Trading Replication Consistency for Performance and Availability: an Adaptive Approach

Chi Zhang * Zheng Zhang†

Abstract

Replication system is one of the most fundamental building blocks of wide-area applications. Due to the inevitable dependencies on wide-area communication, trade-off between performance, availability and replication consistency is often a necessity. While a number of proposals have been made to provide a tunable consistency bound between strong and weak extremes, many of them rely on a statically specified enforcement across replicas. This approach, while easy to implement, neglects the dynamic contexts within which replicas are operating, delivering sub-optimal performance and/or system availability.

In this paper, we analyze the problem of optimal performance/availability for a given consistency level under heterogeneous workload and network condition. Our analysis is based on a consistency model of update window, which is flexible enough to express a number of popular replica control schemes. We prove several optimization rules for different policies. Based on these models, we developed an adaptive update window protocol in which consistency enforcement across replicas is self-tuned to achieve optimal configuration. A prototype system, FRACS, is built and evaluated in this paper. The experiment results demonstrate significant advantages of adaptation over static approach for a variety of workloads.

Keywords: Replication, Bounded Consistency, Performance, Availability, Adaptation

*Department of Computer Science, Princeton University
†Microsoft Research Asia. (Work partially done in HP-Labs)

1 Introduction

Wide-area applications such as remote collaboration, e-commerce, and content delivery network face the challenges on performance and availability from the geographically distributed nature of such applications. Replication relieves this barrier by eliminating expensive remote accesses as well as single failure points. Although strong consistency as ensured by traditional *one-copy serializability* provides most desirable correctness guarantee, it becomes prohibitively expensive across wide-area networks. Optimistic concurrency control [6] lifts the performance barrier by allowing concurrent and potential inconsistent accesses. Bayou [14, 11] further relaxes to an update-anywhere-anytime-anyway consistency model, with the only guarantee of eventually consistent state on replicas.

The "gray zone" between eventual consistency and the strong consistency bound has been an active research area [10, 5, 12, 13, 15, 9, 18]. Those systems introduce various bounds on consistency relaxations: time-based [13, 15] which is appropriate for keeping the freshness of web documents; non-sequential order for better concurrency [10, 9]; multiple-tier consistency [5] that differentiates different clients; or comprehensive approach like [18] that carries multiple higher level semantic meanings.

However, in a number of proposals, the consistency is uniformly relaxed on all replicas without differentiation. Even when some systems allow individually tunable consistency, there lacks an easy way to decide a "correct" configuration for a given "context" in which the replicas are operating. As far as a replication system is concerned, such context includes, but is not limited to, the update activities on each replica, as well as their resource constraints (most predominantly the characteristics of their network connectivity such as latency and bandwidth). Intuitively, busy replicas need looser consistency to reduce overhead. Replicas on remote slow network also cry for relaxation because they can not afford frequent communications. These observations lead us to develop a replication system with adaptive consistency. In this model, user specifies desired consistency level, and replicas adjust their share of inconsistency bound according to their activities and connectivities.

Specifically, we make the following contributions in this paper:

- We propose a quantitative model to describe the consistency in a replicated system. We prove that performance (latency or number of messages) can be optimized with certain configuration of model parameters. Likewise, a similar result can be achieved if total system availability is the optimization target. However, the optimal configuration always depends on what the goal is.
- We develop a distributed algorithm to adjust configuration when adaptation is necessary. The algorithm is light-weight. It requires no additional message passing, and converges without any centralized or group agreement protocols.

Our consistency model is expressive enough to accommodate a number of popular replication control protocols. Such versatility also enabled us to use the same set of mechanisms to control the meta-data of the system, for instance, replica membership and dynamic consistency bound.

The consistency models and adaptation algorithm are implemented in FRACS, a replication system that combines anti-entropy communication protocol with adaptive consistency control. Our experiments show significant benefits of adaptive consistency, as well as some cases where it is less effective.

The remainder of this paper is organized as follows. Section 2 presents the analytical model of window-based consistency bound. Section 3 discusses algorithms for adaptation of model configuration. Section 4 describes the prototype FRACS system. The experimental results in Section 5 evaluate the effectiveness of adaptation under different access patterns. Discussion of related work is offered in Section 6. And we conclude in Section 7.

2 Analytical Models

Among the many goals one may expect from a replication system, performance, consistency and availability are usually found in a subtle tension towards each other. Our approach has been to pick

up a consistency model generic enough to express a range of replication models, build an analytical model to gain enough insight, and then try to configure the system to optimize the performance and availability while keeping a certain consistency promise.

2.1 Bounding obsoleteness

Consistency is highly dependent on application semantics. For example, the loose consistency guarantee provided by NFS is usually strong enough for users in LAN, but hardly acceptable for a database management system. So our first step is to quantitatively model the consistency metrics. Our bottom-line is eventual consistent states on all replicas. Even when this is guaranteed, inconsistency can occur during the transient state when update issued in one replica has not yet reached all other replicas, either because of buffering or communication delay.

Consider the case of External Consistency, where users can communicate with each other via channels outside the replication system. When a user makes an update and notifies another one of the event, this update may be still unavailable in the receiver's replica. In another case, some systems give a "safe" view to application, i.e. no tentative data is exposed to read requests until they get committed. The commonality of both is that read operation sees an "obsolete" view where some amount of recent updates can be missing. We quantify such kind of inconsistency into "obsoleteness", or error of local data view. The measurement can be as simple as the number of missing updates, or more complex schemes that assign different weights to different update operations. We use the former for its simplicity.

The consistency in the system is controlled by limiting the amount of inconsistency in each replica, i.e. the uncommitted updates. Each $replica_i$ is assigned an update window of W_i that allows it to buffer up to W_i tentative updates. When a replica has accumulated W_i updates locally, it has to push out the tentative updates to other replicas. After everyone gets the updates, they are committed. During this time, the user's new update requests are blocked until tentative updates are committed and removed from update window. Read to local replica is always allowed since it does not change system state. We call this an update window system.

A number of popular consistency constraints can be expressed using different configurations of such update window system. Take Jim Gray's models [5] for example:

- Master-eager: this corresponds to $W_i = 1$, and $\forall j \neq i, W_j = 0$. So there is one single master that can make update. It pushes eagerly to all slaves.
- Master-lazy: this is $W_i = \infty$, and $\forall j \neq i, W_j = 0$. The master can buffer updates and push to slaves later.
- Group-eager: $\forall i, W_i = 1$. Everyone makes updates and pushes immediately to others.
- Group-lazy: $\forall i, W_i = \infty$. No consistency guarantee at all.

Furthermore, a host of other more complex requirements can also be described:

- Floating master: a useful variation of Master-lazy where the master can be changed dynamically. Such case is particularly useful for mobile applications where the user moves from one replica to another.
- Two-tier [5]: here $\sum(W_i) = M$, where i belong to the first tier of replicas, and $W_i = 0$ if i belongs to the read-only tier
- Bounded error: when we place a constraint to the window configuration, a certain consistency guarantee can be enforced. The obsoleteness bound can be expressed as $\sum W_i \leq Bound_{obsoleteness}$. A straight-forward enforcement of such a bound is to divide the total bound evenly across all replicas:

$$W_i = \frac{Bound_{obsoleteness}}{N}, 1 \leq i \leq N$$

  where N is the total number of replicas.

  More general and flexible configurations can be constructed in similar ways.

Though quite general, the above model is not easy to deploy because there are a lot of knobs for users to tune. In a lot of cases, users do not care the consistency levels in specific replicas. They

want an overall acceptable consistency level achieved with minimal overhead. In the remaining parts of this paper, we will discuss how to specify the window configuration to achieve optimization goals under consistency constraints.

2.2 Optimizing cost with square-root distribution rule

Our overall consistency constraint is the total obsoleteness, defined as the sum of update windows of all replicas:

One straightforward way of enforcing the constraint would be to split the total window bound evenly to N replicas. However, in a lot of scenarios this makes little sense. For example, consider a bursty replication system: replicas are silent in most of time; once a while, they wake up and do some operations. Allocating a constant window size to everyone is awkward, you either give out too much window that system experience high obsoleteness when they become active together; or get very poor performance in every replica when a small window share is given to everyone to bound worst-case obsoleteness. Similar case can be made for scenarios where activities among replicas are skewed. Not only the application behavior affects the decision of window configuration, network connectivity will make a difference too. Replicas on slow network need bigger window to keep up with other faster ones.

In summary, our goal is to derive a window configuration that minimizes "cost" or maximizes "benefit" in certain operation "context". The context includes, but not limited to the update activity of replicas, network latency and bandwidth. The cost and benefit can be anything interesting to the user. Here we will discuss the number of messages, the latency added to update process, and the availability.

Let's first define cost as the number of messages used to propagate updates. This is usually more meaningful than the number of bytes since the updates are often quite small. When the number of uncommitted updates on $replica_i$ is below the window W_i, $replica_i$ can buffer writes without sending to others. Once update window of W_i is full, $replica_i$ sends all buffered updates to all replicas. Thus the update window reduces messages from $replica_i$ by W_i times. Note, this is

assuming a "lazy" propagation scheme where everyone buffers updates to the maximal limit.

$$MessageRate_i = \frac{(N-1)*A_i}{W_i}$$

$$MessageRate_{total} = \sum_{1 \le i \le N} \frac{(N-1)*A_i}{W_i}$$

In above formulas, A_i is the update rate on $replica_i$, how many updates are generated per unit time; and N is the number of replicas. We can minimize the total message rate with the following theorem.

Theorem 1 *Square-Root Distribution Rule*

Given constraint of $\sum W_i = W$, *and constants* T_i, *amortized total* $\sum \frac{T_i}{W_i}$ *is minimized when* $W_i = W * \frac{\sqrt{T_i}}{\sum \sqrt{T_j}}$.

Proof. Let $X_i = \sqrt{\frac{T_i}{W_i}}$, $Y_i = \sqrt{W_i}$, use Cauchy-Schwarz inequation

$$(X_1^2 + ...X_n^2)(Y_1^2 + ... + Y_n^2) \ge (X_1 * Y_1 + ... + X_n * Y_n)^2$$

where "=" holds when $\frac{X_i}{Y_i}$ is constant for $\forall 1 \le i \le n$. □

Applying this rule to the number of messages, we get the conclusion that optimal configuration in this case is to assign window size $\{W_i\}$ proportional to square root of the replica access rate: $W_i = \frac{W*\sqrt{A_i}}{\sum_{1 \le j \le N} \sqrt{A_j}}$.

If we look at the cost of update latency, things get a little more complex. As in the case for the number of messages, the update window also collects messages and reduce the overall observed communication latency. There is only one round-trip communication delay per W_i updates. So the latency for update propagation on $replica_i$ is $\frac{C_i}{W_i}$. Here C_i is the latency of committing updates on $replica_i$. The overall latency in the whole system is the average of latencies at each replica weighted by the update rate A_i.

$$Latency = \frac{1}{\sum A_i} * \sum \frac{A_i * C_i}{W_i}$$

We can apply square-root rule to this formula and minimize the latency with W_i proportional to $\sqrt{A_i * C_i}$.

2.3 Availability under failure

Availability is another reason that one wants to alleviate consistency. In a replicated system enforcing strong consistency, there is no way to proceed when part of the nodes are not reachable. By buffering some amount of updates in replicas during network partition, replicas can tolerate a short-term failure at the expense of data consistency.

We use an availability concept similar to [17]. In case of failure, system appears to user as available if it continues to accept user request. For a window-based system, if a replica server can not commit the updates due to partitioning, it can only accept updates within its window limit. Once the window is filled up, it will have to refuse any update to avoid breaking its promise of consistency. The overall availability is quantified as (Number of accepted operations)/(Number of submitted operations).

First we look at aggressive update propagation. Since writes are committed as soon as possible, update windows are almost empty during normal operation. When partitioning occurs, the maximal number of updates that $replica_i$ can accept is $Min\{W_i, A_i * T_f\}$, where T_f is failure repair time. Intuitively, we should choose W_i according to the access rate A_i to maximize acceptance.

Theorem 2 *Proportional Distribution Rule*
Given constraint of $\sum_{1\leq i\leq N} W_i = W$, *and constants* $\{C_i\}$, $\sum_{1\leq i\leq N} C_i = C$, *bounded sum* $\sum Min\{W_i, C_i\}$ *is maximized when* $W_i = W * \frac{C_i}{\sum C_j}$. *The maximal value is* $Min\{W, C\}$.

Proof. First, $Min\{W, C\}$ is an upper bound:

$$\sum Min\{W_i, C_i\} \leq Min\{\sum W_i, \sum C_i\} = Min\{W, C\}$$

Second, $Min\{W, C\}$ is achieved when $W_i = \frac{W * C_i}{C}$.

$$
\begin{aligned}
& \sum\{Min\{W_i, C_i\}\} \\
= \; & \sum\{Min\{\tfrac{W*C_i}{C}, C_i\}\} \\
= \; & Min\{\tfrac{W}{C}, 1\} * \sum C_i \\
= \; & Min\{\tfrac{W}{C}, 1\} * C \\
= \; & Min\{W, C\}
\end{aligned}
$$

So the proportional configuration maximizes the sum expression. □

Replacing C_i with $A_i * T_f$ gives the solution to our problem.

The above analysis assumes that no replica can commit any updates during network partition. Since the occurrence of failure is generally unpredictable and independent of the replica hosts, more general assumptions on replication protocol do not change our conclusions.

When partition time is too long or too short, there is no difference between window distribution schemes: either everyone can fully mask the faults, or no one. In between is where adaptation wins out. For more detailed analysis please refer to [19].

When a lazy update propagation policy is used, the system presents less availability, because update window can be quite full and leave no room to buffer updates when failure occurs. For the "laziest" case, where updates are not sent out to other replicas until the window is filled up, the average number of updates already in window when failure occurs should be half of window size. This gives roughly the same availability level as an aggressive propagation scheme with half of total update window.

Optimization Goal	Model	Optimal Value	Optimal Window Configuration	Benefit over Even Distribution
Update Latency	$\frac{1}{\sum A_i} * \sum \frac{A_i * C_i}{W_i}$	$\frac{1}{W} * (\sum \sqrt{A_i * C_i})^2$	$W_i = W * \frac{\sqrt{A_i * C_i}}{\sum_j \sqrt{A_j * C_j}}$	$\frac{N * \sum (A_i * C_i)}{(\sum \sqrt{A_i * C_i})^2}$
Number of Messages	$Time * N * \sum \frac{A_i}{W_i}$	$\frac{Time * N}{W} * (\sum \sqrt{A_i})^2$	$W_i = W * \frac{\sqrt{A_i}}{\sum_j \sqrt{A_j}}$	$\frac{N * \sum A_i}{(\sum \sqrt{A_i})^2}$
Rejections during Partition	$\sum (A_i T_f - min\{W_i, A_i T_f\})$	$\sum (A_i * T_f) - W$	$W_i = W * \frac{A_i}{\sum_j A_j}$	depends on T_f and $\{A_i\}$

Table 1: Optimization models summary.

2.4 Discussion

Table 1 gives a summary of the three optimization goals we discussed. We got several interesting conclusions from these formulas:

- The benefit of optimal window configuration over even distribution does not depend on W. Instead, the benefit ratio solely depends on access distribution and communication cost.
- Optimal configuration can be N times better than even distribution. The extreme case happens when all accesses happens on one replica. This can be meaningful for some workload where most of the accesses are read-only, and concurrent updates are rare. Such access patterns are not uncommon in real word applications.
- Even distribution is near-optimal for balanced systems with comparable update rates on different replicas. For example, in a two server system with 1:2 ratio of update rates, optimal window configuration gives only 3% lower latency than even window distribution. A ratio of 1:4 gives 10% benefit.

To summarize, we expect an even window distribution works well for balanced, steady system where update rates on different replicas are of roughly the same magnitude. However, in other workloads where write accesses are more bursty and irregular, it becomes necessary to dynamically adjust window sizes and keep an optimal configuration. Under skewed update activity and resource distribution, larger number of replicas benefit from adaptation more.

2.5 Extension

Above analysis are all based on a system-wide consistency criterion: worst-case obsoleteness. This is suitable for a lot of applications where consistency in any particular replica is not a concern. However, there are certain scenarios where different replicas have different consistency requirements. Our model can be extended to accommodate per replica consistency guarantees.

In the extended model, each replica specifies its acceptable obsoleteness, and split the bound into update windows for every other replicas. The whole update window system will be a $N \times N$ array, where $W_{i,j}$ specify the amount of tentative updates $replica_j$ can hide from $relica_i$. The optimization strategy in the system are almost same as before. Each replica splits its consistency bound according to the rules we developed above. The result is a consistency semantics similar to that of numerical error defined in [18], but optimized in performance and availability.

3 Distributed Adaptation Algorithm

As described earlier, to optimize for either performance or availability goals, the ideal update window distribution $\{W_i\}$, needs to be a function of a set of input parameters, rather than even distribution among replicas. In a distributed environment, dynamic parameters such as the update rates are measured at each replicas. Consequently, there must be some mechanism to exchange those parameters and maintain a consistent window configuration.

For a small scale system, relying either on centralized control or group anti-entropy to reach and distribute a window configuration agreement is simple and practical. In fact, in our prototype system (see Section 4.2), we use the later approach. However, in other situations, more scalable and fully distributed algorithms are needed. We will explore such a light-weight algorithm in the following paragraphs.

To simplify the discussion, we represent the parameters measured by a replica as its weight C_i. For different optimization purposes, this weight can be the update rate, square root of update rate, or square root of product of update rate and communication latency. However, C_i is local knowledge of each replica. To build a global view of replica weight, the values of C_i are included in update messages and exchanged during group anti-entropy. We represent the weight of $replica_i$ received by replica j as $C_{i,j}$.

As time goes on, each replica will maintain a roughly up-to-date view of weights on all replicas. Then the window share can be computed locally. However, this algorithm introduces the chance

of breaking window bound. When a replica increases its weight and compute its new window share before other replicas ever know the change, the total window size will temporarily exceed the bound. To avoid this flaw, we must refrain the replicas from adopting its newly increased weight. The trick is to use $C_{i,i}$ instead of C_i:

- Record all weight value posted in messages as $\{C^k_{tentative}\}$;
- When update is committed, i.e. received acknowledgement from all replicas, record the weight posted with that update message as C_{commit}, also remove it from $C^k_{tentative}$;
- $C_{i,i} = min(C_{commit}, C^k_{tentative})$. Intuitively, $C_{i,i}$ is a conservative version of C_i. It is guaranteed to be no higher than any $C_{i,j}$ another $replica_j$ may use in its calculation.

The window share is computed locally as:

$$W_i = \frac{W * C_{i,i}}{\sum_j C_{j,i}}$$

Enforcing window distribution without using a group agreement protocol seems rather risky. However, as we shall prove next, the algorithm is correct in that it ensures optimal window distribution at steady state and at no time the window sum exceeds W.

Theorem 3 *Correctness in steady state*

At steady state, this algorithm achieves the optimal window distribution as required.

Proof. Steady state means $C_i = C_{i,j}$ for all $replica_i$, $1 \leq i \leq N$. In that case, $C_{i,i} = C_i = C_{i,j}$. Thus the local computed window shares reflect the optimal window configuration. □

Theorem 4 *Preservation of window bound*

At no time the bound on window sum is violated. i.e., $\sum W_i \leq W$.

Proof. From the definition of $C_{i,i}$, the inequation $C_{i,i} \leq C_{i,j}$ holds.

$$\begin{aligned} \textstyle\sum_i W_i &= \textstyle\sum_i \frac{W * C_{i,i}}{\sum_j C_{j,i}} \\ &\leq \textstyle\sum_i \frac{W * C_{i,i}}{\sum_j C_{j,j}} = \mathrm{W} \end{aligned}$$

Thus $\sum W_i \leq W$. □

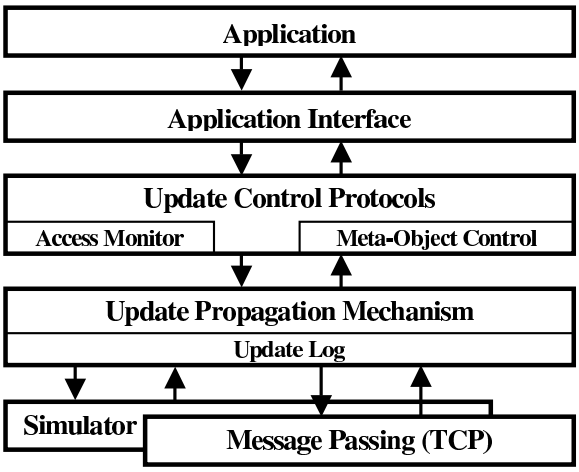

Figure 1: The FRACS architecture.

This algorithm is almost free – there is no additional messages and protocols to communicate window shares. All that is required is to include the update rate in the anti-entropy message and small memory to keep the parameter vectors. The algorithm is conservative only when a replica's share is being increased as a result of its own increased update rate. It will promptly converge to the desired level when anti-entropy is completed. In some cases, we need faster convergence. For example, when a node falls inactive, it will send out updates and new weight less frequently. We can send a null update message to inform others the newly decreased weight. For other details, please refer to [19].

4 Prototype design and implementation

In this section, we describe our prototype implementation of window-based replication system – FRACS, which stands for "Flexible Replication Architecture for a Consistency Spectrum".

4.1 Architecture

FRACS uses a modular design. Figure 1 shows the components and how they are stacked together.

The Application Interface Layer translates the abstractions in FRACS into semantics needed

by certain applications.

Update control layer maintains given consistency guarantees with a distributed update window system at the granularity of volume. Window configuration can be dynamically adjusted with either the protocol discussed in Section 3, or with the meta-object described in Section 4.2. We currently use the latter for its simplicity.

Local update rate is measured at update control layer with a rolling average algorithm. In order to balance the contradictory requirements of measurement stability and sensibility to changes, we use a relatively long averaging window (64 seconds in our test) and give more weight to recent accesses.

The communication modules transfer messages between hosts using standard TCP socket for its reliability and simplicity. A parallel layer, simulator, is included for research purpose. The details are explained in Section 4.3.

The update propagation module is based on Time-Stamped Anti-Entropy algorithm [4], one variant of popular anti-entropy algorithm. Here we do not detail the design of TSAE. The main point is to use timestamp vectors to detect version differences between replicas, so that they can exchange unknown updates to get each other into sync. Any update is kept in log until it has been received by every replica. Thus eventual delivery is guaranteed. Furthermore, FRACS use causal prefix constraint [11] to maintain dependence between updates. Together with TSAE, it guarantees eventual consistency among all replicas.

For efficiency purpose, a FRACS replica server propagates its updates with group anti-entropy instead of one-to-one sessions. It pushes buffered updates to all other replicas together, and pulls in their updates simultaneously. This shortens update propagation time, but is not scalable to large replica set. With hundreds or more replicas, more scalable propagation schemes like multicast trees are needed to avoid communication bottlenecks.

4.2 Dynamic Volume Maintenance

Interesting enough, the first application for FRACS is itself. The meta-data of FRACS volume is maintained with the same protocol as data objects. Such meta-data includes group membership of replica set, global consistency bound, etc. For convenience, we also maintain window configuration as meta-data, though it is more efficient to be managed with the algorithm described in Section 3. To keep a strong consistency for meta-data, we fix the update window as 1 on each replica.

Using such a strong guarantee is not always necessary. For example, adding a replica can be done locally without breaking any consistency promise. To make things worse, the above process will stop making progress when network is partitioned due to failure and reconfiguration is necessary. This can be partially remedied by a sub-group commit of meta-object update, where changes involving only a sub-set of replicas can be committed after reaching this sub-group.

4.3 Integrated Simulator and Validation

	Machine 1	Machine 2
Operating System	Linux	Linux
Location	Princeton, NJ	Stanford, CA
LAN interface	100 Mbps	100 Mbps
Network RTT	79.5 ms	
TCP bandwidth	367 KB/s	

Table 2: Platform characteristics.

Due to the lack of a large WAN environment and the long running time of our tests, the experiment results reported in Section 5 are obtained with the integrated simulator. The simulator substitutes the message passing layer with a event-queue and shares all higher level code with the real FRACS. We simulate a TCP connection as a channel with certain latency and bandwidth.

A two nodes test environment was setup for validation purpose. Table 2 summarizes the configurations and the network properties of the test platform. We run a series of 1-hour experiments

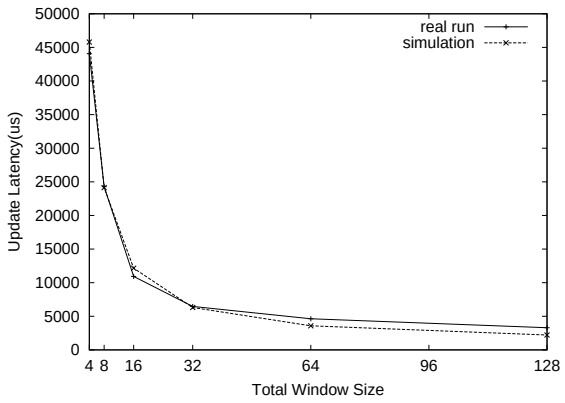

Figure 2: Simulator validation

with different settings of total update window. Figure 2 compares the latency measured from real runs to that of simulations. The curves matches well.

5 Experimental Results

In this section, we evaluate the optimization scheme and adaptation technique in FRACS. We compare the performance and availability of two window configuration alternatives: non-adaptive system with even distribution of window among all replicas; and adaptive system following optimal configuration based on update rate and network condition.

In the experiments, we simulate a transaction processing application replicated on multiple servers on top of FRACS. There are two types of transactions: read-only and read-update-write. Read-only transactions always access local replica, thus can see obsolete data. We externally compare the data to detect out-of-date read results. Each transaction involves only one record.

We use synthetic workloads to access this replicated database. Each server issues a Poisson stream of random read and write accesses. The average inter-arrival rate can vary with time. The read/write ratio, average transaction arrival rate and variation periods are adjustable on a per replica basis. Specifically, we simulate two kinds of dynamic workload: on-off and high-low. In on-off workload, replica servers experience periodic busy/idle cycles. In high-low workload, different

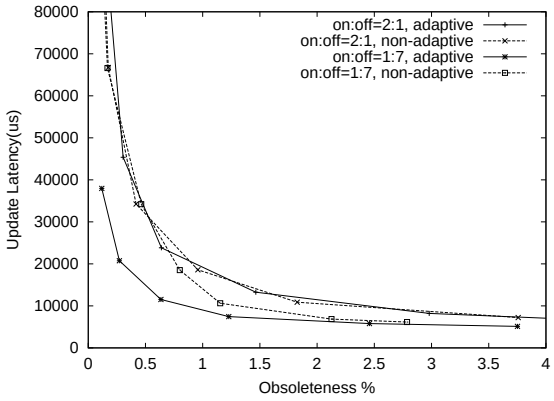

Figure 3: Update latency vs. obsoleteness when we vary the setting of total window size. We use two sets of on-off workload with different ratio of on and off periods. For each workload, we give results of both adaptive and non-adaptive window configurations.

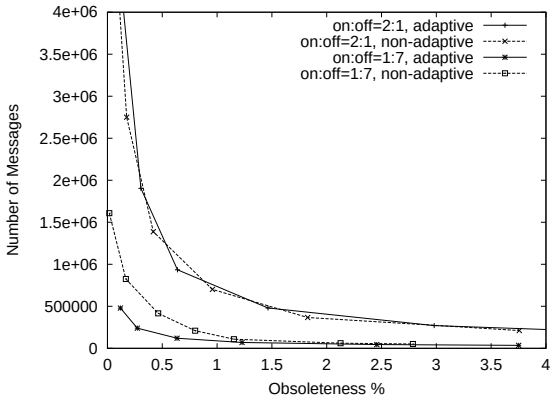

Figure 4: Number of messages vs. obsoleteness.

replicas are subject to different access rate, but the average request rates are constant over time. We believe that these two access patterns are typical in real world applications.

In the experiments, we simulate network connections with bandwidth of 300KB/s, and round trip time 100 mili-seconds, unless otherwise stated.

5.1 Varying the Total Window Size

The first experiment runs 8 replica servers with random on-off periods under Poisson distribution. Each server makes 5 update transactions per second during busy periods. Off periods only see

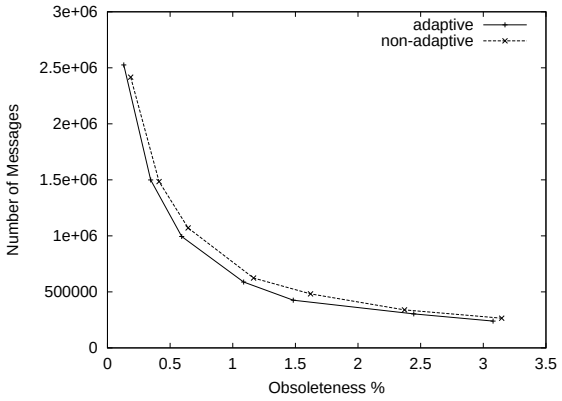

Figure 5: Number of messages vs. obsoleteness under skewed access pattern.

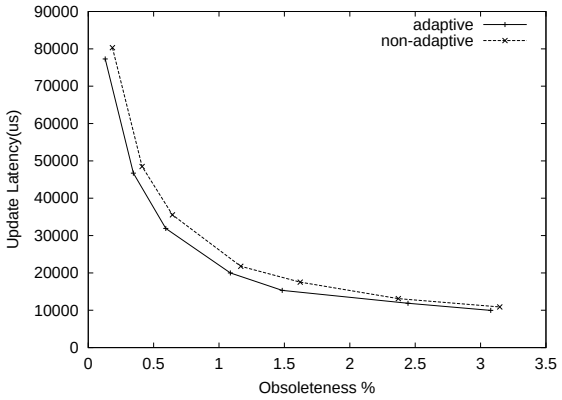

Figure 6: Update latency vs. obsoleteness under skewed access pattern.

background accesses of a few requests per hour. We simulated different ratios between on and off periods, 2:1 and 1:7. The simulation time is 16,000 seconds.

Figure 3 shows the update latency curves under varied consistency bounds. The X-axis is set as actual obsoleteness measured during simulation to give a fair comparison. As expected, adaptive algorithm shows much lower latency than non-adaptive algorithm under workload with on/off ratio of 1:7. When on/off ratio is 2:1, which means most replicas are active at any time, the two algorithms performs similarly because the little benefit from adaptation is almost offset by the cost of adaptation. Figure 4 shows the number of messages vs. obsoleteness. A similar trend can be found from the curves.

The theoretical results implies that the benefit ratio of adaptive system over non-adaptive one is

constant for different total window size. However, the benefit observed in Figure 3 and 4 diminishes when we loosen the consistency bound. This is due to other overhead factors not considered in our analytical model: computation time, adaptation operations, update queuing time, and message size etc. These overheads do not decrease when increasing window bound, thus eventually offset the benefit from adaptation.

In another experiment, we use different update rates at the replicas, but keep them constant all the time. We keep the average Poisson arrival rates on the 8 replica servers as 1:1:1:1:2:2:4:8, with total 12.5 updates per second. Such a configuration is trying to characterize an unbalanced load distribution : a few replicas get very hot while a lot of replicas are relatively idle. The results are shown in Figure 5 and Figure 6. Compared to that in on-off test, adaptation in this case is not very effective, but still gives some benefit.

5.2 Varying the Number of Replicas

Section 2.4 gives conclusions that the the maximal benefit of optimal configuration over even distribution is bounded by N — the number of replicas. In this experiment, we simulate extreme cases to test these conclusions: the hosts enter their active periods in a round-robin fashion where there is only one replica active at any time. The total window size keeps constant(128) throughout all test runs. The average update rate on active replica is also constant. We increase link bandwidth to 1 MB/s to compensate for large communication cost due to increased number of replicas.

The latency results are shown in Figure 7. Theoretically, the curve for adaptive system should be flat. The actual result shows a slow ascending slope, due to the increase of message size that our model fails to take into account. The ratio between results of adaptive and non-adaptive systems is almost linear to N, which is expected.

Figure 8 shows the same comparison for the number of messages. As predicted in Section 2.4, the number of messages for non-adaptive scheme should be $O(N^2)$, and benefit of adaptation be linear to N in this case. The curves roughly match this trend.

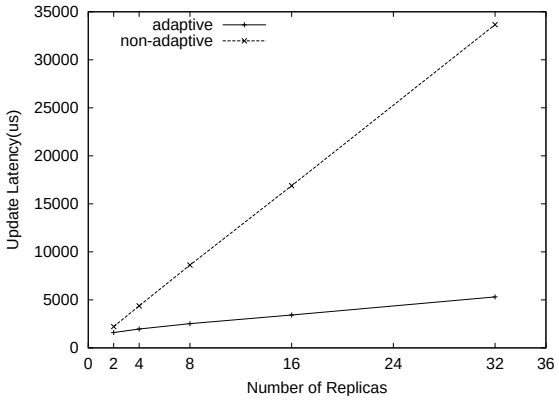

Figure 7: Update latency under varied number of replicas while keeping the total window size constant(128).

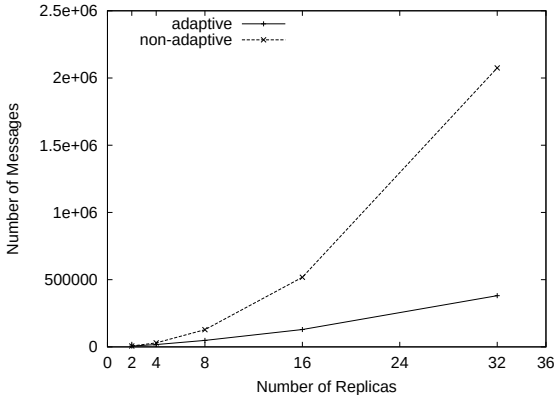

Figure 8: Number of messages under varied number of replicas while keeping the total window size constant(128).

5.3 Availability Results

We run FRACS under a configuration for optimizing availability, i.e., the window share being linear to the access rate of the replica. We experiment with 4 different algorithms as discussed in Section 2.3: eager/lazy propagation cross adaptive/ non-adaptive configuration. We use a random high-low access pattern : 4 replica servers with average Poisson update arrival rates of 1:2:3:4. The total rate is about 25 updates per minute. We also introduce some random network partition events with Poisson arrival rate and duration. Each failure separates the replica servers into two random selected groups.

To explore the effectiveness of adaptation under different failure patterns, we run two set of

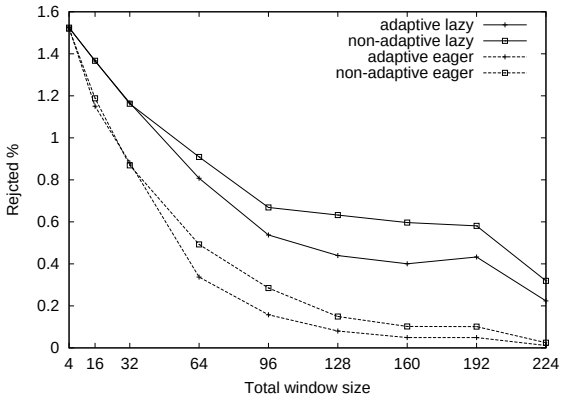

Figure 9: Availability vs. obsoleteness. We give percentage of rejected requests instead of percentage of accepted requests. We vary total window size from 4 to 224. Average failure repair time is 2 minutes

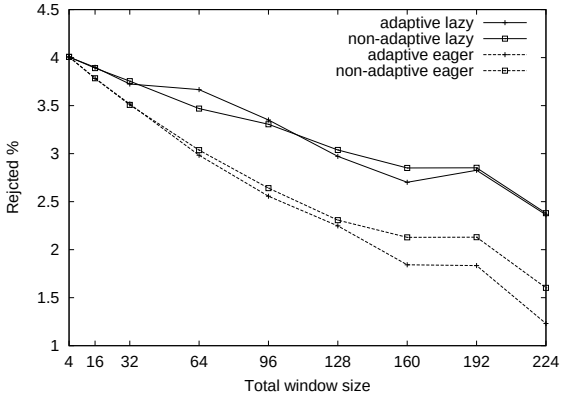

Figure 10: Availability vs. obsoleteness. Average failure repair time is 10 minutes

failures, one has average duration of 2 minutes and another 10 minutes. The average failure interval is 1 hour.

Figure 9 and Figure 10 give the results of running the four algorithms under these failure streams. To ensure fair comparison, we test all alternative algorithms with the same request and failure streams in each of the experiments.

The curves support the conclusions from Section 2.3. Adaptive algorithms give better availability than non-adaptive algorithms. And eagerly pushing out tentative updates also improves availability over lazy propagation schemes. As analyzed in Section 2.3, long failure repair time presents a harsher environment for application availability. And adaptation becomes less effective

in that case.

6 Related Work

The trade-off between replication consistency and performance/availability has been a hot research topic for many years. Strong consistency (one-copy serializability [2]) provides a notion of correctness to replicated database. Examples include the popular primary-copy algorithms [5], quorum consensus systems [8] and atomic broadcast protocols [3]. However, strong constraints on system integrity seriously limit the availability and performance. Many systems take another extreme of optimistic concurrency control. Ficus [6], Coda [7], Bayou [14, 11] provide single-copy availability and non-serialized access performance at the expense of data consistency. They only guarantee eventual consistency.

A number of efforts try to make compromises between the two extremes to benefit a variety of applications which can tolerate a certain bounded violation of integrity. Delta consistency [13] and timed consistency [15] developed models for time-related consistency. N-ignorant system [9] improves concurrency by allowing a transaction to run in parallel with at most N other transactions. This behavior can be directly expressed in our update window system. Lazy replication [10] provides three levels of consistency for operations with different ordering requirements. Epsilon-serializability [12] bounds the amount of divergence from strong consistency in regard of application semantics. TACT [18, 17] gives application more general choices with a set of consistency guarantees. A root difference between FRACS and the above systems is that we use an adaptive, fully-distributed algorithm to enforce the consistency guarantee, while the previous system either rely on centralized algorithms, or specify the consistency enforcement statically.

Adaptation by dynamic monitoring of replica references has been used in data placement and load balancing [16, 1]. As far as we know, there is yet no other work on adaptive update propagation and consistency enforcement.

7 Conclusions

A sound replication system must strike a careful balance among performance, consistency and availability for wide-area applications. In dealing with these trade-offs, the context in which the replicas are operating must be taken into account. In this paper, under the framework of update window protocol, we qualitatively proved and experimentally verified optimal update control distribution. This framework is expressive and versatile enough to capture a number of popular replica control schemes, allowing us to bound the consistency with one single protocol for both data and meta-data objects. The distribution of update control is through a lightweight protocol that does not require any centralized or group agreement protocols. We have built a prototype system, FRACS, which will enable us further researches in this direction.

Our planned future work includes to use this framework to implement more general consistency guarantees, and to understand the implication and application of dynamically adjust consistency bounds.

References

[1] Acharya, S., and Zdonik, S. B. An efficient scheme for dynamic data replication. Technical Report CS-93-43, Brown University, 1993.

[2] Bernstein, P., and Goodman, N. The failure and recovery problems for replicated distributed databases. *ACM Transactions on Database Systems* (December 1984).

[3] Birman, K. P., and Joseph, T. A. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems 5*, 1 (1987), 47–76.

[4] Golding, R. A. *Weak-Consistency Group Communication and Membership.* PhD thesis, University of California at Santa Cruz, 1992.

[5] Gray, J., Helland, P., O'Neil, P., and Shasha, D. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (1996), pp. 173–182.

[6] Guy, R. G., Heidemann, J. S., Mak, W., Page, Jr., T. W., Popek, G. J., and Rothmeir, D. Implementation of the ficus replicated file system. In *Proceedings of Summer USENIX Conference* (June 1990), pp. 63–72.

[7] James J. Kistler, M. S. Disconnected operation in the coda file system. In *Proc. of the ACM Thirteenth Symposium on Operating Systems Principles* (1991).

[8] Keleher, P. Decentralized replicated-object protocols. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)* (1999).

[9] Krishnakumar, N., and Bernstein, A. J. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Transactions on Database Systems 19*, 4 (1994), 586–625.

[10] Ladin, R., Liskov, B., Shrira, B., and Ghemawat, S. Providing availability using lazy replication. *ACM Transactions on Computer Systems 10*, 4 (1992), 360–391.

[11] Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible update propagation for weakly consistent replication. In *Proc. of the ACM Sixteenth Symposium on Operating Systems Principles* (1997).

[12] Pu, C., and A.Leff. Replication control in distributed system: An asynchronous approach. In *Proceedings of ACM SigMod Conference on Management of Data* (May 1991).

[13] Singla, A., and Ramachandran, U. Temporal notions of synchronization and consistency in beehive. In *Proc. of the 9 th Annual ACM Symposium on Parallel Algorithms and Architectures* (1997).

[14] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the ACM Fifteenth Symposium on Operating Systems Principles* (1995).

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

[16] Wolfson, O., and Jajodia, S. Distributed algorithms for dynamic replication of data. In *Proc. ACM PODS'92, Symposium on Principles of Database Systems* (June 1992).

[17] Yu, H., and Vadat, A. The costs and limits of availability for replicated services. In *Proc. of the 18th ACM Symposium on Operating Systems Principles* (2001), pp. 29–42.

[18] YU, H., AND VAHDAT, A. Design and evaluation of a continuous consistency model for replicated services. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (2000).

[19] ZHANG, C., AND ZHANG, Z. Trading replication consistency for performance and availability: an adaptive approach. Tech. Rep. http://www.cs.princeton.edu/ chizhang/fracs.pdf.