A Temporal Consensus Model

Hengming Zou and Farnam Jahanian

Real-time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109
{*zou, farnam*}*@eecs.umich.edu*

Abstract

The active (state-machine) replication protocol has been one of the commonly adopted approaches to provide fault tolerant data services. In such a scheme, service state is replicated at all participating servers and client requests are presented to all *the servers. A strict consensus protocol executed by the servers ensures that all servers receive all the requests in the same order. The output of each request from all the servers are then compared to mask the result of faulty server(s). However, the overhead associated with running the strict consensus protocol tends to slow down the response to client requests, which makes it unsuitable for real-time applications where timing predictability is a necessity. This paper presents a weak consensus model called "temporal consensus" that reduces such overhead and enable the active scheme to meet the timing constraint of real-time applications.*

1 Introduction

Active (state-machine) replication [32] is a commonly used approach by modern systems to provide fault-tolerant data service. In such a system, service state is replicated at all participating servers and client requests are presented to *all* the servers. A strict consensus protocol executed by the servers ensures that all servers receive all the requests in the same order. Output of each request from all the servers is then compared to mask the result of any faulty server(s). The advantage of the active approach is the transparency and speed of failure recovery. The client does not see, nor does it need to know, that there are faulty servers. However, the overhead associated with running the strict consensus protocol tends to slow down the response to client requests, which makes the traditional active replication model unsuitable for real-time applications where timing predictability is a necessity.

Consider a scenario in which a client poses a query concerning an object to an active replication service. If the value of the object is static and does not change as time advances, then the replicas would reach a consensus on the value of the object at some point of time and can return the exact value of the object that is agreed upon by all correct replicas. If the value of the object changes with time, the replicas can still manage to give the client an exact value of the object provided that the frequency of changes is small such that there is enough slack between two successive updates of the object for the replicas to reach a consensus on the value of the object. Now, suppose the object's value changes frequently such that the slack between two successive updates is not enough for the system to reach an agreement. Then it is unclear what the correct behavior would be for the replicas. Since the data could go through many changes as the query is spreading among

the servers, each replica may hold a different value for the object when the query is received at each replica. This would introduce confusion as to what, if anything, should be sent to the client. Traditional solutions to this problem involve either holding back the response until the servers reach a consensus or letting the servers take a vote and sending the value that the majority servers agree on to the client.

However, the methods adopted by traditional active replications in the above situation are inadequate. The flaw of the traditional consensus approach is that if the frequency of changes of data values is high, then there may not be enough slack between any two successive updates for the consensus protocol to run to completion. Hence, the servers may never reach a consensus (provided that data loss is not permitted). Moreover, the time the consensus protocol takes would delay the response to clients, which could lead to violation of timing constraints imposed by the target real-time applications. The problem for the second approach is that an answer is not guaranteed in a real-time environment because the data are continuously changing and there may not exist any single value that is agreed upon by a majority of servers.

The solution to the above problem lies in the relaxation of the requirement that the value returned must be a copy of the object that is agreed upon by all correct replicas. Since many applications do tolerate some staleness of data, a value that not all correct processes agree on could still be acceptable for these applications, provided that such a value is within some limited range of the copies on all correct processes. Hence, we could relax the criteria for the traditional consensus agreement: instead of requiring all correct processes to agree on an exact value of the object, we rather relax the requirement so that all correct processes agree on a limited range of values for the object. We call this kind of consensus *weak consensus*. In such a scenario, a replica can send a response to the client as long as the value is within the acceptable range, and the client can then simply pick any of the responses (if many) it receives as a correct answer. Apparently, this solution avoids the costly consensus protocol and the vote process.

This paper presents a temporal consensus model that relaxes the criteria of the traditional consensus agreement from the time domain to facilitate fast response to client requests and reduced system overhead in maintaining the redundant components. Based on the temporal consensus model, we construct a real-time active replication service that achieves a balance between fault tolerance and timing predictability in active schemes. The rest of the paper is organized as follows: the next section described the concept of temporal consensus. Section 3, 4 and 5 introduce three variations of temporal consensus: strict, sequential, and release temporal consensus. Section 6 presents the development of a real-time active replication. Section 7 addresses implementation issues followed by a performance. analysis in Section 8. Section 9 gives some example applications that can take advantages of the temporal consensus model and the real-time active scheme. Section 10 summaries the related work. Section 11 is our conclusion.

2 Temporal Consensus

Weak consensus is a relaxation of the traditional consensus problem such that instead of requiring that all correct processes agree on an exact value of the concerned object, the requirement for consensus is weakened to that all correct processes agree on a limited range of values for the object. This relaxation is possible because in many real-time applications, different data values for the same object are acceptable provided that the difference between these values are bounded

within some given range. Therefore, we do not have to require that the data copies of the same object are exactly the same on all correct replicas. Two values that are within some bounded range from each other are said to be consistent with each other. The "range" used in the weak consensus model can be specified on a variety of domains. It can be the value domain in which case, the value of the object on each replica must be within some pre-specified bound with the values of the same object on other replicas. It can be the version domain, in which case the version of an object must be within some bound of the versions of the object on other correct replicas. It can also be the time domain, in which case the timestamp of the last update of the object on one replica must be within some bound of the timestamps of the last update for the same object on other replicas.

We choose to use the time domain for our study because we are mainly concerned with real-time applications where time is the basic element. We define a system to be in a *temporal consensus* state if the timestamps of the copies on all correct replicas for the same object are within some bound of some time units with each other. Let T_i^s denote the timestamp of the last update for object i at server s, δ denote the bound on the range of timestamp for the same object, then temporal consensus requires that the difference between the timestamps of the same object on different servers be bounded by the given constraint, i.e. $|T_i^s(t) - T_i^r(t)| \leq \delta, i = 1, 2, \ldots, m$, where m is the total number of objects in the system. This concept is illustrated in Figure 1.

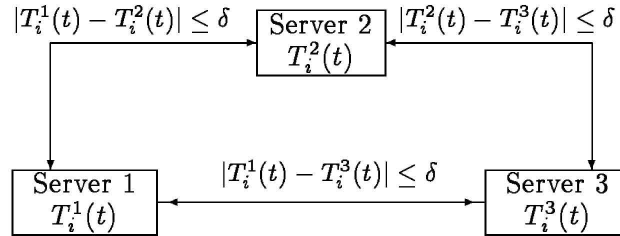


Figure 1: System View of Temporal Consensus

Borrowing from the traditional consistency models used in distributed shared memory, we can define different types of temporal consensus. Each type of temporal consensus is useful for one class of applications. Below, we introduce strict, sequential, and release temporal consensus, which corresponds to the strict, sequential, and release consistency in traditional data consistency model.

3 Strict Temporal Consensus

From the perspective of a client, strict temporal consensus is achieved if any read to a data object in the server group returns a value whose timestamp is no more than δ time units earlier than the timestamp of the last write of the same data object in the system. Let $W(x, t)$ denote the operation that writes data item x with a value of timestamp t, $R(x, t)$ denote the operation that reads data item x with a returned value of timestamp t. Then Figure 2 shows examples of legal and illegal operations under this strict temporal consensus model (x-axis represents real time line).

$P_1:\ W(x,t_1)$		$P_1:\ W(x,t_1)$	
$P_2:$	$R(x,t_2)$	$P_2:$	$R(x,t_2)$
cond: $t_2 \geq t_1 - \delta$		cond: $t_2 < t_1 - \delta$	

(a) Legal operation (b) Illegal operation

Figure 2: Strict Temporal Consensus

Below, we give an update protocol that maintains strict temporal consensus among a collection of cooperative processors. We divide update messages into two categories. One is called ordinary updates issued by the client, the other class is called forced updates issued by another server. A forced update is an update that must be done regardless of the status of the server's last update and acknowledgement is required. A forced update message can also be regarded as a consensus message in some sense. If we let t_h denote the timestamp of the last update for the concerned object at a local host, let t_m denote the timestamp of the incoming message, then a protocol that achieves the strict temporal consensus is as follows:

Every process p executes the following:
receive(m,t_m) occurs as follows:
  if m is an ordinary update message **then**
    if $t_m \leq t_h$ **then**
      discard the message
    else
      update object locally
      multicast a forced update message

  if m is a forced update message **then**
    send acknowledgement to the *sender*
    if $t_m > t_h$ **then**
      update object locally

4 Sequential Temporal Consensus

The previous section described strict temporal consensus. However, not all applications require the same level of temporal consensus. For example, in modern programming practice, people writing parallel programs do not assume any order of execution between instructions of separate programs. But the order within each program is fixed. This kind of practice, in real-time applications, can in effect be supported by an even weaker consensus semantics. Sequential temporal consensus extends the basic form of temporal consensus and addresses this class of practice. It is achieved if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program, and *each read returns a value whose timestamp is no more than δ time units apart*

from the timestamp of the value written by the latest write proceeding the read in this sequence.

$P_1: \quad W(x, t_1)$
$P_2: \quad\quad\quad R(x, t_2), R(x, t_3)$
cond: $t_2 \geq t_1 - \delta, t_3 \geq t_1 - \delta$
A possible sequence: $W(x, t_1), R(x, t_2), R(x, t_3)$

$P_1: \quad W(x, t_1)$
$P_2: \quad\quad\quad R(x, t_2), R(x, t_3)$
cond: $t_2 < t_1 - \delta, t_3 \geq t_1 - \delta$
A possible sequence: $R(x, t_2), W(x, t_1), R(x, t_3)$

$P_1: \quad W(x, t_1)$
$P_2: \quad\quad\quad R(x, t_2), R(x, t_3)$
cond: $t_2 < t_1 - \delta, t_3 < t_1 - \delta$
A possible sequence: $R(x, t_2), R(x, t_3), W(x, t_1)$

(a) Legal operations

$P_1: \quad W(x, t_1)$
$P_2: \quad\quad\quad R(x, t_2), R(x, t_3)$
cond: $t_2 \geq t_1 - \delta, t_3 < t_1 - \delta$
No possible sequential sequence available

(b) An illegal operation

Figure 3: Sequential Consensus

Figure 3 depicts some examples of legal and illegal operations under the sequential temporal consensus model. Figure (a) shows three examples of legal operations. These operations are legal because a possible sequence of execution exists for all three operations that satisfies the listed conditions and preserves the program order (operation order within each process). The operation in (b) is illegal because the only sequence that satisfies the conditions of $t_2 \geq t_1 - \delta$ and $t_3 < t_1 - \delta$ is $R(x, t_3), W(x, t_1), R(x, t_2)$ which violates the program order of processor P_2.

An alternative but equivalent view of the sequential temporal consensus is as follows: if a read returns a value whose timestamp is no more than δ units apart from the timestamps of some writes, then all subsequent reads must return a value that is no more than δ time units earlier than those of the same writes. In other words, a later read is at least as, if not more than, temporally consistent as an earlier one. In light of this alternative view, two processors need not see the same sequence of operations as long as every pair of neighboring reads in each individual sequence is temporally consistent, i.e. their corresponding timestamps are no more than δ time units apart. For example, P_1 may see a sequence of $R_1R_2R_3$ while P_2 may see the sequence of $R_2R_1R_3$. As long as R_1 is temporally consistent with both R_2 and R_3, and R_2 is temporally consistent with R_3, then sequential temporal consensus is preserved in this instance. In fact, we have the following theorem:

Theorem 1: If for every process p, and the sequence of neighboring reads observed by p, $R = \{R_1 R_2 \ldots R_n\}$, the condition $R_{i+1} \geq R_i - \delta$ holds for all $i \in \{1, 2, \ldots, n\}$, then sequential temporal consensus is achieved.

In essence, sequential temporal consensus guarantees that a later read always returns a better value in the temporal sense. It is not difficult to see that sequential temporal consensus is a relaxation of the basic form of temporal consensus since it does not guarantee that any particular read returns a value that is temporally consistent with any particular write. To ensure this, explicit semaphore or synchronization mechanisms should be used. Using the same notations we introduced before, the following is a protocol that achieves sequential temporal consensus among a collection of cooperative processors.

Every process p executes the following:
receive(m,t_m) occurs as follows:
  if m is an ordinary update message **then**
    if $t_m \leq t_h$ **then**
      discard the message
    else
      update object locally
      if $t_m > t_o + \delta$ **then**
        multicast a forced update message

  if m is a forced update message **then**
    if $t_m \leq t_h$ **then**
      discard the message
    else
      send acknowledgement to the *sender*

    if $t_m > t_h$ **then**
      update object locally

5 Release Temporal Consensus

Sequential temporal consensus does not count on any order of execution between the instructions of separate parallel programs, and does not guarantee that a particular read returns the value written by a particular write. When the order of events is essential and a guarantee is needed for a read to return the value of some particular write, synchronization operations should be used. This leads us to release temporal consensus. Under release temporal consensus, a read after an acquire operation is guaranteed to return a value whose timestamp is within δ time units from the timestamp of the value that is last written into the system before a release operation. And all processors see all accesses to synchronization variables in the same order.

Figure 4 shows some examples of valid and invalid operations under the release temporal consensus model. The operations in (a) are legal under release temporal consensus because each read after an acquire operation returns a value that is temporally consistent with the last write before a release operation. The operations in (b) are illegal because they do not meet the above condition.

$$\begin{array}{ll} P_1: & Acq(L), W(x,t_1), Rel(L) \\ \hline P_2: & \qquad\qquad R(x,t_2), Acq(L), R(x,t_3) \\ \text{cond:} & t_2 < t_1 - \delta, t_3 \geq t_1 - \delta \end{array}$$

$$\begin{array}{ll} P_1: & Acq(L), W(x,t_1), Rel(L) \\ \hline P_2: & \qquad\qquad R(x,t_2), Acq(L), R(x,t_3) \\ \text{cond:} & t_2 \geq t_1 - \delta, t_3 \geq t_1 - \delta \end{array}$$

(a) Legal operations

$$\begin{array}{ll} P_1: & Acq(L), W(x,t_1), Rel(L) \\ \hline P_2: & \qquad\qquad R(x,t_2), Acq(L), R(x,t_3) \\ \text{cond:} & t_2 < t_1 - \delta, t_3 < t_1 - \delta \end{array}$$

$$\begin{array}{ll} P_1: & Acq(L), W(x,t_1), Rel(L) \\ \hline P_2: & \qquad\qquad R(x,t_2), Acq(L), R(x,t_3) \\ \text{cond:} & t_2 \geq t_1 - \delta, t_3 < t_1 - \delta \end{array}$$

(b) Illegal operations

Figure 4: Release Consensus

The algorithm for implementing release temporal consensus is straightforward and just like the way semaphore is used. We omit it here.

6 Real-Time Active (RTA) Replication

Based on the temporal consensus model introduced above, we can construct a real-time active replication scheme to achieve a balance between fault tolerance and timing predictability.

As we said before, the single most important issue in an active replication scheme is the consensus protocol. In traditional active replication, a strict consensus must be sought for each update on each data object. This induces a very high system overhead, which consumes precious resources that can be used in processing client requests. The temporal consensus model introduced in this paper relieves the burden in two ways. First, in the temporal consensus model, not every update by client needs to be propagated to all the servers in the system. Second, any two updates whose timestamps are within some given bound from each other can be delivered in any order. Furthermore, one of such two updates can be dropped all together. The relaxation of the strict consensus requirement frees the servers from intense internal communications. Consequently, the servers can dedicate more resources to client request processing, which speeds up system response time and enables the modified active replication scheme to achieve timing predictability. Any of the three temporal consensus semantics can be used in our RTA scheme, but we chose sequential temporal consensus for the reason that it conforms to the most common programming practice in the field.

6.1 System Model and Assumptions

The real-time active replication scheme consists of a group of cooperating replicas. All replicas maintain the system state and interact with clients. Each replica updates the local copies of any object independently from other replicas and notifies such an update to the rest of group members. Unlike the traditional state-machine scheme, this prototype adapts the sequential temporal consensus protocol proposed in this paper, and decouples client read and write operations from communication within the service, i.e. the update and notification transmissions among the replicas. Figure 5 shows the architecture and server composition of the RTA replication service.

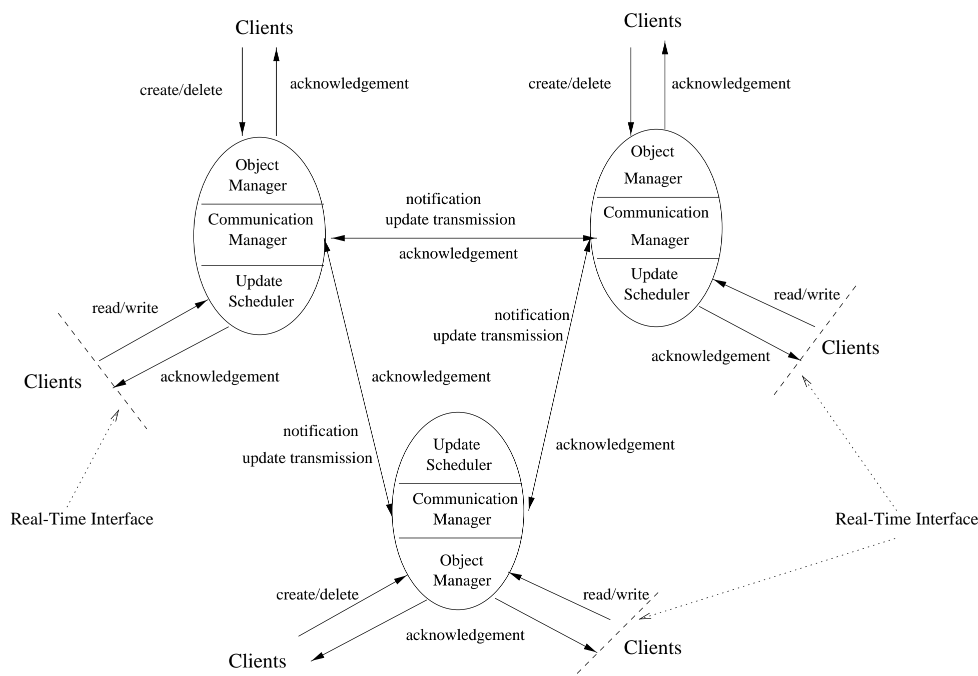


Figure 5: RTA Architecture and Server Composition

The graph only shows three replicas. In practice, there can be as many replicas as needed. Each server has an object manager which handles object registration and admission control administration that determines the merit of all client requests, an update scheduler which handles the writes of the objects in the system, and a communication manager which handles the internal communication between the servers which includes sending messages to other replicas in the system at the behest of the update scheduler or object manager. Since read and write operations do not trigger transmissions to other replicas, client response time depends only on local operations at each server which can be handled much more quickly and effectively. This allows each replica to handle a high rate of client requests while independently sending update messages to other replicas.

A client task periodically or randomly updates objects on the host it interacts with. Each replica balances the demands of the client application's processing and the maintenance of sequential temporal consensus within the system by selectively transmitting messages to other replicas. The primary goal of the temporal consensus model is to determine when updates are sent to the other

group members such that the aforementioned balance is achieved. Each host executes an admission control algorithm as part of object creation to ensure that the update scheduler can schedule sufficient update transmissions for any new object admitted such that the required consensus is achieved. Unlike client reads and writes, object creation and deletion require complete agreement among the replicas.

The system is designed to run under various system conditions and can tolerate a wide range of faults. We assume that processes can suffer crash or performance failures. (We do not assume Byzantine or arbitrary failures.) Send omissions and receive omissions are also allowed, e.g., due to transient link failures, or discard by the receiver, due to corruption or buffer overflow. Permanent link failures (resulting in network partitions) are not considered. We also assume that the environment is deterministic, i.e. the pattern and probability of a fault occurring is known or can be bounded.

6.2 Optimization

However, the discussion on temporal consensus does not minimize the average maximum temporal distance between servers in a system nor the overall CPU overhead in the system used in maintaining a given bound on such distance. The system will be better if we can optimize it from the above two aspects. We have built an optimization mechanism into the real-time replication scheme. Optimization is achieved through careful formulation of objective functions and systematic approach in the selection of update transmissions in the system. Due to space limit, we omit it here.

7 Implementation

We have developed a prototype implementation of the real-time active replication scheme running the sequential temporal consensus model. The reminder of this section describes the relevant issues of this implementation.

7.1 Environment and configuration

The replication system is implemented as a user-level x-kernel based server on the MK 7.2 microkernel from the Open Group[1]. The x-kernel is a protocol development environment which explicitly implements the protocol graph [13]. The protocol objects communicate with each other through a set of x-kernel uniform protocol interfaces. A given instance of the x-kernel can be configured by specifying a protocol graph in the configuration file. A protocol graph declares the protocol objects to be included in a given instance of the x-kernel and their relationships.

We chose the x-kernel as the implementation environment because of its several unique features. First, it provides an architecture for building and composing network protocols. Its object-oriented framework promotes reuse by allowing construction of a large network software system from smaller building blocks. Secondly, it has the capability of dynamically configuring the network software, which allows application programmers to configure the right combination of protocols for their

[1] Open Group is formerly known as the Open Software Foundation (OSF)

applications. Thirdly, its dynamic architecture has the advantage of adapting more quickly to changes in the underlying network technology [27].

Our prototype RTA system consists of four replica servers which are hosted on different machines that are connected by a 100MB fast ethernet line. Each machine has its own client application[2]. The client randomly registers and accesses data objects with/of the system through the server on which it is running. It periodically or randomly updates any object through the local copies. If the updated object is shared, then it is the responsibility of the server on which the update occurred to notify other servers of this event and to limit the inconsistency of the data among the cooperating servers. The servers communicate with each other through an IP multicast protocol and the client accesses the server using Mach IPC-based interface (cross-domain remote procedure call).

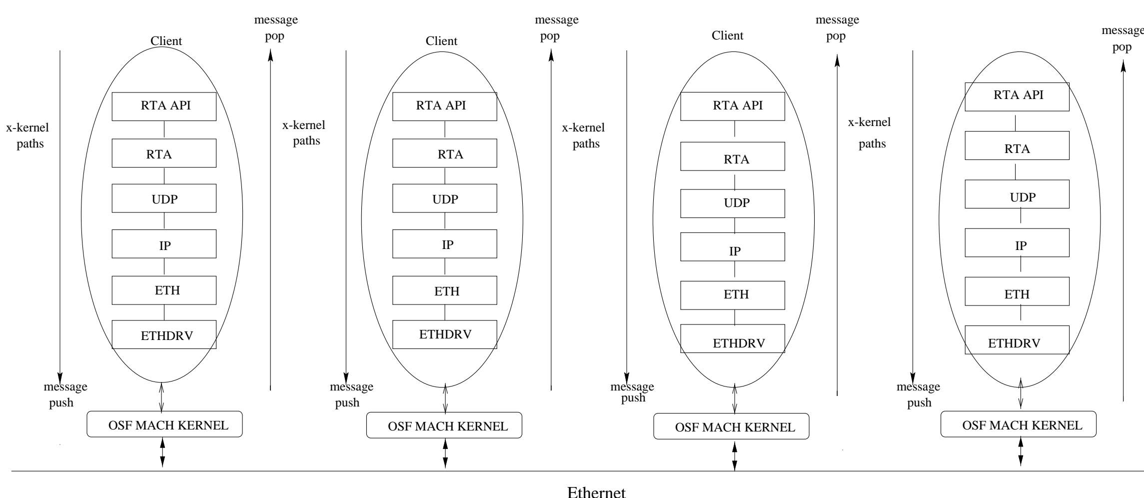


Figure 6: RTA Software and x-kernel Protocol Stack

Figure 6 shows the software and x-kernel protocol stack. At the top level is the application programming interface which connects the outside clients to the Mach server on one end and Mach server to the x-kernel on the other end. The RTA protocol sits right below the API layer. It serves as an anchor protocol in the x-kernel protocol stack. From above, it provides an interface between the x-kernel and the outside host operating system, the OSF Mach in this case. From below, it connects with the rest of the protocol stack through the x-kernel uniform protocol interface. Each host interacts with other hosts through the RTA protocol implemented inside the x-kernel protocol stack (on top of UDP as shown in Figure 6). The underlying transport protocol is UDP.

[2]The model allows stand alone clients as well as clients that run on the same machine as the server replicas. What we mean by client here is the code that reads/modifies the data objects. One can view the latter mode as replicating the application that accesses the data objects. The current implementation supports this mode of operation, i.e., the client application runs on the servers. The implementation can be easily modified to handle requests from remote clients.

7.2 Admission Control

Before a client starts to send updates of a data object to the server it is residing on, it first registers the object with the service so that the host server can perform admission control to decide whether to admit the object into the service. During registration, the client reserves the necessary space for the object on the local server and on the remote servers. In addition, the client specifies the frequency that it will update the object as well as a valid temporal range (temporal consensus constraint) allowed for the object among the cooperating servers. The frequency of update is specified as an update period p if the updates are periodic, and as a mean update rate λ if the updates are random. The host server checks the value of δ, and ℓ, the average maximum communication delay among the cooperating servers. The temporal consensus can be met if $\delta > \ell$. If the test fails, the object is rejected because the temporal consensus cannot be maintained. In such a case, the host server can provide feedback so that the client can negotiate for an alternative quality of service for the object.

If the above test passes, the host server needs to check if it can schedule necessary updates (to the other servers). In other words, it will perform a schedulability test. Specifically, the local server must be able to schedule at least one transmission within $\delta - l$ time units for each update without violating the consistency constraints of all existing objects. For example, the host server will perform a schedulability test based on the Rate-Monotonic scheduling algorithm [24] using the maximum frequency of updates supplied by the client during registration. If all existing update tasks as well as the newly added update task for object i are schedulable, the object passes the schedulability test. After both the schedulability and temporal consensus tests, the server also needs to perform a network bandwidth requirement test. The server computes the system's bandwidth requirement BD, using a number of parameters including the number of objects, their sizes, and the frequency at which these objects are updated. Then it compares the value of BD with the available network bandwidth of the system. If BD is satisfiable, the object is admitted.

In the case of message loss, only slight modification is needed in the above three tests. For the temporal consensus test, the host determines the admissability of the object based on three factors: maximum frequency of updates, the average maximum transmission delay among the server group, and the probability of message loss. Specifically, suppose the probability of message loss is p, and the probability that we want to ensure the delivery of update to remote servers is P, then the number of retransmissions needed is $\log(1-P)/\log p$. Then the minimum time interval of any two consecutive updates from the client must be at least $(\log(1-P)/\log p)\ell$ which requires $\delta \geq (\log(1-P)/\log p)\ell$. If this condition is met, then the temporal consensus constraint of the object can be met. The schedulability test is adjusted so that the system must be able to schedule $\log(1-P)/\log p$ number of transmissions for each update within next $\delta - \ell$ time units, while the bandwidth test is modified such that the bandwidth requirement is recalculated based on the increased number of transmissions that are necessary.

7.3 Update Scheduling

In our model, client updates are decoupled from internal communication among the replicas. Whenever an update occurs, the updating host needs to decide whether or not to propagate the update to the other servers in the cooperating group. The decision is influenced by the timestamp when the immediately older update (than the current one) was received and the timestamp when

the immediately older multicasts of update from this host occurred. Take whichever is greater and compare the difference of such obtained timestamp with the current timestamp. The algorithm described in Section 4 is used for the update scheduling. Specifically, if the current timestamp is more than $\delta - \ell$ apart from the aforementioned timestamp, the host multicasts the update to the servers. If message loss is to be considered, we only need to modify our algorithm to take into account the number of retransmissions that are needed. Specifically, if the current timestamp is more than $\delta - (\log(1-P)/\log p)\ell$ apart from the aforementioned timestamp, the host multicasts the update to the other servers since otherwise sequential temporal consensus of the server group could be compromised. If optimization is desired, it can be achieved by modifying the update scheduling according to the algorithms described in RTPB paper.

7.4 Failure Detection and Recovery

Failure detection and recovery are key components of a replication service, for they determine the availability of the service. Our approach uses the acknowledgement that is required for every forced update and registration messages. These acknowledgement messages serve as the heartbeats among those servers. If an update request goes unacknowledged for some period of time, it will timeout and resend the update message. If there is no response beyond a certain amount of time, the server will declare the other end dead. The local host will then remove the name of the dead server from the name file and consequently all future communication with the dead server is cancelled. A new server can join the group simply by sending a join request to the group. To simplify things, we chose to let a new server send the join request to the host whose name appears last in the name file. After receiving a join request, a server acknowledges the request and transfers state information to the new server. After receiving the state information, the new server can then invoke its own client application and start to serve requests.

8 System Performance

This section summarizes the results of a detailed performance evaluation of the RTA replication service built on the temporal consensus model. The prototype evaluation considers three performability metrics:

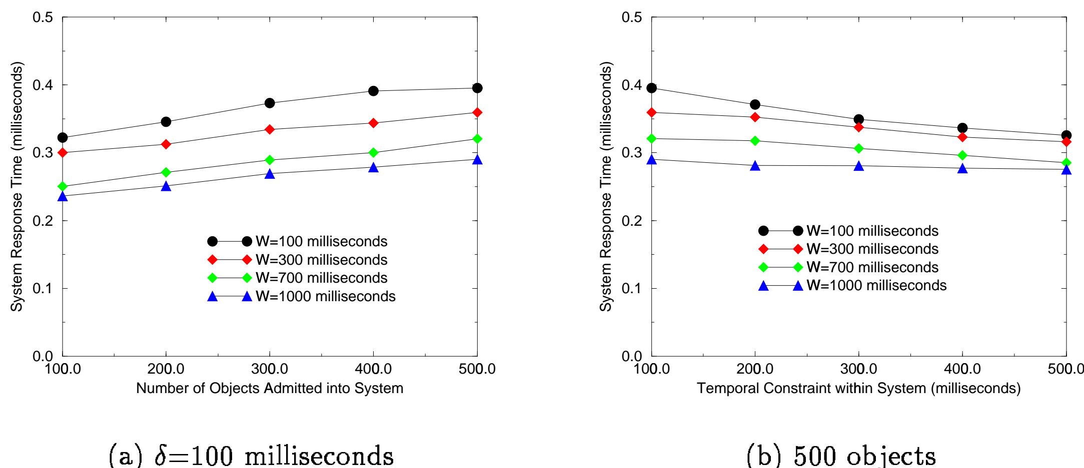
- System response time
- Inter-server temporal distance
- Probability of temporal consensus

These metrics are influenced by several parameters, including client write rate (minimum time gap between two successive updates), temporal consensus constraint, number of objects accepted, communication failure, and optimization.

8.1 System Response Time

To show that the service is suitable for supporting real-time applications, we measured the system response time to client requests. As the system runs, we let the client pose a series of queries to the service, record the time it takes for the service to respond to each request, and

average the results out to get the average response time. Figure 7 depicts the system response time. Graph (a) depicts the metric as a function of the number of objects accepted in the system for a fixed temporal constraint of 100 milliseconds while graph (b) depicts the graph as a function of the temporal constraint placed on the system for a fixed number of objects of 500. We observe a number of things from the graphs. First we see that the average client response time is fast for both graphs, ranging from 0.23 to 0.4 milliseconds, which means the system is indeed suitable for real-time applications. The fast response time was the result of the decoupling of client request processing from server updates and internal communication within the server group. The relaxation of the traditional consensus requirement to temporal consensus is also a significant contributor.



(a) δ=100 milliseconds

(b) 500 objects

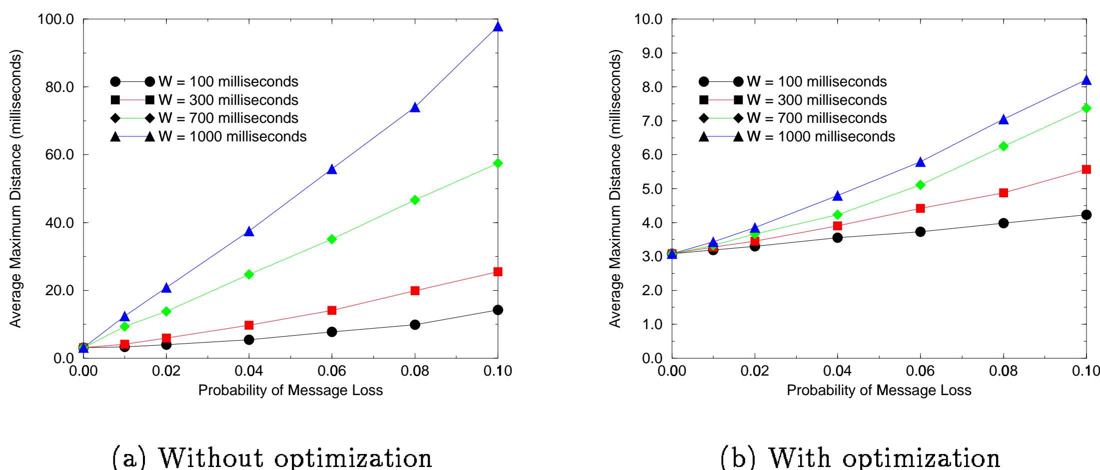
Figure 7: Average Response Time of RTA

Second, we see that the number of objects has a modest impact on the response time. As the number of objects increases, the response time increases slightly. This is due to the fact that the admission control in the system acts as a gate keeper that prevents too many objects from being admitted. Hence, only a limited number of objects can actually get in, which prevents the system performance from degrading. Such a limit is determined by the available system resources at the time of registration. Graph (b) shows that temporal constraint has an inverse impact on the response time. As the temporal constraint increases, the response time decreases slightly. This is because larger temporal constraint gives the replicas more leeway in delaying the handling of the updates while giving priority to client request processing which consequently leads to faster responses. Finally, we observe that for the same number of objects or the same value of temporal constraint, the smaller the minimum time gap (the more frequent the updates are), the longer the response time. This is because more frequent updates consume more resources and hence leave less resources for processing client requests, which consequently increase the response time.

8.2 Inter-Server Temporal Distance

One objective of the temporal consensus model is to bound data inconsistency between any two replicas in the service to a given range. Hence, it is important to measure the average maximum

inter-server temporal distance between any two servers. In this experiment, as the system runs, we take snapshots of the system at regular time intervals, find the timestamps of the objects at each replica, get the maximum difference of the timestamps for each object between the replicas, and then sum them together to derive the average maximum inter-server temporal distance. Figure 8 shows the average maximum inter-server temporal distance as a function of the average probability of message loss between any two communicating servers. The graphs depict two cases with (a) being the case where no optimization techniques are used and (b) being the case where optimization is used. Both graphs plot the average maximum temporal distance for various minimum time gaps between successive updates.



(a) Without optimization

(b) With optimization

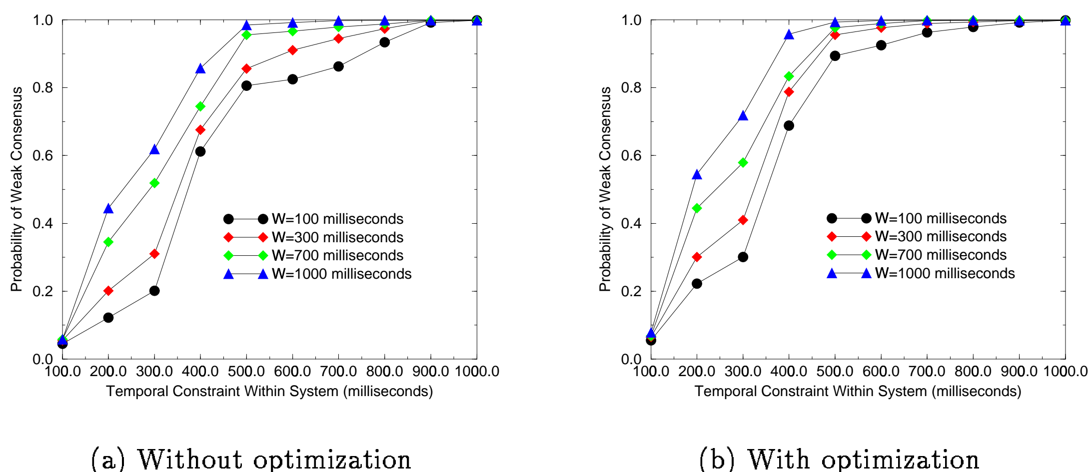
Figure 8: Inter-Server Temporal Distance in RTA

From the figures, we see that the average maximum temporal distance between any two servers is close to zero (roughly the one way message transmission time of 3 milliseconds) when there is no message loss. However, as message loss rate goes up, the distance also increases. For example, when the message loss is 10 percent, the average maximum inter-server temporal distance could exceed 100 milliseconds for graph (a) and 8 milliseconds for graph (b). In general, the distance increases as message loss rate increases or client write rate decreases. Graph (b) shows that optimization has a significant impact on the average maximum temporal distance.

We conclude from the graphs that the performance of the system is good when message loss rate is low. But as message loss rate goes up, the performance could become unacceptable to some applications if no optimization is used. The graph that uses optimization has a better performance under severe message loss because the impact caused by message loss is compensated to some extent by the optimization since it schedules as many updates to the other servers as system resources allow.

8.3 Probability of Temporal Consensus

The other important metric of the system is the average probability that the system is in temporal consensus state at any given time instant. In this experiment, we let the system run normally. Then we take snapshots of the system at regular time intervals. For each snapshot of the system state, if the replicas are in temporal consensus, we count it as a consistent snapshot. We then sum up the total number of temporally consistent snapshots and divide it by the total number snapshots of the experiment to derive our probability. Figure 9 shows the average probability of temporal consensus within the system as a function of the temporal constraint. We measured the outcome under four different minimum time gap between successive updates of objects of 100, 300, 700, and 1000 milliseconds. The difference between (a) and (b) is that (a) does not use optimization while (b) uses optimization to ensure better performance.



(a) Without optimization

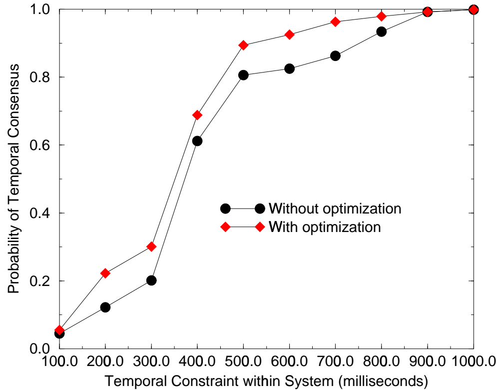
(b) With optimization

Figure 9: Average Probability of Temporal Consensus

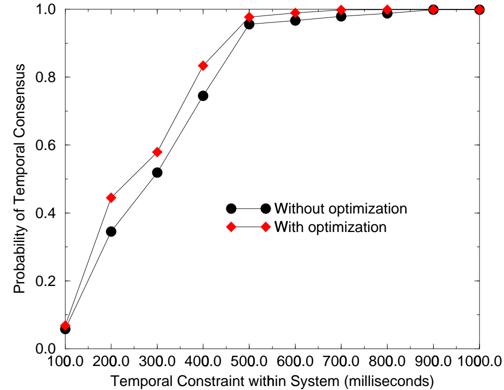
From the graphs, we see that the average probability of temporal consensus is lower for smaller temporal constraint and update time gap. This is because the smaller the temporal constraint, the harder it is to achieve temporal consensus because the message delivery takes time and the temporal constraint may not allow sufficient slack for the message to be delivered. Furthermore, the more frequent the updates are (the smaller the update time gap), the more updates need to be broadcasted to the system to achieve temporal consensus which may be difficult as the available system resources are bounded.

The comparison between (a) and (b) shows that optimization improves performance with respect to probability of temporal consensus in the system. Actually, the improvement is significant and broad based. For example, when temporal constraint is 300 milliseconds and the update time gap is 100 milliseconds, the probability with optimization is about 0.3 while the number is 0.2 when no optimization is allowed. Overall, the performance is improved by about 35%. A re-draw of the graphs in Figure 10 depicts more clearly the improvement.

But optimization or no optimization, we see that the system remains in a temporal consensus state if the temporal constraint is 600 milliseconds or larger. This is because our experiment is



(a) W=300 milliseconds



(b) W=700 milliseconds

Figure 10: Probability of Temporal Consensus

conducted on a local LAN connected by a fast Ethernet and message transmission is very reliable and fast. A temporal constraint of 600 milliseconds is sufficiently large for all servers to reach temporal consensus. We anticipate that the probability of consensus will decrease in wide area environments.

9 Applications

The temporal consensus model developed in this paper can be used in a wide variety of applications that require timing predictability yet tolerate some degree of data staleness. Below, we describe a number of such applications.

9.1 Telephone Directory System

Consider using the temporal consensus model to replicate the records of phone numbers in which a large number of servers are scattered throughout the country. If there is a change in the record, the local server changes immediately, but the other servers need not do the same. Instead, the change at the other servers can be delayed for a carefully selected δ time units. In other words, we maintain the server group in a temporal consensus state. Such relaxation is possible due to two reasons. First, the access to the updated record in a remote server is infrequent and the probability of such an access occurring within δ time units is very small. Second, in the rare case where an access to the updated record does occur within δ time units in a remote server, the return of an old value may be acceptable since such a value can be used in tracing the new value by the users provided that the old value is no more than δ time units apart from the new value.

9.2 Web Server System

In the realm of cyber space dotted with countless web servers, a user usually accesses the local server and occasionally accesses a remote server. When this happens, the client or proxy caches the document fetched from the remote server. Since a user usually accesses a remote server regularly, i.e. at fixed time of the day or week, such a system can be implemented nicely by the temporal consensus model, i.e. the web server, the clients, and the proxies can be organized into an active replication scheme running our temporal consensus model. When there is a change in the original document, the local server is updated immediately, but the remote ones do not need to be changed instantly. Instead, we allow some leeway δ for the change to be reflected in remote servers. Since users are not likely to request accesses to the updated information on remote servers very soon, such a delay is tolerable. Moreover, even if the users do request accesses to the new information from a remote server and get an old value for the requested item, the user can still use the old information or simply try to reload sometime later and the information will be updated.

In essence, the feature of the temporal consensus model in the replication scheme guarantees that cached copies will be brought to the same state as the original one within δ time units after the update at the original site. An alternative view of this is that cached copies are at most δ time units apart from the original document. One thing worth noticing is that Web cache is a special case of active replication scheme in the sense that writes only come from a single host, i.e. the web server, while reads can come to any host in the group (this special case is called the single writer active replication).

9.3 Cache System

A cache system is another good example for the application of the temporal consensus model. A cache system implemented by such a scheme focuses on providing fast response to client reads while delaying some updates. When a cached item is changed, the local cache is updated to reflect this but the remote ones do not need to do the same. Since the next access to the updated item would most likely come from local users, a delay in updating the remote cache entries would be acceptable. The δ bound on such delay for the update of remote cache entries can be computed based on the rate of access in each server, the rate of updates occurring, the distance between the replicas, and optionally the probability distribution of access in any given time period. Experimental data can be collected to evaluate the effect of these parameters on the δ bound or the effect of a chosen δ bound on system performance.

9.4 Hypothesis Test System

In a hypothesis test system, a collection of servers cooperate to provide accurate estimation of the projection trail of a traveling missile. As each radar image becomes available to the system, each server carries out its computation independent of other servers. At the end of the computation for each incoming radar image, the top ten percent of the calculated projection trails from each server is collected and fed back to the system along with the next radar image. The temporal consensus model is well suited for such an application. As the computation goes on, the servers periodically exchange information about their corresponding computation and compare their results. Any significant discrepancy is a cause of alarm. Otherwise, the system proceeds normally as long as the

results are in temporal consensus. The frequency of exchange can be influenced by the δ bound which in turn, can be adjusted accordingly based on the frequency of incoming radar images and other factors.

10 Related work

10.1 Replication models

One can structure a distributed system as a collection of servers that provide services to outside clients or to other servers within the system. A common approach to building fault-tolerant distributed systems is to replicate servers that fail independently. The objective is to give the clients the illusion of service that is provided by a single server. The main approaches for structuring fault-tolerant servers are *passive* and *active* replication. In passive replication schemes [3, 5, 7, 8], the system state is maintained by a primary and one or more backup servers. The primary communicates its local state to the backups so that a backup can take over when a failure of the primary is detected. This architecture is commonly called the *primary-backup* approach and has been widely used in building commercial fault-tolerant systems. In active replication schemes [6, 10, 15, 26, 31, 32, 34], also known as the *state-machine approach*, a collection of identical servers maintain replicated copies of the system state. Updates are applied atomically to all the replicas so that after detecting the failure of a server, the remaining servers continue the service. Schemes based on passive replication tend to require longer recovery time since a backup must execute an explicit recovery algorithm to take over the role of the primary. Schemes based on active replication, however, tend to have more overhead in responding to client requests since an agreement protocol must be run to ensure atomic ordered delivery of messages to all replicas.

Past work on synchronous and asynchronous replication protocols has focused, in most cases, on applications for which timing predictability is not a key requirement. Real-time applications, however, operate under strict timing and predictability constraints that require the system to ensure timely delivery of services and to meet cewcin consistency constraints. Hence, the problem of server replication possess additional challenges in a real-time environment. In recent years, several experimental projects have begun to address the problem of replication in distributed hard real-time systems. For example, TTP [16] is a time-triggered distributed real-time system: its architecture is based on the assumption that the worst-case load is determined *apriori* at design time, and the system response to external events is cyclic at predetermined time-intervals. The TTP architecture provides fault-tolerance by implementing active redundancy. A Fault-Tolerant Unit (FTU) in a TTP system consists of a collection of replicated components operating in active redundancy. A component, consisting of a node and its application software, relies on a number of hardware and software mechanisms for error detection to ensure a fail-silent behavior.

DELTA-4 [4] is a fault-tolerant system based on distribution that provides a reliable networking infrastructure and a reliable group communication and replication management protocols to which computers of different makes and failure behavior can plug in. It also supplies an object-oriented application support environment that allows building application with incremental levels of fault-tolerance while hiding from the programmer the task of ensuring predictability. The leader-follower model was used extensively in DELTA-4 XPA [9] in which decisions are made by one privileged replica called leader that imposes them on the others. Followers are ranked so that if a leader failure

is detected, take-overs is immediate and the new leader simply resumes executing. RTCAST [1] is a lightweight fault-tolerant multicast and membership service for real-time process groups which exchange periodic and aperiodic messages. The service supports bounded-time message transport, atomicity, and order for multicasts within a group of communicating processes in the presence of processor crashes and communication failures. It guarantees agreement on membership among the communicating processors, and ensures that membership changes resulting from processor joins or departures are atomic and ordered with respect to multicast messages.

Rajkumar [12,30] present a publisher/subscriber model for distributed real-time systems. It provides a simple user interface for publishing messages on a logical "channel", and for subscribing to selected channels as needed by each application. In the absence of faults each message sent by a publisher on a channel should be received by all subscribers. The abstraction hides a powcble, analyzable, scalable and efficient mechanism for group communication. It does not, however, attempt to guarantee atomicity and order in the presence of failures, which may compromise consistency.

10.2 Consistency semantics

The approach proposed in this paper bounds the overhead by relaxing the requirements on the consistency of the replicated data. 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 domains. In particular, a number of researchers have observed that serializability is too strict as 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 [11,14,17–23,28,29,35].

For example, a recent work [18] [20] proposed a class of real-time data access protocols called SSP (Similarity Stack Protocol) applicable to distributed real-time systems. The correctness of the SSP protocol is justified by the concept of *similarity* which allows different but sufficiently timely data to be used in a computation without adversely affecting the outcome. Data items that are similar would produce the same result if used as input. SSP schedules are deadlock-free, subject to limited blocking and do not use locks. Furthermore, a schedulability bound can be given for the SSP scheduler. Simulation results show that SSP is especially useful for scheduling real-time data access on multiprocessor systems. Song and Liu [33] defined temporal consistency in database transaction by the difference between a data item's age and its dispersion, and experimented the effect of pessimistic and optimistic concurrency control has on such temporal consistency.

Similarly, the notion of imprecise computation [25] explores weaker application semantics and guarantees timely completion of tasks by relaxing the accuracy requirements of the computation. 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 [2]. This gives the scheduler more flexibility in propagating updates to the cached copies. In the same spirit, the RTPB replication service allows computation that may otherwise be disallowed by existing active or passive protocols that support atomic updates to a collection of replicas.

11 Conclusions

The paper presented a temporal consensus model that relaxes the consensus requirement in traditional active replication schemes. The motive is to free the servers from the costly strict agreement protocol such that system efficiency and response time can be improved. Our implementation and experimental results on the real-time active replication scheme show that the proposed model indeed provides fast response with less overhead.

Avenues for future study include: extension of the model to non-deterministic environment where the pattern and probability of a fault occurring is unknown or can't be bounded, and the development of a dynamic hybrid active-passive replication scheme.

References

[1] T. Abdelzaher, A. Shaikh, S. Johnson, F. Jahanian, and K.G. Shin. Rtcast: Lightweight multicast for real-time process groups. In *IEEE RT Technology and Applications Symposium*, 1996.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

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

[4] P.A. Barrett, A.M. Hilborne, P. Verissimo, L. Rodrigues, D.T. Seaton, and N.A. Speirs. The delta-4 extra performance architecture (xpa). *Digest of Papers, 20nd International Symposium on Fault-Tolerant Computing*, pages 481–488, 1990.

[5] J.F. Bartlett. Tandem: A non-stop kernel. In *ACM Operating System Review*, volume 15, 1991.

[6] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[7] N. Budhiraja and K. Marzullo. Tradeoffs in implementing primary-backup protocols. In *Proc. IEEE Symp. Parallel and Distributed Processing*, pages 280–288, October 1995.

[8] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Primay-backup protocols: Lower bounds and optimal implementations. In *Proceedings of IFIP Working Conference on Dependable Computing*, pages 187–198, 1992.

[9] M. Chereque, D. Powell, P. Reynier, J-L. Richier, and J. Voiron. Active replication in delta-4. *Digest of Papers, 22nd International Symposium on Fault-Tolerant Computing*, 1992.

[10] F. Cristian, B. Dancy, and J. Dehn. Fault-tolerance in the advanced automation system. In *Proceedings Int'l Symp. on Fault-Tolerant Computing*, pages 160–170, June 1990.

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

[12] M. Gagliardi, R. Rajkumar, and L. Sha. Designing for evolvability: Building blocks for evolvable real-time systems. In *Proc. Real-Time Technology and Applications Symposium*, June 1996.

[13] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Janruary 1991.

[14] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In S.H. Son, editor, *Advances in Real-Time systems*, pages 463–486. Prentice Hall, 1995.

[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. In *Proceedings of the IEEE Micro*, pages 25–40, February 1989.

[16] H. Kopetz and G. Grunsteidl. Ttp - a protocol for fault-tolerant real-time systems. In *IEEE Computer*, volume 27, pages 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 *Proceedings of IEEE 14th Real-Time Systems Symposium*, December 1993.

[19] T-W Kuo and A.K. Mok. Real-time database - similarity semantics and resource scheduling. In *ACM SIGMOD Record*, March 1997.

[20] Tei-Wei Kuo, D. Locke, and F. Wang. Error propagation analysis of real-time data intensive application. In *IEEE Real-Time Technology and Applications Symposium*, June 1997.

[21] K-J Lin. Consistency issues in real-time database systems. In *Proc. 22nd Hawaii International Conference on System Sciences*, pages 654–661, January 1989.

[22] K-J Lin and F. Jahanian. Issues and applications. In Sang Son, editor, *Real-time Database Systems*. Kluwer Academic Publishers, 1997.

[23] 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 Research Center, January 1992.

[24] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[25] J.W.S. Liu, W.-K. Shih, and K.-J. Lin. Imprecise computation. In *Proceedings of IEEE*, volume 82, pages 83–94, January 1994.

[26] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[27] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

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

[29] B. Purimetla, R. M. Sivasankaran, K. Ramamritham, and J.A. Stankovic. Real-time databases: Issues and applications. In Sang Hyuk Son, editor, *Advances in Real-Time Systems*, chapter 20. Prentice Hall, first edition, 1995.

[30] R. Rajkumar, M. Gagliardi, and L. Sha. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: Design and implementation. In *Proc. Real-Time Technology and Applications Symposium*, pages 66–75, May 1995.

[31] R.V. Renesse, K.P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[32] F.B. Schneider. Implementing fault-tolerant services using the state-machine approach: A tutorial. *ACM Transactions on Computing Surveys*, 22(4):299–319, December 1990.

[33] Xiaohui Song and J.W.S. Liu. Maintaining temporal consistency: pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, pages 786–796, October 1992.

[34] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *Proc. USENIX Mach Workshop*, pages 73–82, October 1990.

[35] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transactions with temopral constraints: Exploiting data semantics. In *Proc. IEEE Real-Time Systems Symposium*, December 1996.