Design and Evaluation of a Window-Consistent Replication Service

Ashish Mehra, Jennifer Rexford, Hock-Siong Ang and Farnam Jahanian

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
E-mail: {ashish, jrexford, anghs, farnam}@eecs.umich.edu

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 show that the service handles a range of client loads while maintaining bounds on temporal inconsistency.*

1 Introduction

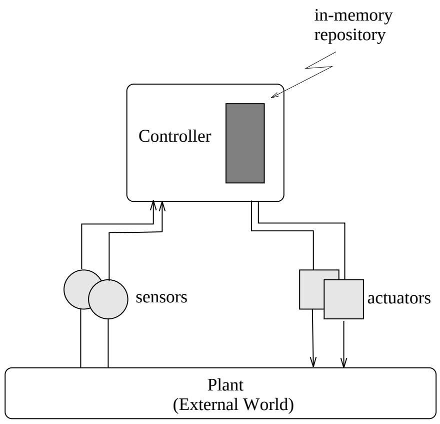
Many embedded real-time applications, such as automated manufacturing and process control, require timely access to a fault-tolerant data repository. Fault-tolerant systems typically employ some form of redundancy to insulate applications from failures. *Time* redundancy protects applications by repeating 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.

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 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.

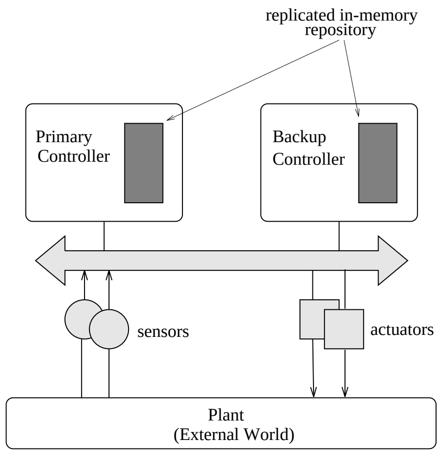
An alternative solution exploits the data semantics in a process-control system by allowing the backup to maintain a less current copy of the data that resides on the primary. The application may have distinct tolerances for the staleness of different data objects. With sufficiently recent data, the 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. 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 [1]. 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 work reported in this paper was supported in part by the National Science Foundation under Grant MIP-9203895 and a grant from the Rackham School of Graduate Studies at the University of Michigan. 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

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 primary-backup architecture and the protocols for maintaining controlled inconsistency within the service. 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. 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 [2–6], a collection of identical servers maintains copies of the system state. Client updates are applied atomically to all of the replicas so that after detecting a server failure, the remaining servers continue the service. *Passive* (primary-backup) replication [7–10], 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 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. Synchronization of redundant servers poses additional challenges in real-time environments; several recent experimental projects support server replication in distributed hard real-time systems [11, 12]. Synchronization overheads, communication delay, and interaction with the external environment complicate the design of replication protocols for real-time applications.

2.2 Consistency Semantics

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 data. This facilitates alternative approaches that exploit weaker consistency amongst the replicas in a real-time system. These schemes may 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 [13–17].

Similarly, imprecise computation[18] ensures that a real-time computation meets its timing constraints by approximating numerical results when there is not sufficient time to compute an exact value. Other non-real-time applications also exploit weak consistency to obtain better performance. For instance, the quasi-copy model [19] permits weak consistency between the central data and its cached copies at remote sites. 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.

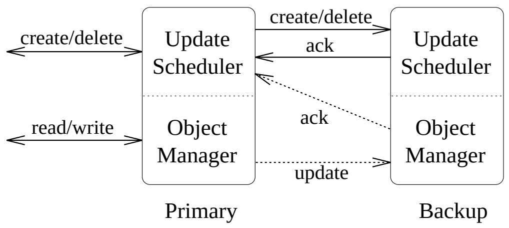


Figure 2: Architecture of the replication service

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. Service availability hinges on the existence of a window-consistent backup to supplant a failed primary.

3.1 System Model

Unlike traditional primary-backup protocols, the window-consistent replication model decouples client read and write operations from communication within the service. As shown in Figure 2, 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 identify successive versions of object O_i at the primary and backup sites. 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 enough." If O_i has window δ_i, a window-consistent backup must believe in data 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

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

The metric $t - t_i'$ represents an object's temporal inconsistency within the replication service. 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.

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

Two components contribute to this read inconsistency: client write patterns and the temporal inconsistency within the service. Window-consistent replication bounds the latter, allowing the client to bound read 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 $\tau_i^P(t) - \tau_i^B(t) \leq \delta_i + w_i$.

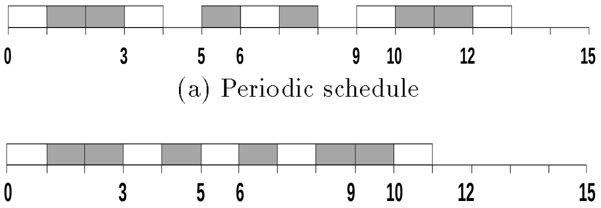
4 Real-Time Update Scheduling

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 failures [20], we assume a bound ℓ on communication latency within the service. 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.

4.1 Periodic Scheduling of Updates

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 [21, 22], 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 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].



(a) Periodic schedule

(b) Compressed periodic schedule

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

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. 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 3; the unshaded boxes denote transmission of O_1, while the shaded boxes signify transmission of O_2. 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 3(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. 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 3(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 4. Although B may not have window-consistent objects during the execu-

[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*.

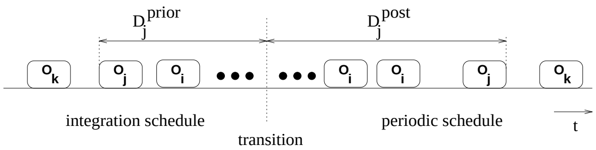


Figure 4: Integrating a new backup repository

tion of the integration schedule, each object must become consistent and remain consistent until its first update in the normal periodic schedule. Referring to Figure 4, 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. Since the periodic task model provides $D_j^{post} \leq p_j$, it suffices to ensure that $D_j^{prior} \leq p_j$.

Limited processing and network capacity necessitate a trade-off between timely integration of a new backup and keeping existing backups window-consistent. If the service already has at least one window-consistent backup, the time required to integrate a new backup is not of paramount importance. 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. 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 3(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). 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. 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 4 $p_i \leq p_j \leq p_k$. In the 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 in $\mathcal{O}(\log N)$ time 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. When a site fails, the remaining replicas must recover in a timely manner to continue the data-repository service. 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. The primary server utilizes these estimates to select the appropriate scheduling policies for coordinating update messages to the backups. For the backup servers, estimates of window consistency and the status of the primary are crucial for both failure detection and recovery.

5.1 Estimating Global System State

For effective failure detection and recovery, all replicas should have an approximate knowledge of the global system state. Since the primary and backup sites exchange regular update and acknowledgement messages, a clock synchronization algorithm can use the transmit times of these messages to bound clock skew within the service. Synchronized clocks enable a backup B to reason about its own window consistency, based on the timestamps of messages from the primary P. Furthermore, we assume that servers may suffer crash failures and the communication subsystem may suffer omission or performance failures.

The update protocols ensure that each site can estimate the status of the other servers, as shown in Figure 5. Although communication delay and message loss prevent P from knowing τ_i^B for every object, P does know the time when it last transmitted an update for O_i. At this time t_i^{xmit}, P sent version $\tau_i^{xmit} = \tau_i^P(t_i^{xmit})$ of O_i to the backup. The

P **sends update to** B	B **receives** (i, O, τ, t)	P **receives** (i, τ)
select object i $\tau_i^{xmit} \leftarrow \tau_i^P$ $t_i^{xmit} \leftarrow$ current_time send B $(i, O_i, \tau_i^{xmit}, t_i^{xmit})$	if $(t > t_i^{xmit})$ $O_i \leftarrow O$ $\tau_i^B \leftarrow \tau$ $t_i^{xmit} \leftarrow t$ send P (i, τ_i^B)	if $(\tau > \tau_i^{ack})$ $\tau_i^{ack} \leftarrow \tau$

Figure 5: Update protocols

backup's acknowledgement message includes the value of τ_i^B; this allows P to determine τ_i^{ack}, the most recent copy of O_i that B has successfully acknowledged. The primary then knows that $\tau_i^{ack} \leq \tau_i^B \leq \tau_i^{xmit}$, giving P both pessimistic and optimistic measures of the backup's window consistency.

Similarly, while B may be unaware of recent client interaction with P for each object, B does know τ_i^B and the time t_i^{xmit} when P transmitted this update. B conservatively estimates that the client wrote a new version of O_i just after P transmitted version τ_i^{xmit}. This estimate plays an important role in failure detection and recovery.

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 $t - t_i^{xmit} \leq \delta_i$.

Since different objects are handled independently by the update protocols, out-of-order or lost messages for different objects do not interfere. For an individual object O_i, the servers can deliver updates and acknowledgements in arbitrary order, since the protocols ensure monotonically non-decreasing values for τ_i^{ack} and τ_i^B. The service can tolerate communication failures, although multiple lost or delayed messages may compromise the window consistency of the backup sites.

5.2 Recovery From Primary Failures

Since the primary sends update messages at regular intervals, the updates and subsequent acknowledgements serve as implicit heartbeats between the servers. If a backup does not receive update messages for some period of time T, the site may diagnose a primary crash and initiate a failover to become the new primary server. The new primary site then 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 server crash through missed update messages, lost or delayed messages could trigger false failure detection, resulting in multiple active primary sites. When the system has multiple backups, the sites vote to select the valid primary. If the replication service has only one backup site, a third-party "witness" [23] can cast 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.

The replication service could reduce the likelihood of false failure detection by having a large timeout value T, but the backups cannot tolerate long periods without receiving update messages from the primary. So, the system must balance the possibility of false failure detection with the likelihood of having no window-consistent backup to replace a crashed primary. Although B is unaware of recent client activity at P, it conservatively estimates that its copy of O_i will become window-inconsistent by time $t_i^{xmit} + \delta_i$, in the absence of further update messages from P.

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. Instead of using a static timeout to diagnose a primary failure, B could adjust its timeout value based on the perceived consistency of the backup objects. In particular, the backup cannot safely supplant a failed primary after time $\min_i\{t_i^{xmit} + \delta_i\}$. Combining this expression with the minimum timeout value T results in an adaptive timeout mechanism that reduces the chance of falsely diagnosing a primary failure without jeopardizing service availability.

5.3 Recovery From Backup Failures

Fault tolerance 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. P can detect possible backup failures through missed acknowledgement messages. After diagnosing a backup crash, P can integrate a fresh backup site into the system while continuing to satisfy client read and write requests.

The primary must balance the likelihood of false failure detection with the time required to integrate a new backup site. If P mistakenly diagnoses a backup crash, the service must function with one less replica while P sends the new site a copy of each object, as described in Section 4. 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 the timeout value to maximize the backups' chance of recovering from a subsequent primary failure.

In this context, various failures manifest themselves as window-consistency violations on the backup. For example, a crashed backup or delayed/lost acknowledgement messages lead P to suspect that B has violated the window-consistency requirements of one or more objects. Based on τ_i^{ack} and τ_i^{xmit}, P can estimate whether B has a window-consistent copy of each object. As the perceived skew between P and B grows, P can invalidate B's membership in the service, regardless of whether this inconsistency stems from communication failures or a true crash.

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 2. The primary implements rate-monotonic scheduling of update transmissions, with an option to enable schedule compression. Tick scheduling allocates the processor for different activities; 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.

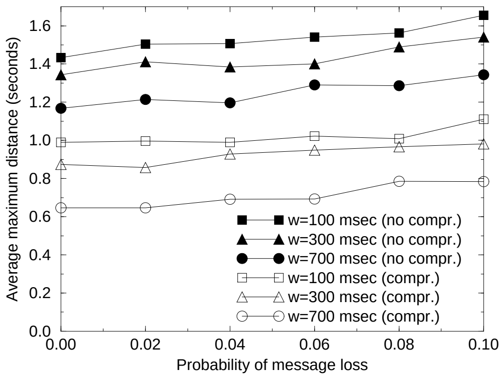
Each server is currently a Sun SPARCstation running Solaris 1.1. The sites communicate over an Ethernet through UDP datagrams using the `Socket++` library [24]. While Solaris 1.1 provides a stable environment for code development and testing, the platform does not support real-time process scheduling or bounded communication delays. A tick period of $100\,ms$ was chosen to minimize the intrusion from other runnable system processes. To further minimize interference, experiments were conducted with lightly-loaded workstations 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 generate a global ordering. 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.

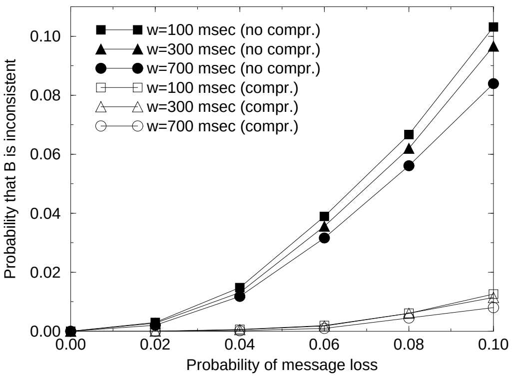
The experiments vary the client write rate by changing the time w_i between successive client writes to an object. We inject communication failures by randomly dropping update messages; this captures the effect of transient network load as well as lost update acknowledgements. The invariants in our evaluation 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 compared to the tick size; the extra time within each tick period is used to process client requests and update acknowledgements.

6.2 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 6(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



(a) Average maximum distance



(b) Probability(backup inconsistent)

Figure 6: Window consistency $t - t'_i$

versions and all objects, reflecting the "goodness" of the replicated data. Figure 6(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 ($w_i = 700\,ms$).

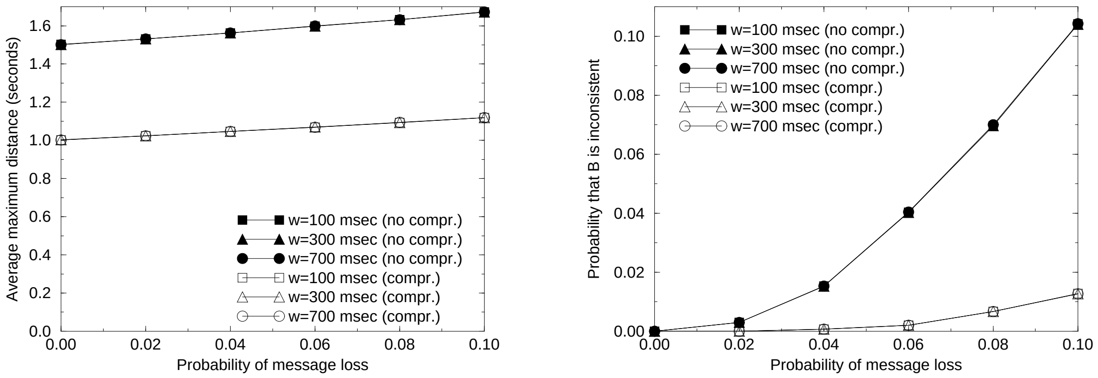
The probability of message loss varies from 0% to 10%; experiments with higher message loss rates reveal similar trends. Message loss increases the distance between the primary and the backup, as well as the likelihood of an inconsistent backup. However, the influence of message loss is not as pronounced 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 6(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 6(b). Also, compression reduces the impact of communication failures, since the extra update transmissions effectively mask lost messages.

6.3 Backup View of Consistency

Although Figure 6 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 7. The backup site uses this metric to evaluate its own window consistency and detect a crashed primary. As in Figure 6 message loss increases the average maximum distance (Figure 7(a)) and the likelihood of an inconsistent backup (Figure 7(b)). Schedule compression also has similar benefits for the backup's estimate of window consistency.

However, unlike Figure 6, 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 6(a) and 7(a); for high client rates relative to the window, $t - t^{xmit}$ and



(a) Average maximum distance

(b) Probability(backup inconsistent)

Figure 7: Backup view $t - t_i^{xmit}$

$t - t'$ 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 consistency.

6.4 Client View (Read Inconsistency)

The client view $(\tau^P(t) - \tau^B(t))$ measures the inconsistency between the primary and backup versions on object reads; better read consistency provides a more accurate system state after failover. Since the client can read at an arbitrary time, Figure 8 shows the time average of read 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 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.

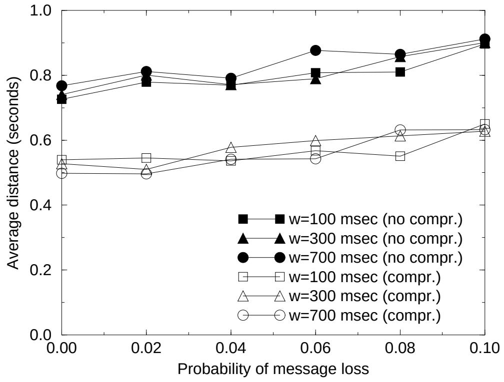


Figure 8: Client view $\tau_i^P(t) - \tau_i^B(t)$

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 on a true real-time platform is needed to validate the ideas identified in this paper.

We have recently implemented the window-consistent replication service on RT-Mach [25] and added support for fault-detection, failover, and integration of new backups. We plan to conduct a more extensive evaluation of the RT-Mach-based implementation, starting with a repeat of the experiments reported here. This exercise would ascertain the usefulness of processor capacity reserves 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, including other scheduling algorithms such as the distance-constrained task model [26].

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.

Alternative replication models: We are studying the additional issues involved in having multiple backups. We are also exploring window consistency in the state-machine replication model.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Primary-backup protocols: Lower bounds and optimal implementations," in *Proc. IFIP Working Conference on Dependable Computing*, pp. 187–198, 1992.

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[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. 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.

[23] 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.

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

[25] C. W. Mercer, J. Zelenka, and R. Rajkumar, "On predictable operating system protocol processing," Technical Report CMU-CS-94-165, Carnegie Mellon University, May 1994.

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