Accepted for publication in *Microelectronics and Reliability*, 1995.

THE EFFECT OF WORKLOAD ON THE PERFORMANCE AND AVAILABILITY OF VOTING ALGORITHMS

Muhammad A. Qureshi and William H. Sanders
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
The University of Illinois at Urbana-Champaign
Urbana, IL 61801 USA

qureshi@crhc.uiuc.edu and whs@crhc.uiuc.edu
+1 (217) 333-0345

ABSTRACT

Voting algorithms are a popular way to provide data consistency in replicated data systems. By maintaining multiple copies of data on distinct servers, they can increase the data's availability, as perceived by a user. Many models have been made to study the degree to which replication increases the availability of data, and some have been made to study the cost incurred in maintaining consistency. However, little work has been done to evaluate the time it takes to serve a request, accounting for server and network failures, or to determine the effect of workload on these measures. The effect of workload can be significant, since failures of system components are not important unless they are needed to deliver a service, and requests can force updates on data that would otherwise be outdated. In this paper, we use stochastic activity networks (SANs), a variant of stochastic Petri nets, to construct two variant models of a replicated file system, one using a static voting algorithm while the other using a dynamic voting algorithm to maintain data consistency. A Markov process representation is automatically constructed from each SAN model and is solved numerically. Specifically, we determine the availability and mean time to respond to write requests as a function of the number of replicated copies and workload offered to the system. The results illustrate that it is indeed possible to determine such measures analytically and that workload, as well as the number of copies, is an important determinant of availability and response time.

Keywords: Replicated Data Systems, Voting Algorithms, Availability, Response Time, Stochastic Activity Networks, Stochastic Petri Nets.

I Introduction

Dependable access to data is critically important to many distributed system applications. This dependability is most often achieved by replication of data on multiple nodes of a system. In this way, the failure of some subset of the nodes can be tolerated, maintaining the availability of data to a user of the system. However, increased availability often comes at a cost in performance, since a user must be provided with a consistent view of the data in spite of possible network and node failures. By consistent we mean that the user is guaranteed to see the most current replicated copy.

Replica control algorithms have been developed to provide data consistency. Many replica control algorithms are based on the idea of voting, which was first presented by Thomas [1] and later generalized by Gifford [2]. In such schemes, each node is assigned a vote v_i. Read and write quorums (numbers of votes of r and w, respectively) are then chosen such that only one write operation can be performed at any time and no read operations can be performed while a write operation is being performed. This behavior can be achieved by requiring that $r + w$ be greater than the total number of votes assigned to the nodes and w be greater than half of the total number of votes assigned to the nodes.

Algorithms which assign a fixed number of votes to each node are called "static" voting algorithms. After Gifford, several variants of the basic static algorithm were introduced to improve availability and performance (for example, see [3, 4, 5, 6]). Algorithms which assign a dynamic number of votes to each host, depending on the state of the system, have also been considered. Dynamic algorithms can increase the availability of data, since votes can be reassigned after a component failure to maintain the desired behavior (single concurrent write and no reads while a write is being performed) among the remaining functioning hosts (e.g., [7, 8, 9, 10, 11]).

Many variants of voting algorithms have been proposed, and a large number of these variants have been evaluated, either via simulation, simple probabilistic arguments, or state-space-based methods (e.g., Markov models, stochastic Petri nets). With a few exceptions, however, the evaluations have focused on determining the availability or reliability of particular algorithms, as determined by the specific algorithm and failure rates of particular system components, or the "cost" associated with implementing the voting algorithm. While these studies provided useful information concerning the subject protocols, they were limited in several respects.

First, by basing the criteria for proper operation (i.e., an available system) on the structural configuration of the system, previous evaluations do not provide a user-oriented view of availability. In particular, such evaluations judge a system as unavailable if the resources to form a quorum are not present, regardless of whether a request to form a quorum has been made. Thus the service delivered by a system would be considered improper even if no request for the service had been made. This view is contrary to that of the dependable computing community ([12]), which bases the proper/improper service criteria on the ability of a system to deliver proper service *when it is requested*. Given this view, there is growing ev-

idence (both experimental and model-based) that workload can affect fault-tolerant system dependability (e.g., see [13, 14, 15, 16]). It is therefore reasonable to assume that workload is also an important determinant of availability in replicated data systems and should be considered.

Second, most studies have been limited to studying the impact of a particular protocol and network on the availability of the data, without simultaneously considering the performance delivered by the system. Failures of system components can significantly affect the performance of a replicated data system as perceived by a user. In many applications, performance measures such as the mean response time can be useful in determining the suitability of particular voting algorithms. This fact has been recognized by several researchers (e.g., see [17, 18, 19, 20]), which underscores the importance of determining the performance as well as availability of voting algorithms.

Stochastic Petri nets can be used to evaluate the performance and availability of voting algorithms in a single model, taking into account the effect of workload. In particular, due to the abstract nature of their model primitives, stochastic Petri nets (e.g., DSPNs, GSPNs, SANs, and SRNs) can be used to represent workload-, performance-, and dependability-related characteristics of a system in a single model. Furthermore, details of voting algorithms and these characteristics can be easily represented as SPNs, resulting in a much more accurate model than is possible using simple-probabilistic or direct Markov-based approaches. A stochastic Petri net representation of a system can be converted automatically to an equivalent Markov process representation (with many more states than could be constructed by hand) and then solved using known numerical techniques. Stochastic Petri nets were first applied to the evaluation of voting algorithms by Dugan and Ciardo [21], who evaluated the effect of changing the number of copies on data availability. They did not, however, study the effect of workload on availability or determine the performance of the algorithm in the presence of faults.

In this paper, we use stochastic activity networks (SANs), a variant of stochastic Petri nets, to evaluate the performance and availability of particular static and dynamic voting algorithms, taking into account the effects of workload and component failures. Both server and network failures are considered. Our measure of performance is the mean response time of a write request to the system, accounting for the fact that a request may have to wait until the number of copies required to form a quorum become available. The results are significant for two reasons. First, they illustrate that one can indeed build accurate and detailed models to analytically evaluate the performance and availability of replicated data systems. Second, they provide interesting information regarding the performance and availability of the specific voting algorithms and hardware configuration studied. Among other things, we show that the mean response time of the voting algorithms considered decreases with an increase in the number of copies employed and an *increase* in workload. The decrease in mean response time with increasing workload is nonintuitive and is linked to the particular voting algorithms we considered.

II Network Architecture, Voting Algorithms, and Workload Model

A Network Architecture Model

We consider a networked LAN environment, where a single broadcast-capable link, i.e., "network," connects the hosts together. A host on the network behaves as either a client or a server. Each server carries a replicated data object. Clients generate requests for the replicated data, while the servers service the requests.

Furthermore, we assume that the number of replicated data objects is equal to the number of servers and each replicated data object resides on a different server. A data object becomes unavailable if either its server or the network fails. For simplicity, we assume that each replicated data object consists of a single file copy and all the data objects carry the copy of the same file. Since each server carries a replicated copy of the file, it is sufficient to monitor a server's status to keep track of the status of the file copy.

Servers and the network fail independently of each other and can fail at any time other than when the replicated file system is in reconstruction. Since reconstruction time is very short relative to the failure rates of the network and servers, it is reasonable to assume that failures do not occur during reconstruction. We assume that failure and repair times are exponentially distributed, with fixed rates. We also assume that a network failure prevents all communication on the LAN. In this case, the replicated file system becomes unavailable and cannot provide any service (including completing service of a request in the system). As soon as the network is repaired, it becomes available and once again begins processing requests, if conditions are such that it can form a quorum.

Repair starts as soon as a server or the network fails. We take a conservative approach, as suggested in [21], and always consider a repaired server as outdated. How outdated a repaired copy is depends upon the number of write requests it has missed. However, in order to model the replicated file system with a reasonably small state space, we treat all outdated copies similarly and ignore the degree to which they are outdated.

B Voting Algorithms

Many voting algorithms could have been used for this study. We consider one static and one dynamic voting algorithm similar to those considered by Jajodia and Mutchler [9]. Since they only differ in the requirement for quorum formation, we describe the static voting algorithm and highlight the differences as necessary.

The protocols make use of two types of version numbers associated with each copy of data, one "physical" and the other "logical." The *physical* version number is an indication of the

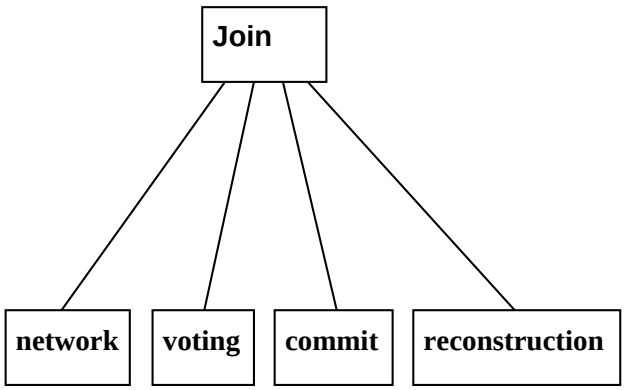

Figure 1: Composed model: *file system model*

version of the data at the server. The *logical* version number is the version that was written during the last quorum in which the copy participated. Note that the physical version number may be less than the logical version number for a copy; this indicates that the copy participated in a quorum formation but (as is possible in the protocol) did not have its data updated. Given these two version numbers, there are three classes into which a copy can fall.

The first class is where both the physical and logical version numbers indicate that the copy participated in the most recent quorum formation. These copies are called *physically current* copies. The second class is where the logical version number indicates that the copy participated in the most recent quorum, but the physical version number is less than the logical version number, indicating that the copy is out of date relative to the most recent update. We call these copies *logically current* copies. The third and final case is when the logical version number indicates that the copy did not participate in the last quorum formation. In this case, the data of the copy are also outdated, since the physical version number must be less than or equal to the logical version number. We call these copies *outdated*. Both the static and dynamic algorithms specify that a write request can be executed if there exists at least one physically current copy and the number of physically or logically current copies participating in quorum formation is greater than half of some number X. The selection of this number X is what differentiates the static and dynamic algorithms. The static algorithm specifies that X is equal to the total number of copies, while the dynamic algorithm requires that it must be equal to the number of copies that took part in the service of the last request.

More precisely, each file f has an associated version number VN which is initially set to 0 and then incremented with each write request on the file. Each copy f_i of the file f has a physical version number PN_i to keep track of the physical status of the copy and logical version number LN_i to keep track of the last quorum that copy participated in. A copy f_i of the file f is said to be physically current if $PN_i = VN$. It is logically current but not physically current if $LN_i = VN$ but $PN_i < VN$. Otherwise, it is outdated. Furthermore, each copy f_i has an associated variable SC_i to keep track of the number of copies that took part in the service of the last request. (Note that variable SC_i is only required in dynamic

voting.) Under the normal operation (when a quorum can be formed), the two phases *voting* and *commit* are executed in response to a write request.

Voting After generating a write request for a file f, the client enters the voting phase. In this phase, the client transmits messages to the servers carrying copies of the file. Each server S_i locks the local copy f_i and sends the physical and logical version numbers PN_i and LN_i back to the client. (Note that in the case of dynamic voting each server also sends variable SC_i back to the client.) After receiving messages from the servers, the client first determines the greatest logical version number among the responses it received. It then checks to see whether it received a number of responses with this logical version number greater than half of the number of copies either in the system for static voting or that took part in the service of the last request for dynamic voting. It then checks to see if at least one of the copies with $LN_i = VN$ has $PN_i = VN$. If so, then the client has obtained a physically current copy and can proceed to do the update and commit the write. If not, the client sends messages to the servers to release their locks, and the file system is forced into reconstruction.

Commit If a quorum could be formed during the voting phase, the system enters the commit phase. In this phase, the client increments the VN associated with the file f and sends the updates to all servers that responded to the request for a quorum formation. Each receiving server S_i updates its local copy f_i, makes the associated $PN_i = LN_i = VN$, and releases the lock. Also, in the case of dynamic voting, each server updates the variable SC_i equal to the number of copies that took part in the quorum formation before releasing the lock. Note that in the variant of the algorithm studied in this paper, we physically update all servers that responded to the last quorum formation request. We do this because we assume a broadcast-capable network where all servers can be updated at a cost equal to that of updating a single server. When the cost to physically update multiple servers is greater than a single server (as it is on a nonbroadcast-capable network, for example), it may not be beneficial to update all servers; the algorithm itself only requires that we increment the logical version number of more than half the servers and physically update at least one server.

Reconstruction is only invoked if the system is unable to form a quorum in the voting phase. All servers, including failed servers just repaired, are physically updated during the reconstruction phase. The write request which causes the invocation of reconstruction waits during the reconstruction phase and triggers voting again once it is complete.

C Workload Model

The workload model captures the demands placed on the data. In a real system, requests for the data may come from many sources and result in contention for the data. The file system can only service one write request at a time, so requests must be queued, and serviced sequentially. Since our goal is to understand the effect of such a workload on the protocol, it suffices to model the workload as a simple generator of requests, with an exponentially

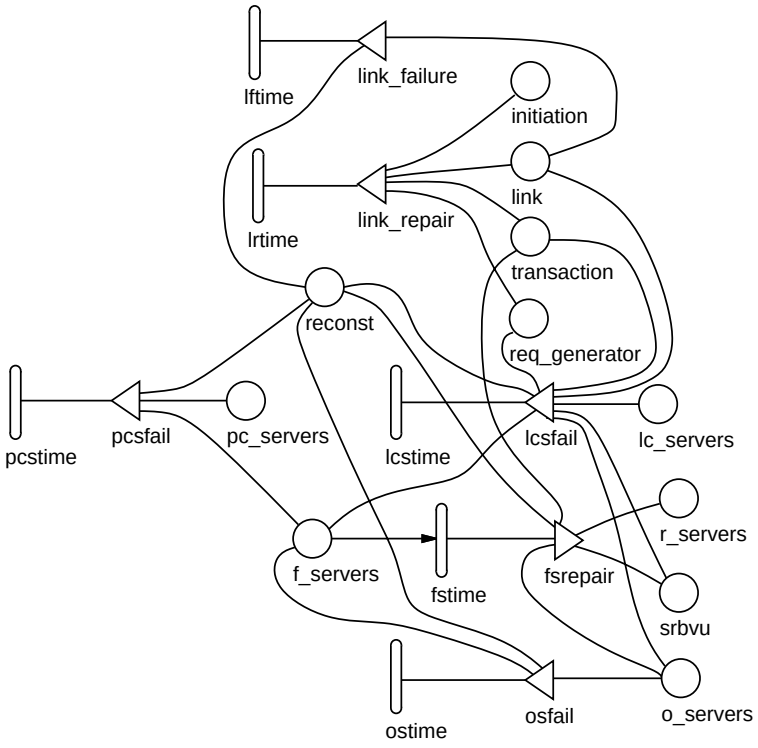

Figure 2: SAN model: *network*

distributed time between the completion of one request and the receipt of the next. This point of view simplifies the model, as there is no need to represent the clients individually. Furthermore, since the workload model insures that there is no more than one write request at a time in the system, we do not explicitly need to model the locking mechanism which insures that there is at most one write request in the system.

III Stochastic Activity Network Representation

We use SANs, a stochastic extension of Petri nets, to represent each replicated file system model. As argued in the introduction, SANs provide a convenient mechanism for representing algorithms and hardware, such as is considered here, and can be automatically converted into Markov processes, which are then numerically solved. Space does not permit a review of the theory of SANs, but interested readers are encouraged to consult [22, 23].

To aid in their understanding, the static and dynamic replicated file system models are divided into four submodels: *network*, *voting*, *commit*, and *reconstruction*, representing the file system architecture, the workload and the voting phase, the commit phase, and the reconstruction process, respectively. The four submodels are then joined by connecting certain of their places together to generate a complete model of the replicated file system. In SAN terminology, this model is referred to as a *composed model* [24] and is given in Figure

1. The remainder of the section describes the four SAN submodels in detail, illustrating how the static algorithm, hardware, and workload described previously are represented as SANs. During the discussion, we also highlight the minimal changes which are required to represent the dynamic algorithm.

A Network

Figure 2 is a SAN model of the network architecture of the replicated file system. It specifies precisely how server and network failures and repairs occur and the effect these failures and repairs have on the replicated file system. Tables 1, 2, and 3 define the components of the SAN. As depicted in Figure 2, the markings of places *pc_servers*, *lc_servers*, *o_servers*, and *f_servers* represent the numbers of physically current servers, logically current servers, outdated servers, and failed servers, respectively. Initially, the marking of place *pc_servers* is set to the number of servers (copies) in the system, while the markings of places *lc_servers*, *o_servers*, and *f_servers* are set to zero. Place *r_servers* is a temporary holding place for servers and represents the number of servers that are repaired while the system is in reconstruction. Place *link* represents the status of the network. As discussed in the previous section, the network is nondegradable, i.e., it either is functioning and can deliver messages or has failed and cannot provide communication. A marking of one in this place indicates the network is functioning; a marking of zero indicates that the network has failed. Four different failure situations need to be considered: failure of a physically current server, failure of a logically current server, failure of an outdated server, and failure of the network. The failure of servers and the network is modeled using input gates and timed activities. In particular, activity *pcstime* and input gate *pcsfail* represent the failures of physically current servers. Input gate *pcsfail* specifies the enabling condition for the timed activity *pcstime*. In particular, as shown in Table 1, activity *pcstime* is enabled when there is at least one physically current server and the system is not in reconstruction. All timed activities are assigned exponential rates with per hour units.

Activity *lcstime* and input gate *lcsfail* represent the failures of logically current servers, in a manner similar to that of physically current servers. However, since we allow servers to fail during the commit phase, these failures must be handled in a manner somewhat different from the failure of physically current servers. This complication need not be considered for physically current servers, since all physically current servers become logically current at the beginning of the commit phase (they have not yet been made physically current by the current write transaction) and become physically current at the end of the commit phase, since the network has broadcast capability. Servers can fail during the commit phase, however, and, if they do, cannot be made physically current. Input gate *lcsfail* thus specifies that the failure of all remaining logically current servers during the commit phase ends the phase by setting the marking of place *transaction* to zero. If the network is available, another request is now allowed, and a token is placed in place *req_generator* to indicate that it should be generated (see the SAN workload representation discussed in the next section).

Table 1: Input gate definitions for SAN model *network*

Gate	*Definition*
pcsfail	Predicate $MARK(reconst) == 0 \;\&\&\; MARK(pc\_servers) > 0$
	Function $MARK(pc\_servers) -= 1;\;\; MARK(f\_servers) += 1;$
link_failure	Predicate $MARK(link) == 1 \;\&\&\; MARK(reconst) == 0$
	Function $MARK(link) = 0;$
link_repair	Predicate $MARK(link) == 0$
	Function $MARK(link) = 1;$ $if\ (\ MARK(initiation) == 0 \;\&\&\; MARK(req\_generator) == 0$ $\&\&\; MARK(transaction) == 0\ )\ \{$ $\quad MARK(req\_generator) = 1;\ \}$
osfail	Predicate $MARK(reconst) == 0 \;\&\&\; MARK(o\_servers) > 0$
	Function $MARK(o\_servers) -= 1;\;\; MARK(f\_servers) += 1;$
lcsfail	Predicate $MARK(reconst) == 0 \;\&\&\; MARK(lc\_servers) > 0$
	Function $MARK(lc\_servers) -= 1;$ $if\ (\ MARK(transaction) == 1 \;\&\&\; MARK(lc\_servers) == 0\ )\ \{$ $\quad MARK(srbvu) += 1;$ $\quad MARK(o\_servers) += (MARK(srbvu) + 1);$ $\quad MARK(srbvu) = 0;\;\; MARK(transaction) = 0;$ $\quad if\ (\ MARK(link) == 1\ )\ MARK(req\_generator) = 1;\ \}$ $else\ MARK(f\_servers) += 1;$

Table 2: Activity time distributions for SAN model *network*

Activity	*Distribution*	*Rate*
pcstime	exponential	$0.01 * MARK(pc\_servers)$
fstime	exponential	$1 * MARK(f\_servers)$
lftime	exponential	*0.002*
lrtime	exponential	*2*
ostime	exponential	$0.01 * MARK(o\_servers)$
lcstime	exponential	$0.01 * MARK(lc\_servers)$

Table 3: Output gate definitions for SAN model *network*

Gate	*Definition*

fsrepair	*if* ($MARK(reconst) == 1$) {
	$MARK(r\_servers) += 1;$ }
	else {
	if ($MARK(transaction) == 1$) {
	$MARK(srbvu) += 1;$ }
	else {
	$MARK(o\_servers) += 1;$ } }

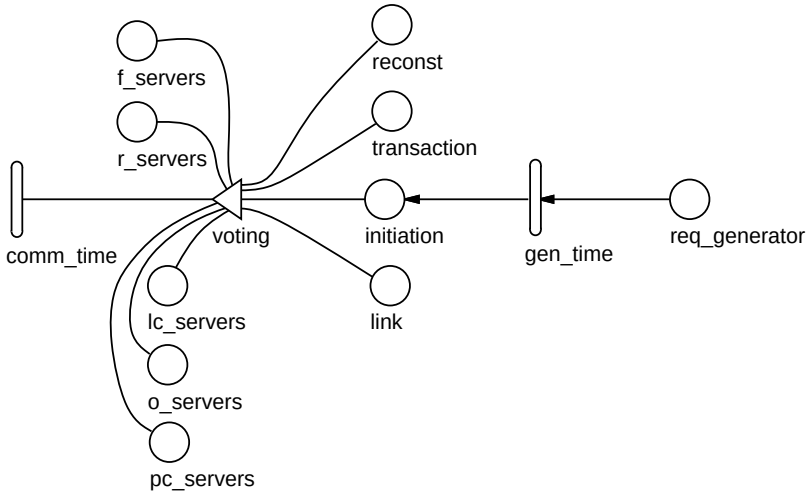

Figure 3: SAN model: *voting*

The failure of the network is represented by activity *lftime* and input gate *link_failure*. As specified in Table 2, we suppose that network failures are exponentially distributed with a rate of 0.002 per hour. Failure of the network will block the completion of service of an ongoing transaction and hence negatively impact its response time.

Server and network repairs begin as soon the respective component has failed and are modeled by activities *fstime* and *lrtime*, respectively. Output gate *fsrepair* specifies what happens when a server is repaired (upon completion of activity *fstime*). Specifically, if a server is repaired during the reconstruction phase (i.e., the marking of place *reconst* is one), it joins the set of repaired servers that need to made physically current (i.e., the marking of place *r_servers* is incremented). If instead the server is repaired during the commit phase, then it is temporarily held in the place *srbvu* so that it can be distinguished from the available servers which formed the quorum in the voting phase and hence are not made physically current. Finally, if the server was repaired when the system was not in the reconstruction or commit phase, it joins the set of outdated servers (marking of *o_servers* is incremented).

Activity *lrtime* specifies that the time to repair the network is exponentially distributed and has a repair rate of two per hour. Input gate *link_repair* specifies what happens when the network is repaired. If a transaction was blocked due to the failed link, it can now continue.

B Workload and Voting

Figure 3 shows the SAN model of the workload and voting phase of the algorithm. Table 4 gives the definitions of the components of the SAN. We first consider how the workload is implemented using SAN components. Recall that as specified in the previous section, we model the time between completion of one request and the initiation of the next as an exponentially distributed random variable. Activity *gen_time* represents this exponential time and is enabled whenever place *req_generator* contains a token. (The initial marking of *req_generator* is one.) Completion of *gen_time* removes the token from *req_generator*, disabling the start of service of a second request. Completion of the request returns the token to place *req_generator*, once again enabling activity *gen_time*. The rate of activity *gen_time* is varied to see the effect of workload on system availability and mean response time. Note that by disabling activity *gen_time* during the processing of a request, we have implicitly insured that at most one request will be serviced by the system at a time, as is required by the protocol.

Completion of *gen_time* places a token in place *initiation* and signifies that the voting phase of the algorithm should begin. At this point, the client attempts to form a quorum by broadcasting the request to all servers. As shown in Table 4, quorum formation will be attempted if there is a request to do so ($MARK(initiation)==1$) and the network is able to provide communication ($MARK(link) ==1$). The time required to broadcast a message to all the servers asking them to participate in the quorum and to receive replies from the nonfailed servers is represented by activity *comm_time*. The rate associated with the timed activity *comm_time* ($45000000/(MARK(pc\_servers)+MARK\ (lc\_servers)+MARK$

Table 4: Input gate definitions for SAN model *voting*

Gate	*Definition*
voting	Predicate $MARK(initiation) == 1 \;\&\&\; MARK(link) == 1$
	Function $MARK(initiation) = 0;$ $if\ (MARK(pc\_servers) > 0\ \&\&$ $(MARK(pc\_servers) + MARK(lc\_servers)) >$ $(MARK(o\_servers) + MARK(f\_servers)))\ \{$ $MARK(transaction) = 1;$ $MARK(lc\_servers) += MARK(pc\_servers);$ $MARK(pc\_servers) = 0;$ $MARK(lc\_servers) += MARK(o\_servers);$ $MARK(o\_servers) = 0;\ \}$ $else\ \{$ $MARK(reconst) = 1;$ $MARK(r\_servers) += MARK(pc\_servers);$ $MARK(pc\_servers) = 0;$ $MARK(r\_servers) += MARK(lc\_servers);$ $MARK(lc\_servers) = 0;$ $MARK(r\_servers) += MARK(o\_servers);$ $MARK(o\_servers) = 0;\ \}$

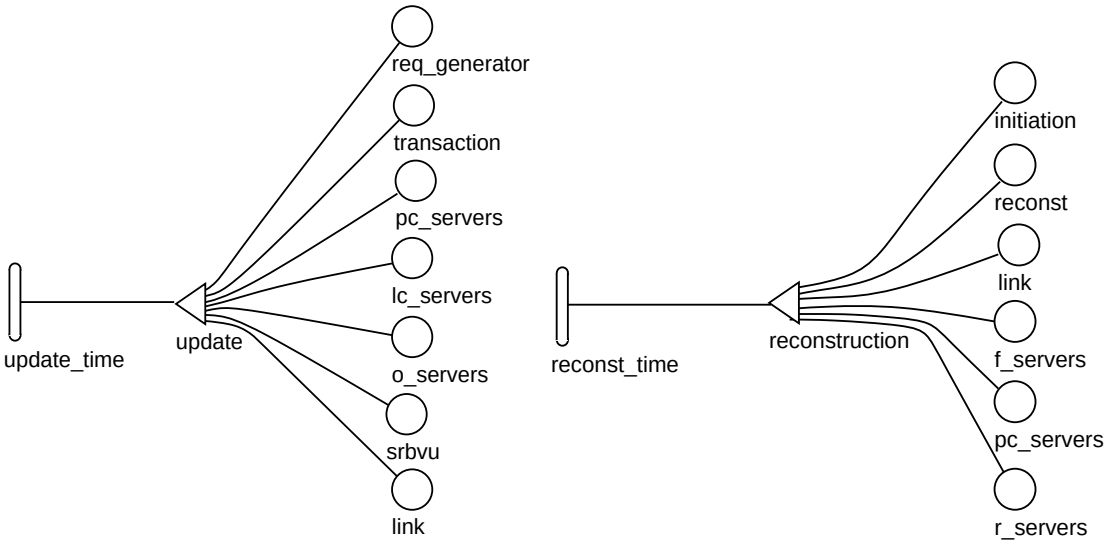

Figure 4: SAN models: *commit* and *reconstruction*

Table 5: Input gate definitions for SAN model *commit*

Gate	*Definition*
update	Predicate $MARK(transaction) == 1 \;\&\&\; MARK(link) == 1$
	Function $MARK(pc\_servers) += MARK(lc\_servers);$ $MARK(lc\_servers) = 0;$ $MARK(o\_servers) += MARK(srbvu);$ $MARK(srbvu) = 0;$ $MARK(transaction) = 0;$ $MARK(req\_generator) = 1;$

$(o\_servers)+1.0))$ is calculated by considering a 10 Mbps (megabits per second) network and a mean message size (exponentially distributed) of 100 bytes for both read and reply messages. After the client receives the reply messages from the nonfailed servers, the function associated with gate *voting* is executed. If the number of servers which are logically current is greater than half the total number of servers and at least one of these servers is physically current, then the attempt to form a quorum is a success in static voting. However, for dynamic voting, the number of logically current servers is required to be greater than half of the number of servers that took part in servicing the last request. To do this, an additional place is added to the SAN in Figure 3 to keep track of the number of servers that took part in servicing the last request. Regardless of the type of algorithm, in the case of success, the client progresses to the commit phase of the algorithm. Otherwise, the file system is forced into reconstruction. In both situations, the token is removed from the place *initiation*. The commit phase is triggered by placing a token in place *transaction*, while the reconstruction phase is initiated by placing a token in the place *reconst*.

C Commit

The left SAN in Figure 4 and Table 5 model the commit phase of the algorithm. The commit phase of the algorithm is entered if a quorum was successfully formed during the voting phase. In this case, all physically current servers become logically current, since the write request in the system has not yet been executed. Placement of a token in place *transaction* during the voting phase signifies that this indeed is the case and enables activity *update_time*. The time associated with the activity *update_time* corresponds to the time to broadcast the update message to all servers that responded to the request to form a quorum and the time required to write the update to the disk at all such servers. Since we consider a network with broadcast capability, the time associated with the activity *update_time* does not depend on the number of such servers and can be calculated by adding a single expected transmission time and disk write time. The rate assigned to *update_time* $(3600/(1000*(8/10^7+1/(2\times10^6))))$ corresponds to a 10 Mbps network, a 2 MBps (megabytes per second) disk, and a mean update message size of 1000 bytes.

Table 6: Input gate definitions for SAN model *reconstruction*

Gate	*Definition*
reconstruction	Predicate $MARK(reconst) == 1$ && $MARK(f\_servers) == 0$ && $MARK(link) == 1$
	Function $MARK(pc\_servers) += MARK(r\_servers);$ $MARK(r\_servers) = 0;\ \ MARK(reconst) = 0;$ $MARK(initiation) = 1;$

The completion of the timed activity *update_time* signifies the completion of the commit phase and triggers the execution of the function associated with the input gate *update*. As can be seen from Table 5, the function first updates the logically current servers then changes the status of the servers which were repaired during the update phase (and collected in place *srbvu*) to *outdated*. Recall that during the commit phase, the SAN collects the repaired servers in place *srbvu* so that they can be distinguished from the available servers which took part in voting. The marking of the place *transaction* is then set to zero, and the activity representing the time between requests to the replicated file system is enabled by placing a token in the place *req_generator*.

D Reconstruction

The right SAN in Figure 4 and Table 6 illustrate the model for the reconstruction phase. Recall, as discussed in the previous section, that the system will be forced into reconstruction if a quorum cannot be formed during the voting phase. This action is represented in the SAN by having input gate *voting* (in the submodel *voting*) place a token in place *reconst*. When this occurs, the algorithm specifies that all failed servers are to be repaired and then all servers are to be made physically current. Input gate *reconstruction* thus specifies that activity *reconst_time* not be enabled until all the failed servers are repaired. Furthermore, in the case of dynamic voting, the marking of the place, which keeps track of the number of servers which participated in servicing the last request, is set to the number of copies in the system.

After they are repaired, *reconst_time* is enabled and specifies the reconstruction time. When reconstruction is completed (*reconst_time* completes), all servers are marked as physically current, and a token is placed in place *initiation* (to initiate another round of voting) so that the write request which forced reconstruction can now be serviced.

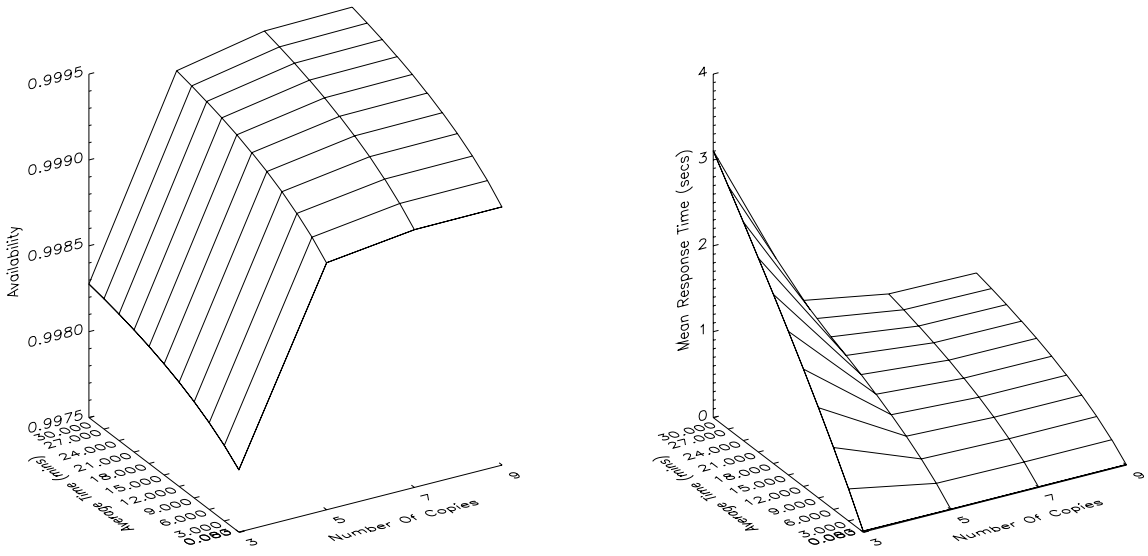

Figure 5: Effect of increasing the number of copies and decreasing workload on availability and response time using static voting

IV Results

Given the SAN models described in the previous section, the stochastic process representations for the static and dynamic models are automatically constructed using *UltraSAN* [23]. The construction procedure results in a continuous-time, discrete-state ergodic Markov process for each set of input parameter values. Depending on the number of servers considered, the state-space sizes for the generated models range from 66 (3 copies) to 375 (9 copies) states and from 120 (3 copies) to 2640 (9 copies) states, respectively. The resulting Markov processes can then be solved (via *UltraSAN*) using successive over-relaxation. Solution time is quick, on the order of a few seconds, so we can evaluate the algorithm for a wide variety of parameter values. In particular, we determine the availability and mean response time of the system to write requests when:

1. The mean time between the completion of one request and the arrival of the next takes on values from .0833 to 30 minutes (denoted as "Average Time" in Figure 5).
2. The number of copies (i.e., servers) takes odd values from 3 to 9.

Availability can be determined directly from the model by measuring the fraction of time the system is not in reconstruction. Mean response time is determined by calculating the average number of requests in the system (since at most one request can be in the system, the average number of requests in the system is equivalent to the probability that there is one request in the system), the average arrival rate of requests to the system, and applying Little's result.

A Static Voting

The left graph in Figure 5 depicts the effect of the change in workload and number of copies on availability. It shows that either an increase in the number of copies or a decrease in workload or both increases availability.

We consider first the increase in availability as the number of copies is increased for a fixed workload. This trend is well understood and has been reported in the context of many voting algorithms. To understand it in terms of the SAN model, consider the different structural configurations the system can be in. By *configuration*, we mean a distribution of servers among physically current, logically current, outdated, and failed classes. Each configuration can be categorized as proper or improper, depending on whether a quorum can be formed in the configuration. Since each server carries a single copy of the replicated file, an increase in the number of copies implies an increase in the number of servers. Each additional server demands more server failures before the system enters an improper configuration. As shown by the model, the number of proper configurations increases, and, furthermore, the probability of an incoming request finding the system in a proper configuration increases. Availability thus increases with an increase in the number of copies.

Looking at the graph, one can see that the increase in availability is significant when the number of copies is increased from 3 to 5 but becomes negligible when the number of copies is increased more than 5. Specifically, for a mean time of 0.0833 minutes between the completion of one request and the arrival of the next, the increase in availability is 1.08×10^{-3} when the number of copies is increased from 3 to 5, while the increase in availability is on the order of 10^{-5} when the number of copies is increased from 5 to 7 or 7 to 9.

The increase in availability with a decrease in workload is not so well known or intuitive. Recall that the system is available whenever it is not in reconstruction. A decrease in workload could in principle push the availability of the system in two different directions. On one hand, since the system is forced into reconstruction only if a request arrives that cannot be serviced, a decrease in rate of requests would imply that the rate at which events occur which can *possibly* cause the system to go into reconstruction decreases, and hence availability might increase. On the other hand, a longer time between requests implies that it is more likely that a server fails and is repaired between requests, leaving it in an outdated state and unable to help in forming a quorum. Thus a decrease in workload implies that the probability of entering reconstruction on a per arrival basis increases (in other words, the "possibility" of reconstruction becomes more probable for each request that occurs). Thus availability might decrease with decreasing workload.

Which trend dominates depends on the particular workload, failure rates, and repair rates considered. For the range of model parameters considered, the first trend dominates, and, as seen in left graph of Figure 5, availability increases with decreasing workload. Specifically, for the failure and repair rates considered, the increase in availability is on the order of 10^{-4} over the range of workload studied. Furthermore, the increase in availability with the

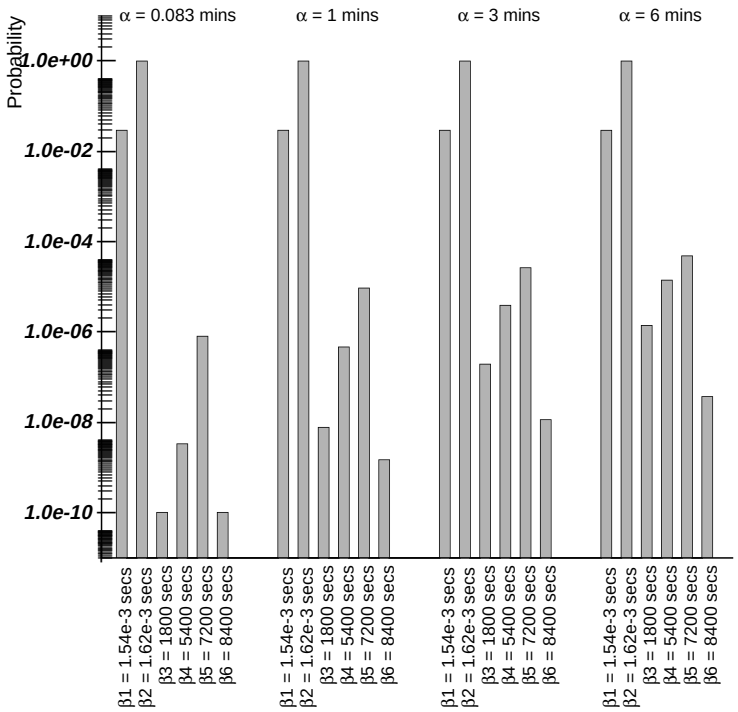

Figure 6: Distribution of the probability of being in structural configurations with different conditional mean response times

increase in time between requests is slightly more significant for higher numbers of copies. In particular, for an increase in the mean time between the completion of one request and the arrival of the next from 0.0833 to 30 minutes, the increase in availability for the number of copies equal to 3, 5, 7, and 9 is 4.15×10^{-4}, 4.51×10^{-4}, 4.90×10^{-4}, and 4.96×10^{-4}, respectively.

The right graph in Figure 5 illustrates that the mean response time decreases with an increase in the number of copies but increases with a decrease in workload. Recall that the mean response time is defined to be the average time it takes the system to respond to a write request, including the time waiting for the system to become available, if it is forced into reconstruction.

The decrease in response time with an increasing number of copies can be attributed to the same mechanism that caused an increase in availability with an increase in the number of copies. Simply put, an increase in the number of copies increases the probability that the file system is in a proper configuration. Since proper configurations have a much shorter conditional mean response time (on the order of milliseconds as compared to thousands of seconds, if reconstruction must be invoked), this increase results in a decrease in mean response time with an increasing number of copies. Note that this argument relies on the fact that the expected response time in proper configurations does not increase with an increasing number of copies. This is true for our scenario (since updates can be done via a broadcast) but would not be true if updates must be done in a serial fashion on the network.

In this case, depending on values given to model parameters, mean response time might increase with an increasing number of copies. Further work must be done to determine exactly when this will be the case.

As was the case with availability, the decrease in mean response time is greatest when the number of copies is increased from 3 to 5 and is still considerable when the number of copies is increased from 5 to 7, but the decrease becomes negligible when the number of copies is decreased from 7 to 9. Specifically, for an average time of 3.0 minutes between the completion of one request and the arrival of the next, the decrease in the response time for an increase in the number of copies from 3 to 5, 5 to 7, and 7 to 9 is 0.195 seconds, 0.012 seconds, and 4.82×10^{-4} seconds, respectively.

The right graph in Figure 5 also illustrates the nonintuitive result that, for a fixed number of copies, the average response time increases with a decrease in workload. This fact can be understood by considering the second trend discussed when considering the effect of workload on availability: the probability an incoming request finds the system in an improper configuration increases with decreasing workload. In particular, recall that as the mean time between the completion of one request and the arrival of another increases, the probability that a host will become outdated, by failing and being repaired, increases. Higher probabilities of configurations with more outdated hosts lead to higher probabilities of improper configurations. Requests that arrive when the system is in an improper configuration take much longer to be serviced, since the system must first undergo reconstruction.

This relationship can be seen quantitatively, by looking at the probability of seeing different expected response times conditioned on being in particular structural configurations. Figure 6 shows this for a system with three replicated copies. On this histogram, the parameter α indicates a specific workload (mean time between completion of one request and an arrival of the next), and the β indicates possible expected response times, conditioned on being in particular structural configurations. Note that it is easy to calculate an expected response time conditioned on being in a structural configuration, since the configuration defines precisely the set of steps the transaction must take to be serviced. Since the expected time to complete each of these steps is known, the total expected response time, conditioned on being in each structural state, can be calculated. $\beta 1$ and $\beta 2$ correspond to configurations where service is proper, while $\beta 3$ to $\beta 6$ correspond to configurations that are improper. The graph shows that the probabilities associated with improper configurations increase slightly with decreasing workload. But even this slight increase in probability significantly affects the (unconditioned) expected response time, since the conditional expected response times within these configurations are so large.

B Dynamic Voting

The results obtained for dynamic voting suggest that availability and mean response time increase either with an increase in the number of copies or a decrease in the workload. These trends are similar to the one we have discussed for static voting and also stem from the

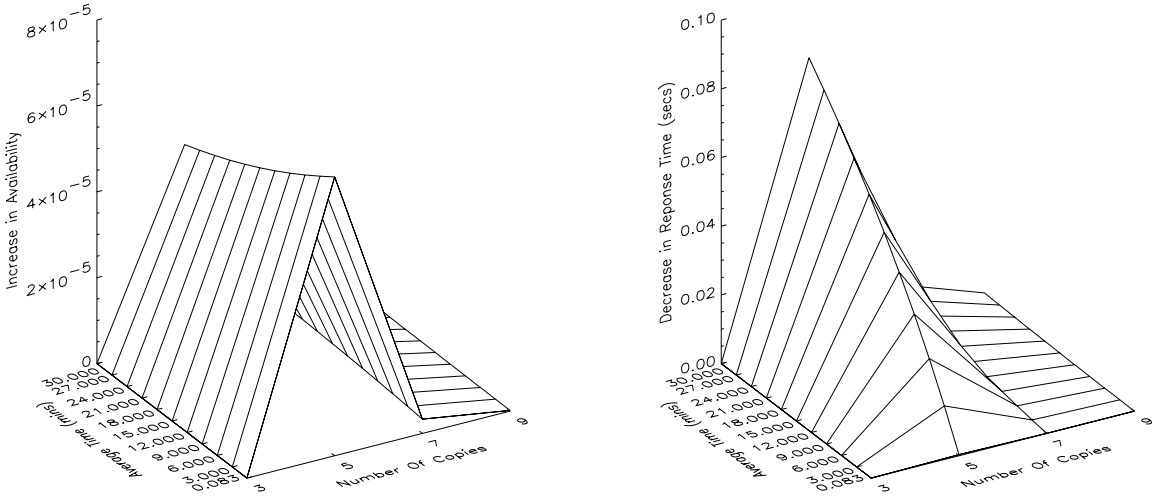

Figure 7: Increase in availability and decrease in response time when the static voting is replaced by dynamic voting

same reasons. However, compared to the results for static voting, the results for dynamic voting show a higher availability and a lower mean response time for a given workload and a fixed number of copies. The reason for higher availability with dynamic voting has already been explained in many previous papers. Due to lack of space, we will not elaborate here.

The right graph in Figure 7 shows a decrease in mean response time when the static voting algorithm is replaced by the dynamic voting algorithm in a replicated file system. This behavior can be explained by realizing that for a given number of copies certain configurations which are improper for the static voting algorithm become proper for the dynamic voting algorithm. We name such configurations as *dual-behavior* configurations. The static voting invokes reconstruction when an incoming request finds the system in any of the dual-behavior configurations. In contrast, the dynamic voting does not require reconstruction for dual-behavior configurations. As the graph shows, the average response time decreases for number of copies equal to 5, 7, or 9 when the static voting is replaced by the dynamic voting. However, for the number of copies equal to 3, the number of proper configurations remains the same for both static and dynamic voting algorithms. Therefore, there is no change in the response time.

Furthermore, by looking more carefully at the graph in Figure 7, one notices that the maximum decrease in the response time is for the number of copies equal to 5. This behavior can be understood by realizing that for higher numbers of copies the probability of finding the system by an incoming request in any of the dual-behavior configurations decreases since the system requires more failures to go into a dual-behavior configuration. Hence, the decrease in response time is smaller for high numbers of copies.

V Conclusion

As previously stated, the results of this paper are significant for two reasons. First, they illustrate that it is indeed possible to construct analytic models of voting protocols that account for the effect of workload and protocol parameter values on performance and availability. Second, they provide interesting results regarding the effect of workload on the performance and availability of the particular voting algorithm studied. Regarding the algorithm itself, the results obtained for availability were as expected. The availability increases either with an increase in the number of copies or a decrease in the workload or both. However, the results obtained for average response time were surprising. The average response time decreases with an increase in the number of copies, while it increases with a decrease in the workload.

While the results are interesting in their own right, they also suggest further work that could be done. For example, factors other than workload and number of copies may also affect a voting algorithm's performance and availability. Specifically, if communication on the network is nonbroadcast, then an important factor would be the number of physical updates which are performed within the commit phase. Seemingly, the availability would increase with increasing numbers of updates, but the behavior of mean response time is not intuitive. Interesting performance/availability tradeoffs may exist that could be investigated using a model similar to the one presented in this paper. Finally, the results suggest that comparisons could be made between different network architectures and voting schemes, taking into account the effect of workload on performance and availability.

REFERENCES

[1] R. H. Thomas, "A majority consensus approach to concurrency control for replicated databases," *ACM Transactions on Database Systems*, vol. 4, no. 2, pp. 180–209, 1979.

[2] D. Gifford, "Weighted voting for replicated data," *Proceedings of 7th ACM symposium on operating systems principle, Pacific Grove, CA*, pp. 150–162, 1979.

[3] D. Agarwal and A. E. Abbadi, "The generalized tree protocol: an efficient approach for managing replicated data," *ACM Transactions on Database Systems*, vol. 17, no. 4, pp. 689–717, 1992.

[4] S. Y. Cheung, M. Ahamad, and M. H. Ammar, "Optimizing vote and quorum assignments for reading and writing replicated data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 3, pp. 387–397, 1989.

[5] J. Paris, "Voting with witnesses: a consistency scheme for replicated files," *Proceedings of 6th Int. Conf. on Data Engineering, Cambridge, MA*, pp. 606–620, 1986.

[6] R. Renesse and A. S. Tanenbaum, "Voting with ghosts," *Proceedings of the 8th IEEE Int. Conf. on on Distributed Computing Systems*, pp. 456–462, 1988.

[7] D. Barbara, H. Garcia-Molina, and A. Spauster, "Increasing availability under mutual exclusion constraints with dynamic vote reassignment," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 394–426, 1989.

[8] M. Herlihy, "Dynamic quorum adjustment for partitioned data," *ACM Transactions on Database Systems*, vol. 12, no. 2, pp. 170–194, 1987.

[9] S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 230–280, 1990.

[10] J. Paris and D. D. E. Long, "Efficient dynamic voting algorithms," *Proceedings of 4th IEEE Int. Conf. on Data Engineering, Los Angeles, CA*, pp. 268–275, 1988.

[11] J. Tang and N. Natarajan, "A scheme for maintaining consistency and availability of replicated files in a partitioned distributed system," *Proceedings of the 5th IEEE Int. Conf. on Data Engineering*, pp. 530–537, 1989.

[12] J. C. Laprie, "Dependability: Basic concepts and terminology," *Dependable Computing and Fault-Tolerant Systems*, vol. 5, 1992.

[13] S. E. Butner and R. K. Iyer, "A statistical study of reliability and system load at slac," *Proceedings of 10th IEEE Int. Symposium on Fault Tolerant Computing*, pp. 207–209, 1980.

[14] X. Castillo and D. P. Siewiorek, "Workload, performance, and reliability of digital computing systems," *Proceedings of 11th IEEE Int. Symposium on Fault Tolerant Computing*, pp. 84–89, 1981.

[15] R. K. Iyer, D. J. Rossetti, and M. C. Hseuh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Transactions on Computer Systems*, vol. 4, no. 3, pp. 214–237, 1986.

[16] J. F. Meyer and L. Wei, "Analysis of workload influence on dependability," *Proceedings of 18th IEEE Int. Symposium on Fault Tolerant Computing, Tokyo*, pp. 84–88, 1988.

[17] M. Ahamad and M. H. Ammar, "Performance characterization of quorum-consensus algorithms for replicated data," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 492–496, 1989.

[18] S. Y. Cheung, M. H. Ammar, and M. Ahamad, "The grig protocol: A high performance scheme for maintaining replicated data," *Proceedings of 6th Int. Conf. on Data Engineering, Cambridge, MA*, pp. 438–445, 1990.

[19] A. Kumar, "Performance analysis of a hierarchical quorum consensus algorithm for replicated objects," *Proceedings of 10th IEEE Int. Conf. on Distributed Computing Systems*, pp. 378–385, 1990.

[20] L. E. Moser, V. Kapur, and P. M. Melliar-Smith, "Probabilistic language analysis of weighted voting algorithms," *Proceedings of ACM Conf. on Measurement and Modeling of Computer Systems*, pp. 67–73, 1990.

[21] J. B. Dugan and G. Ciardo, "Stochastic petri net analysis of a replicated file system," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 394–401, 1989.

[22] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," *Proceedings of International Workshop on Timed Petri Nets, Torino, Italy*, pp. 106–115, 1985.

[23] W. H. Sanders, W. D. Obal, M. A. Qureshi, and F. K. Widjanarko, "The UltraSAN modeling environment," *Performance Evaluation*, vol. 24, pp. 89–115, 1995.

[24] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 25–36, 1991.