Efficient Distributed Precision Control in Symmetric Replication Environments

Uğur Çetintemel
Department of Computer Science
Brown University
ugur@cs.brown.edu

Peter Keleher
Department of Computer Science
University of Maryland
keleher@cs.umd.edu

Abstract

Maintaining strict consistency of replicated data can be prohibitively expensive for many distributed applications and environments. In order to alleviate this problem, some systems allow applications to access stale, imprecise data. Due to relaxed correctness requirements, many applications can tolerate stale data but require that the imprecision be properly bounded.

This paper describes ReBound, a system that provides an adaptive framework for supporting and exploiting data precision vs. efficiency tradeoffs in symmetric replication environments via distributed precision control. Previous work proposed efficient precision control algorithms that support continuous read requests tagged with custom numerical precision ranges. ReBound generalizes and extends previous work with a new algorithm for continuous reads, support for ad-hoc reads, and light-weight adaptation mechanisms for coping with dynamically changing update patterns. This paper also presents preliminary experimental results, based on a prototype implementation, that demonstrate the performance advantages of exploiting precision vs. efficiency tradeoffs.

1. Introduction

Replication is crucial for effectively supporting distributed applications that involve data sharing. The cost of maintaining strict consistency of replicas may be impractical in many environments. On the other hand, many applications do not require strict consistency and can continue to operate with stale data, as long as the divergence from the accurate, up-to-date data is properly bounded. For such applications, it is possible to exploit the tradeoff between the precision of the replicated data (as observed by the applications), and overall system efficiency.

As a motivating scenario, consider a monitoring application that involves a distributed sensor network: sensor units transmit their readings to their closest base stations, which forward the values to the client application. As the sensor values are generated in real-time, it may be very expensive to transmit all the readings. At the same time, this may not be necessary because the monitoring application does not, in most cases, require exact values, but is only interested in certain trends. For instance, an application monitoring the average temperature value of a given region may tolerate an imprecision of ± 3 Fahrenheit. This flexibility can be effectively utilized by the base stations to reduce the volume of data transmitted to the monitoring clients. As another example, consider an inventory-based mobile sales application, where mobile sales people sell consumable or financial products using personal digital assistants with wireless interfaces. The product inventories accessed by the sales force are typically limited, and the products should not be oversold. It is, however, impractical or uneconomical to keep the sales people connected and, thus, up-to-date regarding the inventories at all times. On the other hand, a sales transaction does not need to have the accurate number of products available in the inventory as long as this number is larger than the number of items that is being sold at any instant. For instance, if a transaction involves selling three units of a particular product, the exact number of units left in the inventory does not matter as long as at least three units are still available for sale.

Both scenarios involve (1) numerical data that are replicated and updated at multiple network locations (which we refer to as a *symmetric* replication environment); (2) environments where maintaining strict data consistency is prohibitive due to large system scale, high volume of updates, or communication restrictions; and (3) applications that can tolerate bounded imprecision in the data they observe. In fact many other distributed applications and services—such as wide-area network management [16], commodity distribution [7], distributed load balancing [22], and airline reservation systems—demonstrate similar characteristics.

In this paper, we present the design, implementation, and evaluation of *ReBound*, an adaptable system that provides a flexible framework to support and exploit data precision vs. efficiency tradeoffs in *symmetric* replication environments. Figure 1 illustrates the basic ReBound system model. Multiple servers replicate and update numerical data items. Clients submit read operations that specify precision ranges on the data they cache (a client may be an individual node or a proxy style super-client that represents multiple nodes). These ranges indicate bounds

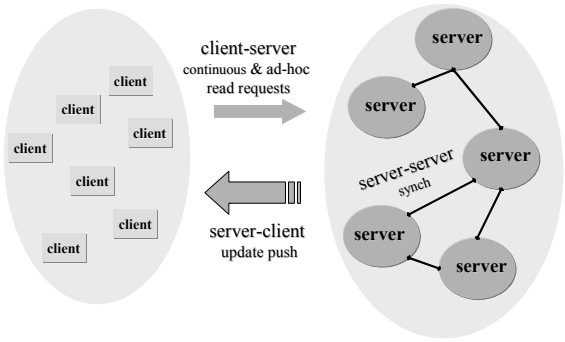

Figure 1: ReBound basic system model

on the quantitative deviation of the data read by the clients from the *accurate*, up-to-date values maintained at the servers.

ReBound supports two types of read requests: *continuous* and *ad-hoc*. Continuous reads require that the data cached at the clients always meet the specified precision constraints. Ad-hoc reads have one-time semantics and indicate a single refresh of the client cache such that the cached data meets the required precision bounds immediately after the refresh.

Recently, Yu and Vahdat proposed an efficient distributed precision control model [22] for symmetric replication environments in the context of the TACT project [21]. TACT's model supports static (i.e., fixed) continuous bounds by efficiently bounding the updates that are committed at servers and are *unknown* (i.e., not yet propagated) to clients.

In this paper, we essentially build on and extend the distributed precision control model presented in [22]. In particular, we propose two server-side algorithms that support both continuous and ad-hoc bounds. Similar to TACT's algorithm, our algorithms maintain continuous precision bounds by limiting update commitment at servers. Upon receiving an update, local commit criteria are used to decide whether to commit the update or not. If the criteria are met, the update is immediately committed. If not, the server must perform remote communication: it either pushes unknown updates to a proper subset of clients (this is the approach taken by TACT), or pulls information from a subset of servers. In our first algorithm, *Share-Bound*, the client-specified precision bounds are shared and cooperatively maintained by the servers. In our second algorithm, *Partition-Bound*, the precision bounds are explicitly partitioned across the servers. In fact, we show that Yu's algorithm is a specific instance of the Partition-Bound algorithm.

ReBound employs simple but practical adaptation mechanisms. Specifically, we use per-replica weights, which define the autonomy of the servers in terms of the volume of updates they can commit locally, and enable dynamic, pair-wise redistribution of these weights to cope with changing update patterns across servers. Our algorithms also address and handle ad-hoc bounds—ReBound exploits the already registered continuous precision bounds, if available, to efficiently select a proper subset of servers whose unknown updates need to be pushed to the clients to satisfy the specified bound.

In summary, we make the following contributions. First, we present an adaptable system for precision control of numerical data in symmetric replication environments. Our model generalizes and extends previous work by proposing decentralized precision control algorithms that efficiently maintain continuous and ad-hoc precision bounds. To the best of our knowledge, our protocols are the first to support ad-hoc reads with precision constraints in these environments. Furthermore, our protocols incorporate light-weight adaptation mechanisms that facilitate dynamic load balancing. Second, we present initial experimental results, based on the *ReBound* prototype, that demonstrate the practicality and the potential performance advantages of our protocols.

The rest of the paper is organized as follows. In Section 2, we give an overview of ReBound, describing its framework and system model. In Section 3, we describe our precision control algorithms in detail. In Section 4, we describe server-client and server-server synchronization in ReBound. In Section 5, we briefly describe the ReBound architecture. In Section 6, we describe the experimental environment and methodology, and present our experimental results. We briefly discuss scalability in Section 7, discuss related work in Section 8, and conclude in Section 9.

2. ReBound Overview

2.1. Framework

In ReBound, clients specify their desired quantitative precision requirements in terms of numerical ranges on the values of cached data items. Servers that replicate and update these items cooperatively maintain the specified ranges.

We first define the *value* of an update as the amount by which the update changes the value of the corresponding data item. We define a precision bound on a data item r with respect to a set of clients C and a set servers S as the sum of the values of updates, u, on r that the servers in S can commit without ensuring that all c in C has observed u. More formally, the algorithms we propose ensure that the following inequality hold at all times:

$$l \le \sum_{s_i \in S} value(U_i) \le h$$

where $l \le 0 \le h$ are, respectively, the lower and upper bounds to be preserved, and $value(U_i)$ is the sum of the values of the updates committed by $s_i \in S$ and not yet reflected to the caches of all clients c. Intuitively, the above constraint limits the total value of updates the servers can commit without refreshing client caches, essentially limit-

ing the numerical *im*precision of the values of the data cached at clients.

We define two types of precision constraints based on the types of read operations with which they are associated. A *continuous* precision constraint is one that is registered to servers only once and then continuously enforced. An *ad-hoc* constraint, on the other hand, is specified and enforced only once. Support for continuous ranges is typically sufficient for most applications that we target. Ad-hoc constraints are, however, desirable in scenarios where higher precision than that already provided by the continuous ranges is required (e.g., when the sensor monitoring application requires a more accurate temperature reading for a specific region).

2.2. System Model

Overview. Our basic system model is similar to that investigated in [21]. We assume that there are n servers and m clients in the system, and denote the set of servers and clients with S and C, respectively. Servers replicate and update numerical data items, whereas clients cache data items. An update u changes the value of an item by an amount equivalent to its value, *value*(u), which can be positive or negative. Clients cache read-only versions of a subset of these items, and can set *precision constraints* on the items they cache (we use the terms bounds, ranges, and constraints interchangeably in the rest of the paper). Servers that replicate an item cooperate to maintain any precision bounds defined on that item.

A server commits an update when it ensures that the update does not violate any client-specified constraints. The server that accepts an update u is called the *initiating* server of u. Updates are always propagated and maintained in the order they are committed at their respective initiating servers. A server reflects the updates it commits to other servers by propagating the corresponding update records, which are consequently applied to the remote databases (note that we do not propagate data item images). Servers can perform update propagation using any information propagation mechanism supported by the underlying communications environment. Because our framework is general, and is designed specifically for wide area, we assume *pair-wise*, epidemic style synchronization sessions [8, 18, 21] for update and control information propagation.

Data structures. For simplicity of exposition, we assume that a single data item that is replicated by all servers and cached by all the clients in the system. We also assume that servers and clients are assigned unique, global identifiers. In our model each server s_i maintains two views (which can be trivially combined in a single view): a *server view* and a *client view* that summarize the updates that other servers and clients have seen, respectively. We represent the server view of s_i as a vector v^i such that $sv^i[j,k]$ gives the number of updates committed by server k and known to server j (as far as s_i knows). Similarly, we represent the client view of s_i as a vector cv^i such that $cv^i[j,k]$ gives the number of updates committed by server k and known to client j (as far as s_i knows).

Views are updated when a server commits a new update locally, or during update propagation. When s_i commits a new update locally, it sets $sv^i[i,i] = sv^i[i,i] + 1$. When s_i propagates the updates unknown to client j, it sets $cv^i[j,k] = sv^i[i,k]$, $\forall s_k \in S$. When a server s_l propagates its server view, sv^l, to s_i, s_i updates its views such that $sv^i[j,k] = \max(sv^i[j,k], sv^l[j,k])$, $\forall s_j, s_k \in S$, and $cv^i[j,k] = \max(cv^i[j,k], cv^l[j,k])$, $\forall c_j \in C$, $\forall s_k \in S$.

Each server s_i maintains a *commit log*, which is a sequence of updates either committed by s_i, or committed by another server and propagated to s_i. We represent the sequence of updates committed by s_k in s_i's log with

$$U_k^i = < u_k^i[1], u_k^i[2], u_k^i[3], ..., u_k^i[v^i[i,k]] >$$

We say that an update u is *unknown* with respect to the client set C if u is not yet observed by all clients $c \in C$. We define the function $U^i(j,k)$ as the *sequence of unknown updates* committed by s_k as seen by node j in s_i's view. More formally,

$$U^i(j,k) = < u_k^i[x+1], u_k^i[x+2], u_k^i[x+3], ..., u_k^i[y] >$$

where $x = \min(cv^i[m,k])$, $\forall c_m \in C$, and $y = sv^i[j,k]$.

The value of a sequence of updates $U = <u_1, u_2, \ldots, u_n>$, *value*($U$), is the sum of the values of the updates in the sequence; i.e.,

$$value(U) = \sum_{i=1}^{n} value(u_i)$$

We now define the *minimum* and *maximum suffix* of a sequence U as, respectively, the suffix *sub*sequences of U,

$$\min(U) = < u_k, u_{k+1}, u_{k+2}, ..., u_n >, \text{and}$$
$$\max(U) = < u_l, u_{l+1}, u_{l+2}, ..., u_n >$$

with minimum and maximum values.

In addition, each server s_i maintains, for each registered constraint, lower and upper *bound weights*

$$0 \le w_i^-, w_i^+ \le 1.0$$

such that ,$\forall i$=1…n:

$$\sum_{i=1}^{n} w_i^- = \sum_{i=1}^{n} w_i^+ = 1.0$$

These weights are abstract measures of the autonomy of servers in committing updates: in general, the more weight a server holds, the more updates it can commit locally. An important system-wide invariant is that the sum of the weights of all servers adds up to a constant value. This invariant needs to be maintained at all times to ensure the correctness of the protocol.

3. Decentralized Precision Control

In this section, we describe two distributed algorithms for enforcing the continuous precision constraints as described above. Both algorithms work by efficiently limiting the sum of the values of updates that can be committed by the servers without synchronizing with the clients. The key insight to the algorithms is to treat the allowed

```
1.  While local_commit_criteria(r, u, [l, h]) not satisfied
2.    Push_to_clients(); and/or
        // push unknown updates to clients (to advance v^i)
3.    Pull-from-servers();
        // redistribute bounds (to relax local commit criteria)
4.  Set r = r + value(u);
```

Figure 2: Basic algorithm executed by a server to commit an update *u* on item *r*

precision bounds as global resources to be consumed by the servers, and distribute these resources across the servers. In the rest of the paper, we use the term *global range* to indicate the client-specified precision range.

In the first algorithm, *Partition-Bound* (*PB*), each global range is *partitioned* among servers, whereas in the second algorithm, *Share-Bound* (*SB*), each global precision range is *shared* among servers. Servers are assigned *per-bound weights* that indicate the size of their *share* in the global precision range. Given the global range, each server computes its *local range* based on its view of the system and its weight. A local range essentially indicates the sum of the values of updates that the server can commit locally, without having to synchronize with any other client or server.

In both algorithms, each server initially attempts to commit a new update based on local information. If this is not possible, the server must perform synchronization to advance its view or increase its weight (thereby expanding its local range). A server can advance its view by pushing unknown updates to clients or servers. A server can increase its weight through *pair-wise weight redistribution*, which involves one server pulling some amount of weight from another. While the pulling server's weight increases (expanding its local range), the pulled server's weight decreases by the *same amount* (shrinking its local range). This style of *sum-preserving* weight redistribution was first explored by the Deno shared-object system [4, 5, 6].

Figure 2 illustrates the basic procedure executed by a server s_i to commit a new update u. We now discuss two algorithms that primarily differ in the way they compute their local update commit criteria.

3.1. The Partition-Bound (*PB*) Algorithm

In this algorithm, the global precision range, $[l, h]$, is explicitly *partitioned* as local ranges across the servers using the local weights at each server. The local bounds at a server s_i are computed as:

$$l_i = w_i^- l \text{ and } h_i = w_i^- h$$

Note that the sum of the local bounds across the servers always sum up to the global bounds:

$$l = \sum_{i=1}^{n} l_i \text{ and } h = \sum_{i=1}^{n} h_i$$

The local bounds at each server s_i bound the sum of the values of the updates that s_i can commit without contacting other servers and refreshing client caches. *PB* does *not* require that a server to take into account the updates committed by other servers when computing local ranges. Since global ranges are partitioned and are thus independent, it is sufficient for each server to limit only its own updates to ensure that the global ranges are maintained properly.

Server s_i commits a new update u using the following *local commit criteria*:

$$value(\min(U_i^i)) + value(u) \geq l_i \quad if\ value(u) < 0$$

$$value(\max(U_i^i)) + value(u) \leq h_i, \quad if\ value(u) > 0$$

In other words, s_i can commit u if the sum of the values of the unknown updates committed by s_i (including u) does not exceed s_i's local bounds, and thus, does not invalidate the shared global bounds.

We note here that the TACT precision control algorithm [22] is a specific instance of the *PB* where all the weights are uniformly partitioned across servers (i.e., $w_i = 1/n$, $\forall i{=}1 \ldots n$).

3.2. The Share-Bound (*SB*) Algorithm

In this algorithm, the global precision range, $[l, h]$, is *shared* by the servers in the system. Upon accepting a new update u, a server s_i checks whether the commitment of u will violate the shared global range. For this purpose, s_i computes a local range, $[l_i, h_i]$, based on (1) the global range, (2) s_i's *local* knowledge about the updates committed by the other servers in the system, and (3) s_i's *weight*, which define s_i's *portion* in the global range. Since s_i's local knowledge may not be up-to-date, s_i conservatively computes its own local bounds by *assuming an upper bound on the ranges of other servers*. These local bounds then indicate the sum of the values of updates that s_i can commit entirely locally, i.e., without contacting any other server or client.

Each server s_i computes its local precision range as follows:

$$\begin{pmatrix} l_i \\ h_i \end{pmatrix} = \begin{pmatrix} w_i^- (l - \sum_{j \neq i} value(\min(U_j^i))) \\ w_i^+ (h - \sum_{j \neq i} value(\max(U_j^i))) \end{pmatrix}$$

Without loss of generality, consider the local upper bound as computed above. Intuitively, the value

$$h - \sum_{j \neq i} value(\max(U_j^i))$$

indicates the total maximum value of updates that can be committed without exceeding the global upper bound h (the sum of the values of updates committed by s_i is factored in the later). This is because the value

$$value(\max(U_j^i))$$

provides an upper bound on the value of the committed updates committed by s_j and potentially unknown to the corresponding client (according to s_i's view).

Server s_i then computes its share of this value by using its upper bound weight, and thus computing the total maximum value of updates that it can commit locally.

Fixing the sum of all weights in this manner, as we discuss in Section 4.2, enables light weight, server-server bound redistribution while maintaining the correctness of algorithm. The commit criteria used by *SB* is the same used by *PB* (described above).

4. Update Synchronization

4.1. Refreshing Client Caches

One way to accommodate a new update is to sufficiently advance the server's view of the clients regarding the updates unknown to those clients. This can be accomplished by refreshing the caches of a proper subset of clients by pushing a subset of the updates unknown to those clients. More specifically, server s_i chooses a subset of clients $C_i \subseteq C$ such that the propagation of unknown updates to each client in C_i will advance the view of s_i, and therefore potentially decrease U_j^i for some j=1…n.

The minimum quantitative view advance s_i requires, referred to as min_adv_i, in terms of the decrease in the sum of the values of the unknown updates required to commit a new update u can be computed as follows (based on the local commit criteria presented in Section 3.2 and Section 3.1).

For the Share-Bound algorithm:

If $value(u) < 0$ then:

$$\min\_adv_i = (w_i^- (l - \sum_{j \neq i} value(\min(U_j^i))) + value(\min(U_i^i)) + value(u)) - l_i$$

else:

$$\min\_adv_i = (w_i^+ (h - \sum_{j \neq i} value(\max(U_j^i))) + value(\max(U_i^i)) + value(u)) - h_i$$

The corresponding equations for the Partition-Bound algorithm are:

If $value(u) < 0$ then:

$$\min\_adv_i = (w_i^- (\min(U_i^i)) + value(u)) - l_i$$

else:

$$\min\_adv_i = (w_i^+ (\max(U_i^i)) + value(u)) - h_i$$

4.2. Bound Redistribution

An alternative to pushing updates to refresh client caches is to *relax local constraints by tightening remote constraints*. This is efficiently and practically accomplished by a pair-wise weight redistribution mechanism: a server that needs to increase its local weight, thereby relaxing its local bound, contacts other servers, and requests some amount of weight. The contacted server computes the amount it can give away and responds with that amount. In effect, the weights of the contacted servers, and therefore their bounds, are redistributed between the two servers. This operation is light weight in that *only* two servers are involved, and since the total amount of weight in the system remains fixed, correctness of the protocol is not affected (provided that the responding server computes the response amount properly, which we discuss below). This style of sum-preserving weight redistribution mechanisms was first explored in the context of the Deno system [5, 6], and was observed to exhibit very interesting dynamic properties [4].

Given an update u that cannot be committed locally at s_i, the minimum amount of extra weight for the upper bound, $min\_w_i^+$, s_i requires in order to commit u is (based on the commit criteria presented in Sections 3.2 and 3.1) is as follows: for Share-Bound:

$$\min\_w_i^+ = \frac{value(\max(U_i^i)) + value(u)}{h - \sum_{j \neq i} value(\max(U_j^i))} - w_i^+$$

For the Partition-Bound algorithm:

$$\min\_w_i^+ = \frac{w(\max(U_i^i)) + w(u)}{\delta^+} - w_i^+$$

The corresponding weights for the lower bounds can be computed similarly. Note that the required weight might be larger than 1.0 due to the already committed updates. Since the total weights held by all servers are fixed at 1.0, it may not be possible to commit the new update solely by weight redistribution. In such a case, client caches must be refreshed before committing new updates.

When a server s_i is contacted for bound redistribution, s_i computes the maximum lower and upper bound weights that it can give away, $extra\_w_i^-$ *and* $extra\_w_i^+$ such that $0 \leq extra\_w_i^- \leq w_i^-$, $0 \leq extra\_w_i^+ \leq w_i^+$, based on the commit criteria presented in Sections 3.2 and 3.1, as follows: For *Share-Bound*

$$\begin{pmatrix} extra\_w_i^- \\ extra\_w_i^+ \end{pmatrix} = \begin{pmatrix} w_i^- - \frac{value(\min(U_i^i))}{l - \sum_{j \neq i} value(\min(U_j^i))} \\ w_i^+ - \frac{value(\max(U_i^i))}{h - \sum_{j \neq i} value(\max(U_j^i))} \end{pmatrix}$$

For *Partition-Bound*, the corresponding values are:

$$\begin{pmatrix} extra\_w_i^- \\ extra\_w_i^+ \end{pmatrix} = \begin{pmatrix} w_i^- - \frac{value(\min(U_i^i))}{l} \\ w_i^+ - \frac{value(\max(U_i^i))}{h} \end{pmatrix}$$

As we can see from the above equations, s_i may give away *only* some of its weight if some of the updates it committed are still unknown to the clients; i.e., $value(U_i^i) \neq 0$. Intuitively, the reason is that the sum of the values of the unknown updates can be thought of as *consuming* some of the precision range available to the server, making the corresponding amount of weight unavailable. If this weight were to be given away and then consequently used by another server, the total amount of

1. Set $Q = \{s_i\}$
2. While *local_read_quorum_criteria* (r, c, $[l_o, h_o]$, Q) not satisfied
   i. Select a new quorum server $s_q \notin Q$
   ii. Pull from s_q those updates unknown to c
   iii. Set $Q = Q \cup \{s_q\}$
3. Push all updates unknown to c

Figure 3: Basic algorithm executed by server s_i to commit an ad-hoc read operation specified by a client c with precision bounds $[l_o, h_o]$ (on item r).

weight in the system used at any one time might potentially exceed the fixed 1.0 value, thereby eliminating any global precision bound guarantees.

4.3. Supporting Ad-Hoc Precision Bounds

In this section, we describe how our algorithms support one-time precision bounds specified by ad-hoc read operations. In both algorithms, the server that received the read, say s, *pulls* unknown updates from a sufficiently large *quorum* of servers, and *pushes* those updates to the client that issued the ad-hoc read. More specifically, s needs to form a read quorum, $Q \subseteq S$, such that the remaining set of *non*-quorum servers, $NQ = S - Q$, cannot commit updates whose sum of weights will invalidate the limit set by the client. In such a case, s does not need to expand Q anymore by pulling from non-quorum servers: the ad-hoc bounds are then satisfied when s pushes those updates committed by the servers in Q as a response to the client's read request. Figure 3 depicts the basic quorum formation algorithm executed by a server s_i to commit an ad-hoc read submitted by client c.

Assume that a client c wants to read an item r with a precision of $[l_o, h_o]$ (the subscript 'o' implies a ad-hoc bound). Two scenarios are possible: (1) there are no registered precision bounds on r; or (2) there already exist registered continuous precision bounds, $[l, h]$, on r. In the former case, the server s that received the read request has no option but to contact all servers (essentially forming a read quorum $Q = S$), and pull all updates unknown to c. This is necessary because no continuous bounds are maintained in the system and, thus, there is virtually no limit on the sum of the values of updates that can be committed locally by any server.

In the latter case, s can exploit the fact that a continuous bound is being maintained in the system to more efficiently execute the read operation. Note that the case where the ad-hoc bounds are more relaxed than the corresponding continuous bounds (i.e., $[l, h] \subseteq [l_o, h_o]$) is trivial: client c can simply complete the read using the data on its cache because the data is already guaranteed to satisfy the precision bounds.

Server s_i decides that the read quorum, $Q \subseteq S$, is sufficiently large to guarantee the ad-hoc bound, $[l_o, h_o]$, given the already registered continuous bounds, $[l, h]$, if the following local conditions are satisfied:

For Share-Bound:

$$\sum_{k \notin Q} w_k^- (l - \sum_{q \in Q} value(\min(U_{k,q}^i))) - \sum_{k \notin Q} value(\min(U_{i,k}^i)) \geq l_o$$

$$\sum_{k \notin Q} w_k^+ [h - \sum_{q \in Q} value(\max(U_{k,q}^i))] - \sum_{k \notin Q} value(\max(U_{i,k}^i)) \leq h_o$$

Intuitively, the left side of each condition computes, based on s_i's view, the sum of the values of the updates that can be committed by a *non*-quorum server s_k, summed over all non-quorum servers $s_k \notin Q$. For each such s_k, the formula computes the local bound of s_k (indicating the total value of updates that can be committed by s_k) less the sum of values of updates that s_k already committed. The result then gives, for each $s_k \notin Q$, the total value of updates that can further be committed by s_k. Notice that the sum

$$\sum_{k \notin Q} w_k^+$$

in the formula (consider the upper bound case) cannot be directly computed by s_i, since, by definition, s_k is not a quorum server and its weight may not be available to s_i. However, since the sum of weights across all servers is fixed to 1.0, s_i can compute the sum indirectly using the weights of quorum servers as: $\sum_{k \notin Q} w_k^+ = 1.0 - \sum_{q \in Q} w_q^+$.

The corresponding conditions for Partition-Bound are:

$$\sum_i (l_i - value(\min(U_i^i)) + \sum_i value(U_i^i) \geq l \quad \text{and}$$

$$\sum_i (h_i - value(\max(U_i^i)) + \sum_i value(U_i^i) \leq h$$

5. ReBound Architecture

This section briefly describes the basic components of a ReBound server. The *Server Manager* is in charge of coordinating the activities of the various components and implementing the basic server API that accepts updates and continuous/one-time reads to data items it maintains. The *Precision Controller* implements the precision control algorithms used by ReBound. In particular, it maintains a *bound list* that contains the precision ranges registered at the server, and a server *view* that compactly summarizes the committed updates propagated in the system. The *Policy Manager* is responsible for implementing efficient divergence control policies. This component implements different synchronization policies that specify when, with whom, and what data to pull or push. The *Update Manager* handles the local execution of updates. It maintains an update queue that contains all active (initiated but not-yet-committed) updates. The *Storage Manager* provides access to the *object store* that stores the committed versions of all replicated items. The object store is currently implemented as an in-memory database.

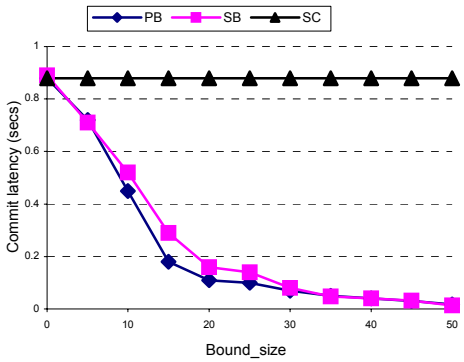

Figure 4: Commit latency vs. continuous bound size (UR=0.2, RR=0.0)

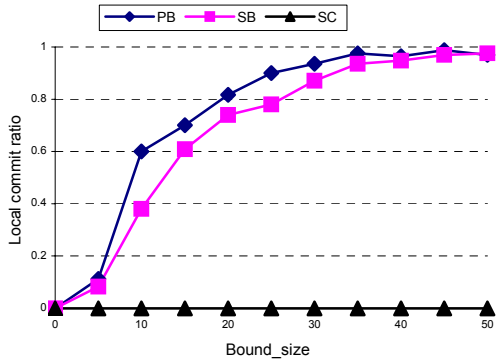

Figure 5: Local commit ratio vs. continuous bound size (*UR*=0.2, *RR*=0.0)

The current ReBound prototype runs on top of Linux and Windows32 platforms.

6. Performance Evaluation

6.1. Environment and Methodology

We now present our performance evaluation that aims to provide a basic understanding of the relative performance characteristics of different protocols and demonstrate potential performance advantages of using them. Even though our protocols are specifically designed for wide-area environments, we conducted our preliminary experiments on a local area network to get repeatable results. In the experiments, we used a cluster of 10 Linux machines, each having two 400 MHz Pentium II's, and 256 MBytes of memory. The machines are connected via a 100Mbps Ethernet network and communication is performed on top of UDP/IP. We artificially injected a 100 milliseconds one-way latency to each outgoing message in order to emulate typical communication latencies over wide area.

In the experiments we present here, we assume that the database consists of a single data item, and that the clients registered a single continuous precision bound to the servers. Each server independently initiates updates based on a uniform *update rate*. We assume that each update has unit value. Each client independently initiates ad-hoc reads based on a uniform *read rate*, and submits the read to a randomly selected server. The ad-hoc bounds are also assumed to be equal (i.e., $l_o = h_o$), and we use the variable *bound ratio* to define the ratio of the ad-hoc bounds to the continuous bound (i.e., h_o/h). The main experimental parameters and settings are shown in Table 1. We note that under these settings, without any bound redistribution, *PB* basically emulates Yu's algorithm [22].

The primary performance metrics we used are: (1) *commit latency*, which indicates the time between the initiation of an update or ad-hoc read and the time to commit; (2) *local commit ratio*, which indicates the ratio of all committed updates that are committed at their initiating server without the need for any synchronization; and (3) *read quorum size*, which indicates the number of servers contacted to satisfy the bounds specified by an ad-hoc read (including the server that initially received the read request).

In the experiments, we employed a simple policy that uses compulsory server-client push, which is initiated only if an update or an ad-hoc read cannot be committed locally. This choice makes sense in our setting because all the messaging latencies are the same, and the push operation is guaranteed to advance the server's view, whereas the server-server pull is not. Our policy does not make pro-active push/pull decisions (i.e., background update propagation and view advance), which potentially would improve the performance of our algorithms significantly. Similarly, servers to be added to a read quorum (when handling ad-hoc queries) are chosen randomly. The numbers we present below are the averaged results of ten independent runs of executing 500 updates/reads in the system.

6.2. Enforcing Continuous Precision Bounds

Figure 4 shows the commit latency results of our algorithms and a hypothetical write-all type strict consistency protocol, labeled *SC*, as a function of bound size. The *SC* protocol is a conventional write-all [2] style protocol that pushes an update to all clients prior to committing the

Notation	Description	Setting
UR	Mean global update generation rate	0.1,0.2 updates/s (uniform)
RR	Mean global read generation rate	0.1 reads/s (uniform)
$\|S\|$	Number of servers	5
$\|C\|$	Number of clients	5
msg_latency	One-way message latency	100ms
bound_size	Continuous bound size: *l* or *h* (both set equal)	[0, 50]
bound_ratio	Ratio of ad-hoc bound size to continuous bound	[0, 1]

Table 1: Primary experimental parameters and settings

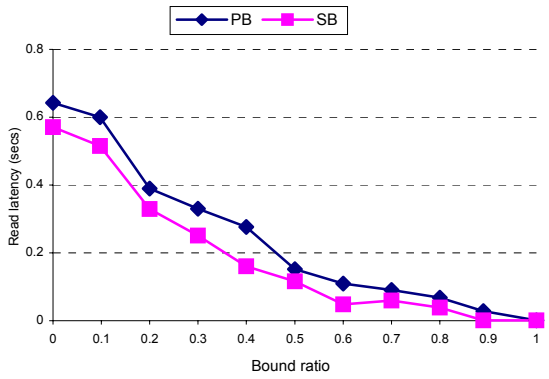

Figure 6: Commit latency vs. ad-hoc bound ratio (*UR*=0.1, *RR*=0.1, *bound_size*=10)

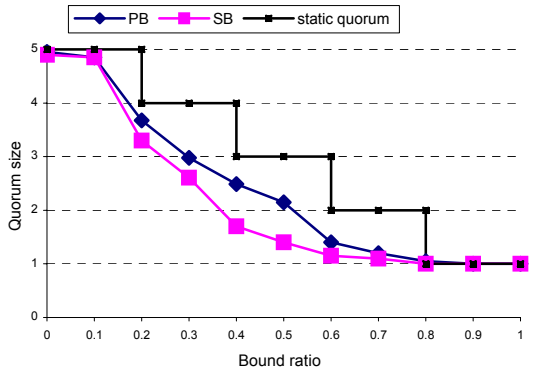

Figure 7: Read quorum size vs. ad-hoc bound ratio (*UR*=0.1, *RR*=0.1, *bound_size*=10)

update. Note that our algorithms demonstrate write-all behavior when the local bounds at each server are sufficiently small (i.e., smaller than the value of an update). This is the reason why the three algorithms converge at a bound size of five, below which none of the servers can locally commit any updates. Our precision control algorithms have lower latency than the strict consistency protocol, and the gap increases significantly with increasing bound size. Clearly, the improvement comes at the expense of data precision. The figure also reveals that *PB* achieves lower latency values than *SB*, especially for relatively small to moderate bound sizes. The difference between the *PB* and *SB* curves quantifies the conservativeness of *SB*'s local commit criteria relative to that of *PB*.

Figure 5 provides further insight by plotting the corresponding local commit ratio curves. It is evident that *SC* cannot commit any updates locally as it requires pushing updates to all clients first. *SP* and *PB*, on the other hand, can commit updates locally most of the time, and local commit ratio increases with increasing bound size. PB commits more updates locally than SB does due to its less conservative local commit criteria, which essentially enables higher server autonomy.

6.3. Enforcing Ad-Hoc Precision Bounds

We now investigate reads that specify ad-hoc precision bounds. Figure 6 presents the commit latency results for increasing ad-hoc bound ratios. For purposes of this experiment, we define *bound ratio* to be the ratio of the ad-hoc bound to the continuous bound as $l_o/l = h_o/h$, where $[l, h]$ is the continuous bound already registered to the system. As discussed earlier, if no continuous bound are registered, the server that accepted the read request needs to pull the unknown updates from all other servers (i.e., form a read quorum containing all servers).

As expected, the latency increases for both algorithms as the bound ratio decreases (i.e., as the read requires higher precision). Unlike the results presented in Section 6.2, the bound sharing algorithm consistently achieves lower latency than the bound partitioning algorithm. In fact, the conservativeness of *SB*'s local commit criteria, which has negative affect on performance when supporting continuous bounds, help *SB* in this case as the non-quorum servers also use the same conservative criteria to compute their local ranges. Since *PB*'s criteria is less conservative, the sum of the values of the unknown updates that can be committed by the non-quorum servers can be more than that in *SB*, requiring more servers to be contacted and included in the read quorum for the precision bound to be ensured.

Figure 7 shows the local commit ratio results for *PB*, *SB*, and a hypothetical variant of our bound partitioning algorithm, *static quorum*. Static-quorum is similar to *PB* in that the global registered bounds are partitioned across all servers, but differ in that servers do not utilize information regarding the updates seen from other servers. We observe that *SB* commits more updates locally than *PB*, due to its more conservative commit criteria. The difference between the curves for *PB* and *static quorum* quantifies the benefits of exploiting views in this case.

6.4. Adaptation Experiments

We conducted several other experiments that investigate the potential performance improvements attainable through adaptation, which is efficiently enabled through our server-server weight redistribution mechanism. The results of the experiments (which we do not present here due to space considerations) reveal that the system can effectively adapt to changing factors such as update-rate distribution, and that adaptation has the potential to yield significant performance gains in real applications (see [3] for complete results).

7. Scalability Issues

The basic system model (described in Section 2.2) assumes that each server maintains views for all clients that register precision bounds. This requires $O(n \times m)$storage (per data item), where n is the number of servers and m is the number of clients. Clearly, this is not a practical approach as we expect to support a large number of clients

and a large number of data items. The issue here is not only storage and synchronization of these potentially large views, which we address below, but also the communication required to keep track of and refresh each individual client. In order to effectively scale, we propose a proxy-based grouping approach. In this approach, proxy servers sit between the clients and servers and act as intermediate *smart* caches. Clients submit their requests to the closest proxy, which then becomes responsible for executing the requests by communicating with the servers. In this model, the servers need to keep track of, communicate with, and refresh the caches of the proxies only (each proxy essentially becomes a virtual *superclient*). It is also possible to organize proxies into hierarchies to further improve scalability. This model, however, is outside the scope of this paper and we plan to investigate it as part of our future work (see Section 9).

It is also possible to use various techniques to reduce the server-side view storage and synchronization overheads at the expense of some accuracy in the representation of precision bounds. One such technique might be to logically group multiple client bounds into a single one, and then represent the entire group's bounds with the tightest bounds in the group. Yu and Vahdat investigated a similar and complementary approach where all clients are enforced to use the same precision bounds for the same data items [22].

8. Related Work

There has been significant research on maintaining consistency constraints on numerical replicated data. Early work, such as the *demarcation protocol* [1], typically addressed the general problem of maintaining integrity constraints in traditional distributed database environments. The demarcation protocol is not designed to support fine-grained, continuous or ad-hoc reads with precision bounds. *Epsilon-serializability* [11, 20] is a generalization of conventional serializability [2], where a limited amount of inconsistency is allowed by multiple reads in a query. Similar to the demarcation protocol, epsilon-serializability addresses a much more general problem, and thus cannot exploit many features specific to our problem. The protocols proposed for the demarcation protocol and epsilon-serializability, being designed for general transaction processing, heavily rely on locking-based techniques, which are prohibitive for the types of environments and applications we address.

Bounded ignorance [13], *precision caching* [12], and several materialized view maintenance algorithms (e.g., [10]) have addressed numerical error bounding in a master-copy model, where only a single node accepts updates to data items. Olston and Widom [16] recently proposed tunable algorithms that provide precision vs. performance tradeoff for aggregation queries over replicated data. This work also does not consider symmetric replication environments.

The most recent and relevant work on distributed precision control that addressed symmetric replication environments is that of Yu and Vahdat [22], which described an efficient algorithm for numerical error bounding for replicated network services. The basic model we present here is essentially based on Yu's model, but generalizes and extends it in the following ways: First, Yu's algorithm addresses only continuous bounds, but does not addresses ad-hoc, per-read bounds. Second, Yu's algorithm is partitioning-based: it can actually be regarded as a specific instance of *PB* where the bounds are statically and uniformly partitioned across servers at startup time. Finally, Yu's algorithm does not address dynamic bound redistribution, which is a crucial mechanism for adapting to dynamically changing update patterns.

Also relevant are *escrow*-based [17] protocols proposed for efficient distributed resource management (e.g., [9, 14, 15, 19]. These protocols are typically based on global state snapshot algorithms [15] or quorum locking techniques [14], restricting their efficiency and practicality in wide-area environments. Of particular relevance is the *Data-value partitioning* approach [19], which basically partitions the numerical values of database items and stores each of the constituent values at different servers.

9. Conclusions and Future Work

We presented the design and implementation of ReBound, a system that provides a practical, flexible framework for efficiently supporting distributed applications that access numerical replicated data and that can tolerate bounded data imprecision. Our work generalizes and extends previous work that proposed efficient algorithms for bounding imprecision in replicated network services. In particular, ReBound provides support for not only continuous bounds but also ad hoc bounds. Furthermore, ReBound incorporates practical, low-overhead adaptation mechanisms can be used to dynamically and asynchronously adjust to the update patterns as observed by the servers in the system.

We described two algorithms, Share-Bound and Partition-Bound, which maintain continuous precision bounds by limiting the sum of the values of the committed updates that are unknown to clients. In Share-Bound, the registered bounds are *shared* among servers, whereas in Partition-Bound, the registered bounds are *partitioned* across servers. For both algorithms, we presented local update commit criteria that, if satisfied, ensure that the registered bounds will still be met after the commitment of an update. If a server's local update commit criteria are not satisfied, then it has to synchronize with other servers or clients. The algorithms handle ad-hoc bounds by exploiting the already registered continuous precision

bounds, if available, to efficiently select a proper subset of servers whose unknown updates need to be pushed to the clients. We also described light-weight adaptation mechanisms based on pair-wise weight distribution among servers.

Using the ReBound prototype, we presented a preliminary performance evaluation of our algorithms under various workloads and scenarios. The results revealed that ReBound could effectively lower the precision of data delivered to the clients for significantly improved system performance.

In this paper, we focused only on efficient precision-control mechanisms, and did not address policy issues. As future work, we are planning to investigate dynamic precision control policies that provide answers to the following questions: when to contact other servers?; which servers to contact?; and what updates to push or pull? Scalability might become an important issue especially when supporting a large client population.

As discussed in Section 7, we are currently investigating proxy-based hierarchical organizations to effectively scale up to large number of clients and data items. In this model, proxies are responsible for executing the requests of their client set by communicating with the servers. We are exploring policies for setting appropriate *aggregate* proxy precision bounds in order to minimize the overall necessary proxy-server pull and server-proxy push communication. If the proxy precision bounds are set too loose, then the proxy would have to make frequent ad-hoc reads. If the bounds are set too tight, on the other hand, then the servers would have to frequently refresh the proxy. Adaptive precision bound setting in the presence of changing update patterns and precision requirements, thus, seem to be an interesting research direction.

References

[1] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Linear Arithmetic Constraints in Distributed Database Systems. In *Proc. of the Intl. Conf. on EDBT*, 1992.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*: Addison-Wesley, 1987.

[3] U. Cetintemel. Decentralized Data Consistency Protocols for Mobile and Wide-Area Environments. *Department of Computer Science*.

[4] U. Cetintemel and P. J. Keleher. Light-Weight Currency Management Mechanisms in Deno. In *Proc. 10th IEEE Workshop on Research Issues in Data Engineering (RIDE)*, San Diego, February 2000.

[5] U. Cetintemel and P. J. Keleher. Performance of Mobile, Single-Object Replication Protocols. In *proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nurnberg, Germany, 2000.

[6] U. Cetintemel, P. J. Keleher, and M. J. Franklin. Support for Speculative Update Propagation and Mobility in Deno. In *IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, Phoenix, 2001.

[7] U. Cetintemel, B. Özden, M. J. Franklin, and A. Silberschatz. Design and Evaluation of Redistribution Strategies for Wide-Area Commodity Distribution. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, Mesa, Arizona, 2001.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. 6th ACM Symp. on Principles of Distributed Computing (PODC)*, Vancouver, 1987.

[9] L. Golubchik and A. Thomasian. Token Allocation in Distributed Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS-12)*, Yokohama, Japan, June 1992.

[10] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, 1995.

[11] W. Hsueush, G. E. Kaiser, C. Pu, K.-L. Wu, and P. S. Yu. Divergence Control for Distributed Database Systems. *Distributed and Parallel Databases*, 3(1):85-109, 1995.

[12] Y. Huang, R. H. Sloan, and O. Wolfson. Divergence Caching in Client Server Architectures. In *Proc. Int. Conf. Parallel and Distributed Information Systems (PDIS)*, 1994.

[13] N. Krishnakumar and A. Bernstein. Bounded Ignorance in Replicated Systems. In *Proc. 10th ACM Symp. on Principles of Database Systems (PODS)*, 1991.

[14] N. Krishnakumar and A. J. Bernstein. High Throughput Escrow Algorithms for Replicated Databases. In *Proc. of the Int. Conf. on Very Large Databases*, Vancouver, Canada, 1992.

[15] A. Kumar and M. Stonebraker. Semantics Based Transaction Management Techniques for Replicated Data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Chicago, USA, June 1988.

[16] C. Olston and J. Widom. Offering a Precision-Performance Tradeoff for Aggregation Queries over Replicated Data. In *Proc. of the 26th VLDB Conf.*, Cairo, Egypt, 2000.

[17] P. E. O'Neil. The Escrow Transactional Method. *ACM Transactions on Database Systems*, 11(4):405-430, 1986.

[18] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *16th ACM Symposium on Operating System Principles*, Saint-Milo France, October 1997.

[19] N. Soparkar and A. Silberschatz. Data-value Partitioning and Virtual Messages. In *Proc. of the Symposium on Principles of Database Systems*, Nashville, 1990.

[20] K.-L. Wu, P. S. Yu, and C. Pu. Divergence Control for Epsilon-Serializability. In *Proc. of Int. Conf. on Data Engineering*, Tempe, USA, 1992.

[21] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[22] H. Yu and A. Vahdat. Efficient Numerical Error Bounding for Replicated Network Services. In *Proc. of the 26th VLDB Conf.*, Cairo, Egypt, 2000.