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 Introduction

Information aggregation is a basic building block in many large-scale distributed applications such as system management [10, 22], service placement [9, 23], file location [5], grid resource monitoring [7], network monitoring [13], and collecting readings from sensors [14]. Certain generic aggregation frameworks [7, 18, 24] 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 [18] 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 [24] 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 [24], 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 [24].

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 [21, 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 [20]. 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 [24], Astrolabe [18], 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 [19], 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 polylogarithmic 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.

Due to space limitations, this submission focuses on conveying the main ideas underlying our results, and some proofs are omitted. The 5-page appendix alluded to below provides some additional proof details, along with a complete description of algorithm RWW. A complete version of our work, which includes all proofs, is available online [17].

An appendix has been submitted to the program chair.

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 [21, 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.

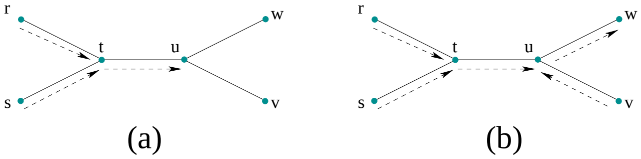
For any request sequence σ and any ordered pair of neighboring nodes (u, v), we define $\sigma(u, v)$ as follows: (1) $\sigma(u, v)$ is a subsequence of σ; (2) for any $write$ request q in σ such that $q.node$ is in $subtree(u, v)$, q is in $\sigma(u, v)$; and (3) for any $combine$ request q in σ such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

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.



(a) (b)

Figure 1: An example tree network.

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 [21, 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.

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

```
    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}
        $\underline{oncombine}$(u);
        foreach v $\in$ tkn() do
          uaw[v] := $\emptyset$; od
        if u $\notin$ pndg $\rightarrow$
          if nbrs() \ tkn() = $\emptyset$ $\rightarrow$
            return gval();
          □ nbrs() \ tkn() $\neq \emptyset$ $\rightarrow$
            sendprobes(u);
            snt[u] := nbrs() \ tkn(); fi fi
$T_2$    true $\rightarrow$ {write q}
        val := q.arg;
        if grntd() $\neq \emptyset$ $\rightarrow$
          id := newid();
          forwardupdates(u, id); fi
$T_3$    □ rcv probe() from w $\rightarrow$
        $\underline{probercvd}$(w);
        foreach v $\in$ tkn() \ {w} do
          uaw[v] := $\emptyset$; od
        if w $\notin$ pndg $\rightarrow$
          if nbrs() \ {tkn() $\cup$ {w}} = $\emptyset$ $\rightarrow$
            sendresponse(w);
          □ nbrs() \ {tkn() $\cup$ {w}} $\neq \emptyset$ $\rightarrow$
            sendprobes(w);
            snt[w] := nbrs() \ {tkn() $\cup$ {w}}; fi fi
$T_4$    □ rcv response(x, flag) from w $\rightarrow$
        $\underline{responsercvd}$(flag, w);
        $\underline{aval[w] := x}$;
        taken[w] := flag;
        foreach v $\in$ pndg do
          snt[v] := snt[v] \ {w};
          if snt[v] = $\emptyset$ $\rightarrow$
            pndg := pndg \ {v};
            if v = u $\rightarrow$
              return gval();
            □ v $\neq$ u $\rightarrow$
              sendresponse(v); fi fi od
$T_5$    □ rcv update(x, id) from w $\rightarrow$
        $\underline{updatercvd}$(w);
        aval[w] := x;
        uaw[w] := uaw[w] $\cup$ id;
        if grntd() \ {w} $\neq \emptyset$ $\rightarrow$
          nid = newid();
          sntupdates := sntupdates $\cup$ {w, id, nid};
          forwardupdates(w, nid);
        □ grntd() \ {w} = $\emptyset$ $\rightarrow$
          forwardrelease(); fi
$T_6$    □ rcv release(S) from w $\rightarrow$
        $\underline{releasercvd}$(w);
        granted[w] := false;
        onrelease(w, S);
    end
```

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

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

procedure sendresponse(node w)
  if (nbrs() \ {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() \ {w} = $\emptyset$);

procedure onrelease(node w, set S)
  Let id is the smallest id in S;
  foreach v $\in$ tkn() \ {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() \ {w} do
    x := f(x, aval[v]); od
  return x;

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

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 $\mathbf{false}$*; 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* $\mathbf{false}$.

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

Proof. See Appendix A. □

Lemma 4.1 is summarized in Figure 4 (see Appendix A). 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 4.2 *Consider a sequential execution of a request sequence σ by a lease-based algorithm. For any node u, $I(u)$ is an invariant.*

Proof. See [17]. □

Lemma 4.3 *Any lease-based aggregation algorithm is nice.*

Proof. See Appendix A. □

From Lemma 4.3 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

We define RWW as an online lease-based aggregation algorithm that follows the policy decisions shown in Figure 5 (see Appendix B) 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 $subtree(u,v)$.

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 $\sigma(u,v)$.

Lemma 5.1 *The algorithm* RWW *is a* $(1,2)$-*algorithm.*

Proof. See Appendix B. □

5.1 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

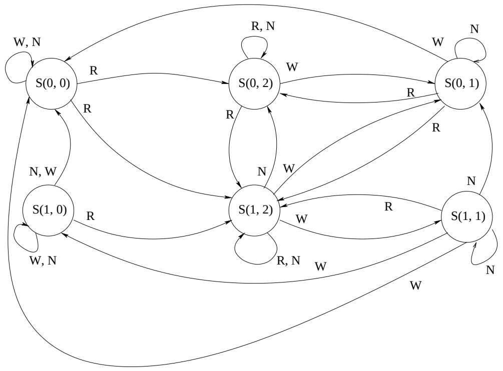


Figure 3: States and state transitions for any pair of nodes (u, v) in executing requests from $\sigma'(u, v)$ (defined in Lemma 5.2).

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).

For any ordered pair of neighboring nodes u and v, we define $\mathit{type}(u, v)$ messages as the following kinds of messages exchanged between u and v: (1) probe messages from v to u; (2) $\mathit{response}$ messages from u to v; (3) update messages from u to v; and (4) $\mathit{release}$ 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 $\mathit{type}(u, v)$ messages in execution of σ by $\mathcal{A}$. Note that the total number of messages exchanged between u and v is the sum 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 $\mathit{combine}$ request, then $F_{\mathrm{RWW}}(u, v)$ is 2; (3) if the last two completed requests in $\sigma(u, v)$ are a $\mathit{combine}$ 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 $u.\mathit{granted}[v]$ holds; otherwise, 0.

Lemma 5.2 *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 $C_{\mathrm{OPT}}(\sigma, u, v)$.*

Proof sketch. 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 $\mathit{type}(u, v)$ 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, we can show that $C_{\mathrm{RWW}}(\sigma, u, v) = C_{\mathrm{RWW}}(\sigma(u, v), u, v)$ (see Lemma B.3 in Appendix B). For RWW, for any request q in $\sigma(u, v)$, we charge all the messages incurred in executing q by RWW to the same request q in $\sigma'(u, v)$. 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, first, for any request q in $\sigma(u, v)$, we charge all $\mathit{type}(u, v)$ messages incurred in executing q by OPT to the same request q in $\sigma'(u, v)$. Second, we can show that any $\mathit{type}(u, v)$ message incurred in execution of $\sigma(v, u)$ can be charged to some noop request in $\sigma'(u, v)$. Thus, for both RWW and OPT, we can charge all $\mathit{type}(u, v)$ messages to requests in $\sigma'(u, v)$ respectively. 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 3, 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 5. 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 4 (see Appendix A).

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 the 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 in Appendix B). 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 5.2, 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 $combine$ 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 B.2). 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 $combine$ 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 (see appendix), 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 $\mathbf{true}$ if and only if $x = f(recent(A, \sigma))$.

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 $I_1(u) \wedge I_2(u) \wedge I_3(u)$.

See [17] for the proofs of the following two lemmas.

Lemma 6.1 *For any node u, $I(u)$ is an invariant.*

For a request sequence σ and a request q, $index(\sigma, q)$ 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, $precedes(\sigma, q_1, q_2)$ is defined to be $\mathbf{true}$ if and only if $index(\sigma, q_1) < index(\sigma, q_2)$.

Lemma 6.2 *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.*

Lemma 6.3 *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 6.2, $u.gwlog$ 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 6.2 again, this step preserves the causal ordering among requests in $u.gwlog'$. □

Lemma 6.4 *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 6.1, $u.log$ 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 6.3, $u.gwlog'$ is causally consistent. By construction, $u.log'$ contains all the requests in $pruned(A, u)$. By Lemma 6.4, $u.log'$ 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 adhoc 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] C. G. Plaxton, M. Tiwari, and P. Yalagandula. Online aggregation over trees. `www.cs.utexas.edu/˜mitult/agg.pdf`.

[18] 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.

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

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

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

[22] 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.

[23] 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.

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