Efficient Numerical Error Bounding for Replicated Network Services

Haifeng Yu
Computer Science Department
Duke University
Durham, NC 27708-0129, USA
yhf@cs.duke.edu

Amin Vahdat*
Computer Science Department
Duke University
Durham, NC 27708-0129, USA
vahdat@cs.duke.edu

Abstract

The goal of this work is to support replicated network services that accept updates to numerical records from multiple wide-area locations. Given the high overhead of maintaining strong consistency, many replicated services can tolerate divergence of their shared data, as long as the numerical error is bounded. Target distributed services include replicated stock quotes services, online auctions, distributed sensor systems, wide-area resource accounting and load balancing for replicated servers.

We present two algorithms to efficiently bound absolute error using only local information. Split-Weight AE *separately bounds increases and decreases, while* Compound-Weight AE *bounds them together. The two algorithms can be combined to provide good performance and low space overhead. Our* Inductive RE *algorithm transforms relative error to absolute error solely based on local knowledge, taking advantage of the fact that the divergence was properly bounded prior to each invocation of the algorithm. We also discuss two optimizations that reduce the space and computational overheads in the algorithms.*

* This work is supported in part by the National Science Foundation (EIA-99772879). Vahdat is also supported by an NSF CAREER award (CCR-9984328).

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

1 Introduction

Many network services store and update numerical records at multiple wide-area sites, which introduces issues of consistency among replicas. Given the high overhead of maintaining strong consistency, many of these services can tolerate divergence from the consistent data, as long as the divergence is bounded. Consider the following replicated services:

Stock Quotes Services Users retrieve stock quotes from replicated stock quotes servers. Each server may also accept updates to the current quotes. Users are concerned with the amount of "error" in the quotes they observe. For example, a user may prefer to see quotes within only ± 1 cent/share (absolute error) or $\pm 1\%$ (relative error) of the accurate quotes.

Online Auctions Each online auction server maintains the highest current bid for a number of items. A user accessing a replica desires guarantees regarding the maximum difference between the highest bid stored locally and the largest global bid.

Distributed Sensor Systems The sensor system takes the average temperature (pollution level, etc.) of an area. Each sensor periodically takes a sample at a fixed point in the area, and updates the average value according to the new sample value taken. User may retrieve the average value at any sensor location. Although users can tolerate approximate value, they may still want an upper bound on the "inaccuracy".

Wide-area Resource Accounting As we move toward global distributed computing, one goal is to account for aggregate consumed resources across multiple providers on a per-user basis[9, 24, 25]. Given the scale of this problem, maintaining accurate resource usage information will incur prohibitive overhead. Allowing bounded error in usage information is a promising approach to solving this problem.

Load Balancing for Replicated Servers For many replicated services, client programs do not directly choose which replica to contact. Instead, a front end forwards requests to the server judged to deliver the highest quality of service for that request. A front end uses its forwarding history to estimate the load of each server. When there are multiple front ends[19, 24], each sees a subset of the request stream, and uses this information to update its estimated load information. Once again, the load information is updated from multiple locations, and it is beneficial to bound the maximum error on load information observed by each front end.

One approach for guaranteeing accurate numerical information is to utilize standard techniques for maintaining strong consistency across wide-area networks. However, the communication costs and latency associated with such techniques often have prohibitively high overhead. We observe that many replicated services, including the ones described above, can tolerate some level of inconsistency in exchange for improved performance and availability, as long as they are provided guarantees regarding the maximum allowable error. In this context, the goal of this work is to develop techniques to efficiently bound numerical inaccuracy by reducing the amount of required wide-area communication.

Despite the importance of bounding numerical error for replicated network services, this topic has not been well studied in the literature. In the context of data caching, [3] proposes the concept of bounded numerical error. However, the authors do not generalize the concept to replicated databases. Efforts exploiting weak consistency [16, 17, 18, 21, 22, 23] concentrate on aspects other than numerical error, such as the number of conflicting transactions. Integrity constraint management algorithms[4, 5, 6, 7, 13, 14] for distributed databases are related to error bounding but are typically inefficient when applied to the special case of bounding numerical error. The demarcation protocol[4] allows easy maintenance of linear inequalities for distributed databases. Error bounding is closely related to enforcing linear inequalities but has three important properties not present in general linear inequalities: i) The copies of a data item are inter-related, ii) servers have approximate information about what writes other servers have seen, iii) during write propagation, writes on all data items are propagated. Because the demarcation protocol addresses a more general problem, it is unable to exploit these properties, resulting in significantly higher communication and space overhead.

In this paper, we present algorithms to efficiently bound numerical error for replicated network services. They are developed in the TACT[26, 27] project, which is a toolkit for building replicated Internet services. Two algorithms *Split-Weight AE* and *Compound-Weight AE* are proposed to bound absolute error. They bound error by limiting the "total weighted writes" accepted by one server but not seen by others. All decisions are based on local information and the algorithms do not incur overhead to acquire global knowledge. Split-Weight AE makes conservative decisions but is amenable to space optimizations, while Compound-Weight AE makes optimal decisions at the cost of higher space overhead. Our *Inductive RE* bounds relative error by transforming it into absolute error and then using Split-Weight AE or Compound-Weight AE to bound the absolute error. By taking advantage of the fact that the divergence was properly bounded prior to each invocation of the algorithm, Inductive RE is able to perform the transformation solely based on local knowledge. We also study the performance of our algorithms through both analysis and simulation.

In summary, this paper makes the following contributions:

- We describe the importance of bounding numerical error to support replicated network services.
- We propose practical algorithms to bound absolute error and relative error using only local information, without incurring the overhead of obtaining global knowledge.
- We explore two optimizations that reduce the space and computational overheads in the algorithms.

The next section describes our replicated database model. We present our error bounding algorithms in Section 3. Section 4 discusses two important optimizations to reduce space and computational overhead for our algorithms. In Section 5, we study the performance of the algorithms. Related work is described in Section 6. In Section 7, we present our conclusions.

2 System Model

The database maintaining the data shared by the network servers is replicated across n servers, $server_1$, $server_2, \ldots, server_n$. The replicated database is composed of multiple *data items*. Each data item has a numerical value for which the service desires to bound *error*. The allowed error for a data item is independent of other data items. We first focus on the case of a single data item and then discuss scalability issues for multiple data items in later sections.

Every server can accept reads (inquires) and writes (updates) from users. Reads return the current value of the data item on the server. A write W increases or decreases the value of a data item by the *weight* ($W.weight$) of the write. $W.weight$ is positive for increases and negative for decreases. While beyond the scope of this paper, our algorithms can be extended to

transactions that consist of multiple read/write operations in a straightforward manner.

The server that accepts a write W from a client is the *originating server* of the write, and is denoted by $W.server$. A server updates other servers by propagating writes. The database image itself is never communicated to other servers. Upon accepting a write, a server does not have to update other servers immediately and divergence among replicas is allowed. Writes with the same originating server are always propagated according to the order they are accepted by that server. *Write propagation* can be done in the form of gossip messages[18], anti-entropy sessions[10, 20], broadcast or even unicast. To reduce communication overhead, some write propagation methods allow multiple writes to be merged into one write during propagation. Our algorithms are orthogonal to the write propagation method used by the database, although the freshness of views (defined later in this section) may be affected.

A server may propagate writes to other servers at any time, and such write propagation is called *voluntary write propagation* or *background write propagation*. The error bounding algorithms may require a server to propagate writes, which is called *compulsory write propagation*. Compulsory write propagation is necessary for the correctness of the algorithms, while voluntary write propagation only affects performance.

Each server maintains a write log, which is an ordered list of the writes the server accepts from clients or sees from other servers. Write log recycling can be done using various techniques[10, 18, 20]. We define the functions $twn(i,j)$ and $twp(i,j)$ as:

$$twn(i,j) = \sum\{W.weight \mid W.weight < 0 \text{ and } W.server = server_j \text{ and } W \in \text{write log of } server_i\}$$

$$twp(i,j) = \sum\{W.weight \mid W.weight > 0 \text{ and } W.server = server_j \text{ and } W \in \text{write log of } server_i\}$$

Intuitively, $twn(i,j)$ is the total negative weight of the writes $server_i$ sees originated from $server_j$, while $twp(i,j)$ is the total positive weight. Distinguishing negative weight and positive weight is necessary because we allow weight on different data items to be totaled in our optimizations. Thus, negative weight on one data item should not offset the positive weight on another data item (see Sections 4.1 and 4.2).

We use V_i to denote the value of the data item on $server_i$, and V_{init} to denote its initial (consistent) value. We use V_{final} to denote the value of the data item if all writes accepted by the system by time t has been applied. Note that as new writes are injected into the system, V_{final} evolves. The following equalities hold for V_i, V_{init} and V_{final}:

$$V_i = V_{init} + \sum_{k=1}^{n}(twn(i,k) + twp(i,k))$$

$$V_{final} = V_{init} + \sum_{k=1}^{n}(twn(k,k) + twp(k,k))$$

For $server_i$, a data item's *absolute error*(AE) is bounded within $[\alpha_i, \beta_i]$ ($\alpha_i \leq 0$ and $\beta_i \geq 0$) if and only if at all times, the following inequality holds:

$$\alpha_i \leq V_{final} - V_i \leq \beta_i \quad (1)$$

Similarly, we say the *relative error*(RE) is bounded within $[\gamma_i, \delta_i]$ ($\gamma_i \leq 0$ and $0 \leq \delta_i \leq 1$) if and only if:

$$\gamma_i \leq 1 - \frac{V_i}{V_{final}} \leq \delta_i \quad (2)$$

For relative error, we assume $V_i > 0,\ 1 \leq i \leq n$.

Each server in the system has approximate knowledge of what writes other servers have seen. We say that each server has its *view* of $twn(i,j)$ and $twp(i,j)$, for $1 \leq i \leq n, 1 \leq j \leq n$. The views are either updated during write propagation or updated with explicit view update messages. The actual view update fashion and view freshness depend on the write propagation method. For example, if we use unicast, then during each write propagation, the two parties can inform each other of the writes they see. For anti-entropy sessions, more efficient view update mechanism can be used and details can be found in [10]. The correctness of our algorithms does not depend on the freshness of the views, but performance is affected. We denote $server_k$'s view of $twn(i,j)$ and $twp(i,j)$ as $twn_k(i,j)$ and $twp_k(i,j)$. Intuitively, $twn_k(i,j)$ is the total negative weight of the writes that $server_k$ believes that $server_i$ sees from $server_j$. During a *view advance*, $server_k$ updates $twn_k(i,j)$ and $twp_k(i,j)$. Views are *conservative* in that $server_k$ will never assume that $server_i$ sees a write that $server_i$ actually does not see.

3 Bounding Numerical Error Using Local Information

This section describes two different algorithms for bounding AE and one algorithm for bounding RE. The idea of the AE bounding algorithms is to bound the total weight of writes accepted by one server but not seen by other servers. The first algorithm, *Split-Weight AE*, bounds positive weight and negative weight separately. The second algorithm, *Compound-Weight AE*, keeps track of the possible range of values on other servers and allows negative weight and positive weight to offset. The basic idea of our relative error bounding algorithm, *Inductive RE*, is to transform the relative error into absolute error. The decisions made in the algorithms are all based only on local information, without incurring the overhead of obtaining global knowledge. All three algorithms require cooperation of other servers in the system to enforce local bounds. For each algorithm, we discuss how $server_j$ acts to bound the error for a single data item on $server_i$. Due to space limitations, correctness proofs for the three algorithms are omitted, but can be found in [28].

3.1 Split-Weight AE

In this algorithm, each $server_j$ maintains two local variables x and y for each $server_i, i \neq j$. They are used to record the total negative and positive weight of the writes accepted by $server_j$ but not seen by $server_i$. $server_j$ uses its view to compute x and y. However, since the view is conservative, x and y are also conservative.

Both variables x and y are initially set to zero and are updated in the following fashion:

1. When $server_j$ accepts a new write W, if $W.weight < 0$, $x = x + W.weight$, else $y = y + W.weight$.
2. When $server_j$ advances its view, if $twn_j(i,j)$ and $twp_j(i,j)$ are updated to $twn'_j(i,j)$ and $twp'_j(i,j)$ respectively, then $x = x - (twn'_j(i,j) - twn_j(i,j))$ and $y = y - (twp'_j(i,j) - twp_j(i,j))$. This subtracts weight of the newly propagated writes from x and y.

When $server_j$ receives a write W from a client, it checks the conditions:

$$x + W.weight \geq \alpha_i/(n-1), \ if \ W.weight < 0 \quad (3)$$

$$y + W.weight \leq \beta_i/(n-1), \ if \ W.weight > 0 \quad (4)$$

If the conditions do not hold, $server_j$ must advance its view for $server_i$ (potentially propagating writes to $server_i$) before the new write may return.

Split-Weight AE is pessimistic, in the sense that $server_j$ may propagate writes to $server_i$ that are unnecessary for bounding the error. For example, the algorithm does not consider the case where negative weight and positive weight may offset each other. In our simulation study, we will quantify how pessimistic Split-Weight AE is under different workloads. However, this simple design enables several optimizations not applicable to Compound-Weight AE (see Section 3.2). For example, in order to optimize the space overhead, several data items may share the same x and y variables (see section 4.1).

3.2 Compound-Weight AE

In Compound-Weight AE, each $server_j$ maintains three local variables z, min and max for each $server_i$, $i \neq j$. Intuitively, z is the total weight of the writes accepted by $server_j$ but not seen by $server_i$ in $server_j$'s view. However, since a view can be stale, $server_i$ may actually see more writes than $server_j$ is aware of. So we use min/max to record the minimum/maximum possible total weight of those writes that $server_i$ may see but are not in $server_j$'s view $twn_j(i,j)$ or $twp_j(i,j)$.

All variables z, min and max are initially set to zero and are updated in the following fashion:

1. When $server_j$ accepts a new write W, $z = z + W.weight$. If $z < min$, then $min = z$. If $z > max$, then $max = z$. Note that since we assume writes from the same originating server are always propagated and applied according to the accept order, min and max are properly maintained in this way.
2. When $server_j$ advances its view for $server_i$, $server_j$ first resets all three variables. Next for each write W in $server_j$'s write log, if $W.server = server_j$ and in $server_j$'s new view, $server_i$ has not seen W, then z, min and max are updated as if W were newly accepted. In this way, z, min and max are re-established for this new view. Rescanning the write log whenever $server_j$ advances its view appears redundant, but is actually necessary for correctness.

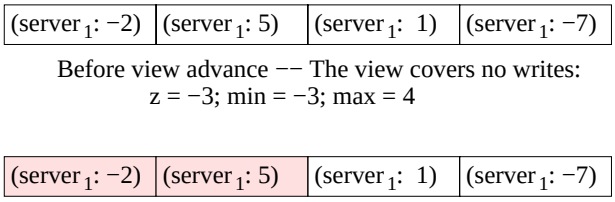

Figure 1: *View Advance in Compound-Weight AE*

Figure 1 illustrates how the three variables on $server_1$ are updated during a view advance. Each write is denoted by the pair ($W.server$: $W.weight$). For simplicity, only writes with $W.server = server_1$ are depicted in the figure. Before the view advance, $server_1$ is not aware of any writes seen by another server, say $server_2$. After the view advance, $server_1$ knows that $server_2$ has seen writes ($server_1$: -2) and ($server_1$: 5). Besides how to update z, min, and max, Figure 1 also explains why rescanning the write log is necessary. Suppose we want to ensure condition $z + W.weight - min \leq 10$ (see inequalities (5) and (6)). Before the view advance, min is -3. If we continue to use this min after the view advance, the algorithm may make incorrect decisions, since min should actually be -6, which makes the condition tighter.

When $server_j$ receives a write W from a client, it checks the following conditions:

$$z + W.weight - max \geq \alpha_i/(n-1) \quad (5)$$

$$z + W.weight - min \leq \beta_i/(n-1) \quad (6)$$

If the conditions do not hold, $server_j$ must advance its view for $server_i$ (potentially propagating writes to $server_i$) before the new write may return. According to these two conditions, it is possible that $server_j$ may need to propagate writes to $server_i$ when its view for $server_i$ advances. To avoid this, $server_j$ can be lazy in advancing its view for $server_i$. That is, the need for view advance is not checked until the above two

conditions are violated. At that time, $server_j$ can deal with view advance and potentially propagate writes to $server_i$.

As opposed to Split-Weight AE, Compound-Weight AE is optimal given only local information. In other words,

$$\alpha_i/(n-1) \leq (twn(j,j) + twp(j,j)) - (twn(i,j) + twp(i,j)) \leq \beta_i/(n-1)$$

holds if and only if conditions (5) and (6) hold. Please see [28] for the proof.

3.3 Inductive RE

Inductive RE transforms relative error to absolute error using only local information. Recall from definition (2) that relative error is bounded within $[\gamma_i, \delta_i]$ if an only if:

$$\gamma_i \leq 1 - \frac{V_i}{V_{final}} \leq \delta_i$$

This definition requires that V_{final} be a parameter of the transformation. However, knowing V_{final} accurately itself requires strong consistency. Since our goal is to avoid the overhead of strong consistency, the system must be able to bound relative error without knowing V_{final}.

One naive way to overcome this is to transform definition (2) to:

$$\gamma_i/(1-\gamma_i) \times V_i \leq V_{final} - V_i \leq \delta_i/(1-\delta_i) \times V_i$$

By setting:

$$\begin{aligned} \alpha_i &= \gamma_i/(1-\gamma_i) \times V_i \\ \beta_i &= \delta_i/(1-\delta_i) \times V_i \end{aligned}$$

we can apply either of the previous AE algorithms to enforce the inequality. However, since V_i changes over time, $server_i$ must constantly update α_i and β_i and inform other servers in the system of this change. This requires that a consensus algorithm be run among all servers whenever V_i decreases. As a result, the performance could degrade significantly.

Inductive RE is based on the observation that for any j, V_j was properly bounded before the invocation of the algorithm and is an approximation of V_{final}. So $server_j$ may use V_j as an approximate norm to bound γ_i and δ_i. Transforming the definition of RE, we have the following two inequalities:

$$\begin{aligned} V_{final} - V_i &\geq \gamma_i \times V_{final} \quad (7) \\ V_{final} - V_i &\leq \delta_i \times V_{final} \quad (8) \end{aligned}$$

On the other hand, on $server_j$, we know that $\gamma_j \leq 1 - V_j/V_{final}$, so $V_{final} \geq V_j/(1-\gamma_j)$. Figure 2 illustrates the relationship between V_j and V_{final}.

Thus the following two inequalities are sufficient conditions for inequality (7) and (8):

$$\begin{aligned} V_{final} - V_i &\geq \gamma_i \times V_j/(1-\gamma_j) \\ V_{final} - V_i &\leq \delta_i \times V_j/(1-\gamma_j) \end{aligned}$$

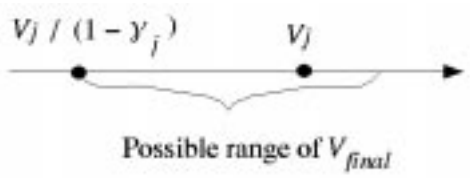

Figure 2: *How to Use V_j as an Approximate Norm to Bound RE*

The right-hand side expressions can be evaluated using only local information. So in order to bound relative error for $server_i$, $server_j$ only needs to apply Split-Weight AE or Compound-Weight AE and use:

$$\begin{aligned} \alpha_i &= \gamma_i \times V_j/(1-\gamma_j) \\ \beta_i &= \delta_i \times V_j/(1-\gamma_j) \end{aligned}$$

Note that since the computed α_i and β_i change with V_j, whenever V_j changes, the limits should be recomputed and re-checked. However, no consensus algorithms are necessary because V_j is known locally.

4 Optimizing for Scalability

With the algorithms described in the last section, numerical error can be efficiently bounded for small scale replicated network services. However, since we are interested in extensive replication, in this section we discuss two optimizations that reduce the space and computational overheads in the algorithms.

4.1 Reducing Space Overhead

We have discussed how to bound AE and RE for a single data item. The algorithms incur a per data item space overhead of $O(n)$, where n is the number of servers. If we simply use multiple instances of the algorithms, the size of the data structure maintained by the algorithms can be n times the size of the database itself in the worst case. If a database maintains tens of thousands of data items, this high space overhead can be prohibitive.

To reduce space overhead, we assume that for all data items, $server_i$ has the same α_i and β_i (or γ_i and δ_i), otherwise the space needed simply for storing α_i and β_i will grow linearly with the number of data items. The application may still use several different α_is(β_is) for different data items by using multiple instances of our algorithm. We reduce space overhead by exploiting the fact that during write propagation, writes to all data items on a server are propagated to another server. So we only need to maintain information for those data items accessed between two write propagations. We also take advantage of the locality among the writes accepted by a server. For "hot" data items, we maintain accurate information needed by the algorithms. For data items seldom accessed, we allow them to share the same data structure and maintain conservative information.

We use a hashtable to store the variables needed by our algorithms. Each server maintains one hashtable for every other server in the system. The hashtables are used to maintain the information on "hot" data items. Whenever $server_j$ receives a write on data item D, it uses D as a key to create or update variables in the hashtables. The total space used by a hashtable is bounded. In the case where a hashtable becomes full, a shared entry is created for all other data items without a hashtable entry. On each write propagation, the hashtable and the shared entry corresponding to the receiving server are cleared and the space is freed.

Care must be taken when maintaining the shared entry. For Split-Weight AE, the shared entry simply consists of two variables x and y, which are updated in the same way as normal hashtable entries. For Compound-Weight AE, it is difficult to maintain a shared entry for multiple data items, so we use Split-Weight AE for the shared entry and Compound-Weight AE for hashtable entries. In Inductive RE, the shared entry must also record the smallest V_j of the data items using that entry, so that the computed α_i and β_i values are tight. Using a shared entry may result in conservative behavior, since weight accumulated on multiple items is coalesced to a single item. However, a server can improve performance at the cost of larger hashtables and higher space overhead.

4.2 Reducing Computational Overhead

While less of a concern than memory overhead, in this section we describe techniques for reducing our algorithms' computational overhead. In our algorithms, a server needs to update one hashtable for every other server in the system when accepting a write. These updates are on the critical path for accepting writes. Thus, if there are a large number of servers, the overhead of updating n hashtables on each write can be high. In this section, we discuss how to reduce this computational overhead.

The first possible optimization is to combine the hashtables for multiple servers. We can group together servers with similar bounds, and enforce the tightest bounds for a group of servers. The servers in the group can then share a single hashtable. A server can trade space for performance by using smaller groups. Note that this optimization also reduces space overhead.

Another optimization is to use a cache, so that in most cases, we only need to update the cache rather than n hashtables. We only discuss how to use a cache for bounding error with Split-Weight AE, because the data structures in Compound-Weight AE make it difficult to utilize a cache.

Table 1 describes the information maintained by each cache entry. To create a cache entry for data item D, suppose x_i and y_i are the values in $server_i$'s hashtable entry for D, we scan all hashtables, and set:

item	database item
x	total negative weight of newly accepted writes since entry creation
y	total positive weight of newly accepted writes since entry creation
limitx	the limit for x
limity	the limit for y
serverx	the server whose limit we use for this entry's *limitx*
servery	the server whose limit we use for this entry's *limity*

Table 1: *Information Maintained by a Cache Entry for Reducing Computational Overhead in Split-Weight AE*

$$\begin{aligned} limitx &= max\{\alpha_i/(n-1) - x_i \mid 1 \le i \le n\ i \ne j\} \\ limity &= min\{\beta_i/(n-1) - y_i \mid 1 \le i \le n\ i \ne j\} \end{aligned}$$

The variables x and y in the cache entry are set to zero. On each cache hit, we check $x + W.weight \ge limitx$ (if $W.weight < 0$) or $y + W.weight \le limity$ (if $W.weight > 0$). As long as the condition holds, we only need to update x or y in the cache entry, rather than updating all hashtables. If the condition does not hold, we writeback the cache entry to the hashtables, and potentially perform compulsory write propagation. After that, a new cache entry can be established for the data item with new *limitx* and *limity* values. The cache must be flushed whenever $server_i$, $1 \le i \le n$ changes α_i or β_i. We consider this an infrequent operation, so the performance penalty will not be excessive.

A further optimization is to use a linked list for each cache entry. The first node in the list has the tightest *limitx* and *limity*, the second node has the second tightest values and so on. When x or y reaches *limitx* or *limity*, we remove the first node and update the hashtable corresponding to *serverx* and *servery*. In this way, we can avoid scanning all hashtables to find the next tightest limits. However, after updating the hashtables for *serverx* and *servery*, we still need to go through the linked list to see whether *serverx* or *servery* now has tighter limits than nodes in the list.

A cache "snapshot" must be made on write propagation. This snapshot is used in the future to create a "diff" when a cache entry is written back. The x and y in the snapshot are subtracted from the cache entry being written back, before the cache entry is added to the hashtable entry.

Applying the cache idea to bounding relative error is subtle. Since the computed α_i and β_i changes with V_j, in order to choose safe *limitx* and *limity* for all servers, we must decouple the limits from V_j. Recall the conditions we want to enforce are:

$$x_i \ge \gamma_i/(1-\gamma_j) \times V_j = s_i \times V_j \quad (9)$$

$$y_i \le \delta_i/(1-\gamma_j) \times V_j = t_i \times V_j \quad (10)$$

Notation	Meaning	Case 1	Case 2
n	number of servers	10	20
t_{apply}	CPU time to apply a write to database	3ms	3ms
t_{check}	time to check limits and update n hashtables in error bounding algorithms	2ms	4ms
t_{delay}	round-trip message delay in write propagation	200ms	500ms
t_{setup}	CPU time on one replica for TCP connection setup	10ms	10ms
t_{send}	CPU time to send *one* write	1ms	1ms
t_{recv}	CPU time to receive *one* write	1ms	1ms
$E(L_i)$	expectation of L_i	N/A	N/A
$E(Q_i)$	expectation of Q_i	100ms	100ms

Table 2: *Symbols and Default Values Used in Performance Analysis*

Let the value of x_i, y_i and V_j be x_i^c, y_i^c and V_j^c, respectively, at the time when the cache entry is established. We have $V_j = (V_j^c - x_i^c - y_i^c) + x_i + y_i$, $1 \leq i \leq n$ and $i \neq j$. By using this equation to substitute V_j in (9) and (10), we have:

$$\begin{aligned} x_i &\geq s_i \times ((V_j^c - x_i^c - y_i^c) + x_i + y_i) \\ y_i &\leq t_i \times ((V_j^c - x_i^c - y_i^c) + x_i + y_i) \end{aligned}$$

Next, by solving these two inequalities and choosing a rectangular solution area, we have sufficient conditions for (9) and (10) as:

$$\begin{aligned} x_i &\geq s_i/(1-s_i) \times (V_j^c - x_i^c - y_i^c) \\ y_i &\leq t_i/((1-t_i)(1-s_i)) \times (V_j^c - x_i^c - y_i^c) \end{aligned}$$

Using these two conditions, we can now set:

$$\begin{aligned} limitx = max\{&s_i/(1-s_i) \times (V_j^c - x_i^c - y_i^c) - x_i^c \mid \\ &1 \leq i \leq n \; and \; i \neq j\} \\ limity = min\{&t_i/((1-t_i)(1-s_i)) \times (V_j^c - x_i^c - y_i^c) \\ &- y_i^c \mid 1 \leq i \leq n \; and \; i \neq j\} \end{aligned}$$

V_j^c only changes when $server_j$ accepts writes from other servers. In that case, the cache should be flushed.

5 Performance Study

In this section, we first build an analytical model to study the performance of the numerical error bounding algorithms. Next through simulation, we gain further understanding on the applicability of the model. We have also implemented a TACT prototype using the algorithms and studied the performance of three applications (Airline Reservation, Bulletin Board and Load Distribution) running across the wide-area network on top of the prototype. Detailed performance results for these applications are available in [27].

5.1 Performance Analysis

We compare our approach to a conventional one-phase protocol in terms of throughput and latency. The one-phase protocol is a variant of a read one, write all quorum system. It achieves strong consistency for most of our target applications by propagating a new write to all other servers before the write may return. For other applications, writes are more complicated than simply increasing or decreasing a numerical value, for example, a write may check the value and decide whether to continue updating or not. In that case, a stronger two-phase update protocol is needed to achieve strong consistency. However, comparing our algorithms against the better performing one-phase protocol understates the potential performance benefits of our algorithms.

Our algorithms and the one-phase protocol treat reads in the same way, so we are mainly interested in the performance for writes and we only consider write workloads. We assume the database consists of a single data item. To simplify discussion, all servers are assumed to have the same error bounds, i.e. α_i and β_i. We also assume that the workload is evenly distributed among the n servers. We do not consider background write propagation or indirect view advance, both of which will improve the performance of our algorithms.

We first describe the terms and notations used in our analysis, as summarized in Table 2. We consider two sets of parameters. Case 1 corresponds to a replicated network service distributed across the United States, while Case 2 models an international replicated network service. An *epoch on* $server_i$ *for* $server_j$ is the period on $server_i$ between two write propagations to $server_j$. We define the *length* of an epoch as the number of writes accepted by a server directly from clients during that epoch.

The performance of the algorithms is dependent on the characteristics of the workload, such as the weight of each write and the inter-arrival time between writes. To make our analysis generally applicable, we abstract the workload characteristics with two high-level random variables L_i and Q_i, where L_i is the length of an epoch on $server_i$ and Q_i is the queuing delay for a write accepted by $server_i$ before the write gets processed by $server_i$. Our goal is to cover the workload spectrum by choosing different distributions for L_i and Q_i. To gain understanding of what $E(L_i)$ and $E(Q_i)$ we can expect in real world cases, we perform simulations for several workloads (see Section 5.2).

We now present the analytical throughput of our algorithms. A detailed analysis is available in [28]. If

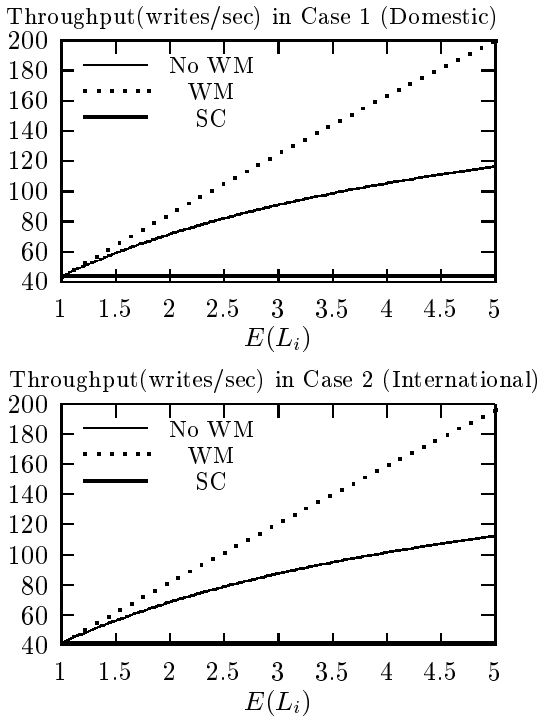

Figure 3: *Throughput of Error Bounding Algorithms vs. Strong Consistency Protocol*

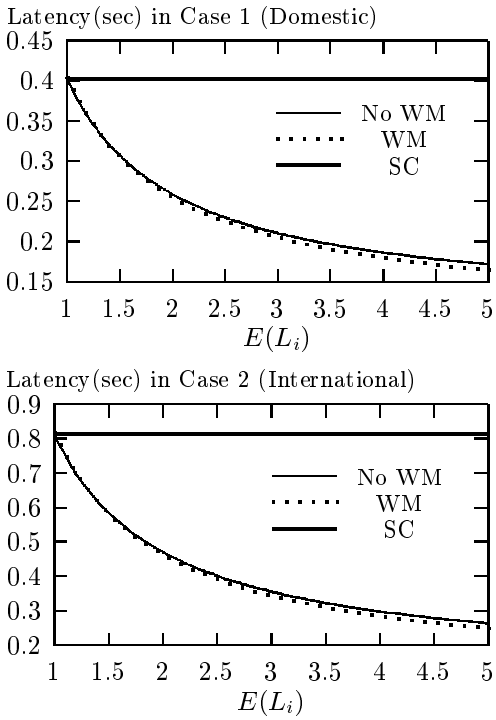

Figure 4: *Latency of Error Bounding Algorithms vs. Strong Consistency Protocol*

writes cannot be merged, we have:

$$Throughput = n/(t_{check} + nt_{apply} + 2(n-1)t_{setup}/E(L_i) + (n-1)t_{send} + (n-1)t_{recv})$$

As mentioned in Section 2, in many cases, it is possible to merge multiple writes into one write in order to save communication overhead. If writes can be merged and in the extreme case where all writes accepted by a server during an epoch can be merged together:

$$Throughput = n \times E(L_i)/(E(L_i)(t_{check} + t_{apply}) + 2(n-1)t_{setup} + (n-1)(t_{send} + t_{recv} + t_{apply}))$$

The throughput of the conventional one-phase protocol is:

$$Throughput = n/(nt_{apply} + 2(n-1)t_{setup} + (n-1)t_{send} + (n-1)t_{recv})$$

Figure 3 shows the throughput of our algorithms versus the one-phase protocol as a function of $E(L_i)$. The key "No WM" stands for error bounding algorithms without write merging, "WM" stands for error bounding algorithms with write merging, and "SC" stands for the strong consistency achieved by the one-phase protocol. We plot the graphs for $E(L_i) \in [1,5]$. By definition, $E(L_i)$ is greater than 1. Since our algorithms perform better as $E(L_i)$ increases, the graphs are conservative by not considering larger $E(L_i)$, which we expect to be common for many applications. As expected, the error bounding algorithms have considerably higher throughput than a strong consistency protocol by reducing wide-area communication. As $E(L_i)$ increases, the throughput of the error bounding algorithms also increases, and in the write merging case, the increase is almost linear. The performance improvement, however, does not come without cost. Larger $E(L_i)$ can sometimes only be gained by tolerating larger numerical error (see Section 5.2). Thus, the gains available to a network service depends upon the magnitude of the numerical error it is willing to tolerate.

Having compared the throughput, we now discuss the latency of the system. For numerical error bounding algorithms, the latency with no write merging is:

$$Latency = E(Q_i) + t_{check} + t_{apply} + (n-1)t_{send} + ((n-1)t_{setup} + t_{delay})/E(L_i)$$

Again, if we consider the possibility of writes merging and in the extreme case where all writes can be merged, the latency will be:

$$Latency = E(Q_i) + t_{check} + t_{apply} + ((n-1)t_{setup} + (n-1)t_{send} + t_{delay})/E(L_i)$$

The average latency in the one-phase protocol is:

$$Latency = E(Q_i) + (n-1)t_{setup} + (n-1)t_{send} + t_{delay} + t_{apply}$$

Figure 4 shows the latency of the error bounding algorithms versus the one-phase protocol as a function of $E(L_i)$. The keys have the same meaning as in Figure 3. We can see that the error bounding algorithms have

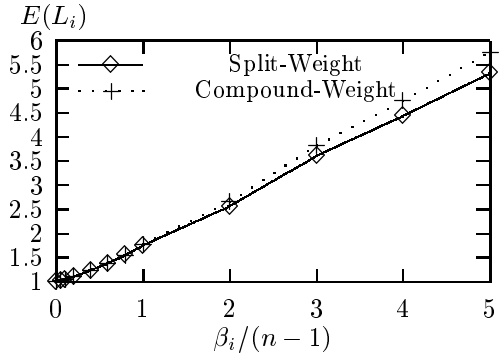

Figure 5: *Split-Weight AE vs. Compound-Weight AE (Weight ∼ U(-1, 3))*

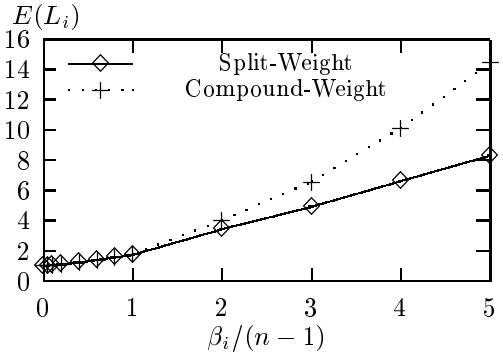

Figure 6: *Split-Weight AE vs. Compound-Weight AE (Weight ∼ U(-2, 2))*

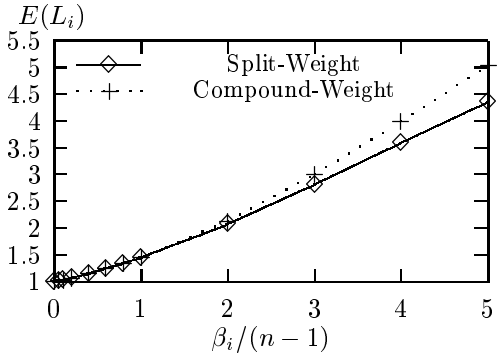

Figure 7: *Split-Weight AE vs. Compound-Weight AE (Weight ∼ N(1, 2))*

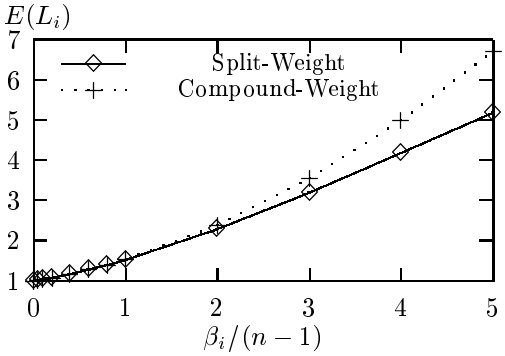

Figure 8: *Split-Weight AE vs. Compound-Weight AE (Weight ∼ N(0, 2))*

smaller latency than the strong consistency protocol. Furthermore, the latency in our algorithms decreases rapidly as $E(L_i)$ increases. However, as with throughput, the performance improvement can come at the cost of data accuracy.

5.2 Simulation Results

In this section, we use simulation to determine a range of typical values for $E(L_i)$ based on the distribution of the weight of individual writes. Although numerous factors affect $E(Q_i)$, it is determined by $E(L_i)$ to a large extent. Thus we believe studying $E(L_i)$ can give us insight into $E(Q_i)$ as well. The simulation results on $E(L_i)$ also quantify the performance difference between Split-Weight AE and Compound-Weight AE. As mentioned earlier, the latter is optimal while the former is better suited for space and computational optimizations. The resulting different $E(L_i)$ and $E(Q_i)$ values for the two algorithms directly affect system performance.

$E(L_i)$ is uniquely determined by the distribution of the weight of writes. We consider two different distributions for the weight: uniform distribution and normal distribution. The weight in the first workload is uniformly distributed within $[-1, 3]$, while those in the second are uniformly distributed within $[-2, 2]$. We denote the distributions by $U(-1, 3)$ and $U(-2, 2)$, respectively. For normal distribution, we consider weight conforming to $N(1, 2)$ and $N(0, 2)$. Each workload consists of one million writes and we measure the average epoch length as a function of $\beta_i/(n-1)$. The bound α_i is set to $-\beta_i$ in our experiments. Figures 5, 6, 7 and 8 summarize the simulation results.

The figures show that in most cases, $E(L_i)$ increases roughly linearly with $\beta_i/(n-1)$. For Compound-Weight AE in Figures 6 and 8, $E(L_i)$ increases faster than linearly. As we expect, $E(L_i)$ in Compound-Weight AE is always bigger than that in Split-Weight AE. The conservativeness of Split-Weight AE is quantified here by the difference between the two curves in each figure. The difference is not so obvious when the distribution is biased toward positive weight and becomes clearer when the distribution is symmetric. Note that we do not consider space overhead in the simulations and that the better performance of Split-Weight AE comes at the cost of increased space overhead.

6 Related Work

Alonso et. al.[3] propose four coherency conditions in the context of "quasi-copy" caching. One of the four conditions is "arithmetic condition," which specifies the allowed numerical error. Since in quasi-copy caching only the master database may accept updates,

maintaining arithmetic condition is a trivial problem. Relative to this effort, we define numerical error for replicated databases and discuss algorithms for bounding the error when updates are accepted by multiple replicas.

Bounding numerical error in a replicated database is closely related to maintaining integrity constraints in distributed databases. In [7], strong theoretical conclusions are made on how to decompose an arbitrary global constraint into a number of local constraints and communication constraints. The conclusions form the basis of the demarcation protocol[4], which applies a number of optimizations to the special case of linear arithmetic inequalities. Bounding numerical error is intrinsically enforcing an inequality. However, we exploit three special properties in this problem, which makes our algorithms practical and efficient for numerical error bounding. First, in the error bounding problem, the copies of a data item are inter-related. For example, if we want to bound the AE on $server_1$, it is not necessary to limit V_1. However, the demarcation protocol will have to put a limit on every variable present in the inequality. Second, in our algorithms, view advance is automatically incorporated and there is no need to explicitly re-adjust limits. On the other hand, the demarcation protocol does not exploit the fact that servers may have knowledge of what writes other servers have seen and limit re-adjustments are always done explicitly. Also, because the demarcation protocol cannot exploit the fact that copies are brought to consistency through write propagation, it is difficult to design efficient limit re-adjustment policies for it. The third property we utilize is that during a write propagation, all writes are propagated and all limits can be reset. This allows us to optimize the space overhead using hashtables. The demarcation protocol incurs $O(n)$ space overhead for each data item, where n is the total number of servers, limiting scalability. A direct performance comparison between our algorithms and the demarcation protocol would be difficult. Using the general limit re-negotiation policies discussed in [4] would result in poor performance and would be unfair to the demarcation protocol, while designing special policies for bounding numerical error is essentially as hard as designing numerical error bounding algorithms from scratch.

Gupta et.al. [13] describe an algorithm to verify a global constraint using only local information. In the case of a tuple insertion, the algorithm uses other "covering tuples" already in the tuple space to prove that the constraint is not affected by the new tuple. The technique cannot be applied to bounding numerical error, since no "covering tuple" can be obtained when users update a numerical data item.

Maintenance of materialized views[1, 8, 12] is closely related to our work. In fact, if the views are approximations of numerical base data, view maintenance can be an application of our error bounding algorithms. Various view maintenance algorithms have been proposed, see [12] for a survey on view maintenance. Relative to our work, view maintenance algorithms usually assume that only the base database can accept updates. In this aspect, our algorithms are more general than view maintenance algorithms.

In Section 4.2, we discussed how to efficiently check n conditions given a new write. This is a special case of how to efficiently check local integrity constraints given an update to the database. The general problem has been well studied[5, 6, 14]. However, most of the studies[6, 14] concentrate on how to filter those local constraints that are unaffected by the update. Others[5] only consider a particular class of local constraints and updates. Thus, none of the techniques is applicable to our case. The n conditions we intend to check are all linear conditions, making related techniques[2, 15, 11] developed in computation geometry also applicable. However, in our case, the linear conditions change frequently, making the cost of reconstructing the data structures [2, 11] outweigh the benefits.

7 Conclusion

In this paper, we argue for efficiently bounding numerical error to support replicated network services. Two algorithms, Split-Weight AE and Compound-Weight AE, are proposed to bound absolute error. They can be combined to achieve good performance and low space overhead. Inductive RE bounds relative error by transforming it into absolute error and applying Split-Weight/Compound-Weight AE. Exploiting the fact that V_j is an approximation of V_{final}, we are able to perform the transformation based on local information. We propose two optimizations to improve the scalability of the error bounding algorithms. Through performance analysis and simulation, we show that a replicated network service using our error bounding algorithms has superior performance in terms of latency and throughput compared to a network service using a traditional strong consistency protocol.

8 Acknowledgments

We thank Misha Rabinovich and the anonymous referees for their careful reviews of this paper.

References

[1] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. Database Support for Efficient Maintaining Derived Data. In *International Conference on Extending Database Technology*, 1996.

[2] Pankaj K. Agarwal, Lars Arge, Jeff Erickson Paolo G. Fanciosa, and Jeffrey Scott Vitter. Efficient Searching with Linear Constraints. In *Proceedings of the 17th*

ACM Symposium on Principles of Database Systems, 1998.

[3] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, September 1990.

[4] Daniel Barbara and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Linear Arithmetic Constraints in Distributed Database Systems. In *Proceedings of the International Conference on Extending Database Technology*, 1992.

[5] Philip Bernstein, Barbara Blaustein, and Edmund Clarke. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. In *Proceedings of the 6th Conference on Very Large Data Bases*, 1980.

[6] Peter O. Buneman and Eric K. Clemons. Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems*, September 1979.

[7] O.S.F. Carvalho and G. Roucairol. On the Distribution of an Assertion. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1982.

[8] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, and Inderpal Singh Mumick. Supporting Multiple View Maintenance Policies. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1997.

[9] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *International Journal of Supercomputer Applications*, volume 11(2), pages 115–128, 1997.

[10] Richard Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California, Santa Cruz, December 1992.

[11] Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft, and Jie-Bing Yu. Processing Queries by Linear Constraints. In *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing*, 1997.

[12] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, June 1995.

[13] Ashish Gupta and Jennifer Widom. Local Verification of Global Constraints in Distributed Databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.

[14] Robert Kowalshi, Fariba Sadri, and Paul Soper. Integrity Checking in Deductive Databases. In *Proceedings of the 13th Conference on Very Large Data Bases*, 1987.

[15] Norbert Beckmannand Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1990.

[16] Narayanan Krishnakumar and Arthur Bernstein. Bounded Ignorance in Replicated Systems. In *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, May 1991.

[17] Narayanan Krishnakumar and Arthur Bernstein. Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems*, 19(4), December 1994.

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, November 1992.

[19] Vivek Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.

[20] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[21] Calton Pu and Avraham Leff. Epsilon-Serializability. Technical Report CUCS-054-90, Columbia University, 1991.

[22] Calton Pu and Avraham Leff. Replication Control in Distributed Systems: An Asynchronous Approach. Technical Report CUCS-053-90, Columbia University, January 1991.

[23] D. Terry, K. Petersen, M. Spreitzer, and M. Theimer. The Case for Non-transparent Replication: Examples from Bayou. In *IEEE Data Engineering*, pages 12–20, December 1998.

[24] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services for Wide-Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems*, Chicago, Illinois, July 1998.

[25] David Wetherall. Active Network Vision and Reality: Lessions from a Capsule-based System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[26] Haifeng Yu and Amin Vahdat. Building Replicated Internet Services using TACT: A Toolkit for Tunable Availability and Consistency Tradeoffs. In *Second International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, June 2000.

[27] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. Technical Report CS-2000-07, Computer Science Department, Duke University, 2000. See `http://www.cs.duke.edu/~yhf/tr2.pdf`.

[28] Haifeng Yu and Amin Vahdat. Efficient Numerical Error Bounding for Replicated Network Services. Technical Report CS-2000-08, Computer Science Department, Duke University, 2000. See `http://www.cs.duke.edu/~yhf/vldbtr.pdf`.