Distributed election in computer networks

Conference Paper · April 1988

DOI: 10.1109/PCCC.1988.10097 · Source: IEEE Xplore

Chung-Ta King
National Tsing Hua University
207 PUBLICATIONS **998** CITATIONS
SEE PROFILE

Lionel M. Ni
University of Macau
501 PUBLICATIONS **20,896** CITATIONS
SEE PROFILE

DISTRIBUTED ELECTION IN COMPUTER NETWORKS[†]

Chung-Ta King,[*] *Thomas B. Gendreau,*[**] *Lionel M. Ni*[*]

[*]Department of Computer Science
Michigan State University
East Lansing, Michigan 48824

[**]Department of Computer Science
Vanderbilt University
Nashville, TN 37235

ABSTRACT: *Election* in a computer network is an operation which selects one process from among a group of processes, perhaps residing in different computers in the network, to perform a particular task. It is found that many problems in computer networks exhibit the behavior of election or can be solved by means of election. Examples include mutual exclusion, load balancing, fault recovery, group joining, and replicated data updating. In this paper, an election is characterized by 1) the *capacities* obtained by the evaluation of a *criterion function* at each candidate process and 2) an agreement reached by all processes in the group to elect the *master* process. A number of election algorithms are presented based on various conditions and environments, including process fault behavior, process timing relations, and communication subsystem supports. These algorithms allow all fault-free processes to elect one and only one process as the *master*, and, by changing the definition of the criterion function, they can be applied to a variety of applications in a computer network.

1. INTRODUCTION

The ultimate goal of a computer network is to provide the users with a unified service environment, in which resource allocation and access are transparent to the users. *Election* is an operation in which one process from among a group of processes is singled out to perform a particular task. In providing unified service environment, many problems in computer networks exhibit the behavior of election or can be solved through election. Examples include mutual exclusion, replicated data updating, group joining, load balancing, and crash recovery.

There has been a substantial amount of work dealing with election on a logical or physical ring network [5, 8]. The problem is to find the process having the maximum value in the ring and the issues considered include synchronous versus asynchronous and uni-directional versus bi-directional communication. Election in arbitrary networks is studied in [6], where two election algorithms: *Bully* and *Invitation algorithm* are proposed. Recently, an election algorithm for distributed clock synchronization is reported [7]. An unreliable network is assumed, in which messages may be lost or delayed and processes can have *fail-stop faults*.

In this paper, we abstract the concept of election to define a very general style of computation in computer networks. The model to describe the election algorithms will be presented in Section 2. The election problem is then defined in Section 3, and algorithms to perform elections under synchronous systems (Section 4), partial synchronous systems (Section 5), and asynchronous systems (Section 6) are given. Finally, applications of the election are discussed in Section 7 and our conclusion is given in Section 8.

2. AN FSM MODEL

A *computation* in a computer network consists of a number of processes, possibly residing on different machines, which solve a given problem through message-based coordination and communication. A model is used to characterize the coordination of processes in a computation. The model consists of a set of assertions, which expresses the properties and conditions that the computation must obey as a whole, and a set of *finite state machines* (*FSM*), which describes the behavior of individual processes. State transitions are triggered by *events*. Three possible events are message arrival, clock time-out, and process failure. Each FSM describes one process and is expressed as follows:

$M = (S,I,O,\delta,s_0,P_F)$, where
$S = P \times V$ = the set of states of the FSM
$I = (\Sigma \times D \times U_V) \cup x \cup F$ = the set of input events to the FSM
$O = (\Sigma \times D \times U_V) \times c$ = the set of outputs from the FSM
$\delta : S \times I \rightarrow S \times O$ = the state transition function
s_0 = the starting state, $s_0 \in S$
P_F = the set of final phases, $P_F \in P$
and
$V = (v_1, v_2, \ldots)$ = *variable vector* of the FSM
P = the set of phases of the FSM
Σ = alphabet of the messages
D = the set of process id of neighboring processes
U_V = the set of all subvectors of V
x = time-out status
F = the set of process failure status
c = clock value

Note that both normal and faulty behaviors of processes can be specified in the model, which makes fault analysis more easier. A clock or timer is associated with each process. Setting a timer can be viewed as sending a message to the *timer process*, and a time-out is just a message (or interrupt) from the timer process.

3. THE ELECTION PROBLEM

Let Q denote the group of processes participating in the election and $n = |Q|$ be the number of processes in Q. A subset of processes in Q will be designated as *candidates*, Q_c, and the election can be characterized by: (1) A *criterion function* $f_i(.)$ evaluated by each candidate process $p_i \in Q_c$ to obtain its *capacity* v_i; *(2) An agreement reached by every process in Q to elect a master* from Q_c based on the capacities. Two sets of processes are identified at the end of an election:

$Q_m = \{ p \in Q \mid p$ assumes itself to be the master $\}$
$Q_{mc} = \{ p \in Q \mid$ there exists $q \in Q$ such that q assumes p to be the master $\}$

We say that an election algorithm satisfies the *deadlock-free property*, if, at the end of the election, there is at least one process elected ($|Q_m| > 0$). An election algorithm satisfies the *uniqueness property*, if, at the end of the election, all processes elect the same process as the master ($|Q_{mc}| = 1$). There are other properties which are specific to applications. For example, the *fairness property*, which states that a candidate will eventually be elected, is very important in mutual exclusion or group joining.

The possibility of process failure requires consideration of the following aspects:

(1) *Fault model*: The faults of a process may range from simple *fail-stop* faults, where a faulty process will stop processing and keep silent, to

[†] This research was supported in part by the State of Michigan REED project and in part by the DARPA ACMP project

Byzantine faults, where a process may act in an arbitrary and malicious manner [9]. Intermediate fault models are possible.

(2) *Requirements under a certain fault model*: Failure of processes causes the properties defined above very hard to hold. Thus, in a Byzantine environment, for example, the deadlock-free and uniqueness properties are have to be relaxed as follows:
- *Deadlock-free Property*: If the elected master is fault-free, then it will assume itself to be master.
- *Uniqueness Property*: All fault-free processes will elect the same process as the master.

Finally, assumptions of the election algorithms are listed as follows:

(1) *Communication subsystem*: A *communication subsystem* collectively refers to the lower layer protocols which provide communication services to the layer where the election is running. A communication subsystem supports *reliable broadcast* if (a) every broadcast message is delivered to all receivers within some known time bound, T; (b) all broadcast messages from all senders are delivered in the same order; and (c) every broadcast message is either delivered to all receivers correctly or not delivered to any of them at all [1, 2].

(2) *Process behavior*: Assumptions related to the behavior of a process include its fault model and the time it spends in replying a message. If this time is negligible to compare with the interprocess communication time, then we say that the process do not pause in responding.

(3) *Timing*: Local clocks of all processes may be assumed to keep full synchrony, run at the same rate, differ only within a known bound, ε, or be out of synchrony completely. Other issues of the timing are when the algorithm starts and whether all processes start executing the algorithm at the same time.

4. ELECTION IN A SYNCHRONOUS ENVIRONMENT

A *synchronous system* is a system in which the communication delay T is defined, processes respond with no pause, local clocks are running at the same rate, and all processes start the election at the same time. Note that $2T$ is the maximum time one has to wait for a reply. Algorithm A introduced in this section is a synchronous algorithm which uses only one round of information exchange. The purpose of introducing Algorithm A is first to motivate our subsequent discussions and to serve as a lower bound for comparison with subsequent algorithms.

```
<Algorithm A>
  1. Conditions:
     1.1. Processes start the election at the same time.
     1.2. Processes are correct at the beginning of the election.
     1.3. Processes do not pause and use clocks of the same rate.
     1.4. Recovery time of a failed process is longer than the election time.
     1.5. Communication subsystem supports reliable broadcast.
  2. Variables:
     2.1. rd = a random value
     2.2. ck = current local clock value
     2.3. Timer = system time-out timer
     2.4. f(.) = criterion function
     2.5. v = capacity; f(.) for a candidate and -1 for a non-candidate
     2.6. ms = id of the elected master
     2.7. S = candidate list
     2.8. max(S) = the process in S with the maximum capacity
  3. FSM Descriptions:
     3.1. $M = (S,I,O,\delta,s_0,P_F)$, where
            $P = \{start,wait,stop,fail\}$
            $V = (v,ms,S)$
            $s_0 = (start,(v,ms=-1,S=\varnothing))$
            $P_F = \{stop,fail\}$
          and
            $\delta((start,(v,-1,\varnothing)),()) = ((wait,()),(all,(v)),ck+T)$
            $\delta((wait,(v,-1,S)),(j,(w)) = ((wait,(v,-1,S\cup\{(j,w)\})),(),()) \; j \in S$
            $\delta((wait,V),time\text{-}out) = ((stop,(v,max(S),S)),(),())$
            $\delta((wait,V),fault) = ((fail,V),(),ck+rd)$
            $\delta((fail,V),time\text{-}out) = ((fail,V),(rd,(rd)),ck+rd)$
□
```

Table 1. The state transition table describing Algorithm A

P	Pres. Phase	start	wait			fail
I	*Time-out?*			true		true
	recv(j,"%")?		*w*			
	Fault?				true	
V	*j* ∈ *S?*		false			
V	*S*←*S*∪%		*{(j,v)}*			
	ms←%			max(*S*)		
O	*Timer*←%	ck+T			ck+rd	ck+rd
	send(%)	all,*v*				rd,rd
P	Next Phase	wait	wait	stop	fail	fail
	Transition	1	2	3	4	5

Table 1 tabulates the state transitions of the FSM. A null entry in the condition part implies a don't care event. The symbol % in the second column is a place-holder, its content is listed in the corresponding row entry.

This algorithm uses only one round of information exchange, which takes T units of time, and requires n broadcast messages. Notice the specification of faulty behavior (Transition 5). If the time-out interval, rd, approaches infinity, then this process experiences a fail-stop fault. On the other hand, if the random message contains a legal capacity value, then this is a Byzantine behavior. Initiation of the algorithm is application dependent, and is not specified.

It is trivial to see that all correct processes will terminate at t_s+T, where t_s is the absolute time the election began. Also, due to reliable broadcasting, all correct processes will receive the same set of messages during the interval t_s through t_s+T in the same sequence. This guarantees that they have the same candidate list S at t_s+T and elect the same process ($ms = max(S)$) as the new master. If the new master is fault-free, it will also have the same candidate list S and assume itself to be the master. Thus deadlock-free and uniqueness property are assured. On the other hand, if any of the conditions listed in Algorithm A were violated, then it is easy to find counter examples which result in inconsistency in candidate lists and cause the algorithm fail.

5. ELECTION WITH PARTIAL SYNCHRONY

Unlike synchronous systems, a *partial synchronous system* is a system that all processes will start their election within the time interval t_s through t_s+T, where t_s is the time the first process starts the election. Note that, after T units of time, all processes will receive at least one election message and be informed of the election.

5.1. Algorithm B.1 — The Fail-stop Case

```
<Algorithm B.1>
  1. Conditions:
     1.1. Processes do not pause and use clocks of the same rate.
     1.2. Recovery time of a failed process is longer than the election time.
     1.3. Communication subsystem supports reliable broadcast.
  2. Variables:
     2.1. candid = true, if the process is a candidate
     see Algorithm A for definitions of other variables.
  3. FSM Descriptions:
     3.1. $M = (S,I,O,\delta,s_0,P_F)$, where
            $P = \{normal,collect,fail,stop\}$
            $V = (candid,v,ms,S)$
            $s_0 = (normal,(true/false,v,ms=-1,S=\varnothing))$
            $P_F = \{stop,fail\}$
          and the state transitions are given in Table 2.
  □
```

It is again trivial to see that the algorithm terminates and all correct processes will have the same S at the end of the election. Note that fail-stop failure will cause problem only when a failed process is elected. In such a case a time-out (with period of $2T$) is sufficient to indicate that the master has failed and a new election should begin. The whole algorithm takes at

Table 2. The state transition table describing Algorithm B.1

P	Pres. Phase	normal				collect		
I	*Time−out?*						true	
	recv(j,"%")?		*w*	*w*		*w*		
	Fault?				true			true
V	*candid=%?*	true	true	false				
V	*S←%*	*{(i,v)}*	*{(i,v),(j,w)}*	*{(j,w)}*		*S∪{(j,w)}*		
	ms←%						*max(S)*	
O	*Timer←%*	*ck+2T*	*ck+2T*	*ck+2T*	*ck+∞*			*ck+∞*
	send(all,"%")	*v*	v					
P	Next Phase	collect	collect	collect	fail	collect	stop	fail
	Transition	1	2	3	4	5	6	7

most $3T$ and $O(n)$ broadcast messages. The worst case ($3T$) happens when one process starts the election at t_s+T and finish at t_s+3T.

5.2. Algorithm B.2 — The Malicious Case

When processes start the election at different times, a malicious process may cause state inconsistency by delaying the sending of some messages such that different processes receive the same message at different phases and make different decisions. Through reliable broadcasting, Algorithm B.2 uses $2T$ to exchange capacities, and another $2T$ to ensure the consistency of all S's. Suppose there are at most t faulty processes, then the algorithm is as follows:

<Algorithm B.2>

1. Conditions:

1.1. Communication subsystem supports reliable broadcast.

1.2. Processes do not pause and use clocks of the same rate.

1.3. Recovery time of a failed process is longer than the election time.

2. Variables:

2.1. Y = the ordered list of S's received from other processes
see Algorithm A for definitions of other variables.

3. FSM Descriptions:

3.1. $M = (S,I,O,\delta,s_0,P_F)$, where
$P = \{normal, coll1, coll2, stop, fail\}$
$V = (candid, v, ms, S, Y)$
$s_0 = (normal,(true|false, v=f(.), ms=-1, S=\varnothing, Y=\varnothing))$
$P_F = \{stop, fail\}$
and the state transitions are given in Table 3.

□

The operation $S+(j,w)$ in Transition 5 denotes the appending of an entry (j,w) to the end of S. The function $el(Y)$ (Transition 9 in Table 3) obtains ms by (1) choosing a sublist Y' from Y, and (2) determining a process in Y' to be ms. Note that the uniqueness property is preserved if every correct process chooses the same Y' and ms out of their local Y. A possible procedure to compute $el(Y)$ is shown below. All elements in the lists are indexed according to the order they are received.

<Function *el(Y)*>

1. input:

1.1. $Y = ((p_1,S_1), (p_2,S_2), ..., (p_j,S_j))$, $n-t \le j \le n$, where
p_l = process id, $1 \le l \le j$
S_l = the S list received from process p_l;

1.2. $S = ((q_1,v_1), (q_2,v_2), ..., (q_k,v_k))$, where
q_l = process id of a candidate, $1 \le l \le k$
v_l = capacity of the candidate q_l.

2. define:

2.1. $Y' = ((p_1,S_1), ..., (p_{n-t},S_{n-t}))$;

3. output:

3.1. $ms \leftarrow max(S_j)$, if there exists $p_j \in Y'$ and $(q_i,v_i) \in S$, such that $p_j = q_i$, and (q_i,v_i) is the first such entry in S;
$ms \leftarrow max(S_1)$, otherwise.

□

Since there are at least $n-t$ correct processes and all messages are delivered in the same sequence, Y in every process is guaranteed to have at least $n-t$ entries (all in the same sequence). Thus, Y', as in Statement 2.1 above, is guaranteed to be the same in every correct process. Next, to select a process in Y' to be the master, we check each candidate according to the order that capacities are received to see whether its S list is also received. If so, then the corresponding S list is used to select the new master (Statement 3.1 in $el(Y)$). Otherwise, we use S_1, because the S list collected by q_1 is probably the one that is least influenced by malicious processes.

Algorithm B.2 takes $5T$ to complete the election and uses $O(2n)$ broadcast messages. Note that it is not necessary to transmit the whole set of S in the second round of information exchange, because all processes receive all messages in the same sequence. A number indicating the size of local S will be enough.

5.3 Malicious Case in Unorder Broadcasting System

To find election algorithms under unordered broadcasting environments, algorithms developed for the *Byzantine General's* problem can be adopted. Byzantine agreement, in its simplest form, is an agreement on either 0 or 1, with all processes starting with either 0 or 1. In order to use Byzantine algorithms, we must transform the election problem, in which processes have diversed capacities initially, into that of reaching agreement on common initial data.

Algorithm B.2 performs the transformation by exchanging S in each process to form a vector Y and reaching agreement on Y [10]. Nevertheless, due to malicious behavior, even these Y lists are not the same for all correct processes. What we need is another level of transformation that decides

Table 3. The state transition table describing Algorithm B.2

P	Pres. Phase	normal				coll1			coll2			fail
I	*Time−out?*						true			true		true
	recv(j,"%")		*init,w*	*init,w*		*init,w*			*vect,T*			
	Fault?				true			true			true	
V	*cadid?*	true	true	false								
V	*S←%*		*(j,w)*	*(j,w)*		*S+(j,w)*						
	Y←%								*Y+(j,T)*			
	ms←%									*el(Y)*		
O	*Timer←%*	*ck+2T*	*ck+2T*	*ck+2T*	*ck+rd*		*ck+2T*	*ck+rd*			*ck+rd*	*ck+rd*
	send(all,"%")	*init,v*	*init,v*				*vect,S*					*rd*
P	Next Phase	coll1	coll1	coll1	fail	coll1	coll2	fail	coll2	stop	fail	fail
	Transition	1	2	3	4	5	6	7	8	9	10	11

Table 4. The state transition table describing Algorithm C

P	Pres. Phase	normal					coll1			coll2		
I	*Time-out?*							true	true			true
	recv(j,"%")		init,*w*	init,*w*	elected	dup:*k,l*	init,*w*			elected	dup:*k,l*	
V	*candid?*	true	true	false								
	i=max(S)?							false	true			
V	*S←%*	$\{(i,v)\}$	$\{(i,v),(j,w)\}$				$S\cup\{(j,w)\}$		$\{i\}$	$S\cup\{j\}$	$S\cup\{k,l\}$	
	ms←%				*j*				*i*			
O	*Timer←%*	*ck*+2*T*	*ck*+2*T*	*ck*+5*T*	*ck*+3*T*	*ck*+3*T*		*ck*+3*T*	*ck*+3*T*			
	send(all,"%")	init,*v*	init,*v*						elected	dup:*i,j*		
P	Next Phase	coll1	coll1	wait1	wait2	wait3	coll1	wait1	coll2	coll3	coll3	master
	Transition	1	2	3	4	5	6	7	8	9	10	11

(a)

P	Present Phase	coll3				wait1			wait2			wait3
I	*Time-out?*			true	true			true			true	true
	recv(j,"%")	lected	dup:*k,l*			elected	dup:*k,l*		elected	dup:*k,l*		
V	*i=max(S)?*			true	false							
V	*S←%*	$S\cup\{j\}$	$S\cup\{k,l\}$	$\{i\}$								
	ms←%			*i*		*j*						
O	*Timer←%*			*ck*+3T	*ck*+3T			*ck*+2T				*ck*+3T
	send(all,"%")	dup:*i,j*		elected					dup:*j,ms*			
	send(sys,"%")							help!				
P	Next Phase	coll3	coll3	coll2	wait1	wait2	wait3	help	wait3	wait3	stop	wait1
	Transition	12	13	14	15	16	17	18	19	20	21	22

(b)

which entry (p_i,S_i) in Y is going to be incorporated into Y'. This is nothing but a 0 or 1 choice, and can be solved by Byzantine algorithms.

6. ELECTION WITH UNRELIABLE BROADCASTING

An unreliable communication subsystem may lose messages or deliver erroneous messages. This can cause the following problems: (1) process failure and message loss sometimes can not be distinguished, (2) any number of candidates (including none) may assume themselves to be elected, (3) different processes may elect different candidates, and (4) some processes may not be aware of the election. Our approach here is to reduce the number of competitors in each successive round by allowing only those duplicated masters to compete in the next round. As will be shown later, the possibility that the algorithm may fail is very small.

<Algorithm C>

1. **Conditions:**
   1.1. Processes do not pause and use clocks of the same rate.
2. **Variables:**
   Same as in Algorithm B.1.
3. **FSM Descriptions:**
   See Table 4a and 4b.

□

In the each round of the election, duplicated masters is detected by receiving messages from more than one master (Transitions 9,12). Should this happen, a warning message, *"dup: k,l"*, is immediately broadcast to inform all others and a new round is necessary. On the other hand, if no duplicated master announcements are received, a candidate will stop and become the new master (Transition 15). Each round of the election will take *3T* to complete, because processes will start a new round within *T* and it takes *2T* to get a response. Note that, in Transition 18, a non-candidate may not receive anything during an interval (*3T* in this case). In our algorithm, an error message (*"help!"*) is returned to the operating system or the user (*sys*) to indicate this situation.

The efficiency of Algorithm C can be characterized by the number of rounds to accomplish the election. Given that a process will miss a broadcast message (due to message error or loss) with probability p, we can find the expected number of rounds of the election to be

$$E[N \mid X_0=a] = \frac{P_{a,a} + \sum_{i=1}^{a-1}(E[N \mid X_0=i]+1)\times P_{a,i}}{1-P_{a,a}} \qquad (1)$$

where $\{X_0=a\}$ is the event that there are a candidates at the beginning of the election, and $P_{i,j}$ is the probability that j candidates will continue to the next round given that there are i candidates at the beginning of the round. Detailed derivation is given in the Appendix. Figure 1 shows the expected number of rounds ($E[N]$) versus the number of candidates (a) at the beginning of the election. It can be seen that with the probability of message loss less than 0.3 the expected number of rounds is no more than 3 (9T).

7. APPLICATIONS OF ELECTION

The election algorithms described so far are very independent of specific applications. What makes applications of the election differ from one another is the different definitions of the criterion functions. In this section we shall concentrate on defining the criterion function for different applications.

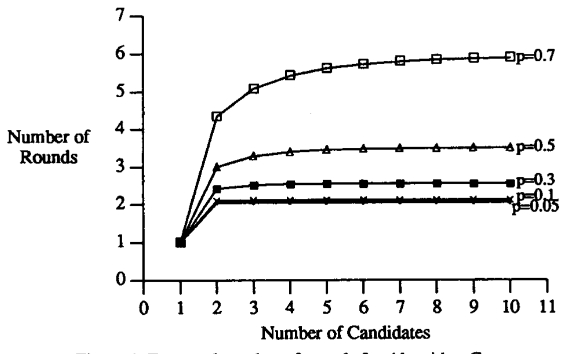

Figure 1. Expected number of rounds for Algorithm C

1. *Group Server*: In a computer network, servers which manage a replicated resource form a *group server* [3]. To provide a unified service environment, the group server should choose the most appropriate server by itself, and one possible way to choose is through election. An election is initiated by a user request and capacities are evaluated according to such criteria as load and location.
2. *Load Balancing*: A load-balancing algorithm using election differs from other algorithms [4,11] in that all the interested processors compete and elect the processor that should execute a new job. The distributed nature of election increases the system's ability of tolerating fault. The criterion function is defined by the load of the processor.
3. *Replicated Data Updating*: All the servers managing a replicated data set form a group server. To prevent data inconsistency, only the process which has gained control on a majority of the replicated data can perform the updating. This can be accomplished through an election. All processes which request the updating become the candidates.
4. *Crash Recovery*: In a group server, if a server crashed in the middle of service, then a new server should be found to continue the service. Again this can be done through election. Another possible situation would be that the whole network is partitioned into several subnetworks. Then an election should be performed to determine a master process to handle the whole recovery process [6]. In this case the process id will be sufficient to serve as the capacities.
5. *Joining Group*: When a resource is installed or repaired, the corresponding server must join the group server to provide coherent service. This can proceed in the form of election, where the new process initiates the election and becomes the candidate. It is admitted to the group if it collects a majority of *yes*'s.
6. *Mutual Exclusion*: The similarity of the election and mutual exclusion has already been pointed out [6]. Whenever a process exits a critical section, all processes intending to enter the critical section can compete through an election and the elected master can enter the critical section. Criterion function can be defined as the time since the last time a process entered the critical section and/or its priority.

8. CONCLUSION

We have presented in this paper the use of election as a style of problem solving in computer networks. The problem is defined and several election algorithms are proposed under various environments. We have shown that election is a very general problem in providing unified service environment in computer networks. By choosing appropriate criterion functions, we can apply the election technique to many different applications in a computer network. As the size of networks increases, systems are becoming more and more complicated. As a result, systems might have to adopt a hierarchical configuration, such as group computing, to handle the increasing complexity. Elections will play a fundamental role in such an environment. Our algorithms are a first step towards developing reliable algorithms for these large scale systems.

REFERENCES

[1] F. Cristian, H. Aghili, R. Strong, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *IEEE 1985 Fault-Tolerant Computing*, 1985, pp. 200-206.

[2] J.M. Chang, N.F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Trans. Computer Systems*, Vol. 2, No. 3, 1984, pp. 251-273.

[3] D.R. Cheriton, W. Zwaenepoel, "Distributed Process Groups in the V Kernel," *ACM Trans. on Computer Systems*, May 1985, pp. 77-107.

[4] D.J. Farber, "The Distributed Computing System," *Proc. Compcon Spring 73*, 1973, pp. 31-34.

[5] G.N. Frederickson, N.A. Lynch, "The Impact of Synchronous Communication on the Problem of Electing a Leader in a Ring," *Proc. 1984 ACM Symp. on Theory of Computing*, 1984, pp. 493-503.

[6] H. Garcia-Molina, "Elections in a Distributed Computing System," *IEEE Trans. on Computers*, Vol. C31, No. 1, Jan. 1982, pp. 48-59.

[7] R. Gusella, S. Zatti, "An Election Algorithm for a Distributed Clock Synchronization Program," *Proc. 6th Int'l Conf. on Distributed Computing System*, May, 1986, pp. 364-367.

[8] G. Le Lann, "Distributed Systems -- Towards a Formal Approach," *Information Processing 77*, 1977, pp. 155-160.

[9] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problems," *ACM Trans. Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 382-401.

[10] B.M. McMillin, L.N. Ni, "Byzantine Fault-Tolerance through Application-Oriented Specification," to appear in the *Proc. of 11th COMPSAC*, Tokyo, Japan, Oct. 1987.

[11] L.M. Ni, C.W. Xu, T.B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," *IEEE Trans. on Software Engineering*, Oct., 1985, pp. 1153-1161.

APPENDIX — ANALYSIS OF ALGORITHM C

Notations:

p	= probability that a process will miss a message
X_i	= the number of duplicated masters at the beginning of the i-th round
a	= number of candidates to begin the election; $X_0 = a$
N	= the number of rounds to complete an election
p_i	= probability that the master with the i-th largest capacity will continue to next round; $p_1 = 1$
$P_{i,j}$	= probability that j masters will continue to the next round given i to begin with

We shall first find $P_{i,j}$ recursively. Suppose $a = 2$, and, then, the duplicated master with the smaller capacity will continue to the next round only if it did not receive any *election* message from the other candidate. That is $p_1 = 1$ and $p_2 = p$. Thus

$P_{2,2}$ = Pr[both candidates continue to the next round] $= p_1 \times p_2 = p$

$P_{2,1}$ = Pr[p_1 continues and p_2 quits] $= p_1 \times (1 - p_2) = 1 - p$

In general, for any $P_{i,j}$, $i \geq j + 1$ we have:

$P_{i,j}$ = Pr[j canidates continue | p_i quits] × Pr[p_i quits] +
Pr[$j-1$ canidates continue | p_i continues] × Pr[p_i continues]
$= P_{i-1,j} \times (1 - p_i) + P_{i-1,j-1} \times p_i$

where $P_{2,2} = p$, $P_{2,1} = 1 - p$, and $p_i = p^{i-1}$. That is

$$P_{i,j} = \begin{cases} P_{i-1,j} \times (1 - p_i) + P_{i-1,j-1} \times p_i & \text{if } i > j \\ p^{i(i-1)/2} & \text{if } i = j \\ 0 & \text{if } i < j \end{cases}$$

We can now recursively derive $E[N \mid X_0 = a]$ as follows:

$$E[N \mid X_0 = 1] = 1$$

$$E[N \mid X_0 = a] = \sum_{i=1}^{a} E[N \mid X_0 = a, X_1 = i] \times Pr[X_1 = i]$$

$$= \sum_{i=1}^{a} (E[N \mid X_0 = i] + 1) \times P_{a,i}$$

It follows that

$$E[N \mid X_0 = a] = \frac{P_{a,a} + \sum_{i=1}^{a-1} (E[N \mid X_0 = i] + 1) \times P_{a,i}}{1 - P_{a,a}}$$