Online Aggregation over Trees

C. Greg Plaxton, Mitul Tiwari
University of Texas at Austin

Praveen Yalagandula
HP Labs

Abstract

Consider a distributed network with nodes arranged in a tree, and each node having a local value. We formulate an aggregation problem as the problem of aggregating values (e.g., summing values) from all nodes to the requesting nodes in the presence of writes. The goal is to minimize the total number of messages exchanged. The key challenges are to define a notion of "acceptable" aggregate values, and to design algorithms with good performance that are guaranteed to produce such values. We formalize the acceptability of aggregate values in terms of certain consistency guarantees similar to traditional consistency models defined in the distributed shared memory literature. The aggregation problem admits a spectrum of solutions that trade off between consistency and performance. The central question is whether there exists an algorithm in this spectrum that provides strong performance and good consistency guarantees. We propose a lease-based aggregation mechanism, and evaluate algorithms based on this mechanism in terms of consistency and performance. With regard to consistency, we generalize the definitions of strict and causal consistency for the aggregation problem. We show that any lease-based aggregation algorithm provides strict consistency in sequential executions, and causal consistency in concurrent executions. With regard to performance, we propose an online lease-based aggregation algorithm, and show that, for sequential executions, the algorithm is constant-competitive against any offline algorithm that provides strict consistency. Our online lease-based aggregation algorithm is presented in the form of a fully distributed protocol, and the aforementioned consistency and performance results are formally established with respect to this protocol. Thus, we provide a positive answer to the central question posed above.

[1] Department of Computer Science, University of Texas at Austin, 1 University Station C0500, Austin, Texas 78712–0233.

[2] Email: plaxton@cs.utexas.edu. Supported by NSF Grant CCR–0310970 and ANI–0326001.

[3] Email: mitult@cs.utexas.edu. Supported by NSF Grant ANI–0326001 and Texas Advanced Technology Program 003658-0608-2003.

[4] HP Labs, 1501 Page Mill Rd MS 1181, Palo Alto, California 94304. Email: praveen.yalagandula@hp.com.

1 Introduction

Information aggregation is a basic building block in many large-scale distributed applications such as system management [10, 21], service placement [9, 22], file location [5], grid resource monitoring [7], network monitoring [13], and collecting readings from sensors [14]. Certain generic aggregation frameworks [7, 17, 23] proposed for building such distributed applications allow scalable information aggregation by forming tree like structures with machines as nodes, and by using an aggregation function at each node to summarize the information from the nodes in the associated subtree.

Some of the existing aggregation frameworks use strategies optimized for certain workloads. For example, in MDS-2 [7], the information is aggregated only on reads, and no aggregation is performed on writes. This kind of strategy performs well for write-dominated workloads, but suffers from unnecessary latency or imprecision on read-dominated workloads. On the other hand, Astrolabe [17] employs the other extreme form of strategy in which, on a write at a node u in the tree, each node v on the path from u to the root node recomputes the aggregate value for the subtree rooted at node v, and the new aggregate values are propagated to all the nodes. This kind of strategy performs well for read-dominated workloads, but consumes high bandwidth when applied to write-dominated workloads. Furthermore, instead of these two extreme forms of workloads, the workload may fluctuate and different nodes may exhibit activity at different times. Therefore, a natural question to ask is whether one can design an aggregation strategy that is adaptive and works well for varying workloads.

SDIMS [23] proposes a hierarchical aggregation framework with a flexible API that allows applications to control the update propagation, and hence, the aggregation aggressiveness of the system. Though SDIMS exposes such flexibility to applications, it requires applications to know the read and write access patterns a priori to choose an appropriate strategy (see our discussion on related work for further details). Thus, SDIMS leaves an open question of how to adapt the aggregation strategy in an online manner as the workload fluctuates.

In this work, we design an online aggregation algorithm, and show that the total number of messages required to execute a given set of requests is within a constant factor of the minimum number of messages required to execute the requests. We give the complete algorithm description in the abstract protocol notation [11], and also believe that our algorithm is practical.

Broader Perspective. The ever increasing complexity of developing large-scale distributed applications motivates a research agenda based on the identification of key distributed primitives, and the design of reusable modules for such primitives. To promote reuse, these modules should be “self-tuning”, that is, should provide near optimal performance under wide range of operating conditions. As indicated earlier, aggregation is useful in many applications. In this work we design a distributed protocol for aggregation that provides good performance guarantees under any operating conditions. Our focus on tree networks is not limiting since many large-scale distributed applications tend to be hierarchical (tree-like) in nature for scalability. If the network is not a tree, one can use standard techniques to build a spanning tree. For example, in SDIMS [23], nodes are arranged in a distributed hash table (DHT), and trees embedded in the DHT are used for the aggregation; these trees are automatically repaired in the face of failures. The present work can be viewed as a case study within the broader research agenda alluded to above. The techniques developed here may find application in the design of self-tuning modules for other primitives.

Problem Formulation. In order to describe our results we next present a brief description of the problem formulation; see Section 2 for a detailed description. We consider a distributed network with nodes arranged in an unrooted tree and each node having a local value. We formulate the aggregation problem as the problem of aggregating values (e.g., computing min, max, sum, or average) from all the nodes to the requesting nodes in the presence of writes. The goal is to minimize the total number of messages exchanged.

The main challenges are to define acceptable aggregate values in presence of concurrent requests, and to design algorithms with good performance that produce the acceptable aggregate values. We define the acceptability of the aggregate values in terms of certain consistency guarantees. There is a spectrum of solutions that trade off between consistency and performance. We introduce a mechanism that uses the concept of leases for aggregation algorithms. Any aggregation algorithm that uses this mechanism is called lease-based aggregation algorithm. The notion of a lease used in our mechanism is a generalization of that used in SDIMS [23].

Results. We evaluate the lease-based aggregation algorithms in terms of consistency and performance. In terms of consistency, we generalize the notions of strict and causal consistency, traditionally defined for distributed shared memory [20, Chapter 6], for the aggregation problem. We show that any lease-based aggregation algorithm provides strict consistency for sequential executions, and causal consistency for concurrent executions.

In terms of performance, we analyze the lease-based algorithms in the competitive analysis framework [19]. In this framework, we compare the cost of an online algorithm with respect to an optimal offline algorithm. An online aggregation algorithm executes each request without any knowledge of the future requests. On the other hand, an offline aggregation algorithm has knowledge of all the requests in advance. An online algorithm is c*-competitive* if, for any request sequence σ, the cost incurred by the online algorithm in executing σ is at most c times that incurred by an optimal offline algorithm.

As is typical in the competitive analysis of distributed algorithms [2, 3], we focus on sequential executions. In this paper we present an online lease-based aggregation algorithm RWW which, for sequential executions, is $\frac{5}{2}$-competitive against an optimal offline lease-based aggregation algorithm. We use a potential function argument to show this result. We also show that the result is tight by providing a matching lower bound. Further, we show that, for sequential executions, RWW is 5-competitive against an optimal offline algorithm that provides strict consistency.

The three main contributions of the work are as follows. First, we design an online aggregation algorithm and show that our algorithm achieves good competitive ratio for sequential executions. Second, we define the notion of causal consistency for the aggregation problem. Third, we show that our algorithm satisfies the definition of causal consistency for concurrent executions.

An interesting highlight of the techniques is the design of the aggregation algorithm that effectively reduces the analysis to reasoning about a pair of neighboring nodes. This reduction allows us to formulate a linear program of small size, independent of tree size, for the analysis.

Related Work. Various aggregation frameworks have been proposed in the literature such as SDIMS [23], Astrolabe [17], and MDS [7]. SDIMS is a hierarchical aggregation framework that utilizes DHT trees to aggregate values. SDIMS provides a flexible API that allows applications to decide how far the updates to the aggregate value due to the writes should be propagated. In particular, SDIMS supports *Update-local*, *Update-all*, and *Update-up* strategies. In Update-local strategy, a write affects only the local value. In Update-all strategy, on a write, the new aggregate value is propagated to all the nodes. In Update-up strategy, on a write, the new aggregate value is propagated to the root node of the hierarchy. Astrolabe is an information management system that builds a single logical aggregation tree over a given set of nodes. Astrolabe propagates all updates to the aggregate value due to the writes to all the nodes, hence, allows all the reads to be satisfied locally. MDS-2 also forms a spanning tree over all the nodes. MDS-2 does not propagate updates on the writes, and each request for an aggregate value requires all nodes to be contacted.

There are some similarities between our lease-based aggregation algorithm and prior caching work. Due to the space limitations, here we are describing the most relevant work. In CUP [18], Roussopoulos and Baker propose a *second-chance* algorithm for caching objects along the routing path. The algorithm removes a cached object after two consecutive updates are propagated to the remote locations due to the writes on that object at the source. The second-chance algorithm has been evaluated experimentally, and shown to provide good performance. In the distributed file allocation [3], Awerbuch et al. consider replication algorithm for a general network. In their algorithm, on a read, the requested object is replicated along the path from the destination to the requesting node. On a write, all copies are deleted except the one at the writing node. Awerbuch et al. showed that their distributed algorithm has poly-logarithmic competitive ratio for the distributed caching problem against an optimal centralized offline algorithm.

The concept of time-based leases has been proposed in literature to maintain consistency between the cached copy and the source. This kind of leases has been applied in many distributed applications such as replicated file systems [12] and web caching [8].

Ahamad et al. [1] gave the formal definition of causal consistency for distributed message passing system. The key difference between their setup and ours is in reading one value compared to aggregating values from all the nodes.

There are several efforts to deal with numerical error in the aggregate value such as [4, 16]. However, in our knowledge, none of these work give a competitive online algorithm for the aggregation problem, and neither of them address the issue of ordering semantics in concurrent executions. In [4], Bawa et al. defined semantics for various scenarios such as approximate aggregation in a faulty environment called *approximate single-site validity*. They designed algorithms that provide such semantics, and evaluated their algorithms experimentally. In [16], Olston and Widom consider one level hierarchy and propose a new class of replication system TRAPP that allows user to control the tradeoff between precision (numerical error) and performance in terms of communication overhead.

Organization. In Section 2 we introduce definitions and aggregation problem statements. In Section 3 we give an informal description of our algorithm and analysis. In Section 4 we define the class of lease-based aggregation algorithms, and establish certain properties of such algorithms. In Section 5 we present our online lease-based aggregation algorithm RWW, and establish bounds on the competitive ratio of RWW for sequential executions. In Section 6 we define the notion of a causally consistent aggregation algorithm, and establish that any lease-based algorithm, including RWW, is causally consistent.

2 Preliminaries

Consider a finite set of nodes (i.e., machines) arranged in a tree network T with reliable FIFO communication channels between neighboring nodes. We are also given an aggregation operator $\oplus$ that is commutative, associative, and has an identity element 0. For convenience, we write, $x \oplus y \oplus z$ as $\oplus(x, y, z)$. For the sake of concreteness in this paper, we assume that the local value associated with each node is a real value, and the domain of $\oplus$ is also real.

The *aggregate value* over a set of nodes is defined as $\oplus$ computed over the local values of all the nodes in the set. That is, the aggregate value over a set of nodes $\{v_1, \ldots, v_k\}$ is $\oplus(v_1.val, \ldots, v_k.val)$, where $v_i.val$ is the local value of the node v_i. The *global aggregate value* is defined as the aggregate value over the set of all the nodes in the tree T.

A request is a tuple $(node, op, arg, retval)$, where $node$ is the node where the request is initiated, op is the type of the request, either $combine$ or $write$, arg is the argument of the request (if any), and $retval$ is the return value of the request (if any). To execute a $write$ request, an aggregation algorithm takes the argument of the request and updates the local value at the requesting node. To execute a $combine$ request, an aggregation algorithm returns a value. Note that this definition admits the trivial algorithm that returns 0 on any $combine$ request. We define certain correctness criteria for aggregation algorithms later in the paper. Roughly speaking, the returned value on a $combine$ request corresponds to the global aggregate value.

The *aggregation problem* is to execute a given sequence of requests with the goal of minimizing the total number of messages exchanged among nodes. For any aggregation algorithm $\mathcal{A}$ and any request sequence σ, we define $C_{\mathcal{A}}(\sigma)$ as the total number of messages exchanged among nodes in executing σ by $\mathcal{A}$. An online aggregation algorithm $\mathcal{A}$ is c-competitive if for all request sequences σ and an optimal offline aggregation algorithm $\mathcal{B}$, $C_{\mathcal{A}}(\sigma) \leq c \cdot C_{\mathcal{B}}(\sigma)$ [6, Chapter 1].

We say T is in quiescent state if (1) there is no pending request at any node; (2) there is no message in transit across any edge; and (3) no message is sent until the next request is initiated. In short, T is in quiescent state if there is no activity in T until the next request is initiated.

In a sequential execution of a request, the request is initiated in a quiescent state and is completed when T reaches another quiescent state. In a sequential execution of a request sequence σ, every request q in σ is executed sequentially. In a concurrent execution of a request sequence, a new request can be initiated and executed while another request is being executed. We refer to the aggregation problem in which the given request sequence is executed sequentially as *sequential aggregation problem*.

The aggregation function f is defined over a set of real values or over a set of write requests. For a set A of real values $x_1, \ldots, x_m$, $f(A)$ is defined as $\oplus(x_1, \ldots, x_m)$. For a set A of write requests $q_1, \ldots, q_m$, $f(A)$ is defined as $f(A) = \oplus(q_1.arg, \ldots, q_m.arg)$.

For any request q in a request sequence σ, let $A(\sigma, q)$ be the set of the most recent writes preceding q in σ corresponding to each of the nodes in T. We say that an aggregation algorithm provides *strict consistency* in executing σ if any $combine$ request q in σ returns $f(A(\sigma, q))$ as the global aggregate value at $q.node$. Note that this definition of strict consistency for an aggregation algorithm is a generalization of the traditional definition of strict consistency for distributed shared memory systems (for further details, see [20, Chapter 6]). We define an aggregation algorithm to be *nice* if the algorithm provides strict consistency for sequential executions.

The set of all nodes in tree T is represented by $nodes(T)$. For any edge (u, v), removal of (u, v) yields two trees, $subtree(u, v)$ is defined to be one of the trees that contains u.

For any request sequence σ and any ordered pair of neighboring nodes (u, v), we define $\sigma(u, v)$ as follows: (1) $\sigma(u, v)$ is a subsequence of σ; (2) for any $write$ request q in σ such that $q.node$ is in $subtree(u, v)$, q is in $\sigma(u, v)$; and (3) for any $combine$ request q in σ such that $q.node$ is in $subtree(v, u)$, q is in $\sigma(u, v)$.

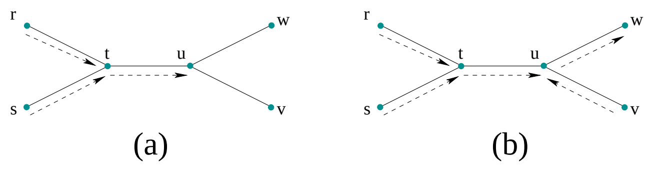

Figure 1: An example tree network.

3 Informal Overview

In this section we present an informal overview of our algorithm and analysis.

Recall that on a combine request at a node u, u returns a value. Roughly speaking, the value corresponds to the global aggregate value. In order to do that, u contacts other nodes and collects the local values from all the other nodes. Note that we can minimize the number of messages by performing aggregation at intermediate nodes, also referred as in-network aggregation.

However, for a combine-dominated workload, one may wish to propagate an updated local value on a write request to minimize the number of messages exchanged on a combine. On the other hand, for a write-dominated workload, such propagation tend to be wasteful. In order to facilitate adaptation of how many messages to send on a combine request versus a write request, we propose a lease mechanism. Here, we illustrate our lease mechanism for just two nodes u and v connected by an edge, and a scenario in which combine requests are initiated at v and write requests are initiated at u. It turns out that the other scenario is symmetric. (See Section 4 for the complete description of the mechanism.)

If the lease from u to v is present, then on a write request at u, u propagates the new local value to v by sending an update message. Hence, in the presence of this lease, a combine request at v is executed locally. On the other hand, if the lease from u to v is not present, then on a combine request at v, a probe message is sent from v to u. As a result, a response message containing the local value at u is sent from u to v. Further, in this case, a write request at u is executed locally. Note that on a combine-dominated scenario, presence of the lease is beneficial. However, on a write-dominated scenario, v may receive many updates while v is not initiating any request. In that case, v can break the lease by sending a release message to u.

In order to make the lease mechanism work for a tree network in a desirable way, we enforce two lease invariants. Consider the tree network in Figure 1 as an example. The presence of a lease on an edge is denoted by a dotted line. To illustrate the first invariant, consider a combine request q at node w with leases as in Figure 1(a). During the execution of q, w sends messages and collects the local values from all the other nodes. If the lease from t to u is present, then u need not send any message to t. However, this would work only if t has leases from r and s. Our first invariant ensures that the lease from t to u is not set unless t has leases from all the other neighboring nodes. Second invariant ensures that the lease from t to u can not be broken if u has given a lease to any other neighboring node, say node w in Figure 1(b).

Given this lease mechanism, an aggregation algorithm can adapt how far an updated value should be propagated on a write request by setting and breaking leases appropriately. The next question is how to set and break the leases dynamically in an optimal manner. We answer this question by providing an online lease-based aggregation algorithm RWW (see Section 5). Roughly, RWW works as follows. For an edge (u, v), RWW sets the lease from u to v during the execution of a combine request at any node in $subtree(v, u)$, and breaks the lease after two consecutive write requests at any node in $subtree(u, v)$. Using a potential function argument, we show that RWW is $\frac{5}{2}$-competitive against any offline lease-based algorithm for sequential executions. We also show that this bound is tight by providing lower bound arguments. Further, we show that RWW is 5-competitive against any offline algorithm that provides strict consistency for sequential executions.

With respect to consistency guarantees, we show that any lease-based aggregation algorithm provides strict consistency for sequential executions. For concurrent executions, it is difficult to provide strict or sequential consistency. Causal consistency is considered to be the next weaker consistency model for the distributed shared memory environment [20, Chapter 6]. At first, it is not clear how to generalize the causal consistency definitions for the aggregation problem.

We define the causal consistency for the aggregation problem and show that any lease-based algorithm provides causal consistency for concurrent executions (see Section 6). First, we introduce a new type of ghost requests $gather$ to associate a combine request with a set of write requests. The concept of gather requests is similar to the way of associating a read request with a unique write request in analyzing distributed shared memory [1, 15]. Second, we define causal ordering among gather and write requests. Third, we extend the lease-based mechanism by adding ghost variables and ghost actions. Finally, we use an invariant style proof technique to show that any lease-based algorithm provides causal consistency in two steps. In the first step, we show that a ghost log maintained at each node, containing gather and write requests, respects causal ordering among requests. In the second step, we show that there is one-to-one correspondence between gather and combine requests, that is, for each gather request there is a combine request and vice-versa, such that the return value of the combine request is same as aggregation function computed over the set of write requests returned by the gather request.

4 Lease-Based Algorithms

In Section 3 we gave a high level description of an aggregation mechanism based on the concept of leases. See Figure 2 for the formal description of this mechanism; the underlined function calls represent stubs for policy decisions of lease setting and breaking. Throughout the remainder of this paper, any aggregation algorithm that uses this mechanism and defines the policy functions is said to be *lease-based*.

The status of the leases for an edge (u, v) is given by two boolean variables $u.taken[v]$ and $u.granted[v]$. Node u believes that the lease from v to u is set if and only if $u.taken[v]$ holds. Also, u believes that the lease from u to v is set if and only if $u.granted[v]$ holds. The local value at u is stored in $u.val$. For each neighbor v_i of u, $u.aval[v_i]$ represents the aggregate value computed over the set of nodes in $subtree(v_i, u)$. The following kinds of messages are sent by a lease-based algorithm: $probe$, $response$, $update$, and $release$.

Informally, for any node u, a lease from a node u to its neighboring node v works as follows. If $u.granted[v]$ holds then, on a $write$ request at any node in $subtree(u, v)$, u propagates the new aggregate value to v by sending an $update$ message. To break the lease (that is, to falsify $u.granted[v]$), a $release()$ message is sent from v to u. On the other hand, if $u.granted[v]$ does not hold then, on a $combine$ request at any node in $subtree(v, u)$, a $probe()$ message is sent from v to u. As a result, a $response$ message is sent from u to v.

```
      $node\ u$
      var $taken[]$ : array$[v_1,\ldots,v_k]$ of boolean;
        $granted[]$ : array$[v_1,\ldots,v_k]$ of boolean;
        $aval[]$ : array$[v_1,\ldots,v_k]$ of real;   $val$ : real;
        $uaw$ : set $\{$int$\}$; $pndg$ : set $\{node\}$;
        $snt[]$ : array$[v_1,\ldots,v_k]$ of set $\{node\}$;
        $upcntr$ : int; $sntupdates$ : set $\{\{node,$ int, int$\}\}$;
      init $val := 0$; $uaw := \emptyset$; $pndg := \emptyset$; $upcntr := 0$;
        $sntupdates := \emptyset$; $\forall v \in nbrs()$, $taken[v] :=$ false;
        $granted[v] :=$ false; $aval[v] := 0$; $snt[v] := \emptyset$;
      begin
$T_1$     true $\rightarrow \{combine\}$
1           $\underline{oncombine}(u)$;
2           foreach $v \in tkn()$ do
3             $uaw[v] := \emptyset$; od
4           if $u \notin pndg \rightarrow$
5             if $nbrs() \setminus tkn() = \emptyset \rightarrow$
6               return $gval()$;
7             $\square$ $nbrs() \setminus tkn() \neq \emptyset \rightarrow$
8               $sendprobes(u)$;
9               $snt[u] := nbrs() \setminus tkn()$; fi fi
$T_2$     true $\rightarrow \{write\ q\}$
1           $val := q.arg$;
2           if $grntd() \neq \emptyset \rightarrow$
3             $id := newid()$;
4             $forwardupdates(u, id)$; fi
$T_3$     $\square$ rcv $probe()$ from $w \rightarrow$
1           $\underline{probercvd}(w)$;
2           foreach $v \in tkn() \setminus \{w\}$ do
3             $uaw[v] := \emptyset$; od
4           if $w \notin pndg \rightarrow$
5             if $nbrs() \setminus \{tkn() \cup \{w\}\} = \emptyset \rightarrow$
6               $sendresponse(w)$;
7             $\square$ $nbrs() \setminus \{tkn() \cup \{w\}\} \neq \emptyset \rightarrow$
8               $sendprobes(w)$;
9               $snt[w] := nbrs() \setminus \{tkn() \cup \{w\}\}$; fi fi
$T_4$     $\square$ rcv $response(x, flag)$ from $w \rightarrow$
1           $\underline{responsercvd}(flag, w)$;
2           $\underline{aval[w] := x}$;
3           $taken[w] := flag$;
4           foreach $v \in pndg$ do
5             $snt[v] := snt[v] \setminus \{w\}$;
6             if $snt[v] = \emptyset \rightarrow$
7               $pndg := pndg \setminus \{v\}$;
8               if $v = u \rightarrow$
9                 return $gval()$;
10              $\square$ $v \neq u \rightarrow$
11                $sendresponse(v)$; fi fi od
$T_5$     $\square$ rcv $update(x, id)$ from $w \rightarrow$
1           $\underline{updatercvd}(w)$;
2           $\underline{aval[w] := x}$;
3           $uaw[w] := uaw[w] \cup id$;
4           if $grntd() \setminus \{w\} \neq \emptyset \rightarrow$
5             $nid = newid()$;
6             $sntupdates := sntupdates \cup \{w, id, nid\}$;
7             $forwardupdates(w, nid)$;
8           $\square$ $grntd() \setminus \{w\} = \emptyset \rightarrow$
9             $forwardrelease()$; fi
$T_6$     $\square$ rcv $release(S)$ from $w \rightarrow$
1           $\underline{releasercvd}(w)$;
2           $granted[w] :=$ false;
3           $onrelease(w, S)$;
      end
```

```
procedure $sendprobes(node\ w)$
  $pndg := pndg \cup \{w\}$;
  foreach $v \in nbrs() \setminus \{tkn() \cup sntprobes() \cup \{w\}\}$ do
    send $probe()$ to $v$; od

procedure $forwardupdates(node\ w, int\ id)$
  foreach $v \in grntd() \setminus \{w\}$ do
    send $update(subval(v), id)$ to $v$; od

procedure $sendresponse(node\ w)$
  if $(nbrs() \setminus \{tkn() \cup \{w\}\} = \emptyset) \rightarrow$
    $granted[w] := \underline{setlease}(w)$; fi
  send $response(subval(w), granted[w])$ to $w$;

boolean $isgoodforrelease(node\ w)$
  return $(grntd() \setminus \{w\} = \emptyset)$;

procedure $onrelease(node\ w,$ set $S)$
  Let $id$ is the smallest id in $S$;
  foreach $v \in tkn() \setminus \{w\}$ do
    Let $A$ be the set of tuples $\alpha$ in $sntupdates$
      such that $\alpha.node = v$ and $\alpha.sntid \geq id$;
    Let $\beta$ be a tuple in $A$
      such that $\beta.rcvid \leq \alpha.rcvid$, for all $\alpha$ in $A$;
    Let $S'$ be the set of ids in $uaw[v]$ with ids $\geq \beta.rcvid$;
    $uaw[v] := S'$;
    if $isgoodforrelease(v) \rightarrow$
      $\underline{releasepolicy}(v)$; fi od
  $forwardrelease()$;

procedure $forwardrelease()$
  foreach $v \in tkn()$ do
    if $isgoodforrelease(v) \rightarrow$
      if $taken[v] \wedge \underline{breaklease}(v) \rightarrow$
        $taken[v] :=$ false;
        send $release(uaw[v])$ to $v$;
        $uaw[v] := \emptyset$; fi fi od

int $newid()$
  $upcntr := upcntr + 1$;
  return $upcntr$;

real $gval()$
  $x := val$;
  foreach $v \in nbrs()$ do
    $x := f(x, aval[v])$; od
  return $x$;

real $subval(node\ w)$
  $x := val$;
  foreach $v \in nbrs() \setminus \{w\}$ do
    $x := f(x, aval[v])$; od
  return $x$;

set $nbrs()$
  return the set of neighboring nodes;
set $tkn()$
  return $\{v \mid v \in nbrs() \wedge taken[v] =$ true$\}$;
set $grntd()$
  return $\{v \mid v \in nbrs() \wedge granted[v] =$ true$\}$;
set $sntprobes()$
  return $\{snt[v_1] \cup \cdots \cup snt[v_k]\}$;
```

Figure 2: Mechanism for any lease-based algorithm. For the node u, $\{v_1, \ldots, v_k\}$ is the set of neighboring nodes.

4.1 Properties of any Lease-Based Algorithm for Sequential Executions

We define a *lease graph* $G(Q)$ in a quiescent state Q, as a directed graph with nodes as the nodes in T, and for any edge (u, v) in T such that $u.granted[v]$ holds, there is a directed edge (u, v) in $G(Q)$. For any two distinct nodes u and v, we define the u-parent of v as the parent of v in tree T rooted at u.

Lemma 4.1 *For a sequential execution of a request sequence, in any quiescent state, for any two neighboring nodes u and v, $u.taken[v] = v.granted[u]$.*

Proof. Consider any node v in $u.nbrs()$. Variable $u.taken[v]$ can be set to $\mathbf{true}$ from $\mathbf{false}$ only in Line 3 of T_4 if the *flag* in the received *response* message is $\mathbf{true}$. However, while sending the *response* message from v to u with *flag* set to $\mathbf{true}$, $v.granted[u]$ is set to $\mathbf{true}$ in $sendresponse()$.

While sending a *release* message from u to v, $u.taken[v]$ is falsified in $forwardrelease()$. However, on receiving the *release* message at v, $v.granted[u]$ is falsified in Line 2 of T_6. $\square$

Lemma 4.2 *For a sequential execution of a request sequence, in any quiescent state, for any node u and any node v in $u.nbrs()$, if $u.granted[v]$ then, for all nodes w in $u.nbrs() \setminus \{v\}$, $u.taken[w]$ holds.*

Proof. By inspection of code, $u.granted[v]$ can be set to $\mathbf{true}$ only in the procedure $sendresponse()$. By inspection of code of $sendresponse()$, $u.granted[v]$ can be set to $\mathbf{true}$ only if $u.nbrs() \setminus \{u.tkn() \cup \{v\}\} = \emptyset$. That is, $u.granted[v]$ can be set to $\mathbf{true}$ only if, for all nodes w in $u.nbrs() \setminus \{v\}$, $u.taken[w]$ holds.

Further, by inspection of code, $u.taken[w]$ is set $\mathbf{false}$ only in the procedure $forwardrelease()$. By inspection of code of $forwardrelease()$, $u.taken[w]$ can be set to $\mathbf{false}$ only if, for all nodes v in $u.nbrs() \setminus \{w\}$, $u.granted[v]$ is $\mathbf{false}$. That is, for any node v in $u.nbrs()$, if $u.granted[v]$ holds then, for any node w in $u.nbrs() \setminus \{v\}$, $u.taken[w]$ is not falsified. $\square$

Lemma 4.3 *Consider a sequential execution of a request sequence σ by a lease-based algorithm. For any $combine$ request q in σ, initiated at node u in a quiescent state Q, let A be the set of nodes v such that $v.granted[w]$ does not hold in Q, where w is the u-parent of v. In Q, for any node v in T, if $v.pndg = \emptyset$ and for any node w in $v.nbrs()$, $v.snt[w] = \emptyset$, then, during the execution of q, (1) $|A|$ $probe$ messages are sent, and any node v in A receives a $probe$ message from the u-parent of v; (2) $|A|$ $response$ messages are sent; any node v in A sends a $response$ message to the u-parent of v; (3) no $update$ or $release$ messages are sent.*

Proof. We prove part (1) by induction on the length of the path from u to any node v in A.

Base case (path length 1). By inspection of code of T_1, *probe* messages are sent to all nodes in $u.nbrs() \setminus \{u.tkn() \cup u.sntprobes() \cup \{u\}\}$. Since in the quiescent state Q, for any node v in T and any node w in $v.nbrs()$, $v.snt[w] = \emptyset$, $u.sntprobes() = \emptyset$. Hence, a *probe* message is sent to any node v in $u.nbrs()$ such that $u.taken[v]$ does not hold. By Lemma 4.1, in Q, $u.taken[v] = v.granted[u]$. Hence, any node v in A such that v is in $u.nbrs()$ and $v.granted[u]$ does not hold, receives a *probe* message from u.

Induction hypothesis. Any node v in A such that the length of the path from u to v is i receives a *probe* message from the u-parent of v.

Induction step. Consider a node v in A such that the length of the path from u to v is $(i+1)$. Let the u-parent of v is w. By the definition of A, $v.\mathit{granted}[w]$ does not hold in Q. Hence, by Lemma 4.1 and Lemma 4.2, $w.\mathit{granted}[u$-parent of $w]$ does not hold in Q. Thus, w is in A, and by induction hypothesis w receives a probe message from w'. By inspection of code of T_3, w sends a probe message to any node w' in $w.\mathit{nbrs}()$ such that $w.\mathit{taken}[w']$ does not hold. Since $w.\mathit{taken}[v]$ does not hold and the communication channels are reliable, v receives a probe message from w, the u-parent of v.

From above arguments, during the execution of q at least $|A|$ probe messages are sent. By the inspection of code, any node v in $A \cup \{u\}$ does not send any probe message to any node in $v.\mathit{tkn}() \setminus \{u$-parent of $v\}$. And so, it is straightforward to see that any node v in $\mathit{nodes}(T) \setminus A$ does not receive any probe message. Hence, during the execution of q only $|A|$ probe messages are sent.

We prove part (2) by reverse induction on the length of the path from u to any node v in A. Let the maximum length of the path from u to any node v in A be l.

Base case. Consider a node v in A such that the length of the path from u to v is l. By part (1), v receives a probe message from w, the u-parent of v. In the quiescent state Q, let B be $v.\mathit{nbrs}() \setminus \{v.\mathit{tkn}() \cup \{u$-parent of $v\}\}$. By Lemma 4.1, B must be $\emptyset$, otherwise, there would be a node in A with the length of the path from u equal to $l+1$. By inspection of code of T_3, if B is empty, then v sends back a $\mathit{response}$ message to w.

Induction hypothesis. Let any node v in A with the length of path from u equal to i, sends a $\mathit{response}$ message to the u-parent of v.

Induction step. Consider a node v in A such that the length of the path from u to v is $i-1$. Since v is in A, $i-1$ must be greater than 0. In Q, let B be $v.\mathit{nbrs}() \setminus \{v.\mathit{tkn}() \cup \{u$-parent of $v\}\}$.

By part (1), v receives a probe message from the u-parent of v. By given condition, in Q, $v.\mathit{sntprobes}()$ is empty. By inspection of code of T_3, if B is empty, then v sends a $\mathit{response}$ message back to the u-parent of v. Hence, the induction step succeeds.

Otherwise, v sends probe messages to each of the node in B, and sets $v.\mathit{pndg} = \{u$-parent of $v\}$ and $v.\mathit{snt}[u$-parent of $v] = B$. Since we are dealing with sequential execution, no node initiates any request during the execution of q. And so, v does not initiates any request or receives a probe message during the execution of q. Hence, $v.\mathit{pndg} \leq 1$.

By Lemma 4.1 and definition of A, any node in B is also present in A. Further, the length of the path from u to any node in B is i. Hence, by induction hypothesis, any node w in B sends a $\mathit{response}$ message to v. By inspection of code of T_4, on receiving the $\mathit{response}$ message, v removes w from $v.\mathit{snt}[u$-parent of $v]$. If $v.\mathit{snt}[u$-parent of $v]$ becomes empty, then v sets $v.\mathit{pndg} = \emptyset$, and sends a $\mathit{response}$ message to the u-parent of v. Hence, the induction step succeeds.

(3) Follows from the inspection of code. □

Lemma 4.4 *For any sequential execution of a request sequence σ, in any quiescent state, for any node u, (1) $u.\mathit{pndg} = \emptyset$; (2) for any node v in $u.\mathit{nbrs}()$, $u.\mathit{snt}[v] = \emptyset$;*

Proof. We prove by induction on the number of requests executed.

Base case: Initially, for any node v, $v.\mathit{pndg} = \emptyset$ and for any node w in $v.\mathit{nbrs}()$, $v.\mathit{snt}[w] = \emptyset$.

Induction hypothesis: In the quiescent state Q just after execution of i requests, for any node v, $v.\mathit{pndg} = \emptyset$ and for any node w in $v.\mathit{nbrs}()$, $v.\mathit{snt}[w] = \emptyset$.

Induction step: Consider the execution $(i+1)$st request q initiated in Q. If q is a write request, then by inspection of code, no probe or $\mathit{response}$ message are generated. Hence, for any node v,

$v.pndg$ and any node w in $v.nbrs()$, $v.snt[w]$ are not modified. Therefore, the execution of $(i+1)$st request preserves the claim of the lemma.

Otherwise, q is a $combine$ request, say at u. Consider execution of q. Let A be the set of nodes v such that $v.granted[w]$ does not hold at Q, where $w = u$-parent of v.

By hypothesis, in Q, for any node v, $v.pndg = \emptyset$ and for any node w in $v.nbrs()$, $v.snt[w] = \emptyset$.

First, consider any node v in $nodes(T) \setminus \{A \cup \{u\}\}$. By inspection of code, for any node v, $v.pndg$ and for any node w in $v.nbrs()$, $v.snt[w]$ can be modified only in T_1 (on a $combine$ request at v), in T_3 (on receiving a $probe$ message), or in T_4 (on receiving a $response$ message). In sequential execution of σ, v does not initiate any request during the execution of q. By Lemma 4.3, during the execution of q, any node in A receives a $probe$ message, and only $|A|$ $probe$ messages are sent. Hence, v does not receive any $probe$ message during the execution of q. By definition of A, u-parent of any node in A is in $A \cup \{u\}$. By Lemma 4.3, during the execution of q, $|A|$ $response$ messages are generated and any node in A sends a $response$ message to the u-parent of the node. Hence, v does not receive any $response$ message during the execution of q. Hence, $v.pndg$ and for any node w in $v.nbrs()$, $v.snt[w]$ remain unchanged, that is, $\emptyset$, during the execution of q.

Second, consider $v = u$. By inspection of code of T_1, if $u.nbrs() \setminus u.tkn() = \emptyset$, then u returns $gval()$, and so, $u.pndg$ and for any node w in $u.nbrs()$, $u.snt[w]$ remain unchanged, that is, remain $\emptyset$. Further, by Lemma 4.1 and Lemma 4.2, $|A| = \emptyset$. Hence, from the arguments in the previous paragraph, induction step succeeds, and the lemma follows.

Otherwise, if $u.nbrs() \setminus u.tkn() \neq \emptyset$. Then, since $u.sntprobes() = \emptyset$ by induction hypothesis, u sends a $probe$ message to each of the node in the set $u.nbrs() \setminus u.tkn()$, and u adds u to $u.pndg$ and sets $u.snt[u] = nodes.nbrs() \setminus u.tkn()$. Since in a sequential execution, a new request can be generated only in a quiescent state, no node generates any request until q is completed. Hence, u does not generate any request until q is completed, and by Lemma 4.3, u does not receive any $probe$ message from any node. Therefore, $|u.pndg| \leq 1$. By definition of A, any node w in $u.nbrs() \setminus u.tkn()$ is also in A. By Lemma 4.3, w sends back a $response$ message to u. By inspection of code of T_4, on receiving the $response$ message, u removes w from $u.snt[u]$. When $u.snt[u] = \emptyset$, that is, u has received $response$ messages from all the nodes to whom u has sent a $probe$ message, then, u sets $u.pndg = \emptyset$, and returns $gval()$.

Finally, consider any node v in A. By Lemma 4.3, v receives a $probe$ message from the u-parent of v, say w. Let C be $v.nbrs() \setminus \{v.tkn() \cup \{w\}\}$. By inspection of code of T_3, if $C = \emptyset$, then v sends a $response$ message to w, and $v.pndg$ and for any node w' in $v.nbrs()$, $v.snt[w']$ remains unchanged, that is, remains $\emptyset$.

Otherwise, if $C \neq \emptyset$. Then, since $v.sntprobes() = \emptyset$, v sends a $probe$ message to each of the node in C. By inspection of code of T_3, while sending a $probe$ messages, v adds w to $v.pndg$ and sets $v.snt[w] = C$. As argued in the preceding paragraph, in a sequential execution, $|v.pndg| \leq 1$. By Lemma 4.3, any node w' in C sends back a $response$ message to v. By inspection of code of T_4, on receiving the $response$ message, v removes w' from $v.snt[v]$. When $v.snt[w] = \emptyset$, that is, v has received $response$ messages from all the nodes in C, then, w sets $v.pndg = \emptyset$, and sends a $response$ message back to w.

Hence, after execution of q, for any node v in A, $v.pndg = \emptyset$ and for any node w in $v.nbrs()$, $v.snt[w] = \emptyset$. □

Lemma 4.5 *Consider a sequential execution of a request sequence σ by a lease-based algorithm. For any $write$ request q in σ initiated at node u in a quiescent state Q, let A be the set of nodes* *in T reachable from u in $G(Q)$. Then, during the execution of q, (1) any node v in A receives an $update$ message from the u-parent of v; (2) $|A|$ $update$ messages are sent; and (3) no $probe$ or $response$ messages are sent.*

Proof. (1) We prove by induction on the length of the path from u to any node v in A.

Base case (path length 1). By the inspection of code of T_2, $update$ messages are sent to all nodes in $u.grntd()$. That is, an $update$ is sent to any node v in A such that the length of the path from u to v is 1.

Induction hypothesis. Any node v in A such that the length of the path from u to v is i, receives an $update$ message from the u-parent of v.

Induction step. Consider a node v in A such that the length of the path from u to v is $(i+1)$. By induction hypothesis, the u-parent of v, say w, receives an $update$ message. By definition of A, $w.granted[v]$ holds. By inspection of code of T_5, w sends an $update$ message to v. Since the communication channels are reliable, v receives an $update$ message from w, the u-parent of v.

(2) From above arguments, at least $|A|$ $update$ messages are sent. By the inspection of code, any node v in $A \cup \{u\}$ does not send any $update$ message to any node in $v.nbrs() \setminus \{v.grntd() \cup \{u\text{-parent of } v\}\}$. And so, it is straightforward to see that any node v in $nodes(T) \setminus A$ does not receive any $update$ message. Hence, during the execution of q only $|A|$ $probe$ messages are sent.

(3) Follows from the inspection of code. □

Lemma 4.6 *For any node u, $u.granted[v]$ is set to $\mathbf{true}$ only while sending a $response$ message to v with flag set to $\mathbf{true}$.*

Proof. For any node u, $u.granted[v]$ can be set to $\mathbf{true}$ only in $sendresponse$ procedure. By the inspection of code, the lemma follows. □

Lemma 4.7 *For any node u, $u.granted[v]$ is set to* **false** *only on receiving a $release$ message from v.*

Proof. Follows from the inspection of code. □

Lemma 4.8 *Consider a sequential execution of a request sequence σ by a lease-based algorithm and any two neighboring nodes u and v.*

1. *Let a $combine$ request q in $\sigma(u,v)$ be initiated in a quiescent state Q. If $u.granted[v]$ does not hold in Q, then in execution of q, (i) a $probe$ message is sent from v to u; (ii) a $response$ message is sent from u to v; (iii) $u.granted[v]$ can be set to $\mathbf{true}$ while sending the $response$ message from v to u; and (iv) no $update$ or $release$ messages are sent. Otherwise, if $u.granted[v]$ holds, then in execution of q, no messages are exchanged between u and v.*

2. *Let a $write$ request q in $\sigma(u,v)$ be initiated in a quiescent state Q. If $u.granted[v]$ does not hold in Q, then in execution of q, no messages are exchanged between u and v. Otherwise, if $u.granted[v]$ holds in Q, then in execution of q, (i) an $update$ message is sent from u to v; (ii) a $release$ message from v to u can be sent; (iii) on receiving the $release$ message at u, $u.granted[v]$ is set to* **false***; and (iv) no $probe$ or $response$ messages are sent.*

3. *Let a $write$ request q in $\sigma(v,u)$ be initiated in a quiescent state Q. If $u.granted[v]$ holds in Q, then in execution of q, a $release$ message can be sent from v to u, and on receiving the $release$ message at u, $u.granted[v]$ is set to* **false**.

4. *In the execution of a $combine$ request in $\sigma(v,u)$, $u.granted[v]$ is not affected.*

Proof. Part (1) follows from Lemma 4.3, Lemma 4.4, and 4.6. Part (2) follows from Lemma 4.5, Lemma 4.7, and the inspection of code. Part (3) follows from Lemma 4.7 and the inspection of code. Part (4) follows from Lemma 4.3, Lemma 4.4, and Lemma 4.6. □

$u.granted[v]$ in Q	Request q in $\sigma(u,v)$	$u.granted[v]$ in Q'	Cost
false	R	**false**	2
false	R	**true**	2
false	W	**false**	0
false	N	**false**	0
true	R	**true**	0
true	W	**false**	2
true	W	**true**	1
true	N	**false**	1
true	N	**true**	0

Figure 3: For any two neighboring nodes u and v, possible changes in the value of $u.granted[v]$ and costs incurred by any lease-based algorithm in executing any request q from $\sigma(u,v)$. Here, q is initiated in the quiescent state Q and completed in the quiescent state Q'. A $release$ message sent during the execution of a $write$ request in $\sigma(v,u)$ is associated with a $noop$ (N) request.

Lemma 4.8 is summarized in Figure 3. A $release$ message sent during the execution of a $write$ request in $\sigma(v,u)$ is associated with a $noop$ (N) request in this figure.

For any node u, we define $I_1(u)$, $I_2(u)$, and $I_3(u)$ as follows. (1) $I_1(u)$: For the most recent $write$ request q at u, $u.val = q.arg$; (2) $I_2(u)$: For any $update$ or $response$ message m from any neighboring node v to u, $m.x = f(A)$, where A is the set of most recent write requests at each of the nodes in $subtree(v,u)$; and (3) $I_3(u)$: For any quiescent state Q and any node v in $u.tkn()$, $u.aval[v] = f(A(v))$, where $A(v)$ is the set of the most recent $write$ request at each of the nodes in $subtree(v,u)$. Let $I(u)$ be $I_1(u) \wedge I_2(u) \wedge I_3(u)$.

Lemma 4.9 *Consider a sequential execution of a request sequence σ by a lease-based algorithm. For any node u, if $I_1(u)$ and $I_3(u)$ hold just before an $update$ message m is sent from u to any node v in $u.nbrs()$, then $m.x = A$, where A is the set of the most recent $write$ requests at each of the nodes in $subtree(u,v)$.*

Proof. By Lemma 4.2, for any node v in $u.nbrs()$, if $u.granted[v]$ then, for all nodes w in $u.nbrs() \setminus \{v\}$, $u.taken[w]$ holds.

For any node w in $u.nbrs()$, let $A(w)$ be the set of the most recent $write$ requests preceding q in σ at each of the nodes in $subtree(w,u)$. By $I_3(u)$, if $u.taken[w]$ then, $u.aval[w] = f(A(w))$.

By the inspection of code, for any node v in $u.grntd()$, an $update$ message m is sent to v with $m.x = u.subval(v)$. Let $\{w_1, \ldots, w_k\}$ be $u.nbrs() \setminus \{v\}$ and B be the set of the most recent $write$ requests at each on the node in $subtree(u, v)$.

$$\begin{aligned} m.x &= subval(v) \\ &= f(u.val, aval[w_1], \ldots, aval[w_k] \\ &= f(q.arg, f(A(w_1)), \ldots, f(A(w_k))) \\ &= f(B) \end{aligned} \tag{1}$$

In the above equation, the second equality follows from the definition of function $subval()$. The third equality follows from $I_1(u)$ and $I_3(u)$. The last equality follows from the fact that $subtree(u, v) = \{u\} \cup subtree(w_1, u) \cup \cdots \cup subtree(w_k, u)$. □

Lemma 4.10 *Consider a sequential execution of a request sequence σ by a lease-based algorithm. For any node u, $I(u)$ is an invariant.*

Proof.

Initially, there are no $write$ request at u and $u.tkn()$ is empty. Hence, $I(u)$ holds.

$\{I(u)\}T_1\{I(u)\}$. $I_1(u)$, $I_2(u)$, and $I_3(u)$ are not affected.

$\{I(u)\}T_2\{I(u)\}$. Let the $write$ request q is initiated in the quiescent state Q. In execution of T_2, $I_1(u)$ is only affected in Line 1. By the inspection of code, Line 1 preserves $I_1(u)$. $I_3(u)$ is not affected in execution of T_2. If $u.grntd() \neq \emptyset$ in the quiescent state Q, then $I_2(u)$ is affected in the procedure $forwardupdates()$, invoked in Line 4. By Lemma 4.9, $I_2(u)$ is preserved in Line 4.

Therefore, $I_1(u) \wedge I_2(u) \wedge I_3(u)$ is preserved in the execution of T_2.

$\{I(u)\}T_3\{I(u)\}$. By the inspection of code, $I_1(u)$ and $I_3(u)$ are not affected. $I_2(u)$ is affected only in the procedure $sendresponse()$, invoked in Line 6 to send a $response$ message m to w. However, Line 6 is executed only if $u.nbrs() \setminus \{u.tkn() \cup \{w\}\}$ is empty. By $I_3(u)$, for any node v in $u.nbrs()$, if $u.taken[v]$, then $u.aval[v] = f(A)$, where A is the set of the most recent $write$ requests at each of the nodes in $subtree(v, u)$. As in the proof of Lemma 4.9, $m.x = f(B)$, where B is the set of the most recent $write$ requests at each of the node in $subtree(u, w)$.

$\{I(u)\}T_4\{I(u)\}$. $I_1(u)$ is not affected in T_4. In T_4, $I_3(u)$ is affected in Line 2 and $I_2(u)$ is affected in $sendresponse()$ procedure, invoked in Line 11.

In the following, for any node w' in $u.nbrs()$, let $B(w')$ be the set of the most recent $write$ requests at each of the node in $subtree(w, u)$.

Since $I_2(u)$ holds for the received $response$ message, after execution of Line 2, $u.aval[w] = f(B)$, where $B(w)$. Hence, $I_3(u)$ holds in the execution of Line 2.

To argue that $I_2(u)$ holds in Line 11, we show that just before the execution of Line 11, for each node w' in $u.nbrs() \setminus \{v\}$, $u.aval[w'] = f(B(w'))$.

By Lemma 4.3 and Lemma 4.5, a $response$ message from w is received during the execution of a $combine$ request, say q. We can assume that $q.node \neq u$, since Line 11 is executed only if $q.node \neq u$.

From Lemma 4.3, u is $q.node$-parent of w and v is $q.node$-parent of u. Let q be initiated in the quiescent state Q, and in quiescent state Q, let A be the set of nodes $u.nbrs() \setminus \{u.tkn() \cup \{v\}\}$.

Again by Lemma 4.3, during execution of q, u sends a $probe$ message to each of the node in A and receives a $response$ message from each of them. For each the received $response$ message from w, as argued above, after execution of Line 2, $u.aval[w] = f(B(w))$. By the inspection of code of T_3, while sending $probe$ messages, u sets $u.snt[v] = A$. By the inspection of code of T_4, on receiving a $response$ message from a node w, w is removed from $u.snt[v]$. Hence, Line 11 is executed only when u has received $response$ messages from all the nodes in A. Hence, just before execution of 11, for each of the node w' in A, $u.aval[w'] = B(w')$. By I_2, for each of the node w' in $u.tkn()$, $u.aval[w'] = B(w')$. Hence, just before the execution of Line 11, for each of the node w' in $u.nbrs \setminus \{v\}$, $u.aval[w'] = B(w')$. Hence, as in the proof of Lemma 4.9, for the $response$ message m sent to v, $m.x = f(C)$, where C is the set of the most recent $write$ requests at each of the node in $subtree(u, v)$.

$\{I(u)\}T_5\{I(u)\}$. $I_1(u)$ is not affected in the execution of T_5.

$I_3(u)$ is affected only in Line 2. Let A be the set of the most recent $write$ requests at each of the node in $subtree(w, u)$. By $I_2(u)$, $m.x = f(A)$. After Line 2 $u.aval[w] = f(A)$. Hence, $I_3(u)$ is preserved in Line 2.

If $u.grntd() \neq \emptyset$ in quiescent state Q, then $I_2(u)$ is affected in the procedure $forwardupdates()$, invoked in Line 7. By Lemma 4.9, $I_2(u)$ is preserved in Line 7.

Therefore, $I_1(u) \wedge I_2(u) \wedge I_3(u)$ is preserved in the execution of T_5.

$\{I(u)\}T_6\{I(u)\}$. $I_1(u)$, $I_2(u)$, and $I_3(u)$ are not affected. Hence, $I(u)$ is preserved. □

Lemma 4.11 *Any lease-based aggregation algorithm is nice.*

Proof. Follows from Lemma 4.3 and Lemma 4.10. □

From Lemma 4.11 and the definition of a nice aggregation algorithm, we have that any lease-based aggregation algorithm provides strict consistency in a sequential execution.

5 Competitive Analysis Results for Sequential Executions

```
var $lt$ : array$[v_1 \ldots v_k]$ of int;
  $granted$ : array$[v_1 \ldots v_k]$ of boolean;

procedure $\underline{oncombine}()$
  foreach $v \in tkn()$ do
    $lt[v] := 2$; od
procedure $\underline{probercvd}(node\ w)$
  foreach $v \in tkn() \setminus \{w\}$ do
    $lt[v] := 2$; od
boolean $\underline{setlease}(node\ w)$
  $lg[w] :=$ true;
  return true;
```

```
procedure $\underline{responsercvd}$(boolean $flag$, $node\ w$)
  if $flag \wedge (taken[w] =$ false$) \rightarrow$
    $lt[w] := 2$; fi
procedure $\underline{updatercvd}(node\ w)$
  if $(grntd() \setminus \{w\} = \emptyset) \wedge lt[w] > 0 \rightarrow$
      $lt[w] := lt[w] - 1$; fi
procedure $\underline{releasepolicy}(node\ v)$
  $lt[v] := max(0, lt[v] - |uaw[v]|)$;
procedure $\underline{releasercvd}(node\ w)$
  $lg[w] :=$ false;
boolean $\underline{breaklease}(node\ w)$
  return$(lt[w] = 0)$;
```

Figure 4: Policy decisions for RWW

We define RWW as an online lease-based aggregation algorithm that follows the policy decisions shown in Figure 4 for setting or breaking a lease.

Informally, RWW works as follows. For any edge (u, v), RWW sets the lease from u to v during the execution of a $combine$ request at any node in the $subtree(v, u)$, and breaks the lease after two consecutive $write$ requests at any nodes in $subtree(u, v)$.

5.1 Properties of RWW

For positive integers a and b, an online lease-based algorithm $\mathcal{A}$ is in the class of (a, b)*-algorithms* if, in a sequential execution of any request sequence σ by $\mathcal{A}$, for any edge (u, v), $\mathcal{A}$ satisfies the following condition: (1) if $u.granted[v]$ is $\mathbf{false}$, then it is set to $\mathbf{true}$ after a consecutive $combine$ requests in $\sigma(u, v)$; and (2) if $u.granted[v]$ is $\mathbf{true}$, then it is set to $\mathbf{false}$ after b consecutive $write$ requests in $\sigma(u, v)$.

For any ordered pair of neighboring nodes u and v, we define $type(u, v)$ messages as the following kinds of messages exchanged between u and v: (1) $probe$ messages from v to u; (2) $response$ messages from u to v; (3) $update$ messages from u to v; and (4) $release$ messages from v to u. For a lease-based algorithm $\mathcal{A}$ and a request sequence σ, we define $C_{\mathcal{A}}(\sigma, u, v)$, as the number of $type(u, v)$ messages in execution of σ by $\mathcal{A}$.

Lemma 5.1 *Consider a sequential execution of a request sequence σ by* RWW *and any two neighboring nodes u and v. Then, during the execution of any request from $\sigma(v, u)$, $u.granted[v]$ is not affected.*

Proof. First, consider the execution of any $combine$ request in $\sigma(v, u)$. By Lemma 4.3 and Lemma 4.4, no $update$ or $release$ messages are sent. Further, no $response$ message from u to v are sent. Hence, $u.granted[v]$ is not affected during the execution of any $combine$ request in $\sigma(v, u)$.

Second, consider the execution of any $write$ request in $\sigma(v, u)$. By Lemma 4.5, no $probe$ or $response$ messages are sent. Further, no $update$ message from u to v is sent. By the inspection of code of RWW, a $release$ message from v to u can sent during execution of a $write$ request in $\sigma(u, v)$. Hence, $u.granted[v]$ is not affected during the execution of any $write$ request in $\sigma(v, u)$. $\square$

Let $I_4(u)$ be the following predicate. For any node v in $u.nbrs()$, if $u.taken[v]$ does not hold then, $u.uaw[v] = \emptyset$. Otherwise, if $u.grntd() \backslash \{v\} = \emptyset$ then, $(u.lt[v] + |u.uaw[v]| = 2) \wedge u.lt[v] > 0$; else $u.lt[v] = 2$.

Lemma 5.2 *Consider a sequential execution of a request sequence by* RWW. *For any node u, $I_4(u)$ is an invariant.*

Proof. Initially, for any node v in $u.nbrs()$, $u.taken[v]$ does not hold and $u.uaw[v] = \emptyset$.

$\{I_4(u)\}T_1\{I_4(u)\}$. For any node v in $u.tkn()$, $u.lt[v]$ is set to 2 in $oncombine$ procedure and $u.uaw[v]$ is set to $\emptyset$ in Line 3. Hence, $I_4(u)$ is preserved.

$\{I_4(u)\}T_2\{I_4(u)\}$. $I_4(u)$ is not affected.

$\{I_4(u)\}T_3\{I_4(u)\}$. For any node v in $u.tkn() \setminus \{w\}$, $u.lt[v]$ is set to 2 in $probercvd()$ procedure and $u.uaw[v]$ is set to $\emptyset$ in Line 3. Hence, $I_4(u)$ is preserved.

$\{I_4(u)\}T_4\{I_4(u)\}$. By Lemma 4.3, a $response$ message is received from w as a result of an earlier $probe$ message sent to w during execution of a $combine$ request, say q. By Lemma 4.3 again, in the quiescent state Q in which q is initiated, $u.taken[w]$ does not hold. Hence, if $I_4(u)$ holds before execution of T_4 then, $u.uaw[w]$ is empty.

If *flag* is $\mathbf{true}$ then, $u.lt[w]$ is set to 2 in $responsercvd()$ procedure, and $u.taken[w]$ is set to $\mathbf{true}$ in Line 3. Since $u.uaw[w]$ remains empty, $I_4(u)$ holds after execution of T_4.

$\{I_4(u)\}T_5\{I_4(u)\}$. By Lemma 4.5 and 4.1, u receives an $update$ message from w iff $u.taken[w]$ holds.

If $u.grntd()\backslash\{w\} = \emptyset$ then, $u.lt[w]$ is decremented by 1 in $updatercvd()$ procedure. Otherwise, $u.lt[w]$ is not affected. In Line 3, $|uaw[w]|$ is incremented by 1. Hence, if $u.lt[w]$ remains greater than 0, then $I_4(u)$ is preserved.

If $u.lt[w]$ is decremented to 0 then, a $release$ message is sent to w in $forwardrelease()$ procedure invoked in Line 9. In $forwardrelease()$ procedure, $u.taken[w]$ is set to **false**, and $u.uaw[w]$ is set to $\emptyset$. Hence, $I_4(u)$ is preserved.

$\{I_4(u)\}T_6\{I_4(u)\}$. Fix v to be an arbitrary node in $u.nbrs() \setminus \{w\}$.

By the inspection of code, if $u.grntd() \setminus \{v\} \neq \emptyset$ then, $u.lt[v]$ is not affected. Hence, $I_4(u)$ is preserved in execution of T_6.

Now we argue that, if $u.grntd() \setminus \{v\} = \emptyset$, then also $I_4(u)$ is preserved.

First, we argue that $|S| = 2$. By the inspection of code, a $release$ message from node w to u is sent only in $forwardrelease()$ procedure containing $w.uaw[u]$. Since any $release$ message is sent only if $w.breaklease(u)$ returns $\mathbf{true}$, $w.lt[u]$ is 0 while sending $release$ message. Since $I_4(u)$ holds before execution of T_6, $|S| = 2$.

Second, we argue that in $onrelease()$ procedure, the number of tuples α in $sntupdates$ with $\alpha.sntid$ greater or equal to the smallest id in S is at most 2. From the inspection of code, (1) identifiers of all received $update$ messages at node w from u are added to $w.uaw[u]$; (2) identifiers of sent $update$ messages from u are always incremented; (3) an identifier is not removed from the middle in $w.uaw[u]$, that is, identifiers in $w.uaw[u]$ are contiguous; and (4) on receiving an $update$ message, identifier of the forwarded $update$ message to node w is added to $sntupdates$. Hence, S contains identifiers of last two $update$ messages sent to w from u, that is, S contains two highest identifiers of $update$ messages sent to w. Since S may contain identifiers corresponding to the $update$ messages due to $write$ requests at u, the number of tuples α in $sntupdates$ with $\alpha.sntid$ greater or equal to the smallest id in S is at most 2.

Third, because of above arguments, $|A|$ is at most 2, where A is as defined in $onrelease()$ procedure.

Fourth, we argue that $|S'|$ is at most 2. Identifiers of the received $update$ messages are in increasing order. Before receiving the $release$ message, $u.granted[w]$ holds. On receiving an $update$ message from v, identifier of the received $update$ message is added to $u.uaw[v]$. Since $u.granted[w]$ holds, on receiving an $update$ with id, an $update$ message is sent to w with nid, and a tuple $\{v, id, nid\}$ is added $sntupdates$. Hence, the size of the set of identifiers in $u.uaw[v]$ (i.e., $|S'|$) with identifiers $\geq \beta.rcvid$, where β is as defined in $onrelease()$ procedure, is at most 2.

Finally, we argue that $|u.uaw[v]| + u.lt[v] = 2$. Since before receiving the $release$ message, $u.granted[w]$ and $I_4(u)$ hold, $u.lt[v] = 2$ before the invocation of $releasepolicy$. In $releasepolicy$, $u.lt[v]$ is set to $u.lt[v] - |u.uaw[v]|$. Hence, after execution of $releasepolicy$, $|u.uaw[v]| + u.lt[v] = 2$.

If $u.lt[v]$ becomes 0 then, in $forwardrelease()$ procedure, $u.tkn[v]$ is set **false**, $u.uaw[v]$ is set to $\emptyset$, and a $release$ message is sent to v.

Hence, $I_4(u)$ is preserved in execution of T_6. □

Lemma 5.3 *Consider a sequential execution of a request sequence σ by* RWW *and any two neighboring nodes u and v. (1) In the quiescent state after execution of any $combine$ request in $\sigma(u, v)$, $u.granted[v]$ holds. (2) In the quiescent state after execution of two consecutive $write$ requests in $\sigma(u, v)$, $u.granted[v]$ does not hold.*

Proof. (1) Let the the $\mathit{combine}$ request q is initiated in the quiescent state Q and completed in the quiescent state Q'.

If $u.\mathit{granted}[v]$ in Q, then no $\mathit{type}(u,v)$ messages are sent during the execution of q, and so $u.\mathit{granted}[v]$ holds in Q'.

Otherwise, if $u.\mathit{granted}[v]$ does not hold in Q, then by Lemma 4.3, during the execution of q, a probe message is sent from v to u and a $\mathit{response}$ message is sent from u to v. By inspection of code of $\mathit{sendresponse}$, RWW's function $\mathit{setlease}$ is invoked. By inspection of code of RWW, $\mathit{setlease}$ always returns $\mathbf{true}$, and so $u.\mathit{granted}[v]$ is set to $\mathbf{true}$. Hence, after execution of q, $u.\mathit{granted}[v]$ holds.

(2) Let the two consecutive write requests are q_1 and q_2, initiated in quiescent states Q and Q' respectively. Let q_2 is completed in the quiescent state Q''.

By Lemma 4.5, if $u.\mathit{granted}[v]$ does not hold in Q, then during the execution of q_1, no $\mathit{type}(u,v)$ messages are exchanged between u and v. Hence, $u.\mathit{granted}[v]$ is not affected and remains $\mathbf{false}$ in Q' and Q''.

Otherwise, if $u.\mathit{granted}[v]$ in Q, then without loss of generality we can assume that the request preceding q_1 in $\sigma(u,v)$ is a $\mathit{combine}$ request q.

Since, by Lemma 5.1, any request in $\sigma(v,u)$ does not affect $u.\mathit{granted}[v]$, without loss of generality we can also assume that there are no request in $\sigma(v,u)$ such that the request lies between q_1 and q_2 in σ.

By part (1), in Q, there is a path from u to $q.\mathit{node}$ (say w) in the lease graph $G(Q)$. Further, in Q, $w.\mathit{uaw}[u$-parent of $w]$ is empty and $w.\mathit{lt}[u$-parent of $w]$ is 0. By Lemma 4.5, w receives an update message during the execution of q_1. By the inspection of code of T_5, $w.\mathit{taken}[u$-parent of $w]$ holds in Q'. Hence, by Lemma 4.2 and Lemma 4.1, $u.\mathit{granted}[v]$ holds in Q'.

It is sufficient to show that during the execution of q_2, a $\mathit{release}$ message is sent from v to u, falsifying $u.\mathit{granted}[v]$.

Let A be the set of reachable nodes in the lease graph $G(Q')$ from u following the edge (u,v).

Let $\mathit{id}(q_1,w)$ be the id of the update message received at w during the execution of q_1.

First, we show that the following properties hold. Fix w to be an arbitrary node in A. (1) Node w receives an update message during the execution of q_1. (2) In quiescent state Q', $w.\mathit{uaw}[u$-parent of $w]$ contains $\mathit{id}(q_1,w)$. (3) In quiescent state Q', if $w.\mathit{grntd}()\setminus\{u$-parent of $w\}$ is empty, $|w.\mathit{uaw}[u$-parent of $w]| = 1$ and $w.\mathit{lt}[u$-parent of $w] = 1$.

(1) By Lemma 4.5, no probe or $\mathit{response}$ messages are sent during the execution of q_1. By the inspection of code, an edge is added in the lease graph only while sending and receiving a $\mathit{response}$ message. Hence, if an edge is present in the lease graph $G(Q')$, then the edge is also present in the lease graph $G(Q)$. Hence, by Lemma 4.5, each node in A receives an update message during the execution of q_1.

(2) From (1) and Lemma 4.5, w receives an update message from u-parent of w. From the inspection of code of T_5, $\mathit{id}(q_1,w)$ is added to $w.\mathit{uaw}[u$-parent of $w]$. In quiescent Q', since the identifiers of update messages sent from the u-parent of w to w are in increasing order and q_1 is the latest write request, $\mathit{id}(q_1,w)$ is the highest identifier in $w.\mathit{uaw}[u$-parent of $w]$. Hence, $w.\mathit{uaw}[u$-parent of $w]$ contains $\mathit{id}(q_1,w)$.

(3) Without loss of generality assume that $w.\mathit{grntd}()\setminus\{u$-parent of $w\}$ is empty. By (2), in quiescent state Q', $|w.\mathit{uaw}[u$-parent of $w]| > 0$.

By the inspection of code, $w.\mathit{lt}[u$-parent of $w] > 0$. Hence, by Lemma 5.2, $|w.\mathit{uaw}[u$-parent of $w]| \leq 2$.

By contradiction, we show that $|w.uaw[u\text{-parent of } w]| \neq 2$. Assume that $|w.uaw[u\text{-parent of } w]| = 2$ in Q'. By Lemma 5.2 and the inspection of code of T_5 and T_6, if $w.grntd() \setminus \{u\text{-parent of } w\}$ is empty and $|w.uaw[u\text{-parent of } w]| = 2$, then $w.lt[u\text{-parent of } w]$ is 0 in Q'. Hence, w must send a $release$ message to the u-parent of w and set $w.taken[u\text{-parent of } w]$ to **false** during the execution of q_1. But w is in A, hence, contradiction.

Therefore, $|w.uaw[u\text{-parent of } w]| = 1$, and by Lemma 5.2, (3) follows.

Second, We show the desired result by showing that every node w in A, including v, sends a $release$ message to u-parent of w containing $\{id(q_1, w), id(q_2, w)\}$.

We prove this claim by reverse induction on the length of the path from u to any node in A. Let the maximum length of the path from u to any node in A be l.

Base case. Consider a node w in A such that the length of the path from u to w is l. By definition of A, $w.grntd() \setminus \{u\text{-parent of } w\}$ is empty. By Claim 2 and Claim 3, $w.uaw[u\text{-parent of } w] = \{id(q_1, w)\}$ and $w.lt[u\text{-parent of } w] = 1$.

By Lemma 4.1 and Lemma 4.2, w is reachable from $q_2.node$ in the lease graph $G(Q')$. Hence, by Lemma 4.5, during the execution of q_2, w receives an $update$ message from the u-parent of w.

By inspection of code of T_5, $updatercvd()$ function of RWW is invoked. In $updatercvd()$, $w.lt[u\text{-parent of } w]$ is set to 0. By inspection of code of T_5, $forwardrelease()$ procedure is invoked. By inspection of code of RWW, $breaklease()$ returns **true**. Hence, $w.granted[u\text{-parent of } w]$ is set to **false** and a $release$ message is sent to the u-parent of w containing $\{id(q_1, w), id(q_2, w)\}$.

Induction hypothesis. Let any node w in A with the length of the path from u to w is i, where $i > 1$, sends a $release$ message to the u-parent of w containing $\{id(q_1, w), id(q_2, w)\}$.

Induction step. Consider a node w in A such that the length of the path from u to w is $i - 1$. As argued in the base case, during the execution of q_2, w receives an $update$ message from the u-parent of w.

By property (2) and above arguments, $w.uaw[u\text{-parent of } w]$ contains $id(q_1, w)$ and $id(q_2, w)$.

By induction hypothesis, for each node w' in $w.nbrs()$ such that w is u-parent of w', w receives a $release$ message from w'.

By the inspection of the code of T_6, after receiving a $release$ message from all the nodes w' such that $w.granted[w']$ in Q', w sets $w.lt[u\text{-parent of } w]$ to 0, and sends a $release$ message to u-parent of w containing $\{id(q_1, w), id(q_2, w)\}$.

Therefore, during the execution of q_2, a $release$ message is sent from v to u, falsifying $u.granted[v]$. □

Lemma 5.4 *The algorithm* RWW *is a* $(1, 2)$*-algorithm.*

Proof. Follows from Lemma 5.3. □

Lemma 5.5 *Consider a sequential execution of any request sequence* σ *by* RWW. *For any quiescent state* Q*, and for any ordered pair of neighboring nodes* (u, v), $F_{\mathrm{RWW}}(u, v)$ *is greater than* 0 *if and only if* $u.granted[v]$ *holds.*

Proof. Follows from Lemma 5.1 and Lemma 5.3. □

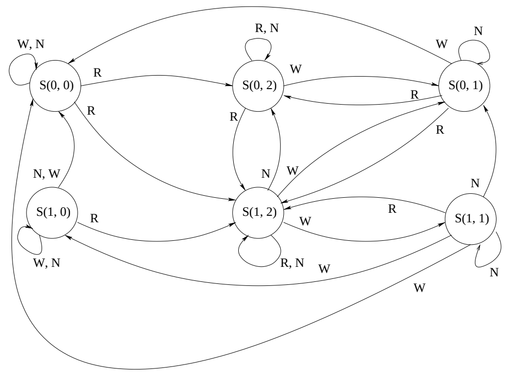

Figure 5: States and state transitions for any pair of nodes (u, v) in executing requests from $\sigma'(u, v)$ (defined in Lemma 5.8).

5.2 Competitive Ratio of RWW

In this section we show that RWW is $\frac{5}{2}$-competitive against an optimal offline lease-based algorithm OPT for the sequential aggregation problem (see Theorem 1). We also show that RWW is 5-competitive against a nice optimal offline algorithm for the sequential aggregation problem (see Theorem 2). Further, we show that, for any lease-based aggregation algorithm $\mathcal{A}$, there exist a request sequence σ and an offline algorithm such that, in a sequential execution of σ, the cost of $\mathcal{A}$ is at least $\frac{5}{2}$ times that of the offline algorithm (see Theorem 3).

Lemma 5.6 *In a sequential execution of any request sequence σ, for any two neighboring nodes u and v, $C_{\mathrm{RWW}}(\sigma, u, v) = C_{\mathrm{RWW}}(\sigma(u, v), u, v)$.*

Proof. Follows from Lemma 4.8 and Lemma 5.1. □

Lemma 5.7 *Consider a sequential execution of a request sequence σ by a lease-based algorithm $\mathcal{A}$. For any two neighboring nodes u and v, the total number of messages exchanged between u and v in executing σ is the sum of $C_{\mathcal{A}}(\sigma, u, v)$ and $C_{\mathcal{A}}(\sigma, v, u)$.*

Proof. Follows from the definitions of $C_{\mathcal{A}}(\sigma, u, v)$ and $C_{\mathcal{A}}(\sigma, v, u)$. □

Consider a sequential execution of an arbitrary request sequence σ by RWW. For any quiescent state Q, and for any ordered pair of neighboring nodes (u, v), we define the configuration of RWW, denoted $F_{\mathrm{RWW}}(u, v)$, as follows: (1) if Q is the initial quiescent state, then $F_{\mathrm{RWW}}(u, v)$ is 0; (2) if the last completed request in $\sigma(u, v)$ is a $combine$ request, then $F_{\mathrm{RWW}}(u, v)$ is 2; (3) if the last two completed requests in $\sigma(u, v)$ are a $combine$ request followed by a $write$ request, then $F_{\mathrm{RWW}}(u, v)$ is 1; (4) if the last two completed requests in $\sigma(u, v)$ are $write$ requests, then $F_{\mathrm{RWW}}(u, v)$ is 0.

For any quiescent state Q and ordered pair of neighboring nodes (u, v), we define the configuration of OPT $F_{\mathrm{OPT}}(u, v)$ to be 1 if $u.granted[v]$ holds; otherwise, 0.

Lemma 5.8 *Consider a sequential execution of a request sequence σ by* RWW *and* OPT. *For any two neighboring nodes u and v, $C_{\mathrm{RWW}}(\sigma, u, v)$ is at most $\frac{5}{2}$ times $C_{\mathrm{OPT}}(\sigma, u, v)$.*

$$
\begin{array}{lcllcl}
\text{minimize} & : & c & & & \\
\Phi(0,2) & - & \Phi(0,0) & + \; 2 & \leq & 2 \cdot c \\
\Phi(1,2) & - & \Phi(0,0) & + \; 2 & \leq & 2 \cdot c \\
\Phi(0,0) & - & \Phi(0,0) & & \leq & 0 \\
\Phi(1,2) & - & \Phi(1,0) & + \; 2 & \leq & 0 \\
\Phi(0,0) & - & \Phi(1,0) & & \leq & 2 \cdot c \\
\Phi(1,0) & - & \Phi(1,0) & & \leq & c \\
\Phi(0,0) & - & \Phi(1,0) & & \leq & c \\
\Phi(0,2) & - & \Phi(0,2) & & \leq & 2 \cdot c \\
\Phi(1,2) & - & \Phi(0,2) & & \leq & 2 \cdot c \\
\Phi(0,1) & - & \Phi(0,2) & + \; 1 & \leq & 0 \\
\Phi(1,2) & - & \Phi(1,2) & & \leq & 0 \\
\Phi(0,1) & - & \Phi(1,2) & + \; 1 & \leq & 2 \cdot c \\
\Phi(1,1) & - & \Phi(1,2) & + \; 1 & \leq & c \\
\Phi(0,2) & - & \Phi(1,2) & & \leq & c \\
\Phi(0,2) & - & \Phi(0,1) & & \leq & 2 \cdot c \\
\Phi(1,2) & - & \Phi(0,1) & & \leq & 2 \cdot c \\
\Phi(0,0) & - & \Phi(0,1) & + \; 2 & \leq & 0 \\
\Phi(1,2) & - & \Phi(1,1) & & \leq & 0 \\
\Phi(0,0) & - & \Phi(1,1) & + \; 2 & \leq & 2 \cdot c \\
\Phi(1,0) & - & \Phi(1,1) & + \; 2 & \leq & c \\
\Phi(0,1) & - & \Phi(1,1) & & \leq & c
\end{array}
$$

Figure 6: LP formulation of the costs associated with state transitions.

Proof. Once a request q in σ is initiated in a quiescent state, without loss of generality, we assume that RWW executes q, and then OPT executes q.

We construct a new request sequence $\sigma'(u,v)$ from $\sigma(u,v)$ as follows: (1) insert a $noop$ request in the beginning and at the end of $\sigma(u,v)$; and (2) insert a $noop$ request between every pair of successive requests in $\sigma(u,v)$.

In the rest of the proof, first, for both RWW and OPT, we argue that we can charge each of the $type(u,v)$ messages to a request in $\sigma'(u,v)$. Then, to prove the lemma, we use potential function arguments to show that $C_{\mathrm{RWW}}(\sigma'(u,v),u,v)$ is at most $\frac{5}{2}$ times $C_{\mathrm{OPT}}(\sigma'(u,v),u,v)$.

For RWW, from Lemma 5.6, we have, $C_{\mathrm{RWW}}(\sigma,u,v) = C_{\mathrm{RWW}}(\sigma(u,v),u,v)$. For RWW, we do not charge any message to a $noop$ request in $\sigma'(u,v)$. Hence, we have, $C_{\mathrm{RWW}}(\sigma,u,v) = C_{\mathrm{RWW}}(\sigma'(u,v),u,v)$.

For OPT, from lemma 4.3, during the execution of a $combine$ request in $\sigma(v,u)$, no $type(u,v)$ messages are sent. Also from Lemma 4.5 and part 3 of Lemma 4.8, during the execution of a $write$ request in $\sigma(v,u)$ by OPT, only a $release$ message from v to u can be sent. Consider a $type(u,v)$ $release$ message m sent during the execution of a $write$ request q in $\sigma(v,u)$ by OPT. On receiving m, $u.granted[v]$ is falsified. From Lemma 4.5, Lemma 4.3, Lemma 4.6, and part 3 and 4 of Lemma 4.8, $u.granted[v]$ is not set to $\mathbf{true}$ before executing another $combine$ request in $\sigma(u,v)$. Hence, at most one $type(u,v)$ $release$ message can be associated with a $noop$ request. Thus, we can associate all $type(u,v)$ messages with a request in $\sigma'(u,v)$.

Therefore, we can restrict our attention to messages sent in executing requests in $\sigma'(u,v)$ in comparing $C_{\mathrm{RWW}}(\sigma,u,v)$ and $C_{\mathrm{OPT}}(\sigma,u,v)$.

For the ordered pair (u,v), in Figure 5, we show a state diagram depicting possible changes in $F_{\mathrm{RWW}}(u,v)$ and $F_{\mathrm{OPT}}(u,v)$ in executing a request from $\sigma'(u,v)$. In the state diagram, a state labeled $S(x,y)$ represent a state of the algorithms in which $F_{\mathrm{OPT}}(u,v)$ is x and $F_{\mathrm{RWW}}(u,v)$ is y. Observe that the changes in $F_{\mathrm{RWW}}(u,v)$ in executing a request is deterministic as specified by the algorithm in Figure 4. On the other hand, the changes in $F_{\mathrm{OPT}}(u,v)$ in executing a request is not known in advance. Hence, more than one possible changes in $F_{\mathrm{OPT}}(u,v)$ in executing a request are depicted by non-deterministic state transitions. Recall that the cost of processing a request in a particular configuration for any lease-based algorithm is given in Figure 3.

We define a potential function $\Phi(x,y)$ as a mapping from a state $S(x,y)$ to a positive real number. The amortized cost of any transition is defined as the sum of the change in potential $\Delta(\Phi)$ and the cost of RWW in the transition. For any transition, we write that the amortized cost is at most c times the cost of OPT in the transition, where c is a constant factor. We solve these inequalities by formulating a linear program with an objective function to minimize c (see Figure 6). By solving the linear program, we get $c = \frac{5}{2}$, $\Phi(0,0) = 0$, $\Phi(0,1) = 2$, $\Phi(0,2) = 3$, $\Phi(1,0) = \frac{5}{2}$, $\Phi(1,1) = 2$, and $\Phi(1,2) = \frac{1}{2}$.

Hence, for any state transition due to the execution of a request q from $\sigma'(u,v)$, the amortized cost is at most $\frac{5}{2}$ times the cost of OPT in the execution of q. Recall that, in the initial quiescent state, $F_{\mathrm{RWW}}(u,v)$ and $F_{\mathrm{OPT}}(u,v)$ are 0, and the potential for any state is non-negative. Therefore, in execution of $\sigma'(u,v)$, the total cost of RWW is at most $\frac{5}{2}$ times that of OPT. That is, $C_{\mathrm{RWW}}(\sigma,u,v)$ is at most $\frac{5}{2}$ times $C_{\mathrm{OPT}}(\sigma,u,v)$. □

Theorem 1 *Algorithm* RWW *is* $\frac{5}{2}$*-competitive with respect to any lease-based algorithm for the sequential aggregation problem.*

Proof. From Lemma 5.8, in a sequential execution of a request sequence σ, for any two neighboring nodes u and v, $C_{\mathrm{RWW}}(\sigma, u, v)$ is at most $\frac{5}{2}$ times $C_{\mathrm{OPT}}(\sigma, u, v)$. By symmetry, $C_{\mathrm{RWW}}(\sigma, v, u)$ is at most $\frac{5}{2}$ times $C_{\mathrm{OPT}}(\sigma, v, u)$. Hence, the total number of messages exchanged between u and v in execution of σ by RWW is at most $\frac{5}{2}$ times that of OPT. Summing over all the pairs of neighboring nodes, we get that $C_{\mathrm{RWW}}(\sigma)$ is at most $\frac{5}{2}$ times $C_{\mathrm{OPT}}(\sigma)$. Hence, the theorem follows. □

Theorem 2 *Algorithm* RWW *is* 5*-competitive with respect to any nice algorithm for the sequential aggregation problem.*

Proof sketch. Let $\mathrm{OPT_N}$ be the optimal nice algorithm for the sequential aggregation problem. Consider any pair of neighboring nodes (u, v). We compare the cost of RWW and $\mathrm{OPT_N}$ in executing request sequences $\sigma(u, v)$ and $\sigma(v, u)$ separately.

First, consider the execution of requests in $\sigma(u, v)$. We define an *epoch* as follows. The first epoch starts at the beginning of the request sequence. An epoch ends with a write to $\mathit{combine}$ transition in $\sigma(u, v)$, and a new epoch starts at the same instant. By the definition of a nice algorithm, $\mathrm{OPT_N}$ provides strict consistency for the sequential execution problem. Hence, $\mathrm{OPT_N}$ sends at least one message in the any epoch. We are able to show that the algorithm RWW sends at most 5 messages in any epoch (follows from Lemma 5.3). Summing over all the epochs, we get that the cost of RWW in executing $\sigma(u, v)$ is at most 5 times that of $\mathrm{OPT_N}$. By symmetry, the cost of RWW in executing $\sigma(v, u)$ is at most 5 times that of $\mathrm{OPT_N}$. By summing over all the pair of neighboring nodes, the desired result follows. □

Theorem 3 *For any lease-based algorithm* $\mathcal{A}$*, there exist a request sequence* σ *and an offline algorithm such that the cost* $\mathcal{A}$ *in executing* σ *is at least* $\frac{5}{2}$ *times that of the offline algorithm.*

Proof sketch. We give an adversarial request generating argument to sketch the desired result. Consider an example of a tree consisting of just two nodes u and v such that there is an edge between u and v. The adversarial request generating algorithm ADV is as follows. The algorithm ADV generates a $\mathit{combine}$ requests at v such that there is a lease from u to v after execution of a-th request. And then, ADV generates b write requests at u such that there is no lease from u to v after execution of b-th request. Using potential function arguments, we can show that, for a sufficient long request sequence σ generated by ADV, the cost of $\mathcal{A}$ in executing σ is at least $\frac{5}{2}$ times that of an optimal offline algorithm, which is tailored to the request sequence σ. □

6 Consistency Results for Concurrent Executions

In this section we generalize the traditional definition of causal consistency [1] for the aggregation problem, and show that any lease-based aggregation algorithm is causally consistent. As mentioned earlier, the key difference between the setup in [1] and ours is in reading one value compared to aggregating values from all the nodes. See Section 3 for an informal discussion on this section.

6.1 Definitions

Request. For the convenience of the analysis of this section, we extend the definition of a request from Section 2 as follows. A request q is a tuple $(node, op, arg, retval, index)$, where (1) $node$ is the node where the request is initiated; (2) op is the type of of the request, $combine$, $gather$, or $write$; (3) arg is the argument of the request (if any); (4) $retval$ is the return value of the request (if any); and (5) $index$ is the number of requests that are generated at $q.node$ and completed before q is completed.

An aggregation algorithm executes $write$ and $combine$ requests as described in Section 2. To execute a $gather$ request, an aggregation algorithm returns a set A of pairs of the form $(node, index)$ such that (1) for each node u in T, there is a tuple (u, i) in A, where $i \geq -1$; (2) for any tuple (u, i) in A, if $i \geq 0$, then there is a $write$ request q such that $q.node = u$ and $q.index = i$; and (3) $|A|$ is equal to the number of nodes in T.

Miscellaneous. For the convenience of analysis of this section, we extend the definition of function f from Section 2 as follows. In the extended definition, f can also take a set of pairs A of the form $(node, index)$ as an argument, and $f(A) = f(B)$, where B is a set of $write$ requests such that for any tuple (u, i) in A with $i \geq 0$, there is a $write$ request q in B with $q.node = u$ and $q.index = i$.

A *combine-write* sequence (set) is a sequence (set) of requests containing only $combine$ and $write$ requests. A *gather-write* sequence (set) is a sequence (set) of requests containing only $gather$ and $write$ requests. Let A be a set of requests. Then, $pruned(A, u)$ is a subset of A such that, for any request q in A, q is in $pruned(A, u)$ if and only if $q.op = write$ or $q.node = u$.

For any sequence of requests S and any request q in S, we define $recentwrites(S, q)$ as a set of pairs such that the size of $recentwrites(S, q)$ is equal to the number of nodes in T, and for any node u in T: (1) if q' is the most recent $write$ request at u preceding q in S, then $(u, q'.index)$ is in $recentwrites(S, q)$; (2) if there is no $write$ request at u preceding q in S, in which case, $(u, -1)$ is in $recentwrites(S, q)$.

Let A be a gather-write set, and S be a linear sequence of all the requests in A. Then, S is called a *serialization* of A if and only if, for any $gather$ request q in S, $q.retval = recentwrites(S, q)$.

For any two request sequences σ and τ, $\sigma - \tau$ is defined to be the subsequence of σ containing all the requests q in σ such that q is not present in τ. For any two request sequences σ and τ, $\sigma.\tau$ is defined to be σ appended by τ.

Compatibility. Let q_1 be a $combine$ or $write$ request and q_2 be a $gather$ or $write$ request. Then, q_1 and q_2 are *compatible* if and only if (1) $q_1.op = write$ and $q_1 = q_2$; or (2) $q_1.op = combine$, $q_2.op = gather$, $q_1.retval = f(q_2.retval)$, and the $node$, arg, and $index$ fields are equal for q_1 and q_2. A combine-write sequence σ and a gather-write sequence τ are compatible if and only if (1) σ and τ are of equal length; and (2) for all indices i, $\sigma(i)$ and $\tau(i)$ are compatible. Let A be a combine-write set and B be a gather-write set. Then, A and B are compatible if and only if for any node u in T, there exists a linear sequence S of all the requests in $pruned(A, u)$, and a linear sequence S' of all the requests in $pruned(B, u)$ such that S and S' are compatible.

Causal Consistency. We define *causal ordering* ($\rightsquigarrow$) among any two requests q_1 and q_2 in a gather-write execution-history A as follows. First, $q_1 \overset{1}{\rightsquigarrow} q_2$ if and only if (1) $q_1.node = q_2.node$ and $q_1.index < q_2.index$; or (2) q_1 is a write request, q_2 is a $gather$ request, and q_2 returns $(q_1.node, q_1.index)$ in $q_2.retval$. Second, $q_1 \overset{i+1}{\rightsquigarrow} q_2$ if and only if there exists a request q' such that $q_1 \overset{i}{\rightsquigarrow} q' \overset{1}{\rightsquigarrow} q_2$. Finally, $q_1 \rightsquigarrow q_2$ if and only if there exists an i such that $q_1 \overset{i}{\rightsquigarrow} q_2$.

The execution-history of an aggregation algorithm is defined as the set of all requests executed by the algorithm. A gather-write execution-history A is *causally consistent* if and only if, for any node u in T, there exists a serialization S of $pruned(A, u)$ such that S respects the causal ordering $\rightsquigarrow$ among all the requests in $pruned(A, u)$. A combine-write execution-history A is causally consistent if and only if there exists a gather-write execution-history B such that A and B are compatible and B is causally consistent.

6.2 Algorithm

In Figure 7, we present the mechanism for any lease-based aggregation algorithm with *ghost actions* (in the curly braces). The ghost actions are presented for the convenience of analysis.

For any node u, $u.log$ is a ghost variable. For any node u, $u.wlog$ is a subsequence of $u.log$ containing all the $write$ requests in $u.log$.

Initially, for any node u, $u.val := 0$, $u.uaw := \emptyset$, $u.pndg := \emptyset$, $u.upcntr := 0$, $u.sntupdates := \emptyset$. For each node v in $u.nbrs()$, $u.taken[v] :=$ **false**, $u.granted[v] :=$ **false**, $u.aval[v] := 0$, $u.snt[v] := \emptyset$, and $u.log$ is empty.

Function $request(combine)$ generates and returns a $combine$ request q' as follows. $q'.node = u$, $q'.op = combine$, $q'.arg = \emptyset$, $q'.retval = gval()$, and $q'.index$ is 1 plus the number of completed requests at u. Function $request(write, q)$ generates and returns a $write$ request q' as follows. $q'.node = u$, $q'.op = write$, $q'.arg = q.arg$, $q'.retval = \emptyset$, and $q'.index$ is 1 plus the number of completed requests at u.

6.3 Analysis

For each node u in T, we construct a gather-write sequence $u.gwlog$ from $u.log$ as follows: (1) if $u.log(i)$ is a $write$ request then $u.gwlog(i) = u.log(i)$; (2) if $u.log(i)$ is a *combine* q_1 then, $u.gwlog(i)$ is a $gather$ q_2 such that $q_2.node = q_1.node$, $q_2.op = gather$, $q_2.index = q_1.index$, and $q_2.retval = recentwrites(u.log, q_1)$.

For each node u in T, we construct $u.log'$ and $u.gwlog'$ from $u.log$ and $u.gwlog$ as follows. First, initialize $u.log'$ to $u.log$, and $u.gwlog'$ to $u.gwlog$. Then, for each node v in T except u repeat the following steps: (1) $u.log' = u.log'.(v.wlog - u.log')$; (2) $u.gwlog' = u.gwlog'.(v.wlog - u.gwlog')$.

For any set of nodes A and a request sequence σ, $recent(A, \sigma)$ returns a set of $|A|$ pairs such that, for any node $u \in A$: (1) if q' is the most recent $write$ request at u in σ, then $(u, q'.index)$ is in $recent(\sigma, q)$; (2) if there is no $write$ request at u in σ, then $(u, -1)$ is in $recent(S, q)$.

For a set of nodes A, a real value x, and a request sequence σ, we define $corresponds(A, x, \sigma)$ to be $\mathtt{true}$ if and only if $x = f(recent(A, \sigma))$.

For a set of nodes A and a request sequence σ, $projectwrites(A, \sigma)$ returns the sub-sequence of σ containing all the $write$ requests at any node in A.

For request sequences σ and τ, $prefix(\sigma, \tau)$ is defined to be $\mathtt{true}$ if and only if τ is a prefix of σ. *Remark:* An empty sequence is considered prefix of any other request sequence.

Lemma 6.1 *For any $update$ or $response$ message m from any node v to any neighboring node u, let S be the $v.wlog$ after m has been sent. Then, $prefix(S, m.wlog)$ holds.*

```
     node $u$
     var $taken$ : array$[v_1 \dots v_k]$ of boolean;
       $granted$ : array$[v_1 \dots v_k]$ of boolean;
       $aval$ : array$[v_1 \dots v_k]$ of real;   $val$ : real;
       $uaw$ : set {int}; $pndg$ : set $\{node\}$;
       $snt[]$ : array$[v_1, \dots, v_k]$ of set $\{node\}$;
       $upcntr$ : int; $sntupdates$ : set $\{\{node$, int, int$\}\}$;
     begin
$T_1$    true $\rightarrow \{combine\ q\}$
1          $\underline{oncombine}(u)$;
2          foreach $v \in tkn()$ do
3            $uaw[v] := \emptyset$; od
4          if $u \notin pndg \rightarrow$
5            if $nbrs() \setminus tkn() = \emptyset \rightarrow$
6              {append $request(combine)$ to $log$};
7              return $gval()$;
8            $\square\ nbrs() \setminus tkn() \neq \emptyset \rightarrow$
9              $sendprobes(u)$;
10             $snt[u] := nbrs() \setminus tkn()$; fi fi
$T_2$    true $\rightarrow \{write\ q\}$
1          $val := q.arg$; {append $request(write, q)$ to $log$}
2          if $grntd() \neq \emptyset \rightarrow$
3            $id := newid()$;
4            $forwardupdates(u, id)$; fi
$T_3$    $\square$ rcv $probe()$ from $w \rightarrow$
1          $\underline{probercvd}(w)$;
2          foreach $v \in tkn() \setminus \{w\}$ do
3            $uaw[v] := \emptyset$; od
4          if $w \notin pndg \rightarrow$
5            if $nbrs() \setminus \{tkn() \cup \{w\}\} = \emptyset \rightarrow$
6              $sendresponse(w)$;
7            $\square\ nbrs() \setminus \{tkn() \cup \{w\}\} \neq \emptyset \rightarrow$
8              $sendprobes(w)$;
9              $snt[w] := nbrs() \setminus \{tkn() \cup \{w\}\}$; fi fi
$T_4$    $\square$ rcv $response(x, flag)$ from $w \rightarrow$
         {rcv $response(wlog_w, flag)$ from $w$} $\rightarrow$
1          $\underline{responsercvd}(flag, w)$;
2          $\underline{aval}[w] := x$; $\{log := log.(wlog_w - log)\}$;
3          $taken[w] := flag$;
4          foreach $v \in pndg$ do
5            $snt[v] := snt[v] \setminus \{w\}$;
6            if $snt[v] = \emptyset \rightarrow$
7              $pndg := pndg \setminus \{v\}$;
8              if $v = u \rightarrow$
9                {append $request(combine)$ to $log$};
10               return $gval()$;
11             $\square\ v \neq u \rightarrow$
12               $sendresponse(v)$; fi fi od
$T_5$    $\square$ rcv $update(x, id)$ from $w \rightarrow$
         {rcv $update(wlog_w, id)$ from $w$ } $\rightarrow$
1          $\underline{updatercvd}(w)$;
2          $\underline{aval}[w] := x$; $\{log := log.(wlog_w - log)\}$;
3          $uaw[w] := uaw[w] \cup id$;
4          if $grntd() \setminus \{w\} \neq \emptyset \rightarrow$
5            $nid = newid()$;
6            $sntupdates := sntupdates \cup \{w, id, nid\}$;
7            $forwardupdates(w, nid)$;
8          $\square\ grntd() \setminus \{w\} = \emptyset \rightarrow$
9            $forwardrelease()$; fi
$T_6$    $\square$ rcv $release(S)$ from $w \rightarrow$
1          $\underline{releasercvd}(w)$;
2          $granted[w] :=$ false;
3          $onrelease(w, S)$;
     end
```

```
procedure $sendprobes(node\ w)$
  $pndg := pndg \cup \{w\}$;
  foreach $v \in nbrs() \setminus \{tkn() \cup snt \cup \{w\}\}$ do
    send $probe()$ to $v$; od

procedure $forwardupdates(node\ w, int\ id)$
  foreach $v \in grntd() \setminus \{w\}$ do
    send $update(subval(v), id)$ to $v$;
    {send $update(wlog, id)$ to $v$}; od

procedure $sendresponse(node\ w)$
  if $(nbrs() \setminus \{tkn() \cup \{w\}\} = \emptyset) \rightarrow$
    $granted[w] := \underline{setlease}(w)$; fi
  send $response(subval(w), granted[w])$ to $w$;
  {send $response(wlog, granted[w])$ to $w$; }

boolean $isgoodforrelease(node\ w)$
  return $(grntd() \setminus \{w\} = \emptyset)$;

procedure $onrelease(node\ w$, set $S)$
  Let $id$ is the smallest id in $S$;
  foreach $v \in tkn() \setminus \{w\}$ do
    Let $A$ be the set of tuples $\alpha$ in $sntupdates$
      such that $\alpha.node = v$ and $\alpha.sntid \geq id$;
    Let $\beta$ be a tuple in $A$
      such that $\beta.rcvid \leq \alpha.rcvid$, for all $\alpha$ in $A$;
    Let $S'$ be the set of ids in $uaw[v]$ with ids $\geq \beta.rcvid$;
    $uaw[v] := S'$;
    if $isgoodforrelease(v) \rightarrow$
      $\underline{releasepolicy}(v)$; fi od
  $forwardrelease()$;

procedure $forwardrelease()$
  foreach $v \in tkn()$ do
    if $isgoodforrelease(v) \rightarrow$
      if $taken[v] \wedge \underline{breaklease}(v) \rightarrow$
        $taken[v] :=$ false;
        send $release(uaw[v])$ to $v$;
        $uaw[v] := \emptyset$; fi fi od

int $newid()$
  $upcntr := upcntr + 1$;
  return $upcntr$;

real $gval()$
  $x := val$;
  foreach $v \in nbrs()$ do
    $x := f(x, aval[v])$; od
  return $x$;

real $subval(node\ w)$
  $x := val$;
  foreach $v \in nbrs() \setminus \{w\}$ do
    $x := f(x, aval[v])$; od
  return $x$;

set $nbrs()$
  return the set of neighboring nodes;
set $tkn()$
  return $\{v \mid v \in nbrs() \wedge taken[v] = \textbf{true}\}$;
set $grntd()$
  return $\{v \mid v \in nbrs() \wedge granted[v] = \textbf{true}\}$;
```

Figure 7: Mechanism for any lease-based algorithm with ghost actions. For the node u, $\{v_1, \dots, v_k\}$ is the set of neighboring nodes.

Proof. By the inspection of code ($forwardupdates()$ and $sendresponse()$), $m.wlog = v.wlog$ when m is being sent. Since $v.wlog$ grows only at the end, the lemma follows. □

Lemma 6.2 *For any two $update$ or $response$ messages m_1 and m_2 from a node v to any neighboring node u such that m_2 is sent after m_1, $prefix(m_2.wlog, m_1.wlog)$ holds.*

Proof. By Lemma 6.1, $m_1.wlog$ is a prefix of $v.wlog$ after m_1 has been sent. By the inspection of code ($forwardupdates()$ and $sendresponse()$), $m_2.wlog = v.wlog$ when m_2 is being sent. Hence, the lemma follows. □

Lemma 6.3 *Just before the execution of T_4 (T_5) at u, on receiving a $response$ message (an $update$ message) m sent from v, let σ be $projectwrites(A, m.wlog)$ and τ be $projectwrites(A, u.log)$, where $A = subtree(v, u)$. Then, (1) $prefix(\sigma, \tau)$ holds; (2) $projectwrites(nodes(T) \setminus A, m.wlog - u.log)$ is an empty set.*

Proof. (1) We prove by induction on the number of $update$ or $response$ messages from v to u.

Base case. Since $v.granted[u]$ does not hold initially, the first message of our interest is a $response$ message m. Since u receives any $write$ requests in A only from v, τ is empty. Hence, $prefix(\sigma, \tau)$ holds.

Induction step. Since communication channels are FIFO, $(n + 1)$st $update$ or $response$ message m reaches u after nth message m'. By induction hypothesis, just before receiving m', $projectwrites(A, u.log)$ is prefix of $projectwrites(A, m'.wlog)$. In line 2 of T_4 (T_5), $u.log = u.log.(m'.wlog - u.log)$, that is, all the $write$ requests in $m'.wlog$ not present in $u.log$ are appended to $u.log$. Hence, $projectwrites(A, u.log) = projectwrites(A, m'.wlog)$ after execution of Line 2 of T_4 (T_5).

By Lemma 6.2, $m'.wlog$ is a prefix of $m.wlog$. Hence, just before receiving m, $projectwrites(A, u.log)$ is a prefix of $projectwrites(A, m.wlog)$.

(2) Let B be $nodes(T)\setminus A$. By Lemma 6.1, Lemma 6.2, and part (1), at any instant $projectwrites(B, v.log)$ is a prefix of $projectwrites(B, u.log)$. By Lemma 6.1, $m.wlog$ is a prefix of $v.wlog$ after m has been sent. Hence, just before receiving m, $projectwrites(B, m.wlog)$ is a prefix of $projectwrites(B, u.log)$. Therefore, $projectwrites(B, m.wlog - u.log)$ is empty. □

For any node u, (1) $I_1(u)$: $corresponds(A, u.gval(), u.log)$, where A is the set of all nodes in T; (2) $I_2(u)$: for any $update$ or $response$ message m from u to any node v in $u.nbrs()$, $corresponds(A, m.x, m.wlog)$, where A is the set of all nodes in $subtree(u, v)$; and (3) $I_3(u)$: for any node v in $u.nbrs()$, $corresponds(A, u.aval[v], u.log)$, where A is the set of all nodes in $subtree(v, u)$. Let $I(u)$ be $I_1(u) \wedge I_2(u) \wedge I_3(u)$.

Lemma 6.4 *For any node u, if $I_1(u)$ and $I_3(u)$ hold just before an $update$ or a $response$ message m is sent from u to a node v in $u.nbrs()$, then $corresponds(A, m.x, m.wlog)$, where $A = subtree(u, v)$.*

Proof. Initially, $u.val$ is 0 and $u.log$ is empty. Hence, initially,

$$u.val \quad = \quad f(recent(\{u\}, u.log)) \tag{2}$$

The only line of code that modifies $u.val$ is Line 1 of T_2. This line preserves equation 2. Hence, equation 2 holds just before sending any *update* or *response* message.

In the following equation, let $\{v_1, \ldots, v_k\} = u.nbrs() \setminus \{v\}$ and $S_i = subtree(v_i, u)$

$$\begin{aligned} m.x &= u.subval(v) \\ &= f(u.val, u.aval[v_1], \ldots, u.aval[v_k]) \\ &= f(f(recentwrites(\{u\}, u.log)), f(recent(S_1, u.log)), \ldots, f(recent(S_k, u.log))) \\ &= f(recent(\{u\} \cup S_1 \cup \cdots \cup S_k), u.log) \\ &= f(recent(A, u.log)) \\ &= f(recent(A, m.wlog)) \end{aligned} \tag{3}$$

In the above equation, the first equality follows from the algorithm. The second equality follows from the definition of $subval(v)$. The third equality follows from I_3 and equation 2. The fourth and fifth equalities follows from the fact that $\{u\}, S_1, \ldots, S_k$ are disjoint sets of nodes and their union is $subtree(T, u, v)$. The last equality follows from the fact that $m.wlog = wlog$ and $recent(A, log) = recent(A, wlog)$.

Hence, the lemma follows. $\square$

Lemma 6.5 *For any node u, $I(u)$ is an invariant.*

Proof. Initially, for any node u, $u.gval()$ is 0 and $u.log$ is empty. Hence, $I_1(u)$ holds. There are no *update* or *response* messages. Hence, $I_2(u)$ holds. For any node v in $u.nbrs()$, $u.aval[v]$ is 0 and $u.log$ is empty. Hence, $I_3(u)$ holds.

$\{I(u)\}T_1\{I(u)\}$. In the execution of T_1, for any node v in $u.nbrs()$, $u.aval[v]$ and $u.val$ remain unchanged. No *update* or *response* messages are generated in execution of T_1. No *write* request is added to $u.log$. Hence, $I_1(u)$, $I_2(u)$, and $I_3(u)$ are not affected in execution of T_1.

$\{I(u)\}T_2\{I(u)\}$. In the execution of T_2, only part of the code affecting $I_1(u)$ is the line 1. Note that Line 1 does not affect $I_2(u)$ and $I_3(u)$. In the following equation, let $\{v_1, \ldots, v_k\} = u.nbrs()$ and $S_i = subtree(T, v_i, u)$.

$$\begin{aligned} f(u.aval[v_1], \ldots, u.aval[v_k]) &= f(f(recent(S_1, u.log)), \ldots, f(recent(S_k, u.log))) \\ &= f(recent(S_1, u.log) \cup \cdots \cup recent(S_k, u.log)) \\ &= f(recent(S_1 \cup \cdots \cup S_k, u.log) \\ &= f(recent(nodes(T) \setminus \{u\}, u.log)) \end{aligned} \tag{4}$$

In the above equation, the first equality follows from $I_3(u)$. The second equality follows from the fact that $S_1, \ldots, S_k$ are disjoint sets of nodes.

Let q be the *write* request appended to $u.log$ in Line 1. After Line 1, val is $q.arg$, and $\{q\}$ is $recent(\{u\}, log)$. Hence, after Line 1,

$$u.val = f(recent(\{u\}, u.log)) \tag{5}$$

Therefore, after Line 1,

$$\begin{aligned} u.gval() &= f(u.val, u.aval[v_1], \ldots, u.aval[v_k]) \\ &= f(u.val, f(u.aval[v_1], \ldots, u.aval[v_k])) \\ &= f(f(recent(\{u\}, u.log)), f(recent(nodes(T) \setminus \{u\}, u.log)) \\ &= f(recent(\{u\}, u.log) \cup recent(nodes(T) \setminus \{u\}, u.log)) \\ &= f(recent(nodes(T), u.log)) \end{aligned} \tag{6}$$

In the above equation, the first equality follows from the definition of $u.gval()$. The second equality follows from the associativity property of f. The third equality follows from the equations 4 and 5.

Hence, $corresponds(nodes(T), u.gval(), u.log)$ holds after line 1. That is, $I_1(u)$ holds after Line 1. Therefore, for each line of the code in T_2 if $I_1(u) \wedge I_2(u) \wedge I_3(u)$ holds before the execution of the line then $I_1(u)$ holds after the execution of the line.

In the execution of T_2, the only part of the code affecting $I_2(u)$ is the invocation of procedure $forwardupdates()$ in Line 4. By Lemma 6.4, $I_2(u)$ holds after Line 4. Therefore, for each line of the code in T_2 if $I_1(u) \wedge I_2(u) \wedge I_3(u)$ holds before the execution of the line then $I_2(u)$ holds after the execution of the line.

In T_2, $I_3(u)$ is not affected.

$\{I(u)\}T_3\{I(u)\}$. $I_1(u)$ and $I_3(u)$ are not affected in the execution of T_3. Only part of the code that affects $I_2(u)$ is the invocation of procedure $sendresponse()$ in Line 6. By Lemma 6.4, $I_2(node)$ holds after line 6.

$\{I(u)\}T_4\{I(u)\}$. Only lines that affect $I(u)$ are Line 2 and Line 12. Line 2 does not affect $I_2(u)$, but affects $I_1(u)$ and $I_3(u)$ since the line modifies $u.aval[w]$ and $u.log$. First we show that $I_3(u)$ is preserved in Line 2, and so, $I_1(u)$ is also preserved.

Let m be the $response$ message received and A be $subtree(w, u)$. By part (1) of Lemma 6.3, after the execution of Line 2, $u.aval[w] = m.x$ and $recent(A, u.log) = recent(A, m.wlog)$. Hence, by $I_2(u)$, $u.aval[w] = f(recent(A, u.log))$.

By part (2) of Lemma 6.3, for all v in $u.nbrs() \setminus \{w\}$, $recent(B, u.log)$ is not affected, where $B = subtree(v, u)$, and so, $corresponds(B, u.aval[v], u.log)$ remains unchanged. Hence, along with the arguments in the preceding paragraph, $I_3(u)$ is preserved in Line 2, and so, preserved in the execution of T_4.

By part (2) of Lemma 6.3, $recent(\{u\}, u.log)$ is not affected. Therefore, $I_1(u)$ is also preserved in Line 2, and so, preserved in the execution of T_4.

Line 12 only affects $I_2(u)$. By Lemma 6.4, $I_2(u)$ holds in Line 12.

Therefore, $I_1(u) \wedge I_2(u) \wedge I_3(u)$ is preserved in the execution of T_4.

$\{I(u)\}T_5\{I(u)\}$. Only lines that affect $I(u)$ are Line 2 and Line 7. Line 2 does not affect $I_2(u)$, but affects $I_1(u)$ and $I_3(u)$. Line 7 affects only $I_2(u)$.

By part (2) of Lemma 6.3, $recent(\{u\}, u.log)$ is not affected in Line 2. Therefore, $I_1(u)$ is preserved in Line 2, and so, preserved in the execution of T_5.

Let m be the $update$ message received and A be $subtree(w, u)$. By part (1) of Lemma 6.3, after the execution of Line 2, $u.aval[w] = m.x$ and $recent(A, u.log) = recent(A, m.wlog)$. Hence, by $I_2(u)$, $u.aval[w] = f(recent(A, u.log))$.

By part (2) of Lemma 6.3, for all nodes v in $u.nbrs() \setminus \{w\}$, $recent(B, u.log)$ is not affected, where $B = subtree(v, u)$, and so, $corresponds(B, u.aval[v], u.log)$ remains unchanged. Hence, along with the arguments in the preceding paragraph, $I_3(u)$ is preserved in Line 2, and so, preserved in the execution of T_5.

Line 7 affects only $I_2(u)$. By Lemma 6.4, $I_2(u)$ holds in Line 7.

Therefore, $I_1(u) \wedge I_2(u) \wedge I_3(u)$ is preserved in the execution of T_5.

$\{I(u)\}T_6\{I(u)\}$. In the execution of T_6, $I_1(u)$, $I_2(u)$, and $I_3(u)$ are not affected. Hence, $I(u)$ is preserved in the execution of T_6. □

For a request sequence σ and a request q, $\mathit{index}(\sigma, q)$ returns the index of q in σ if present, otherwise, returns -1. For any request sequence σ, and requests q_1 and q_2 in σ, $\mathit{precedes}(\sigma, q_1, q_2)$ is defined to be $\mathbf{true}$ if and only if $\mathit{index}(\sigma, q_1) < \mathit{index}(\sigma, q_2)$.

Lemma 6.6 *Let q_1 and q_2 be any gather or write requests such that $q_1.\mathit{node} = q_2.\mathit{node}$ and $q_1.\mathit{index} < q_2.\mathit{index}$. Then, q_1 and q_2 belong to $q_1.\mathit{node}.\mathit{gwlog}$, and $\mathit{precedes}(q_1.\mathit{node}.\mathit{gwlog}, q_1, q_2)$ holds.*

Proof. From given condition, q_1 and q_2 belong to $q_1.\mathit{node}.\mathit{log}$ and $\mathit{precedes}(q_1.\mathit{node}.\mathit{log}, q_1, q_2)$. By the construction of gwlog, the lemma follows. □

Lemma 6.7 *Let u and v be distinct nodes and let q_1 and q_2 be write requests in $v.\mathit{gwlog}$ such that $q_2.\mathit{node} = v$, $\mathit{precedes}(v.\mathit{gwlog}, q_1, q_2)$, and q_2 belongs to $u.\mathit{gwlog}$. Then, q_1 belongs to $u.\mathit{gwlog}$ and $\mathit{precedes}(u.\mathit{gwlog}, q_1, q_2)$.*

Proof. By induction on the length of path from v to u, say l.

Base case. $l = 1$, that is, u and v are neighboring nodes. Let u receives q_2 in an update or a $\mathit{response}$ message m, that is, q_2 belongs to $m.\mathit{wlog}$ and q_2 does not belong to $u.\mathit{log}$ just before receiving m. By the inspection of code, $m.\mathit{wlog} = v.\mathit{wlog}$. Hence, just before m is sent, q_2 belongs to $v.\mathit{log}$. Since $\mathit{precedes}(v.\mathit{log}, q_1, q_2)$, $\mathit{precedes}(m.\mathit{wlog}, q_1, q_2)$. If q_1 is in $u.\mathit{log}$ just before receiving m, then on receiving m, q_2 belongs to $u.\mathit{log}$, and so, $\mathit{precedes}(u.\mathit{gwlog}, q_1, q_2)$ holds. Otherwise, on receiving m, $u.\mathit{log} = u.\mathit{log}.(u.\mathit{log} - m.\mathit{wlog}_w)$, and so, $\mathit{precedes}(u.\mathit{log}, q_1, q_2)$ holds. Hence, by construction of $u.\mathit{gwlog}$, $\mathit{precedes}(u.\mathit{gwlog}, q_1, q_2)$ holds.

Induction hypothesis. For some i, such that $l = i$, q_1 belongs to $u.\mathit{gwlog}$ and $\mathit{precedes}(u.\mathit{gwlog}, q_1, q_2)$.

Induction step. Consider $l = i + 1$. Let w be the node such that w belongs to $u.\mathit{nbrs}()$ and v belongs to $\mathit{subtree}(T, w, u)$. Let u receives q_2 from w in an update or a $\mathit{response}$ message m. By the inspection of code, q_2 belongs to $w.\mathit{log}$, and so, by construction of $w.\mathit{gwlog}$, q_2 also belongs to $w.\mathit{gwlog}$. By induction hypothesis and by construction of $w.\mathit{gwlog}$, q_1 belongs to $w.\mathit{log}$ and $\mathit{precedes}(w.\mathit{log}, q_1, q_2)$ holds when m is sent. Since $m.\mathit{wlog} = w.\mathit{wlog}$ when m is sent, q_1 belongs to $m.\mathit{wlog}$ and $\mathit{precedes}(m.\mathit{log}, q_1, q_2)$ holds. As in the base case, regardless of whether q_1 belongs to $u.\mathit{log}$ just before receiving m, q_1 belongs to $u.\mathit{log}$ and $\mathit{precedes}(u.\mathit{log}, q_1, q_2)$ on receiving m. Hence, by construction of $u.\mathit{gwlog}$, $\mathit{precedes}(u.\mathit{gwlog}, q_1, q_2)$ holds. □

Lemma 6.8 *Let q_1 and q_2 be gather requests such that $q_1.\mathit{node} \neq q_2.\mathit{node}$, and for integer $i > 1$, $q_1 \overset{i}{\rightsquigarrow} q_2$. Then, there is a write request q' such that $q'.\mathit{node} = q_1.\mathit{node}$ and for integer j, $q_1 \overset{j}{\rightsquigarrow} q' \overset{i-j}{\rightsquigarrow} q_2$, where $i > j \geq 1$.*

Proof. By contradiction. Assume that there is no such $write$ request at $q_1.node$. Let $q_1 \overset{1}{\rightsquigarrow} \ldots \overset{1}{\rightsquigarrow} q' \overset{1}{\rightsquigarrow} q'' \overset{1}{\rightsquigarrow} \ldots \overset{1}{\rightsquigarrow} q_2$ such that q'' is the first request in this chain that is not at $q_1.node$. That is, in this chain, $q_1, \ldots, q'$ are at $q.node$. We can find such a request (q'') since $q_2.node \neq q_1.node$. By causal ordering ($\overset{1}{\rightsquigarrow}$) definition, $q' \overset{1}{\rightsquigarrow} q''$ if and only if q' is a $write$ request and q'' is a $gather$ request. Hence, the contradiction. Therefore, the lemma follows. □

Lemma 6.9 *For any node u and $i = 1, 2$, let q_i be a request such that $(q_i.op = write) \vee (q_i.op = gather \wedge q_i.node = u)$. Further assume that $q_1 \rightsquigarrow q_2$ and q_2 belongs to $u.gwlog$. Then, q_1 belongs to $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$ holds.*

Proof. By definition, $q_1 \rightsquigarrow q_2$ if and only if there exists i such that $q_1 \overset{i}{\rightsquigarrow} q_2$. We prove the lemma by induction on i.

Base case: $i = 1$, that is, $q_1 \overset{1}{\rightsquigarrow} q_2$. There are two cases $q_1 \overset{1}{\rightsquigarrow} q_2$ by rule (1) or by rule (2).

First case, $q_1 \overset{1}{\rightsquigarrow} q_2$ by rule (1), that is, $q_1.node = q_2.node$ and $q_1.index < q_2.index$. There are two cases, (a) $u = q_1.node$; (b) $u \neq q_1.node$. Case (a), that is, $u = q_1.node$. By lemma 6.6, q_1 and q_2 belong to $u.gwlog$, and $precedes(u.gwlog, q_1, q_2)$ holds. Case (b), that is, $u \neq q_1.node$. Let v be $q_1.node$. By lemma 6.6, $precedes(v.gwlog, q_1, q_2)$ holds. Since $u \neq v$, q_1 and q_2 are $write$ requests. Since q_2 belongs to $u.gwlog$, by lemma 6.7, q_1 is in $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$ holds.

Second case, $q_1 \overset{1}{\rightsquigarrow} q_2$ by rule (2), that is, q_1 is a $write$ request and q_2 is a $gather$ request such that q_2 returns $(q_1.node, q_1.index)$ in $q_2.retval$. Since q_2 returns $(q_1.node, q_1.index)$, q_1 is in $u.log$ and $precedes(u.log, q_1, q_2)$ holds. By construction of $u.gwlog$, q_1 is in $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$ holds.

Induction step: $q_1 \overset{i}{\rightsquigarrow} q' \overset{1}{\rightsquigarrow} q_2$. Consider the two cases, (1) $(q'.op = write) \vee (q'.op = gather \wedge q'.node = u)$, and (2) $(q'.op = gather \wedge q'.node \neq u)$.

Case (1), that is, $(q'.op = write) \vee (q'.op = gather \wedge q'.node = u)$. By induction hypothesis, q' belongs to $u.gwlog$, $precedes(u.gwlog, q', q_2)$ holds. Also by induction hypothesis, q_1 belongs to $u.gwlog$, $precedes(u.gwlog, q_1, q')$ holds. Hence, q_1 belongs to $u.gwlog$, and $precedes(u.gwlog, q_1, q_2)$ holds.

Case (2), that is, $(q'.op = gather \wedge q'.node \neq u)$. Let $q'.node$ be v. Since $q'.op = gather$, $q' \overset{1}{\rightsquigarrow} q_2$ could only be by rule (1), that is, $q_2.node = v$ and $q'.index < q_2.index$. Since $v \neq u$, q_2 must be a $write$ request. By Lemma 6.6, $precedes(v.gwlog, q', q_2)$ holds. Now consider the two possible cases for q_1, (a) $q_1.op = write$, and (b) $q_1.op = gather \wedge q_1.node = u$. Case (a), that is, $q_1.op = write$. By induction hypothesis, q_1 belongs to $v.gwlog$ and $precedes(v.gwlog, q_1, q')$ holds. From above, q_1 and q_2 belong to $v.gwlog$ and $precedes(v.gwlog, q_1, q_2)$. By lemma 6.7, q_1 belongs to $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$.

Case (b), that is, $q_1.op = gather \wedge q_1.node = u$. Since $q_1.node \neq q'.node$, $q_1 \overset{i}{\rightsquigarrow} q'$, and q_1 and q' are $gather$ requests, i must be greater than 1. By Lemma 6.8, there is a $write$ request q'' such that $q''.node = u$ and $q_1 \overset{j}{\rightsquigarrow} q'' \overset{i-j}{\rightsquigarrow} q'$, for some $j, i > j \geq 1$. By induction hypothesis, q'' belongs to $v.gwlog$ and $precedes(v.gwlog, q'', q')$ holds. Hence, from above, $precedes(v.gwlog, q'', q_2)$ holds. Since q'' and q_2 are $write$ requests, $q_2.node = v$, q_2 belongs to $u.gwlog$, and $precedes(v.gwlog, q'', q_2)$ holds, by Lemma 6.7, $precedes(u.gwlog, q'', q_2)$ holds. From above, q'' belongs to $u.gwlog$ and $q_1 \overset{j}{\rightsquigarrow} q''$ for some $j \geq 1$. Hence, by induction hypothesis, $precedes(u.gwlog, q_1, q'')$ holds. From above, it follows that, q_1 belongs to $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$ holds. □

Lemma 6.10 *For any node u, $u.gwlog'$ respects the causal ordering among requests in $u.gwlog'$.*

Proof. We prove this lemma by induction on the number of iterations in the construction of $u.gwlog'$. For the base case, by Lemma 6.9, $u.gwlog$ respects the causal ordering among requests in $u.gwlog$. In each iteration in the construction, the additional requests are added at the end of $u.gwlog'$. By Lemma 6.9 again, this step preserves the causal ordering among requests in $u.gwlog'$. □

Lemma 6.11 *For any node u, $u.log'$ and $u.gwlog'$ are compatible.*

Proof. We prove this lemma by induction on the number of iterations in the construction of $u.log'$ and $u.gwlog'$. For the base case, by Lemma 6.5, $u.log$ and $u.gwlog$ are compatible. In each iteration of the construction, by the base case and the induction hypothesis, additional requests appended to both the request sequences are mutually compatible. Hence, $u.log'$ and $u.gwlog'$ are compatible. □

Theorem 4 *Let set A be the execution-history of any lease-based algorithm $\mathcal{A}$. Then, A is causally consistent.*

Proof. Consider any node u in T. By construction, $u.gwlog'$ is a serialization of all the requests in $u.gwlog'$. From this observation and Lemma 6.10, $u.gwlog'$ is causally consistent. By construction, $u.log'$ contains all the requests in $pruned(A, u)$. By Lemma 6.11, $u.log'$ and $u.gwlog'$ are compatible. Hence, by definition, A is causally consistent. □

7 Discussion

What we have done in this paper is a useful case study in the design and analysis of self-tuning distributed algorithm for an important key primitive. Although we have focussed on fault-free case, we can extend some of our results to faulty environment, especially with respect to causal consistency, by keeping track of time-stamps with writes.

An open problem for future research is to design a self-tuning algorithm for the approximate aggregation problem, where one allows a certain numerical error in the aggregate value, and analyze the algorithm in competitive analysis framework.

References

[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9:37–49, 1995.

[2] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. *Journal of Algorithms*, 28:67–104, 1998.

[3] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. *Information and Computation*, 185:1–40, 2003.

[4] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 515–526, June 2004.

[5] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX Symposium on Networked Systems Design and Implementation*, May 2006.

[6] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, UK, 1998.

[7] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. T. Foster. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, pages 181–194, August 2001.

[8] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A strong consistency mechanism for the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 15:1266–1276, 2003.

[9] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 133–148, October 2003.

[10] Ganglia: Distributed monitoring and execution system. `http://ganglia.sourceforge.net`.

[11] M. G. Gouda. *Elements of Network Protocol Design*. John Wiley & Sons, New York, 1998.

[12] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, December 1989.

[13] N. Jain, P. Yalagandula, M. Dahlin, and Y. Zhang. INSIGHT: A distributed monitoring system for tracking continuous queries. In *Work-in-Progress Session at SOSP 2005*, pages 23–26, October 2005.

[14] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

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

[16] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 144–155, September 2000.

[17] R. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21:164–206, 2003.

[18] M. Roussopoulos and M. Baker. CUP: Controlled update propagation in peer-to-peer networks. In *USENIX Annual Technical Conference*, pages 167–180, June 2003.

[19] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[20] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995.

[21] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *Proceedings of the 2nd Workshop on Hot Topics in Networks*, November 2003.

[22] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15:757–768, 1999.

[23] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proceedings of the ACM SIGCOMM Conference*, pages 379–390, August/September 2004.