Design and Evaluation of a Window-Consistent Replication Service

Ashish Mehra†, Jennifer Rexford‡, and Farnam Jahanian†

†Real-Time Computing Lab
Department of EECS
The University of Michigan
Ann Arbor, MI 48109
{*ashish,farnam*}*@eecs.umich.edu*

‡Network Mathematics Research
Networking and Distributed Systems
AT&T Labs Research
Florham Park, NJ 07932
jrex@research.att.com

Abstract

Real-time applications typically operate under strict timing and dependability constraints. Although traditional data replication protocols provide fault tolerance, real-time guarantees require bounded overhead for managing this redundancy. This paper presents the design and evaluation of a *window-consistent* primary-backup replication service that provides timely availability of the repository by relaxing the consistency of the replicated data. The service guarantees controlled inconsistency by scheduling update transmissions from the primary to the backup(s); this ensures that client applications interact with a window-consistent repository when a backup must supplant a failed primary. Experiments on our prototype implementation, on a network of Intel-based PCs running RT-Mach, show that the service handles a range of client loads while maintaining bounds on temporal inconsistency.

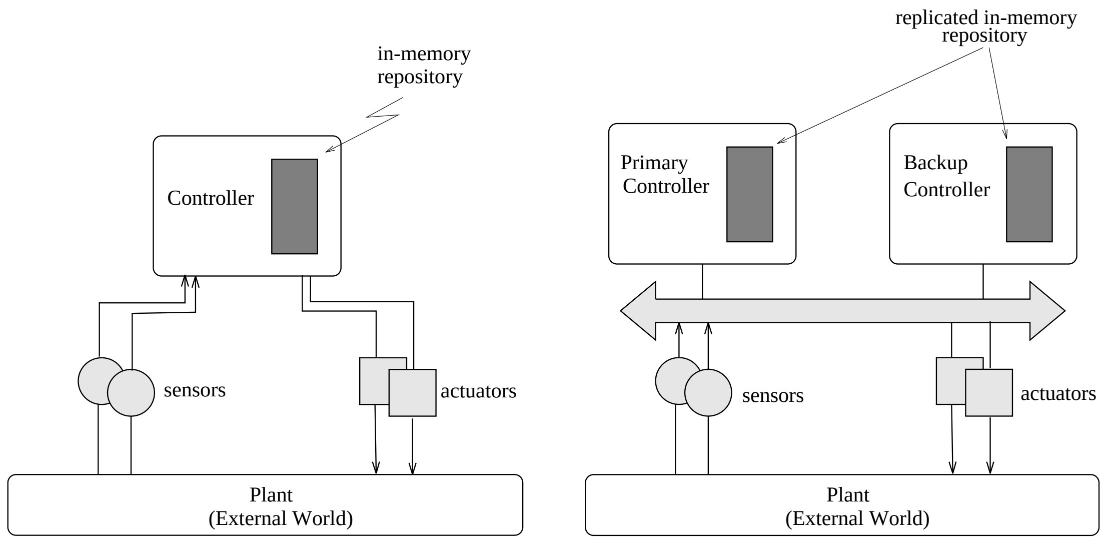
Index terms: real-time systems, fault tolerance, replication protocols, temporal consistency, scheduling

1 Introduction

Many embedded real-time applications, such as automated manufacturing and process control, require timely access to a fault-tolerant data repository. Fault-tolerant systems typically employ some form of redundancy to insulate applications from failures. *Time* redundancy protects applications by repeating computation or communication operations, while *space* redundancy masks failures by replicating physical resources. The time-space tradeoffs employed in most systems may prove inappropriate for achieving fault tolerance in a real-time environment. In particular, when time is scarce and the overhead for managing redundancy is too high, alternative approaches must balance the trade-off between timing predictability and fault tolerance.

For example, consider the process-control system shown in Figure 1(a). A *digital controller* supports monitoring, control, and actuation of the plant (external world). The controller software executes a

The work reported in this paper was supported in part by the National Science Foundation under Grant MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the NSF.



(a) Digital controller interacting with a plant (b) Primary-backup control system

Figure 1: Computer control system

tight loop, sampling sensors, calculating new values, and sending signals to external devices under its control. It also maintains an in-memory data repository which is updated frequently during each iteration of the control loop. The data repository must be replicated on a backup controller to meet the strict timing constraint on system recovery when the primary controller fails, as shown in Figure 1(b). In the event of a primary failure, the system must switch to the backup node within a few hundred milliseconds. Since there can be hundreds of updates to the data repository during each iteration of the control loop, it is impractical (and perhaps impossible) to update the backup synchronously each time the primary repository changes.

An alternative solution exploits the data semantics in a process-control system by allowing the backup to maintain a less current copy of the data that resides on the primary. The application may have distinct tolerances for the staleness of different data objects. With sufficiently recent data, the backup can safely supplant a failed primary; the backup can then reconstruct a consistent system state by extrapolating from previous values and new sensor readings. However, the system must ensure that the distance between the primary and the backup data is bounded within a predefined time window. Data objects may have distinct tolerances in how far the backup can lag behind before the object state becomes stale. The challenge is to bound the distance between the primary and the backup such that consistency is not compromised, while minimizing the overhead in exchanging messages between the primary and its backup.

This paper presents the design and implementation of a data replication service that combines fault-tolerant protocols, real-time scheduling, and temporal consistency semantics to accommodate such system requirements [24,29]. A client application registers a data object with the service by declaring the consistency requirements for the data, in terms of a *time window*. The primary selectively transmits to the backup, as opposed to sending an update every time an object changes, to bound both resource utilization and data inconsistency. The primary ensures that each backup site maintains a version of the object that was valid on the primary within the preceding time window by *scheduling* these update messages.

The next section discusses related work on fault-tolerant protocols and relaxed consistency semantics, with an emphasis on supporting real-time applications. Section 3 describes the proposed *window-consistent* primary-backup architecture and replication protocols for maintaining controlled inconsistency within the service. This replication model introduces a number of interesting issues in scheduling, fault detection, and system recovery. Section 4 considers real-time scheduling algorithms for creating and maintaining a window-consistent backup, while Section 5 presents techniques for fault detection and recovery for primary, backup, and communication failures. In Section 6, we present and evaluate an implementation of the window-consistent replication service on a network of Intel-based PCs running RT-Mach [32]. Section 7 concludes the paper by highlighting the limitations of this work and discussing future research directions.

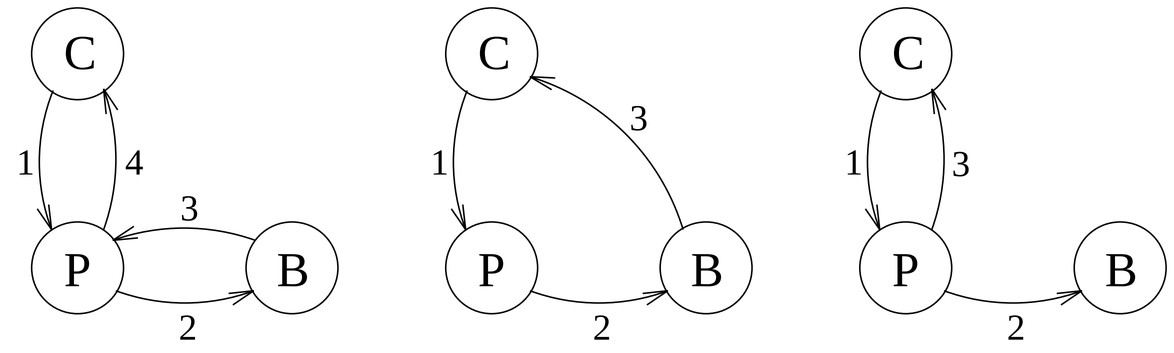
2 Related Work

2.1 Replication Models

A common approach to building fault-tolerant distributed systems is to replicate servers that fail independently. In *active* (state-machine) replication schemes [6,30], a collection of identical servers maintains copies of the system state. Client write operations are applied atomically to all of the replicas so that after detecting a server failure, the remaining servers can continue the service. *Passive* (primary-backup) replication [2,9], on the other hand, distinguishes one replica as the primary server, which handles all client requests. A write operation at the primary invokes the transmission of an update message to the backup servers. If the primary fails, a failover occurs and one of the backups becomes the new primary.

In recent years, several fault-tolerant distributed systems have employed state-machine [7,11,26] or primary-backup [4,5,9] replication. In general, passive replication schemes have longer recovery delays since a backup must invoke an explicit recovery algorithm to replace a failed primary. On the other hand, active replication typically incurs more overhead in responding to client requests since the service must execute an agreement protocol to ensure atomic ordered delivery of messages to all replicas. In both replication models, each client write operation generates communication within the service to maintain agreement amongst the replicas. This artificially ties the rate of write operations to the communication capacity in the service, limiting system throughput while ensuring consistent data.

Past work on server replication has focused, in most cases, on improving throughput and latency for client requests. For example, Figure 2(a) shows the basic primary-backup model, where a client write operation at the primary P triggers a synchronous update to the backup B [4]. The service can improve



(a) Blocking (b) Efficient blocking (c) Non-blocking

Figure 2: Primary-backup models

response time by allowing the backup B to acknowledge the client C [2], as shown in Figure 2(b). Finally, the primary can further reduce write latency by replying to C immediately after sending an update message to B, without waiting for an acknowledgement [8], as shown in Figure 2(c). Similar performance optimizations apply to the state-machine replication model. Although these techniques significantly improve *average* performance, they do not guarantee bounded *worst-case* delay, since they do not limit communication within the service.

Synchronization of redundant servers poses additional challenges in real-time environments, where applications operate under strict timing and dependability constraints; server replication for hard real-time systems is under investigation in several recent experimental projects [15,16,33]. Synchronization overheads, communication delay, and interaction with the external environment complicate the design of replication protocols for real-time applications. These overheads must be quantified precisely for the system to satisfy real-time constraints.

2.2 Consistency Semantics

A replication service can bound these overheads by relaxing the data consistency requirements in the repository. For a large class of real-time applications, the system can recover from a server failure even though the servers may not have maintained identical copies of the replicated state. This facilitates alternative approaches that trade atomic or causal consistency amongst the replicas for less expensive replication protocols. Enforcing a weaker correctness criterion has been studied extensively for different purposes and application areas. In particular, a number of researchers have observed that serializability is too strict a correctness criterion for real-time databases. Relaxed correctness criteria facilitate higher concurrency by permitting a limited amount of inconsistency in how a transaction views the database state [12,17,18,20,28].

Similarly, imprecise computation guarantees timely completion of an application by relaxing the accuracy requirements of the computation [22]. This is particularly useful in applications that use discrete samples of continuous-time variables, since these values can be approximated when there is not sufficient time to compute an exact value. Weak consistency can also improve performance in non-real-time applications. For instance, the quasi-copy model permits some inconsistency between the central data and its cached copies at remote sites [1]. This gives the scheduler more flexibility in propagating updates to the cached copies. In the same spirit, window-consistent replication allows computations that may otherwise be disallowed by existing active or passive protocols that require atomic updates to a collection of replicas.

3 Window-Consistent Replication

The window-consistent replication service consists of a primary and one or more backups, with the data on the primary shadowed at each backup site. These servers store objects which change over time, in response to client interaction with the primary. In the absence of failures, the primary satisfies all client requests and supplies a data-consistent repository. However, if the primary crashes, a window-consistent backup performs a failover to become the new primary. Hence, service availability hinges on the existence of a window-consistent backup to supplant a failed primary.

3.1 System Model

Unlike the primary-backup protocols in Figure 2, the window-consistent replication model decouples client read and write operations from communication within the service. As shown in Figure 3, the primary object manager (OM) handles client data requests, while sending messages to the backups at the behest of the update scheduler (US). Since read and write operations do not trigger transmissions to the backup sites, client response time depends only on local operations at the primary. This allows the primary to handle a high rate of client requests while independently sending update messages to the backup sites.

Although these update transmissions must accommodate the temporal consistency requirements of the objects, the primary cannot compromise the client application's processing demands. Hence, the primary must match the update rate with the available processing and network bandwidth by *selectively* transmitting messages to the backups. The primary executes an admission control algorithm as part of object creation, to ensure that the US can schedule sufficient update transmissions for any new objects. Unlike client reads and writes, object creation and deletion requires complete agreement between the primary and all the backups in the replication service.

3.2 Consistency Semantics

The primary US schedules transmissions to the backups to ensure that each replica has a sufficiently recent version of each object. Timestamps $\tau_i^P(t)$ and $\tau_i^B(t)$ identify successive versions of object O_i at the primary and backup sites, respectively. At time t the primary P has a copy of O_i written by the client application at time $\tau_i^P(t)$, while a backup B stores a, possibly older, version originally written on P at time $\tau_i^B(t)$. While B may have an older version of O_i than P, the copy on B must be "recent

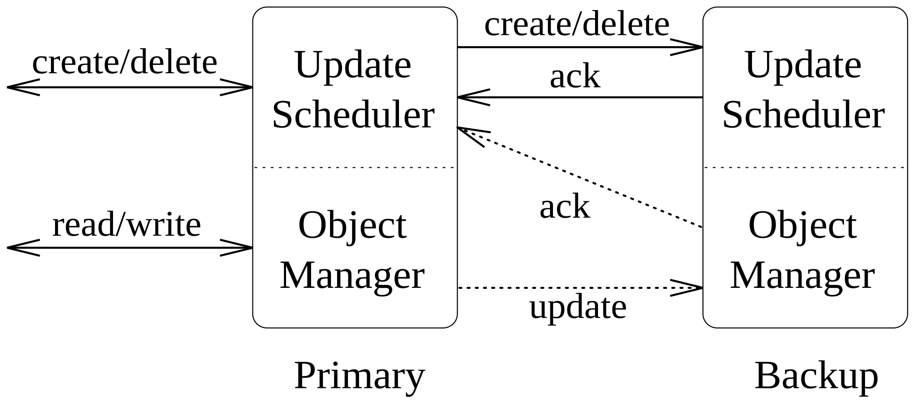


Figure 3: Window-consistent primary-backup architecture

enough." If O_i has window δ_i, a window-consistent backup must believe in data that was valid on P within the last δ_i time units.

Definition 1: At time t, a backup copy of object O_i has *window-inconsistency* $t - t_i'$, where t_i' is the maximum time such that $t_i' \leq t$ and $\tau_i^P(t_i') = \tau_i^B(t)$. Object O_i is *window-consistent* if and only if $t - t_i' \leq \delta_i$; a backup B is window-consistent if and only if all of its objects are window-consistent.

In other words, B has a window-consistent copy of object O_i at time t if and only if

$$\tau_i^P(t - \delta_i) \leq \tau_i^B(t) \leq \tau_i^P(t).$$

For example, in Figure 4, P performs several write operations on O_i, on behalf of client requests, but selectively transmits update messages to B. At time t the primary has the most recent version of the object, written by the client at time d. The backup has a copy first recorded on the primary at time b; the primary stopped believing this version at time c. Thus, $\tau_i^P(t) = d$, $\tau_i^B(t) = b$, and $\tau_i^P(t - \delta_i) = a$. Since $a \leq b \leq d$, B has a window-consistent version of O_i at time t. The backup object has inconsistency $t - c$, which is less than its window-consistency requirement δ_i. A small value of $t - c$ allows the client to operate with a more recent copy of the object if the backup must supplant a failed primary.

The metric $t - t_i'$ represents an object's temporal inconsistency within the replication service, as seen by an "omniscient" observer. Since the backup site does not always have up-to-date knowledge of client operations, the backup has a more conversative view of temporal consistency, as discussed in Section 5.2. The client may also require bounds on the staleness of the backup's object, relative to the primary's copy, to construct a valid system state when a failover occurs. In particular, if the client reads O_i at time t on P, it receives the version that it wrote $t - \tau_i^P(t)$ time units ago. On the other hand, if B supplants a failed primary, the client would read the version that it wrote $t - \tau_i^B(t)$ time units ago. This version is $\tau_i^P(t) - \tau_i^B(t)$ older than that on the primary; in Figure 4, this "client view" has inconsistency $d - b$.

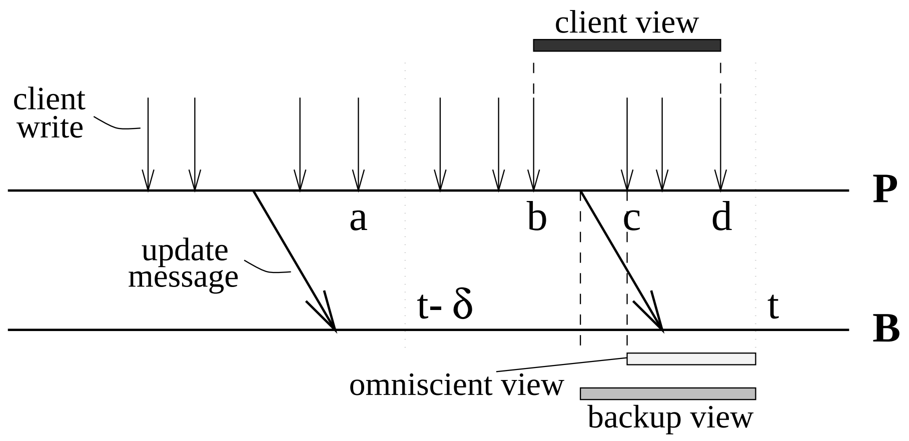


Figure 4: Window-consistency semantics

Definition 2: At time t, object O_i has *recovery inconsistency* $\tau_i^P(t) - \tau_i^B(t)$.

Two components contribute to this recovery inconsistency: client write patterns and the temporal inconsistency within the service. Window-consistent replication bounds the latter, allowing the client to bound recovery inconsistency based on its access patterns. For example, suppose consecutive client writes occur at most w_i time units apart; typically, w_i is smaller than δ_i, since the primary sends only *selective* updates to the backup sites. The window-consistency bound $t - t'_i \leq \delta_i$ then ensures that the backup's copy of the object was written on the primary no earlier than time $t - (\delta_i + w_i)$. Since $\tau_i^P(t) \leq t$, window consistency guarantees that $\tau_i^P(t) - \tau_i^B(t) \leq \delta_i + w_i$.

4 Real-Time Update Scheduling

This section describes how the primary can use existing real-time task scheduling algorithms to coordinate update transmissions to the backups. In the absence of link (performance or crash) failures [10], we assume a bound ℓ on the end-to-end communication latency within the service. For example, a real-time channel [14,23] with the desired bound could be established between the primary and the backups. Several other approaches to providing bounds on communication latency are discussed in [3].

If a client operation modifies O_i, the primary must send an update for the object within the next $\delta_i - \ell$ time units; otherwise, the backups may not receive a sufficiently recent version of O_i before the time-window δ_i elapses. In order to bound the temporal inconsistency within the service, it suffices that the primary send O_i to the backups at least once every $\delta_i - \ell$ time units. While bounding the temporal inconsistency, the primary may send additional updates to the backups if sufficient processing and network capacity are available; these extra transmissions increase the service's resilience to lost update messages and the average "goodness" of the replicated data.

In addition to sending update transmissions to the backups, the primary must allow efficient integration of new backups into the replication service. Limited processing and network capacity necessitate a trade-off between timely integration of a new backup and keeping existing backups window-consistent. The primary should minimize the time to integrate a new replica, especially when there are no other window-consistent backups, since a subsequent primary crash would result in a server failure. The primary constructs a schedule that sends each object to the backup exactly once, and allows the primary to smoothly transition to the update transmission schedule. While several task models can accommodate the requirements of window-consistent scheduling and backup integration, we initially consider the periodic task model [19, 21].

4.1 Periodic Scheduling of Updates

The transmissions of updates can be cast as "tasks" that run periodically with deadlines derived from the objects' window-consistency requirements. The primary coordinates transmissions to the backups by scheduling an update "task" with period p_i and service time e_i for each object O_i [1]; for window consistency, this permits a maximum period $p_i = (\delta_i - \ell)/2$. The end of a period serves as both the deadline for one invocation of the task and the arrival time for the subsequent invocation. The scheduler always runs the ready task with the highest priority, preempting execution if a higher-priority task arrives. For example, rate-monotonic scheduling statically assigns higher priority to tasks with shorter periods [19, 21], while earliest-due-date scheduling favors tasks with earlier deadlines [21].

The scheduling algorithm, coupled with the object parameters e_i and δ_i, determines a schedulability criterion based on the total processor and network utilization. The schedulability criterion governs object admission into the replication service. The primary rejects an object registration request (specifying e_i and δ_i) if it cannot schedule sufficient updates for the new object without jeopardizing the window consistency of existing objects, i.e., it does not have sufficient processing and network resources to accommodate the object's window-consistency requirements. The scheduling algorithm maintains window consistency for all objects as long as the the collection of tasks does not exceed a certain bound on resource utilization (e.g., 0.69 for rate-monotonic and 1 for earliest-due-date) [21].

4.2 Compressing the Periodic Schedule

While the periodic model can guarantee sufficient updates for each object, the schedule updates O_i only once per period p_i, even if computation and network resources permit more frequent transmissions. This restriction arises because the periodic model assumes that a task becomes ready to run only at period boundaries. However, the primary can transmit the current version of an object at any time. The scheduler can capitalize on this "readiness" of tasks to improve both resource utilization and the window consistency on the backups by *compressing* the periodic schedule.

Consider two objects O_1 (with $p_1 = 5$ and $e_1 = 2$) and O_2 ($p_2 = 3$ and $e_2 = 1$), as shown in Figure 5; the unshaded boxes denote transmission of O_1, while the shaded boxes signify transmission of O_2. The scheduler must send an update requiring 1 unit of processing time once every 3 time units

[1]The size of O_i determines the time e_i required for each update transmission. In order to accommodate preemptive scheduling and objects of various sizes, the primary can send an update message as one or more fixed-length *packets*.

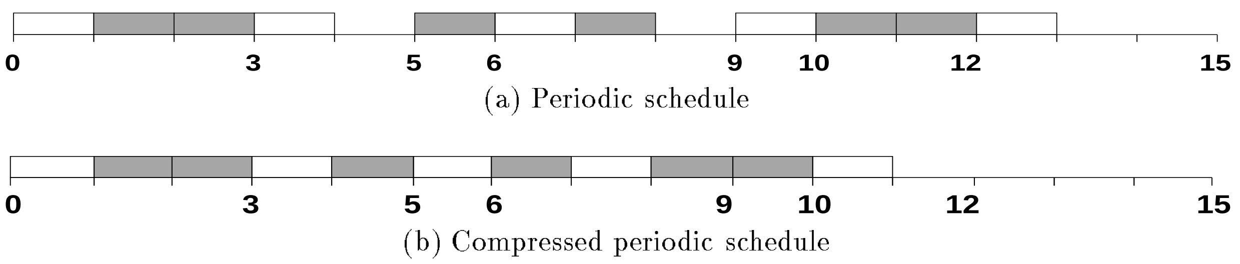


Figure 5: Compression ($p_1 = 5$, $e_1 = 2$, $p_2 = 3$, $e_2 = 1$)

(unshaded box) and an update requiring 2 units of processing time once every 5 time units (shaded box). The schedule repeats after each major cycle of length 15. Each time unit corresponds to a *tick* which is the granularity of resource allocation for processing and transmission of a packet. For this example, both the rate-monotonic and earliest-due-date algorithms generate the schedule shown in Figure 5(a).

While each update is sent as required in the major cycle of length 15, the schedule has 4 units of slack time. The replication service can capitalize on this slack time to improve the average temporal consistency of the backup objects. In particular, the periodic schedule in Figure 5(a) can provide the *order* of task executions without restricting the *time* the tasks become active. If no tasks are ready to run, the scheduler can advance to the earliest pending task and activate that task by advancing the logical time to the start of the next period for that object. With the compressed schedule the primary still transmits an update for each O_i at least once per period p_i but can send more frequent update messages when time allows. As shown in Figure 5(b), compressing the slack time allows the schedule to start over at time 11. In the worst case, the compressed schedule degrades to the periodic schedule with the associated guarantees.

4.3 Integrating a New Backup

To minimize the time the service operates without a window-consistent backup, the primary P needs an efficient mechanism to integrate a new or invalid backup B. P must send the new backup B a copy of each object and then transition to the normal periodic schedule, as shown in Figure 6. Although B may not have window-consistent objects during the execution of the integration schedule, each object must become consistent and remain consistent until its first update in the normal periodic schedule.

As a result, B must receive a copy of O_i within the "period" p_i before the periodic schedule begins; this ensures that B can afford to wait until the next p_i interval to start receiving periodic update messages for O_i. In order to integrate the new backup, then, the primary must execute an *integration schedule* that would allow it to transition to the periodic schedule while maintaining window consistency. Referring to Figure 6, a window-consistent transition requires $D_j^{prior} + D_j^{post} \leq \delta_j - \ell$; D_j^{prior} is the time elapsed from the last transmission of O_j to the end of the integration schedule, while D_j^{post} is the time from the start of the periodic schedule until the first transmission of O_j. This

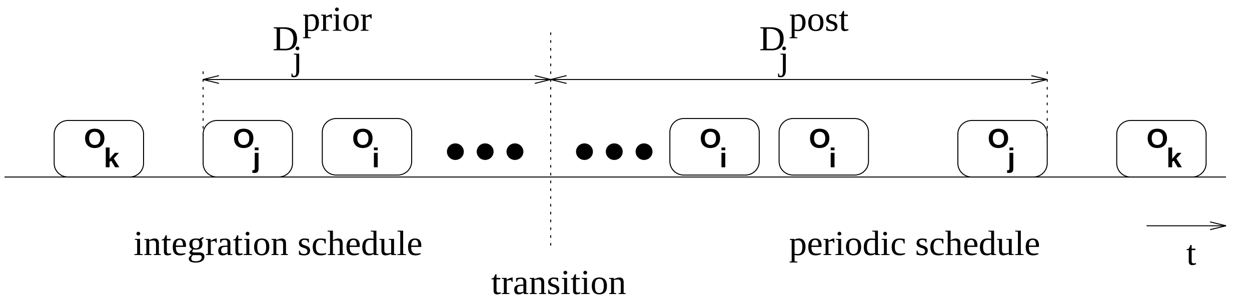


Figure 6: Integrating a new backup repository

ensures window consistency for each object, even across the schedule transition. Since the periodic task model provides $D_j^{post} \leq p_j$, it suffices to ensure that $D_j^{post} \leq p_j = (\delta_i - \ell)/2$.

A simple schedule for integration is to send objects to the new backup using the normal periodic schedule already being used for update transmissions to the existing replicas. This incurs a worst-case delay of $2\max_i\{p_i\}$ to integrate the new backup into the service. However, if the service has no window-consistent backup sites, the primary should minimize the time required to integrate a new replica. In particular, an efficient integration schedule should transmit each object exactly once before transitioning to the normal periodic schedule.

The primary may adapt the normal periodic schedule into an efficient integration schedule by removing duplicate object transmissions. In particular, the primary can transmit the objects in order of their *last* update transmissions before the end of a major cycle in the normal schedule. For example, for the schedule shown in Figure 5(a), the integration schedule is $[O_1, O_2]$ because the last transmission for O_1 (O_2) before time 15 is at time 10 (12). A transition from the integration schedule to the normal schedule sustains window consistency on the newly integrated backup since the normal schedule guarantees window consistency across major cycles. Since the integration schedule is derived from the periodic schedule, it follows that $D_j^{prior} \leq D_j^{post} \leq p_j$.

The normal schedule order can be determined when objects are created or during the first major cycle of the normal schedule. Since the schedule transmits each object only once, the integration delay is $\sum_i^N e_i$, where N is the number of registered objects. Although this approach is efficient for static object sets, dynamic creation and deletion of objects introduces more complexity. Since the transmission order in the normal schedule depends on the object set, the primary must recompute the integration schedule whenever a new object enters the service. The cost of constructing an integration schedule, especially for dynamic object sets, can be reduced by sending the objects to B in *reverse period* order, such that the objects with larger periods are sent before those with smaller periods.

For object O_j, this ensures that only objects with smaller or equivalent periods can follow O_j in the integration schedule; these same objects can precede O_j in the periodic schedule. This guarantees that the integration schedule transmits O_j no more than p_j time units before the start of the periodic schedule, ensuring a window-consistent transition. For example, in Figure 6 $p_i \leq p_j \leq p_k$. In the

```
**$P$ sends update to $B$**
    select object $i$
    $\tau_i^{xmit} \leftarrow \tau_i^P$
    send $B$ $(i, O_i, \tau_i^P, t)$
```

$\Longrightarrow$

```
**$B$ receives $(i, O, \tau, s)$**
    if $(s > t_i^{xmit})$
        $O_i \leftarrow O$
        $\tau_i^B \leftarrow \tau$
        $t_i^{xmit} \leftarrow s$
    if $(s > t^{last})$
        $t^{last} \leftarrow s$
    send $P$ $(i, \tau_i^B, t)$
```

$\Longrightarrow$

```
**$P$ receives $(i, \tau, s)$**
    if $(\tau > \tau_i^{ack})$
        $\tau_i^{ack} \leftarrow \tau$
    if $(s > t^{last})$
        $t^{last} \leftarrow s$
```

Figure 7: Update protocols

periodic schedule objects O_i with $p_i \leq p_j$ are transmitted *at least* once within time D_j^{post} but *exactly* once within time D_j^{prior}; it follows that $D_j^{prior} \leq D_j^{post} \leq p_j$. After object creations or deletions, the primary can construct the new integration schedule by sorting the new set of periods. The primary minimizes the time it operates without a window-consistent backup by transmitting each object exactly once before transitioning to the normal periodic schedule.

5 Fault Detection and Recovery

Although real-time scheduling of update messages can maintain window-consistent replicas, processor and communication failures potentially disrupt system operation. We assume that servers may suffer crash failures and the communication subsystem may suffer omission or performance failures; when a site fails, the remaining replicas must recover in a timely manner to continue the data-repository service. The primary attempts to minimize the time it operates without a window-consistent backup, since a subsequent primary crash would cause a service failure. Similarly, the backup tries to detect a primary crash and initiate failover before any backup objects become window-inconsistent. Although the primary and backup cannot have complete knowledge of the global system state, the message exchange between servers provides a measure of recent service activity.

5.1 Update Protocols

Figure 7 shows how the primary and backup sites exchange object data and estimate global system state. We assume that the servers communicate only by exchanging messages. Since these messages include temporal information, P and B cannot effectively reason about each other unless server clocks are synchronized within a known maximum bound. A clock synchronization algorithm can use the transmit times for the update and acknowledgement messages to bound clock skew in the service. Using the update protocols, P and B each approximate global state by maintaining the most recent information received from the other site.

Before transmitting an update message at time t, the primary records the version timestamp τ_i^{xmit}

for the selected object O_i. Since $\tau_i^B \leq \tau_i^{xmit}$, this information gives P an optimistic view of the backup's window consistency. The primary's message to the backup contains the object data, along with the version timestamp and the transmission time. B uses the transmission time to detect out-of-order message arrivals by maintaining t_i^{xmit}, the time of the most recent transmission of O_i that has been successfully received; the sites store monotonically non-decreasing version timestamps, without requiring reliable or in-order message delivery in the service. Upon receiving a newer transmission of O_i, the backup updates the object's data, the version timestamp τ_i^B, and t_i^{xmit}; as discussed in Section 5.2, the backup uses t_i^{xmit} to reason about its own window consistency.

To diagnose a crashed primary, B also maintains t^{last}, the transmission time of the last message received from P regarding *any* object; that is, $t^{last} = \max_i\{t_i^{xmit}\}$. Similarly, P tracks the transmission times of B's messages to diagnose possible crash failures. Hence, the backup's acknowledgement message to P includes the transmission time t, as well as τ_i^B, the *most recent* version timestamp for O_i on B. Using this information, the primary determines τ_i^{ack}, the most recent version of O_i that B has successfully acknowledged. Since $\tau_i^B \geq \tau_i^{ack}$, this variable gives P a pessimistic measure of the backup's window consistency; as discussed in Section 5.3, the primary uses τ_i^{ack} and τ_i^{xmit} to select policies for scheduling update transmissions to the backup.

5.2 Backup Recovery From Primary Failures

A backup site must estimate its own window consistency and the status of the primary to successfully supplant a crashed primary. While B may be unaware of recent client interaction with P for each object, B does know the time t_i^{xmit} when P transmitted verion τ_i^B of object O_i. Although P may continue to believe version τ_i^{xmit}, even after transmitting the update message, B conservatively estimates that the client wrote a new version of O_i *just after* P transmitted the object at time t_i^{xmit}. In particular,

Definition 3: At time t, the backup copy of object O_i has *estimated* inconsistency $t - t_i^{xmit}$; the backup knows that O_i is window-consistent if $t - t_i^{xmit} \leq \delta_i$.

Figure 4 shows an example of this "backup view" of window consistency.

Using this consistency metric, the backup must balance the possibility of becoming window-inconsistent with the likehood of falsely diagnosing a primary crash. If B believes that all of its objects are still window-consistent, B need not trigger a failover until further delay would endanger the consistency of a backup object; in particular, the backup conservatively estimates that its copy of O_i could become window-inconsistent by time $t_i^{xmit} + \delta_i$, in the absence of further update messages from P. However, to reduce the likelihood of false failure detection, failover should only occur if B has not received *any* messages from P for some minimum time β.

In this adaptive failure detection mechanism, B diagnoses a primary crash at time

$$t^{crash} = \min_i\{t_i^{xmit} + \delta_i\}$$

if and only if $t^{crash} \geq t^{last} + \beta$. After failover, the new primary site invokes the client application and begins interacting with the external environment. For a period of time, the new P operates with some partially inconsistent data but gradually constructs a consistent system state from these old values and new sensor readings. The new P later integrates a fresh backup to enhance future service availability.

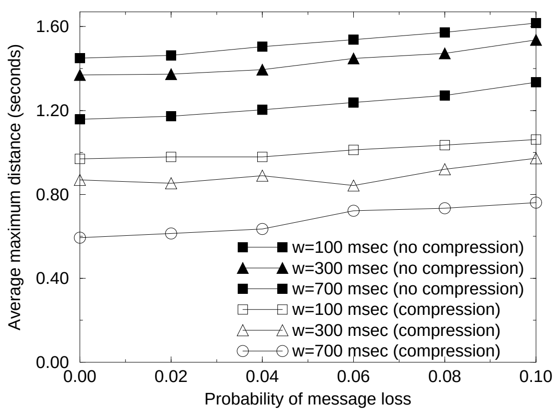
Since B diagnoses a primary crash through missed update messages, lost or delayed messages could still trigger false failure detection, resulting in multiple active primary sites. When the system has multiple backups, the replicas can vote to select a single, valid primary. However, when the service has only two sites, communication failures can cause each site to assume the other has failed. In this situation, a third-party "witness" [27] can select the primary site. This witness does not act as a primary or backup server, but casts the deciding vote in failure diagnosis. In a real-time control system, the actuator devices could implicitly serve as this witness; if a new server starts issuing commands to the actuators, the devices could ignore subsequent instructions from the previous primary site.

5.3 Primary Recovery From Backup Failures

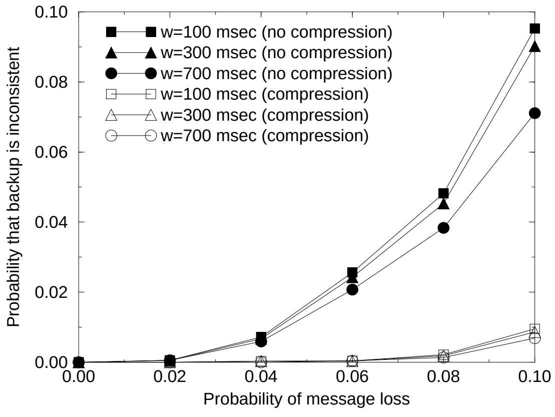
Service availability also depends on timely recovery from backup failures. Since the data-repository service continues whenever a valid primary exists, the primary can temporarily tolerate backup crashes or communication failures without endangering the client application. Ultimately, though, P should minimize the portion of time it operates without a window-consistent backup, since a subsequent primary crash would cause a service failure. The primary should diagnose possible backup crashes and efficiently integrate new backup sites. If P believes that an operational backup has become window-inconsistent, due to lost update messages or transient overload conditions, the primary should quickly refresh the inconsistent objects.

As in Section 5.2, timeout mechanisms can detect possible server failures. The primary assumes that the backup has crashed if P has not received *any* acknowledgement messages in the last α time units (i.e., $t - t^{last} \geq \alpha$). After detecting a backup crash, P can integrate a fresh backup site into the system while continuing to satisfy client read and write requests. If the P mistakenly diagnoses a backup crash, the system must operate with one less replica while the primary integrates a new backup site; this new backup does not become window-consistent until the integration schedule completes, as described in Section 4.3. However, if the backup has actually failed, a large timeout value increases the failure diagnosis latency, which also increases the time the system operates without sufficient backup sites. Hence, P must carefully select α to maximize the backups' chance of recovering from a subsequent primary failure.

Even if the backup site does not crash, delayed or lost update messages can compromise the window consistency of backup objects, making B ineligible to replace a crashed primary. Using τ_i^{ack} and τ_i^{xmit}, P can estimate the consistency of backup objects and select the appropriate policy for scheduling update transmissions. The primary may choose to *reintegrate* an inconsistent backup, even when $t - t^{last} < \alpha$, rather than wait for a later update message to restore the objects' window



(a) Average maximum distance



(b) Probability(backup inconsistent)

Figure 8: *Window consistency* $t - t'_i$: The graphs show the performance of the service as a function of the client write rate, message loss, and schedule compression. Although object inconsistency increases with message loss, compressing the periodic schedule reduces the effects of communication failures. Inconsistency increases as the client writes more frequently, since the primary changes it object soon after transmitting an update message to the backup.

consistency. Suppose the primary thinks that B's copy of O_i is window-inconsistent. Under periodic update scheduling, P may not send another update message for this object until some time $2p_i = \delta_i - d$ later. If this object has a large window δ_i, the primary can reestablish the backup's window consistently more quickly by executing the integration schedule, which requires time $\sum_i e_i$, where e_i is the service time for object O_i, as described in Section 4.1.

Still, the primary cannot accurately determine if the backup object O_i is inconsistent, since lost or delayed acknowledgement messages can result in an overly pessimistic value for τ_i^{ack}. The primary should not be overly aggressive in diagnosing inconsistent backup objects, since reintegration temporarily prohibits the backup from replacing a failed primary. Instead, P should ideally "retransmit" the offending object, without violating the window consistency of the other objects in the service. For example, P can schedule a special "retransmission" window for transmitting objects that have not received acknowledgement messages for past updates; when this "retransmission object" is selected for service, P transmits an update message for one of the existing objects, based on the values of τ_i^{ack} and τ_i^{xmit}. This improves the likelihood of having window-consistent backup sites, even in the presence of communication failures.

6 Implementation and Evaluation

6.1 Prototype Implementation

We have developed a prototype implementation of the window-consistent replication service to demonstrate and evaluate the proposed service model. The implementation consists of a primary and a backup server, with the client application running on the primary node as shown in Figure 3. The primary implements rate-monotonic scheduling of update transmissions, with an option to enable schedule compression. Tick scheduling allocates the processor for different activities, such as handling client requests, sending update messages, and processing acknowledgements from the backup. At the start of each tick, the primary transmits an update message to the backup for one of the objects, as determined by the scheduling algorithm. Any client read/write requests and update acknowledgements are processed next, with priority given to client requests.

Each server is currently an Intel-based PC running the Real-Time Mach [25,32] operating system[2]. The sites communicate over an Ethernet through UDP datagrams using the `Socket++` library [31], with extensions to the UNIX `select` call for *priority-based* access to the active sockets. At initialization, sockets are registered at the appropriate priority such that the socket for receiving client requests has a higher priority over that for receiving update acknowledgements from the backup. A tick period of $100\,ms$ was chosen to minimize the intrusion from other runnable system processes[3]. To further minimize interference, experiments were conducted with lightly-loaded machines on the same Ethernet segment; we did not observe any significant fluctuations in network or processor load during the experiments.

The primary and backup sites maintain in-memory logs of events at run-time to efficiently collect performance data with minimal intrusion. Estimates of the clock skew between the primary and the backup, derived from actual measurements of round-trip latency, are used to adjust the occurrence times of events to calculate the distance between objects on the primary and backup sites. The prototype evaluation considers three main consistency metrics representing window consistency and the backup and client views. These performability metrics are influenced by several parameters, including client write rate, communication failures, and schedule compression.

The experiments vary the client write rate by changing the time w_i between successive client writes to an object. We inject communication failures by randomly dropping update messages; this captures the effect of transient network load as well as lost update acknowledgements. The invariants in our evaluation are the tick period ($100\,ms$), the objects' window size ($\delta_i = 30$ ticks), and the number of objects ($N = 10$); given the tick period and δ_i, N is determined by the schedulability criterion of the rate-monotonic scheduling algorithm. All objects have the same update transmission time of one tick, with the object size chosen such that the time to process and transmit the object is reasonably small

[2]earlier experiments on Sun workstations running Solaris 1.1 show similar results [24].

[3]The $100\,ms$ tick period has the same granularity as the process scheduling quantum to limit the interference from other jobs running on the machine. However, smaller tick periods are desirable in order to allow objects to specify tighter windows (the window size is expressed in number of ticks) and respond to client requests in a timely manner.

compared to the tick size; the extra time within each tick period is used to process client requests and update acknowledgements. Experiments ran for 45 minutes for each data point.

6.2 Omniscient View (Window Consistency)

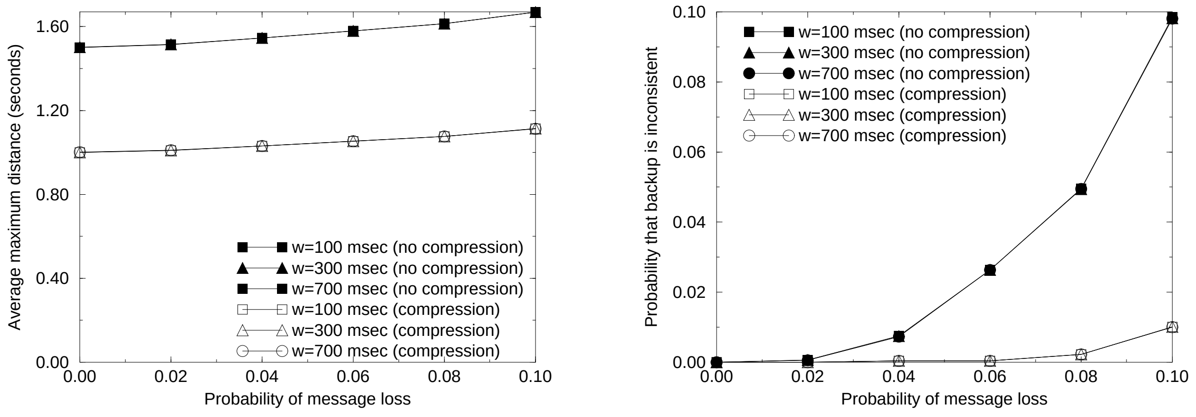
The window-consistency metric $(t - t')$ captures the actual temporal inconsistency between the primary and the backup sites, and serves as a reference point for the performance of the replication service. Figure 8(a) shows the *average maximum distance* between the primary and the backup as a function of the probability of message loss for three different client write periods, with and without schedule compression. This measures the inconsistency of each backup object just before receiving an update, averaged over all versions and all objects, reflecting the "goodness" of the replicated data. Figure 8(b) shows the *probability of an inconsistent backup* as a function message loss; this "fault-tolerance" metric measures the likelihood that the backup has one or more inconsistent objects. In these experiments, the client writes each object once every tick ($w_i = 100\,ms$), once every 3 ticks ($w_i = 300\,ms$) and once every 7 ticks ($w_i = 700\,ms$).

The probability of message loss varies from 0% to 10%; experiments with higher message loss rates reveal similar trends. Message loss increases the distance between the primary and the backup, as well as the likelihood of an inconsistent backup. However, the influence of message loss is not as pronounced due to conservative object admission in the current implementation. This occurs because, on average, the periodic model schedules updates twice as often as necessary, in order to guarantee the required worst-case spacing between update transmissions. Message loss should have more influence in other scheduling models which permit higher resource utilization, as discussed in Section 7. Higher client write rates also tend to increase the backup's inconsistency; as the client writes more frequently, the primary's copy of the object changes soon after sending an update message, resulting in staler data at the backup site.

Schedule compression is very effective in improving both performance variables. The average maximum distance between the primary and backup under no message loss (the y-intercept) reduces by about 30% for high client rates in Figure 8(a); similar reductions are seen for all message loss probabilities. This occurs because schedule compression successfully utilizes idle ticks in the schedule generated by the rate-monotonic scheduling algorithm; the utilization thus increases to 100% and the primary sends approximately 30% more object updates to the backup. Compression plays a relatively more important role in reducing the likelihood of an inconsistent backup, as can be seen from Figure 8(b). Also, compression reduces the impact of communication failures, since the extra update transmissions effectively mask lost messages.

6.3 Backup View (Estimated Consistency)

Although Figure 8 provides a system-wide view of window consistency, the backup site has limited knowledge of the primary state. The backup's view $(t - t^{xmit})$ is a good, albeit pessimistic, estimate of the actual window consistency, as shown in Figure 9. The backup site uses this metric to evaluate its



(a) Average maximum distance (b) Probability(backup inconsistent)

Figure 9: *Backup view* $t - t_i^{xmit}$*:* The plots show system performance from the backup's conservative viewpoint, as a function of the client write rate, message loss, and schedule compression. As in Figure 8, temporal consistency improves under schedule compression but worsens under increasing message loss. The backup's view is impervious to the client write rate.

own window consistency to detect a crashed primary and effect a failover. As in Figure 8 message loss increases the average maximum distance (Figure 9(a)) and the likelihood of an inconsistent backup (Figure 9(b)). Schedule compression also has similar benefits for the backup's estimate of window consistency.

However, unlike Figure 8, the client write rate does not influence the backup's view of its window consistency. The backup (pessimistically) assumes that the client writes an object on the primary immediately after the primary transmits an update message for that object to the backup. For this reason, the backup's estimate of the average maximum distance between the primary and the backup is always worse than that derived from the omniscient view. It follows that this estimate is more accurate for high client write rates, as can be seen by comparing Figures 8(a) and 9(a); for high client rates relative to the window, $t - t^{xmit}$ and $t - t'$ are virtually identical. The window-consistent replication model is designed to operate with high client write rates, relative to communication within the service, so the backup typically has an accurate view of its temporal consistency.

6.4 Client View (Recovery Consistency)

The client view ($\tau^P(t) - \tau^B(t)$) measures the inconsistency between the primary and backup versions on object reads; better recovery consistency provides a more accurate system state after failover. Since the client can read at an arbitrary time, Figure 10 shows the time average of recovery inconsistency, averaged across all objects, with and without compression. We attribute the minor fluctuations in the graphs to noise in the measurements.

The distance metric is not sensitive to the client write rate, since frequent client writes increase

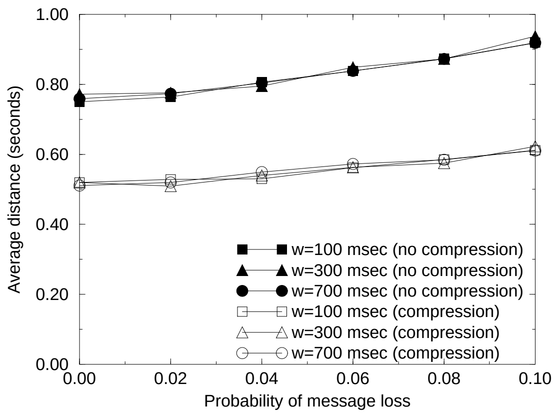


Figure 10: *Client view* $\tau_i^P(t) - \tau_i^B(t)$*:* This graph presents the time average of recovery inconsistency, as a function of the client write rate, message loss, and schedule compression. Compressing the update schedule improves consistency by generating more frequent update transmission, while message loss worsens read consistency. The metric is largely independent of the client write rate.

both $\tau_i^P(t)$ and $\tau_i^B(t)$; when the client writes more often, the primary copy changes frequently (i.e., $\tau_i^P(t)$ is close to t), but the backup also receives more recent versions of the data (i.e., $\tau_i^B(t)$ is close to t_i^{xmit}). Moderate message loss does not have a significant influence on read inconsistency, especially under schedule compression. As expected, schedule compression improves the read inconsistency seen by the client significantly ($\approx 30\%$). It is, therefore, an effective technique for improving the "goodness" of the replicated data.

7 Conclusion and Future Work

Window consistency offers a framework for designing replication protocols with predictable timing behavior. By decoupling communication within the service from the handling of client requests, a replication protocol can handle a higher rate of read and write operations and provide more timely response to clients. Scheduling the selective communication within the service provides bounds on the degree of inconsistency between servers. While our prototype implementation has successfully demonstrated the utility of the window-consistent replication model, more extensive evaluation is needed to validate the ideas identified in this paper. We have recently added support for fault-detection, failover, and integration of new backups. Further experiments on the current platform will ascertain the usefulness of processor capacity reserves [25] and other RT-Mach features in implementing the window-consistent replication service. The present work extends into several fruitful areas of research:

Object admission/scheduling: We are studying techniques to maximize the number of admitted objects and improve objects' window consistency by optimizing object admission and update scheduling. For the window-consistent replication service, the periodic task model is overly conservative in accepting object registration requests; that is, it may either limit the number of objects that are accepted or it may accept only those objects with relatively large windows. This occurs because, on average, the periodic model schedules updates twice as often as necessary, in order to guarantee the required worst-case spacing between update transmissions. We are exploring other scheduling algorithms, such as the distance-constrained task model [13] which assigns task priorities based on separation constraints, in terms of their implementation complexity and ability to accommodate dynamic creation/deletion of objects.

We are also considering techniques to maximize the "goodness" of the replicated data. As one possible approach, we are exploring ways to incorporate the client write rate in object admission and scheduling. An alternate approach is to optimize the object window size itself by proportionally shrinking object windows such that the system remains schedulable; this should improve each object's *worst-case* temporal inconsistency. The selection of object window sizes can be cast as an instance of the linear programming optimization problem. Schedule compression can still be used to improve the utilization of the remaining available resources.

Inter-object window consistency: We are extending our window-consistent replication model to incorporate temporal consistency constraints *between* objects. Our goal is to bound consistency in a *replicated* set of related objects; new algorithms may be necessary for real-time update scheduling of such object sets. This is related to the problem of ensuring temporally consistent objects in a real-time database system; however, our goal is to bound consistency in a *replicated* set of related objects.

Alternative replication models: Although the current prototype implements a primary-backup architecture with a single backup site, we are studying the additional issues involved in supporting multiple backups. In addition, we are also exploring window consistency in the state-machine replication. This would enable us to investigate the applicability of window consistency to alternative replication models.

Acknowledgements

The authors wish to thank Sreekanth Brahmamdam and Hock-Siong Ang for their help in running experiments and post-processing the collected data, and the reviewers for their helpful comments.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina, "Data caching issues in an information retrieval system," *ACM Trans. Database Systems*, vol. 15, no. 3, pp. 359–384, September 1990.

[2] P. Alsberg and J. Day, "A principle for resilient sharing of distributed resources," in *Proc. IEEE Int'l Conf. on Software Engineering*, Los Angeles, 1976.

[3] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne, "Real-time communication in packet-switched networks," *Proc. of the IEEE*, vol. 82, no. 1, pp. 122–139, January 1994.

[4] J. F. Bartlett, "A NonStop kernel," in *Proc. ACM Symp. on Operating Systems Principles*, 1981.

[5] A. Bhide, E. N. Elnozahy, and S. P. Morgan, "A highly available network file server," in *Winter USENIX Conference*, pp. 199–205, January 1991.

[6] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Computer Systems*, vol. 5, no. 1, pp. 47–76, 1987.

[7] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37–53, December 1993.

[8] N. Budhiraja and K. Marzullo, "Tradeoffs in implementing primary-backup protocols," Department of Computer Science TR-92-1307, Cornell University, 1992.

[9] N. Budhiraja and K. Marzullo, "Tradeoffs in implementing primary-backup protocols," in *Proc. IEEE Symposium on Parallal and Distributed Processing*, pp. 280–288, October 1995.

[10] F. Cristian, "Understanding fault tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, February 1991.

[11] F. Cristian, B. Dancy, and J. Dehn, "Fault-tolerance in the advanced automation system," in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 6–17, June 1990.

[12] S. B. Davidson and A. Watters, "Partial computation in real-time database systems," in *Proc. Workshop on Real-Time Operating Systems and Software*, pp. 117–121, May 1988.

[13] C.-C. Han and K.-J. Lin, "Scheduling distance-constrained real-time tasks," in *Proc. Real-Time Systems Symposium*, pp. 300–308, 1992.

[14] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[15] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The MARS approach," *IEEE Micro*, pp. 25–40, February 1989.

[16] H. Kopetz and G. Grunsteidl, "TTP – a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, January 1994.

[17] H. F. Korth, N. Soparkar, and A. Silberschatz, "Triggered real time databases with consistency constraints," in *Proc. Int'l Conf. on Very Large Data Bases*, August 1990.

[18] T.-W. Kuo and A. K. Mok, "Ssp: a semantics-based protocol for real-time data access," in *Proc. Real-Time Systems Symposium*, pp. 76–86, December 1993.

[19] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proc. Real-Time Systems Symposium*, pp. 166–171. IEEE, December 1989.

[20] K.-J. Lin, F. Jahanian, A. Jhingran, and C. D. Locke, "A model of hard real-time transaction systems," Technical Report RC 17515, IBM T.J. Watson Reseach Center, January 1992.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[22] J. W. S. Liu, W.-K. Shih, and K.-J. Lin, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, January 1994.

[23] A. Mehra, A. Indiresan, and K. G. Shin, "Structuring communication software for quality-of-service guarantees," in *Proc. 17th Real-Time Systems Symposium*, pp. 144–154, December 1996.

[24] A. Mehra, J. Rexford, H.-S. Ang, and F. Jahanian, "Design and evaluation of a window-consistent replication service," in *Proc. IEEE Real-Time Technology and Applications Symposium*, pp. 182–191, May 1995.

[25] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proc. IEEE International Conference on Multimedia Computing and Systems*, pp. 90–99, May 1994.

[26] S. Mishra, L. L. Peterson, and R. D. Schlichting, "Consul: A communication substrate for fault-tolerant distributed programs," Technical Report 91-32, University of Arizona, November 1991.

[27] J.-F. Paris, "Using volatile witnesses to extend the applicability of available copy protocols," in *Proc. Workshop on the Management of Replicated Data*, pp. 30–33, November 1992.

[28] C. Pu and A. Leff, "Replica control in distributed systems: An asynchronous approach," in *Proc. ACM SIGMOD*, pp. 377–386, May 1991.

[29] J. Rexford, A. Mehra, J. Dolter, and F. Jahanian, "Window-consistent replication for real-time applications," in *Proc. Workshop on Real-Time Operating Systems and Software*, pp. 107–111, May 1994.

[30] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, December 1990.

[31] G. Swaminathan, *C++ Socket Classes*, University of Virginia, June 1993.

[32] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Toward a predictable real-time system," in *Proc. USENIX Mach Workshop*, pp. 73–82, October 1990.

[33] P. Verissimo, P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton, "The extra performance architecture (XPA)," in *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, D. Powell, editor, 1991.