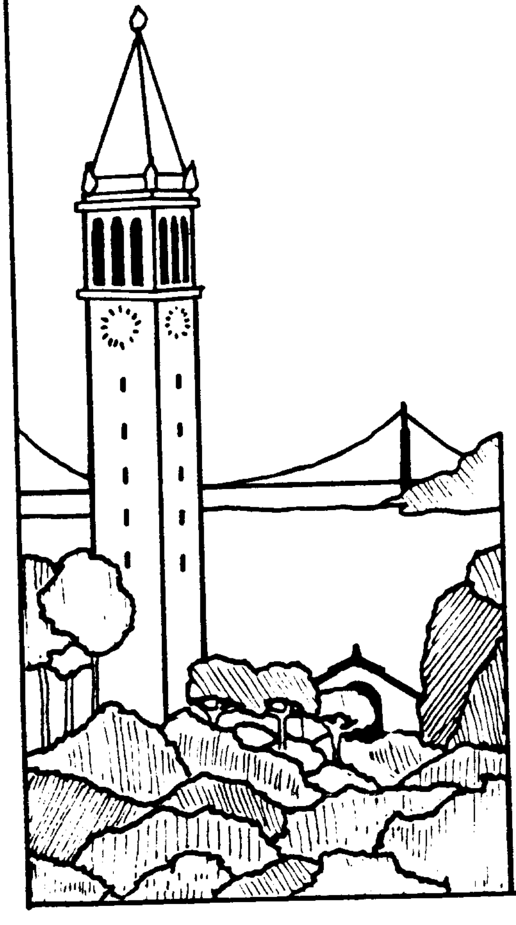
An Election Algorithm for a Distributed Clock Synchronization Program

Riccardo Gusella and Stefano Zatti



Report No. UCB/CSD 86/275

December 1985
PROGRES Report No. 85.17

Computer Science Division (EECS)
University of California
Berkeley, California 94720

An Election Algorithm for a Distributed Clock Synchronization Program

Riccardo Gusella and Stefano Zatti

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

This paper describes the election algorithm that guarantees the reliability of TEMPO, a distributed clock synchronizer running on Berkeley UNIX[†] 4.3BSD systems. TEMPO is a distributed program based on a master-slave scheme that is comprised of *time daemon* processes running on individual machines.

The election algorithm chooses a new master from among the slaves after the crash of the machine on which the original master was running. When the master is working, it periodically resets an election timer in each slave. If the master disappears, the slave whose timer expires first will become a candidate for the new master. The election algorithm covers this normal case, as well as the infrequent case where there may be two or more simultaneous candidates. It also handles the case in which, due to a network partition that has been repaired, two masters are present at the same time.

1. Introduction

In this paper we describe the election algorithm we have designed for TEMPO, a distributed network clock synchronizer for Berkeley UNIX 4.3BSD systems.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089, and by the CSELT Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency, of the US Government, or of CSELT.

[†] UNIX is a Trademark of AT&T Bell Laboratories.

TEMPO, which works in a local area network, consists of a collection of *time daemons* (one per machine) and is based on a master-slave structure[Gusella1984,Gusella1985b]. The present implementation keeps processor clocks synchronized within 20 milliseconds.

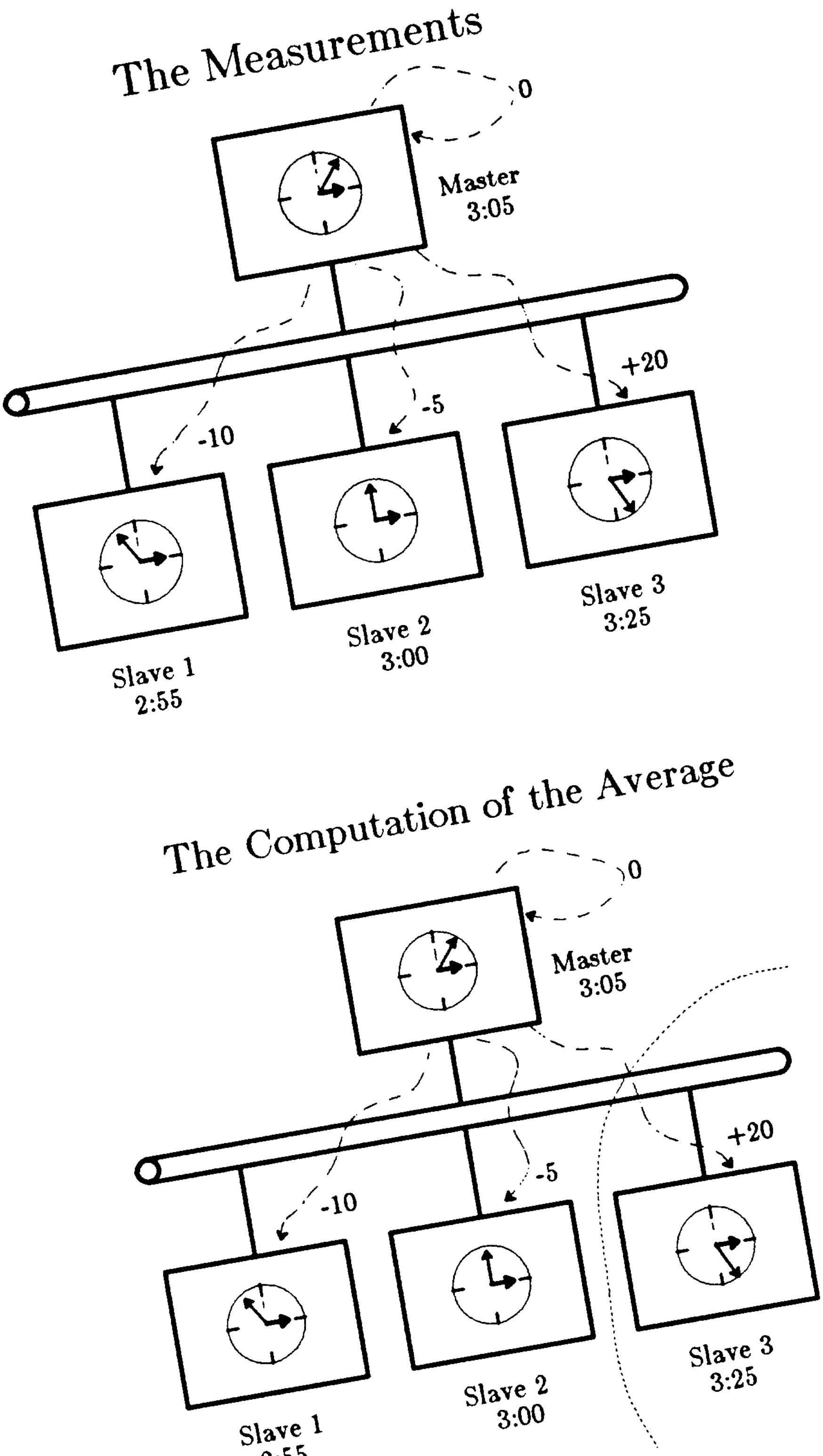
Figures 1a and 1b sketch the way TEMPO works. A *master time daemon* measures the time difference between the clock of the machine on which it is running and those of all other machines. The master computes the *network time* as the average of the times provided by nonfaulty clocks.[1] It then sends to each *slave time daemon* the correction that should be performed on the clock of its machine. This process is repeated periodically. Since the correction is expressed as a time difference rather than an absolute time, transmission delays do not interfere with synchronization. When a machine comes up and joins the network, it starts a slave time daemon, which will ask the master for the correct time and will reset the machine's clock before any user activity can begin. TEMPO therefore maintains a single network time in spite of the drift of clocks away from each other.

To ensure that TEMPO provides continuous, and therefore reliable, service, it is necessary to implement an election algorithm that will elect a new master should the machine running the current master crash, the master terminate (for example, because of a run-time error), or the network be partitioned. Under our algorithm, slaves are able to realize when the master has stopped functioning and to elect a new master from among themselves. It is important to note that, since the failure of the master results only in a gradual divergence of clock values, the election need not occur immediately.

The election algorithm must be able to perform the following tasks:

- Allow a time daemon that is a candidate for master to collect information about the system's topology.
- Mask communication failures such as loss, delay, and duplication of messages.
- Deal with network partitions.
- Withstand machine failures that occur *during* the election.

[1] TEMPO considers faulty a clock whose value is more than a small specified interval apart from the majority of the clocks of the machines on the same network. See [Gusella1984,Gusella1985b] for more details.

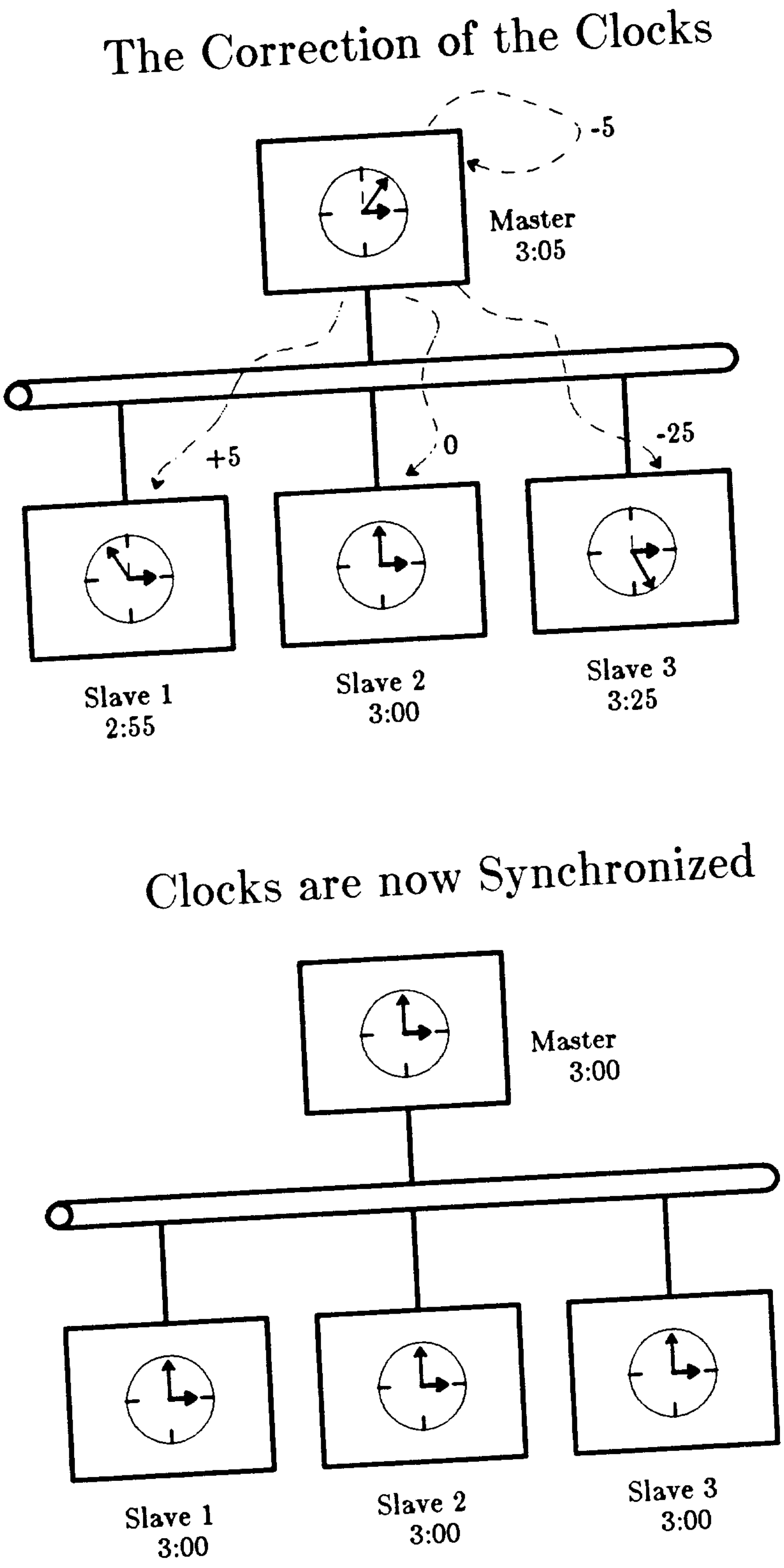


The Measurements

The Computation of the Average

$$Av = \frac{0 - 10 - 5}{3} = -5$$

Figure 1a



The Correction of the Clocks

Clocks are now Synchronized

Figure 1b

To maximize overall efficiency, the algorithm must be very simple and fast for the normal case[Lampson1983]. Yet, at the same time, it must be able to deal with abnormal cases, e.g., when two slave time daemons simultaneously become candidates for master, either by solving these problems or by simply signaling their existence to system managers.

The algorithm we designed displays the following characteristics:

- Simplicity in the normal case.
- Network traffic efficiency.
- Uniformity: all time daemons, regardless of their state (slave, master, or candidate for master), run the same software.
- Conservativeness: a delay in the choice of the new master is favored over the prospect of having two masters.

2. Some Past Solutions and Relevant Ideas

The problem of electing a coordinator in a loosely coupled distributed system has received substantial attention. Most of the solutions proposed in the past were for us only of theoretical interest, either because they were excessively complex or because the hypotheses on which they were based did not apply to our local area network environment. An election algorithm need not be algorithmically complex: it has been shown that processor agreement can be reached by exchanging a *polynomial* number of messages[Dolev1982,Frederickson1984]

A well-known paper[LeLann1977] presents an algorithm of quadratic complexity to elect a coordinator in a network of machines logically arranged as a ring whose nodes are statically ordered. All nodes must talk to the others before an agreement can be reached, and the winner will simply be the highest one in the ordering. Our election algorithm does not require machines to be logically arranged in a ring, nor does it require ordering among them. In the normal case, its message complexity is linear with respect to the number of machines.

Vitanyi[Vitanyi1984] introduced a concept that is important in our implementation: only if clocks of our processors maintain *Archimedean* times, i.e. if there is always an integer multiple of one clock value that exceeds the others,[2] do elections or, in general, any sort of distributed synchronization, become possible.

[2] See: Archimedes, "**Κυάδρατυρε ο̄φ θὲ Παράβολα**," *Syracuse Monthly*, Syracuse, 223 B.C..

Without physical time and clocks to measure it, in fact, we cannot distinguish a pausing process from one that has crashed. With an Archimedean timing system a process can use a *timer* to tell if some process on a different processor has crashed. The Archimedean time requirement is easily satisfied in practice.

The *bully* algorithm[Garcia-Molina1982] requires conditions hard to fulfill in practice, like an atomically writable storage that is preserved across system crashes or a reliable transmission mechanism, and has high complexity, i.e. $O(n^2)$. A new election is started every time a machine joins the network. Since it is unnecessary for the purposes of TEMPO to start an election whenever a new time daemon comes up, we did not employ this method. Moreover, like LeLann's algorithm, the bully algorithm requires a predefined ordering of the machines that must be known by all processes, and successively updated and maintained.

3. The Hypotheses

Our election algorithm is based on the following hypotheses:

A) Communication mechanism assumptions:

- Messages are not spontaneously generated in the network.
- Messages are not (maliciously) forged by somebody connected to the network.
- Messages are process-to-process datagrams that can be lost, delayed, duplicated, or received in altered order.
- Transmission errors are detected: a message is either received as it was sent or discarded and considered lost.
- The communication subsystem provides a broadcast mechanism that enables one to send a message to all machines on the network.

B) Assumptions about the processors:

- Fail-stop: when a processor fails, it stops running and loses all its state information. A processor never pauses and resumes working later.
- Byzantine failures are not considered.
- Machines have clocks with Archimedean time functions. This assumption allows the use of timers.

Note that no unrealistic assumptions are made on the reliability of the communication channel (messages may be lost), on the order of reception (messages may be out of order), on the transmission time, or on the permanence of the information stored in memory (core memory is completely erased in case of machine crash).

4. The Communication Protocol

All messages exchanged by time daemons have their structure defined by a specifically designed protocol called the Time Synchronization Protocol (TSP)[Gusella1985a]. TSP, built on the DARPA UDP Protocol, serves a dual purpose. First, it supports messages for the synchronization of the clocks of the various hosts in a local area network. Second, it supports messages for the election that occurs among slave time daemons when, for any reason, the master disappears.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |  Version No.  |         Sequence No.          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Seconds of Time                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Microseconds of Time                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Machine Name                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2

While some messages need not be sent in a reliable way, most communication in TSP does require reliability. Reliability is achieved by the use of acknowledgements, sequence numbers, and retransmission when message losses occur. When a message that requires acknowledgment is not acknowledged, the time daemon which has sent the message will assume that the addressee is down. This mechanism guarantees that either the message arrives at its destination, or, if no acknowledgement is received after a certain number of retransmissions, that the sender can assume the addressed time daemon is not active.

The message format in TSP (see Fig. 2) is the same for all message types, though in some instances one or more fields are not used. A message contains a message type field, a protocol version number field, a sequence number field, a field to store timing information, and a field that contains the name of the machine from which the message is sent.

5. The Election Algorithm

5.1. The Normal Case

Normally the master time daemon periodically synchronizes the clocks of all controlled machines. At start-up time, a slave time daemon randomly selects from a predefined range R a value for its election timer that is greater than the interval between synchronization messages. It is unlikely, though possible, that two slaves will have the same timer value. In the normal case we make the assumption that only one slave has selected the lowest timer value. Every time the master sends a slave a synchronization message (every four minutes in the current implementation)[3], the slave reinitializes its timer to that random value. If the master crashes, the absence of synchronization messages will cause the slave whose timer expires first to automatically become a *candidate* for master.

The candidate broadcasts an *Election* message. The other slaves, still waiting for a synchronization message from the master, then reinitialize their election timers to prevent other candidacies. They also reply to the *Election* message with *Accept* messages that inform the candidate of the senders' names.

The candidate acknowledges the slaves' *Accept* messages, and builds a list with their names. If elected it will synchronize the machines on this list. In the absence of events that will make the candidate resign (these events will be described in the next section), the candidate becomes master after a predetermined period of time following the receipt of the last *Accept* message has elapsed. The length of this time period has been chosen to give slaves the necessary time to answer the candidate's request.

Upon becoming master, the time daemon broadcasts a *Masterup* message to make slaves that may not have received its candidacy offer aware of the presence of a new master. The slaves will reply with *Slaveup* messages, which enable the master to obtain the names of all slaves.

5.2. The Case of Two (or more) Candidates

It is possible that the lowest timer value be selected by two or more time daemons which then time out simultaneously. As a result, the slaves receive two or more *Election* messages. However, they will reply with an *Accept* message only

[3] See [Gusella1984, Gusella1985b] for a justification of this value.

to the first *Election* message received. Any other messages will be replied to with a *Refuse* message. A candidate that receives a *Refuse* message will withdraw its candidacy and return to the humble state of Slave.

In the Ethernet local area network, broadcast messages do not overlap. Every machine receives messages in the order they are sent. However, we wanted our election algorithm to work with other network topologies (for example, ring networks) where broadcast message ordering can be arbitrary. In this latter case, it is possible that all candidates might receive *Refuse* messages. To make the election algorithm behave consistently in all network types, we designed the algorithm so that candidates themselves reply to *Election* messages with *Refuse* messages.

When two time daemons run for master simultaneously, each candidate, upon receiving the other's *Election* message, replies with a *Refuse* message. Both candidates therefore return to the state of Slave. They also reselect their election timer values using an exponential backoff mechanism similar to the one used in the Ethernet to avoid simultaneously timing out once again. This procedure also handles the degenerate case where only two time daemons are left in the network after the master has disappeared and they time out at the same time.

In the Appendix we give a proof of the fact that the probability that two or more slaves time out at the same time is less than $\frac{N\delta}{R}$, where N is the number of time daemons, δ is the time window after an *Election* message during which conflicts may arise, and R is the width of the interval over which time daemons select their election timers; the timer values are supposed to be uniformly distributed over R.

The following scenario illustrates another case in which a conflict may occur. Suppose that the election procedure takes M seconds to complete and that there are N slaves in the network. The *Election* message broadcast by a candidate is not received by a slave whose election timer expires within M seconds[4]. In this situation, even though we also have two candidates, there is no need to postpone the election. Rather, the first candidate and all the other slaves, upon receiving the *Election* from the second candidate, reply to it with *Refuse* messages, leaving the first candidate free to complete the election procedure.

[4] This can happen because the operating system allocates only a limited amount of buffer space for incoming messages, and the UDP protocol does not provide a reliable transport subsystem.

5.3. The Case of Two Masters

A network partition creates two disjoint sets of slaves: one with the original master and one without a master. This latter set will elect a master. When the network partition is repaired, there will be two independent sets of slaves, each with its own master. This anomalous situation is stable and will only be detected by a starting time daemon's *Masterreq* message.

The following sequence of events will lead to the detection of the abnormal case and restoration of the normal situation:

- A newly started time daemon will broadcast a *Masterreq* message.
- It will receive two *Masterack* messages, one from each master.
- Since a slave does not have the authority to kill a master, it simply notifies the first master −the first to reply to its request− that a second one is present with a *Conflict* message.
- The informed master broadcasts a *Conflict* message to find out the name of the other master.
- The other master replies with a *Masterack*.
- The first master then sends a *Quit* message to the other master, which returns to the **Slave** state.
- The first (and now sole) master broadcasts a *Masterup* message to collect the names of all of the second master's slaves.

A slave that is overlooked because it does not receive the *Masterup* message will timeout and will start an election. However, the present master will immediately cut short the slave's attempt by sending it a *Quit* message, and adding its name to the list of synchronized machines.

6. The Election Algorithm: A Finite State Model

We will use a state model to describe the details of how the election algorithm works. In their lifetime, time daemons can be in one of a finite number of states. Transitions from one state to another are caused by either the arrival of a message or the expiration of a timer. A state transition may cause a time daemon to send out a message, which triggers subsequent transitions in other time daemons. It is important to clarify that, in explaining the model, we focus only on the state of one time daemon, and not on the state of the entire distributed program.

Figure 3 represents the state diagram for a time daemon. Circles represent states; arrows represent transitions. A transition occurs either upon the arrival of a message or upon the expiration of a timer; these events are shown on the upper part of the labels superimposed on the arrows. The lower part of the label shows the message that is sent at the time of the corresponding transition. A *null* label signifies that no message is sent or received. Messages with an asterisk indicate that the message is broadcast, i.e. sent to all the other time daemons.

For example, if a time daemon is in the **Master** state and receives a *Conflict* message, it will broadcast a *Resolve* message and change its state to the **Conflict** state. The following description refers to Figure 3.

The State Diagram

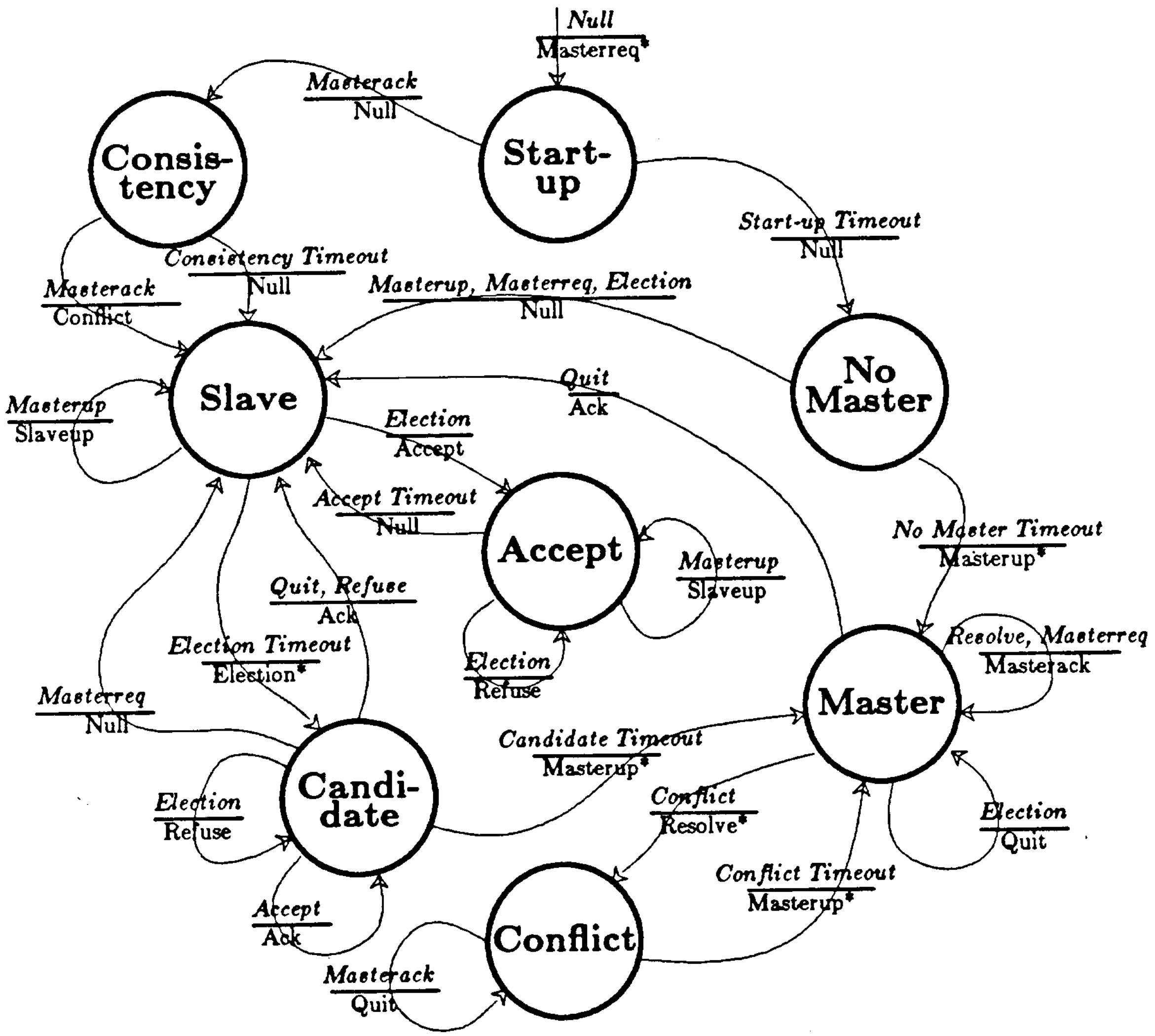


Figure 3

6.1. Description of the States

Start-up: Upon start-up, a time daemon broadcasts a *Masterreq* message to inform the master that a new time daemon exists.

No Master: This state results when no message is received by a time daemon in the **Start-up** state and its start-up timer expires. At this stage, the time daemon assumes that there is no master present and is ready to become the master. However, there are three cases where the time daemon will not become the master and will become a slave instead: first, if it receives a *Masterreq* from a newly started time daemon; second, if it receives an *Election* message from a slave; and third, if it receives a *Masterup* message from a candidate that is about to become the master. If none of these messages is received after the no master timer expires, then the time daemon will enter the **Master** state and broadcast a *Masterup* message.

Master: The **Master** state is reached either when an election is won, or when no master is found at start-up time.

Slave: In this state a time daemon receives periodical adjustment messages from the master; when this occurs, it reinitializes its election timer.

Candidate: This state is reached when the election timer in a slave expires. Since these timers are randomly set from a large interval, it is unlikely that two or more of them will expire simultaneously. A time daemon will remain in the **Candidate** state as long as it receives *Accept* messages from slaves; if instead it receives a *Refuse* message, it will revert back to the **Slave** state. If the candidate timer that is reinitialized after any *Accept* message is received expires, the candidate will become master, broadcasting a *Masterup* message.

Accept: A process in the normal **Slave** state receives an *Election* message and sends to the candidate an *Accept* message entering this state. It will then reply with a *Refuse* to all the following *Election* messages, until the accept timeout occurs, which resets the state to normal **Slave**.

Conflict: In this state the master has received a *Conflict*, and looks for one, or possibly more, rival masters to kill. It will leave the state after waiting for the conflict timeout to expire following the resolution of the conflict.

Consistency: In this state a newly started time daemon that has received a *Masterack* message from a master waits to check that no other master is active. If it receives a *Masterack* from another master, it sends a *Conflict* message to the first one, which will eliminate the anomalous situation, and immediately becomes a slave. If no *Masterack* is received, after the consistency timeout expires, the time daemon enters the **Slave** state.

7. Message Complexity of the Algorithm

The algorithm we have described is very efficient in terms of network utilization. In fact, it requires a linear number of messages to elect a new master. Suppose that, after the master's crash, there are N machines in the network. Suppose also, for the purpose of simplifying the discussion, that there are no message losses.

1) Case of one candidate:
The election starts with the *Election* message, which is followed by $N-1$ *Accept* messages sent by the slaves. Then there are $N-1$ acknowledgments from the candidate, the *Masterup* message, and finally, the $N-1$ *Slaveup* replies. The total message count is $2+3*(N-1) = 3*N-1$.

2) Case of two candidates:
The election starts with two *Election* messages. Each of the $N-2$ remaining slaves replays with an *Accept* message to the first candidate and with a *Refuse* message to the second one. Each candidate acknowledges the $N-2$ messages received. Moreover, the two candidates send a *Refuse* message, which is also acknowledged, to each other. In this case, the total message count is $4*N-2$. The election is not successful and must be repeated; however, as we have shown, in each repetition it becomes less and less probable that two candidates will appear.

8. Conclusions

We have presented an election algorithm that ensures the reliability of the master-slave based distributed program TEMPO in spite of master crashes. Elections occur when randomly set election timers expire. The algorithm is very simple and efficient in the normal case. In the unusual cases in which there are two or more candidates, it handles election conflicts in a conservative way by making candidates withdraw their candidacies and manipulating timers in order to reduce the probability of conflicts reoccurring. This is appropriate because synchronized clocks diverge only slightly during an election. The algorithm also handles the case in which there are two masters after a network partition is repaired.

Perhaps the most significant drawback of the algorithm is its reliance on the existence of a broadcast channel. This limits its usability in an internetwork environment. We are working to extend the algorithm to this more general environment. We are satisfied with the performance of our election algorithm for TEMPO, and we believe that its simplicity and efficiency make it attractive for other applications as well.

9. Acknowledgments

Domenico Ferrari stimulated our efforts in designing a better and more accurate algorithm. Claude Belisle helped in the derivation shown in the Appendix. The authors are also grateful to Mike Karels, Luis Felipe Cabrera, and Stuart Sechrest. A special *grazie* to Phyllis Chang for editorial advice.

References

Dolev1982.

D. Dolev and R. Strong, "Polynomial Algorithms for Multiple Processor Agreement," *Proceedings of the 14th Annual Symposium on Theory of Computation*, pp. 401-407, ACM, May 1982.

Frederickson1984.

G. N. Frederickson and N. A. Lynch, "The Impact of Synchronous Communication on the Problem of Electing a Leader in a Ring," *Proceedings of the 16th Annual Symposium on Theory of Computation*, pp. 493-503, ACM, May 1984.

Garcia-Molina1982.

H. Garcia-Molina, "Elections in a Distributed Computing System," *IEEE Transactions on Computers*, vol. C-31 No.1, pp. 48-59, IEEE, January 1982.

Gusella1984.

R. Gusella and S. Zatti, "TEMPO - A Network Time Controller for a Distributed Berkeley UNIX System," *Distributed Processing Tech. Comm.*

Newsletter, vol. 6 NoSI-2, pp. 7-15, IEEE, June 1984.

Gusella1985a.

R. Gusella and S. Zatti, "The Berkeley UNIX 4.3BSD Time Synchronization Protocol: Protocol Specification," *Report No. UCB/CSD 85/250*, University of California, Berkeley, June 1985.

Gusella1985b.

R. Gusella and S. Zatti, "Clock Synchronization in a Local Area Network," *(in preparation)*.

Lampson1983.

B. Lampson, "Hints for Computer System Design," *Procs. of the 9th SOSP, Operating System Review*, vol. 17,5, pp. 33-48, ACM, October 1983.

LeLann1977.

G. LeLann, "Distributed Systems – Towards a Formal Approach," *Proceedings of the IFIP Congress 77*, pp. 155-160, 1977.

Vitanyi1984.

P.M.B. Vitanyi, "Distributed Elections in an Archimedean Ring of Processors," *Proceedings of the 16th Annual Symposium on Theory of Computation*, pp. 542-547, ACM, May 1984.

Appendix

In this section we will compute the probability that an election attempt results in a collision. We will ignore the case, described in section 5.2, of collisions generated by the loss of election messages. Being this last case rare, it is more a curiosity than a real problem.

Every time an election timer expires and a time daemon starts an election, there is a window of δ seconds during which a collision may happen. Since upon receiving the election message each time daemon reinitializes its election timer, δ is equal to the transmission time over the network plus the longest scheduling delay of the various time daemon processes. This window is, therefore, very small.

The problem can be formalized as follows. Let $T_1, T_2, \cdots, T_N$ be the values of the election timers of N time daemons. They are independent random variables with uniform distribution over the interval $[r, r+R]$, where r is larger than the time between two subsequent synchronization rounds, as discussed in section 5.1. In order to simplify the following derivation, we will work instead with the random variables $X_1, X_2, \cdots, X_N$ uniformly distributed over $[0, R]$. Let M_1 and M_2 be the smallest and the second smallest values among the X_i's respectively. Then, the probability that an election attempt results in a collision is $P[M_2-M_1 \leq \delta]$. This probability can be computed in the following way:

$$P[M_2-M_1 \leq \delta] = \int_0^R P[M_2-M_1 \leq \delta | M_1=x] f_{M_1}(x)dx$$

$$= \int_0^R P[M_2 \leq \delta+x | M_1=x] f_{M_1}(x)dx \qquad (1)$$

where f_{M_1} is the probability density function of M_1.

Given that $M_1=x$, the values of the remaining $N-1$ random variables are uniformly distributed over $[x, R]$. Therefore, for $0 \leq x \leq R-\delta$:

$$P[M_2 \leq \delta+x | M_1=x] = P[\min(U_1, U_2, \cdots, U_{N-1}) \leq \delta]$$

$$= 1 - \left[\frac{R - x - \delta}{R - x}\right]^{N-1}$$

where $U_1, U_2, \cdots, U_{N-1}$ are independent random variables with uniform distribution over $[0, R-x]$.

For $R-\delta \leq x \leq R$, we have instead:

$$P[M_2 \leq \delta+x | M_1=x] = 1 \ ,$$

thus (1) becomes:

$$P[M_2 - M_1 \le \delta] = \int_0^{R-\delta} \left[1 - \left[\frac{R - x - \delta}{R - x} \right]^{N-1} \right] f_{M_1}(x)dx \; + \int_{R-\delta}^{R} 1 f_{M_1}(x)dx \; . \qquad (2)$$

The density function $f_{M_1}(x)$ can be written as the derivative of the distribution function:

$$f_{M_1}(x) = \frac{d}{dx} F_{M_1}(x) = \frac{d}{dx} P[M_1 \le x] = \frac{d}{dx} \left[\begin{array}{ll} 0 & \text{if } x<0 \\ 1-(\frac{R-x}{R})^N & \text{if } 0 \le x < R \\ 1 & \text{if } x \ge R \end{array} \right]$$

$$= \begin{cases} \frac{N}{R} \left[\frac{R-x}{R} \right]^{N-1} & \text{if } 0 \le x < R \\ 0 & \textit{otherwise} \end{cases} .$$

Thus finally:

$$P[M_2 - M_1 \le \delta] = \int_0^{R-\delta} \left[1 - \left[\frac{R-x-\delta}{R-x} \right]^{N-1} \right] \frac{N}{R} \left[\frac{R-x}{R} \right]^{N-1} dx \; + \int_{R-\delta}^{R} \frac{N}{R} \left[\frac{R-x}{R} \right]^{N-1} dx$$

$$= 1 - \left[1 - \frac{\delta}{R} \right]^N . \qquad (3)$$

Observe that for all $0 \le x < 1$ and for all positive integers N , we have:

$$1-(1-x)^N \le Nx$$

with very good approximation when x is small, in the sense that:

$$\lim_{x \to 0} \frac{1-(1-x)^N}{Nx} = 1 \; .$$

This provides us with a sharp upper bound for (3):

$$1 - \left[1 - \frac{\delta}{R} \right]^N \le \frac{N\delta}{R} \; .$$