynomial only in the special case when both k and our specific measure of write concurrency are bounded by constants.

5.1 Algorithm description

The algorithm uses the graph-theoretic representations of the input history H described in Section 3 as Definitions 1 and 2. Its complexity depends on both the size of these graph structures and on a formal notion of concurrency among write operations captured in Definition 3.

DEFINITION 3. *The* write concurrency *of a history H is the maximum number of writes that any one write overlaps with, including itself.*

The algorithm works by analyzing possible topological orderings of G_W starting with small permutations of values called *configurations*, which are formalized in Definition 4. Intuitively, each configuration is a candidate contiguous subsequence of a k-atomic total order for H, and must therefore satisfy constraints related to edges of G_W and G_R in accordance with Lemma 2. Furthermore, it must be possible to compose smaller configurations to form larger contiguous subsequences of a k-atomic total order for H. These requirements have two implications regarding the size of a configuration S. First, for Lemma 2 the order of the $k-1$ immediate predecessors of at least one value in S (i.e., the last one) must be fixed, and so S must contain at least k values. Second, for composability the set of values outside of S must be partitioned into those that can only precede S in a topological ordering of G_W, and those that can only follow S. As a result, the number of values in S must be greater than or equal to the write concurrency of H, denoted in this section by m. Thus, S must have size at least $\max(m, k)$.

DEFINITION 4. *An (m, k)-configuration with respect to a read value graph $G_R(V, E_R)$ and write graph $G_W(V, E_W)$ is any sequence $S = \langle v_1, v_2, \ldots, v_l \rangle$ of $l = \max(m, k)$ values that satisfies the following properties:*

(a) S is a contiguous subsequence of some topological ordering of G_W;

(b) if G_R contains an edge (v, v') such that both v and v' are values in S, say $v = v_i$ and $v' = v_j$, then $j > i - k$; and

(c) if G_R contains an edge (v, v') such that v is the last value in S and v' is not in S, then S contains a value v'' such that G_W contains the edge (v'', v').

The algorithm attempts to find a k-atomic total order for the input history by stringing together (m, k)-configurations. To that end, the algorithm searches for a sufficiently long path in a directed graph of configurations where the edge set represents the *extends* relation formalized in Definition 5.

DEFINITION 5. *For any (m, k)-configurations C and C' with respect to a read value graph $G_R(V, E_R)$ and write graph $G_W(V, E_W)$, we say that C'* extends *C if and only if C' is obtained from C by removing the first value v and appending a value $v' \neq v$ such such that $(v', v) \notin E_W$.*

As we show later on in Lemma 8, a path through the directed graph of configurations induces a non-repeating sequence of values whose order obeys the constraints imposed by the happens before relation $<_H$ for the input history H. As a result, the length of the longest path in this graph can be used to decide k-atomicity. The full procedure is presented as Algorithm 3, where the parameter k is given as input and the write concurrency of the input history, denoted by m, is computed explicitly at line 8.

```
   Input: history H and integer k, 2 ≤ k ≤ n.
   Output: YES if H is k-atomic, NO otherwise.
 1 G_R(V, E_R) := read value graph for H
 2 G_W(V, E_W) := write graph for H
 3 if |V| ≤ k then
 4 |   return YES
 5 else if ∃v ∈ V s.t.
   |{v' ∈ V | (v', v) ∈ E_W ∧ (v, v') ∈ E_R}| ≥ k then
 6 |   return NO
 7 else
 8 |   m := write concurrency of H
 9 |   C := set of (m, k)-configurations with respect to
   |   G_R and G_W
10 |   E_C := {(c, c') ⊆ C × C | configuration c' extends
   |   configuration c}
11 |   G_C := configuration graph (C, E_C)
12 |   if G_C is non-empty and contains a path with
   |   |V| − max(m, k) edges then
13 |   |   return YES
14 |   else
15 |   |   return NO
16 |   end
17 end
```

Algorithm 3: The CGS algorithm.

5.2 Examples

With respect to the example history presented in Figure 1, CGS works correctly for all $k \geq 1$. The write concurrency for this particular input is $m = 5$. Recall from Section 3 that the input history is 3-atomic but not 1-atomic or 2-atomic. For $k = 1$ the condition at line 5 holds with $v = 1$ and $v' = 2$, and so the algorithm correctly returns NO at line 6.

For $k = 2$ the condition at line 5 is false and so the algorithm proceeds to build the set $\mathcal{C}$ of $(5, 2)$-configurations at

line 9. According to Definition 4 there are no possible $(5,2)$-configurations because it is impossible to satisfy clause (b). As a result, the configuration graph G_C computed at line 11 is empty. Thus, the condition at line 12 is false and CGS correctly returns NO at line 15.

For $k = 3$ the condition at line 5 is false and so the algorithm proceeds to build the set $\mathcal{C}$ of $(5,3)$-configurations at line 9. According to Definition 4 the only possible $(5,3)$-configurations are $\langle 5,2,1,3,4\rangle$ and $\langle 5,2,3,1,4\rangle$, which are the two possible 3-atomic total orders for H. The configuration graph G_C computed at line 11 is therefore non-empty and contains a path with $|V| - \max(m,k) = 5-5 = 0$ edges. Thus, the condition at line 12 holds and CGS correctly returns YES at line 13.

Finally, consider the case when $w(5)$ is removed from the input. In that case the write concurrency becomes $m = 3$. For $k = 2$ the condition at line 5 is false and so the algorithm proceeds to build the set $\mathcal{C}$ of $(3,2)$-configurations at line 9. The possible $(3,2)$-configurations are: $\langle 2,1,4\rangle$, $\langle 2,3,4\rangle$, $\langle 1,3,4\rangle$, $\langle 1,4,3\rangle$, and $\langle 3,1,4\rangle$. However, the configuration graph G_C computed at line 11 does not contain any edges, and hence does not contain a path with $|V| - \max(3,2) = 4-3 = 2$ edges. Thus, the condition at line 12 is false and CGS correctly returns NO at line 15. For $k = 3$ the condition at line 5 is again false and the possible $(3,3)$-configurations include $\langle 2,1,3\rangle$, $\langle 1,3,4\rangle$, and several others. Since $\langle 1,3,4\rangle$ extends $\langle 2,1,3\rangle$ it follows that the graph G_C computed at line 11 is non-empty and contains a path with $|V| - \max(3,3) = 4-3 = 1$ edges. Thus, the condition at line 12 holds and CGS correctly returns YES at line 13.

5.3 Analysis

In this section we suppose that Assumptions 1–3 hold and we show that the algorithm returns YES if and only if the input history H is k-atomic. In our analysis any references to the *configuration graph* G_C pertain to the graph structure computed at line 11 of the algorithm. Detailed proofs of correctness are omitted due to lack of space.

DEFINITION 6. *Let $P = \langle C_1, C_2, C_3, \ldots, C_l\rangle$ be a finite path in the configuration graph G_C. The* sequence S of values induced by P *is defined recursively as follows: if $l = 1$ then $S = C_1$, otherwise S is the sequence of values induced by $\langle C_1, C_2, C_3, \ldots, C_{l-1}\rangle$ followed by the last value of C_l.*

LEMMA 8. *For any finite path $P = \langle C_1, C_2, C_3, \ldots, C_l\rangle$ in the configuration graph G_C, the sequence S of values induced by P is a subsequence of some topological ordering of G_W.*

COROLLARY 9. *Suppose that H contains $n_w \geq k$ writes. Then any path P through the configuration graph G_C has length (number of edges) at most $n_w - \max(m,k)$.*

LEMMA 10. *Suppose that H contains $n_w \geq k$ writes. Then H is k-atomic if and only if the configuration graph G_C is non-empty and has a path with exactly $n_w - \max(m,k)$ edges.*

THEOREM 11. *The algorithm returns YES if H is k-atomic, and NO otherwise.*

PROOF SKETCH. If $|V| \leq k$ then any topological ordering of G_W shows that H is k-atomic (Lemma 2) under Assumptions 1–3, and so the algorithm returns YES at line 4. Otherwise if the condition at line 5 holds then H is not k-atomic because in any topological ordering of G_W one of the vertices v' precedes v by more than $k-1$ positions (Lemma 2), and so the algorithm returns NO at line 6. Finally, if the algorithm reaches line 12 then k-atomicity is decided using Lemma 10 and the algorithm returns the correct response at line 13 or line 15. □

LEMMA 12. *The graph G_C of (m,k)-configurations for a history H with n_w writes has at most $n_w \cdot (2m-1)^{\max(m,k)-1}$ vertices and at most $n_w \cdot (2m-1)^{\max(m,k)}$ edges.*

PROOF SKETCH. In each (m,k)-configuration there are at most n_w choices for the first value, and for each value v there are at most $2m-1$ choices for the successor of v: at most $m-1$ values whose writes overlap with $w(v)$, and an independent set of at most m out-neighbors of v in G_W. □

THEOREM 13. *The time complexity of the algorithm is $O(n^2 + n \cdot \max(m,k)^2 \cdot (2m-1)^{\max(m,k)-1})$ where n denotes the number of operations in the input history H, and m is the write concurrency of H.*

PROOF SKETCH. $O(n^2)$ steps are required to construct G_R and G_W, as well as some intermediate structures that speed up the construction of G_C. Computing m at line 8 requires $O(n^2)$ steps. The total number of vertices in G_C is at most $n \cdot (2m-1)^{\max(m,k)-1}$ by Lemma 12 and each vertex can be identified along with its in-edges in $O(\max(m,k)^2)$ steps given that the condition tested at line 5 of the algorithm is false. □

6. PRACTICAL APPLICABILITY

In this section we investigate the time complexity of algorithm CGS (Section 5) in practice by computing the parameters n (number of operations) and m (write concurrency, see Definition 3) from experimental data. The data set is borrowed from the experimental study of Γ-atomicity [14].

Experimental environment.

The experiments were conducted using ten 64-bit 2.2 GHz dual-core AMD Opteron servers equipped with 7200 RPM SATA disks and Gigabit Ethernet. Five servers were used to execute Cassandra 1.2.4 [17] and another five to run the Yahoo Cloud Serving Benchmark (YCSB) 0.1.4 [6]. The software environment included CentOS 5.5 Linux and OpenJDK 1.7.0_19. Clocks skew across servers was estimated using the `ntpq` command as less than 10ms.

Translation of execution histories into k-AV test cases.

The data set includes five sets of experiments, where each experiment comprises multiple (3-9) runs with different combinations of storage system settings and workload parameters. In each run the benchmark (YCSB) applies operations to the storage system (Cassandra) for 60 seconds, usually at the peak throughput level of approximately 1000 operations/s/server, which yields an execution history containing hundreds of thousands of operations.

Although the history generated by each of the 49 experimental runs could be used directly as an instance of k-AV, each object accessed in the history can be analyzed separately thanks to the locality property of k-atomicity, as explained in Section 2. Therefore, we first project the history obtained from a given run into a set of sub-histories corresponding to distinct objects. Furthermore, each of these sub-histories can often be decomposed into smaller fragments

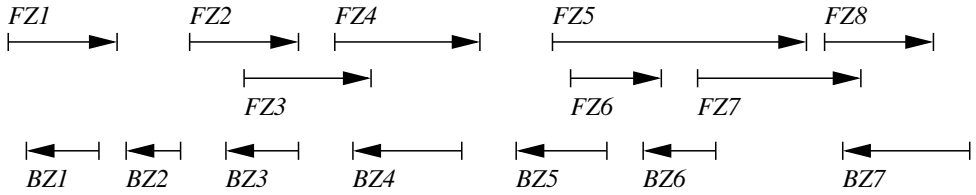

Figure 2: Example of zones in a history, reproduced from [12].

that can be analyzed separately—a technique used by the FZF algorithm for solving 2-AV [12]. The decomposition is based upon Gibbons and Korach's method of clustering operations into *zones*, each of which comprises a write operation and its dictated reads [10]. A zone is represented as the time interval from the minimum finish time of any operation in the zone to the maximum start time of any operation in the zone. A *forward zone* occurs when the minimum finish time is less than the maximum start time, meaning that at least one dictated read starts after some other operation in the zone finishes. A *backward zone* occurs when the minimum finish time is greater than or equal to the maximum start time, meaning that all operations in the zone overlap at a common point in time. For completeness we include in Figure 2 a copy of Figure 3 from [12], which illustrates a possible combination of zones in one history. In this figure *FZx* and *BZx* indicate forward and backward zones, respectively, and time increases from left to right.

The decomposition procedure groups zones into maximal subsets called *chunks* such that (i) two forward zones are in the same chunk if they intersect, and (ii) a backward zone belongs to a chunk if its time interval is contained entirely in the union of the time intervals of the forward zones in that chunk. As an example, in Figure 2 there are three chunks: $\{FZ_1, BZ_1\}$, $\{FZ_2, FZ_3, FZ_4, BZ_3, BZ_4\}$, and $\{FZ_5, FZ_6, FZ_7, FZ_8, BZ_6\}$. The remaining zones BZ_2, BZ_5 and BZ_7 can be ignored in the context of k-AV, intuitively because their operations can be linearized easily in-between chunks. It is straightforward to show that under Assumptions 1–2 an execution history is k-atomic if and only if each chunk of that history is k-atomic.

Experimental results.

We applied the above translation procedure to the 49 execution histories in the data set, which generated more than two million chunks. For each history we computed the total number of chunks, total number of zones, total number of operations, maximum number of operations per chunk, maximum write concurrency, number of chunks with $m \leq 5$, number of chunks that satisfy Assumption 4, and number of chunks where $m > 5$ and Assumption 4 is not satisfied.

The detailed results, which we omit due to lack of space, show that the chunks are quite small, generally comprising fewer than 200 operations and 4-5 operations per zone on average. Thus, the problem instances are fairly small but not small enough to permit solving k-AV by brute force. More than 99.3% of the chunks satisfy Assumption 4 and can therefore be analyzed efficiently using the GPO algorithm from Section 4. More than 99.9% of the chunks have write concurrency $m \leq 5$, and hence they are likely good candidates for the CGS algorithm from Section 5 provided that k is similarly small. The remaining chunks, where $m > 5$ and Assumption 4 does not hold, account for fewer than 0.1% of the test cases. These inputs may be too complex for our CGS algorithm.

7. CONCLUSION AND DISCUSSION

In this paper we have presented two algorithms that solve the k-AV problem for arbitrary $k \geq 2$ in special cases. Our algorithms assume that each read has exactly one dictating write (Assumption 1), which circumvents NP-completeness when $k \leq 2$. The first algorithm (GPO) places an additional restriction on the structure of the execution history but always runs in polynomial time. The second algorithm (CGS) does not place any additional restrictions on the input but its running time is polynomial only if both k and our measure of write concurrency (Definition 3) are bounded by constants. The time complexity of k-AV under Assumption 1, with no additional restrictions, remains unsettled.

Our algorithms enable precise measurement of version-based staleness, which in turn has several applications: (i) analyzing and understanding the behavior of eventually consistent storage systems (e.g., [7, 17, 23]); (ii) validating mathematical models of consistency (e.g., [3]); as well as (iii) verifying that a storage system or consistency tuning framework (e.g., [2, 24, 29]) delivers a promised level of consistency, and quantifying the severity of any consistency violations observed.

Measurement of version-based staleness entails computing the k-value of a history H, denoted k_H, which is the smallest integer $k \geq 1$ for which H is k-atomic. This value can be computed by a brute force method where k-AV is solved for consecutive values of k starting at $k = 1$, or by executing a binary search over $1 \leq k \leq n_w$ where n_w is the number of writes in H. When k_H is sufficiently large compared to m, the brute force approach is optimal for an exponential time algorithm such as CGS because the time complexity is dominated by the cost of solving k-AV for $k = k_H + 1$. Otherwise the computation may benefit from binary search with the overall time complexity bounded by $O(\log n_w)$ times the cost of solving k-AV for $k = n_w$.

Given that our k-AV algorithms do not guarantee polynomial running time without additional restrictions, their practical value depends on how often these restrictions hold in real data sets. To shed light on this point we analyzed the execution histories from [14] to determine representative values of n (number of operations) and m (write concurrency, Definition 3). The results suggest that in practice n is typically small (<1000), and the vast majority (>99%) of k-AV instances satisfy Assumption 4, which makes GPO applicable. In most of the remaining cases m is small (≤ 5) and hence CGS is efficient for small k (e.g., $k \leq m$). A very small fraction of cases (<0.1%) fail to satisfy both Assumption 4 and $m \leq 5$, and may be too complex for CGS.

8. REFERENCES

[1] A. Aiyer, L. Alvisi, and R. A. Bazzi. On the availability of non-strict quorum systems. In *Proc. of the 19th International Symposium on Distributed Computing (DISC)*, pages 48–62, September 2005.

[2] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–381, 2014.

[3] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8):776–787, 2012.

[4] D. Bermbach and S. Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proc. of the 6th Workshop on Middleware for Service Oriented Computing (MW4SOC)*, 2011.

[5] E. A. Brewer. Towards robust distributed systems (abstract). In *Proc. of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, June 2010.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM Symposium on Operating System Principles (SOSP)*, pages 205–220, October 2007.

[8] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, December 1976.

[9] M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics*, 34(3):477–495, May 1978.

[10] P. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26:1208–1244, August 1997.

[11] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[12] W. Golab, J. Hurwitz, and X. Li. On the k-atomicity-verification problem. In *Proc. of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[13] W. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. In *Proc. of the 30th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 197–206, June 2011.

[14] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *Proc. of the 34th International Conference on Distributed Computing Systems (ICDCS)*, pages 493–502, 2014.

[15] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[16] D. J. Kleitman and R. V. Vohra. Computing the bandwidth of interval graphs. *SIAM Journal on Discrete Mathematics*, 3(3):373–375, August 1990.

[17] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[18] L. Lamport. On interprocess communication, Part I: Basic formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, June 1986.

[19] H. Lee and J. L. Welch. Randomized registers and iterative algorithms. *Distributed Computing*, 17(3):209–221, 2005.

[20] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[21] C. H. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, September 1976.

[22] J. Saxe. Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM Journal on Algebraic and Discrete Methods*, 1(4):363–369, December 1980.

[23] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project Voldemort. In *Proc. of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[24] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proc. of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 309–324, 2013.

[25] 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 *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–182, 1995.

[26] F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *Proc. of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 163–172, 1999.

[27] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, Oct. 2008.

[28] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. In *Proc. of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2011.

[29] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.