1.3 Related Work

The problem of synchronizing clocks has been a topic of interest recently. A seminal paper was Lamport's work [6], defining logical clocks and describing an algorithm to synchronize them. Several algorithms to synchronize real time clocks have appeared in the literature [5, 6, 7, 9]. Those of Lamport [6] and Marzullo [9] have the processes updating their clocks whenever they receive an appropriate message; these messages are assumed to arrive every so many real seconds, or more often. In contrast, the algorithms in Halpern, Simons and Strong [5], Lamport and Melliar-Smith [7], and this thesis run in rounds. During a round, a process updates its clock once. The rounds are determined by the times at which different processes' local clocks reach the same times. There is an impossibility result due to Dolev, Halpern and Strong [2], showing that it is impossible to synchronize clocks without digital signatures if one third or more of the processes are subject to Byzantine failures. Dolev, Halpern and Strong's paper [2] also contains a lower bound similar to ours (proved independently), but characterizing the closeness of synchronization obtainable along the real time axis, that is, a lower bound on how closely in real time two processes' clocks can read the same value.

The three algorithms of Lamport and Melliar-Smith [7], as well as our maintenance algorithm, require a reliable, completely connected communication network, and handle arbitrary process faults. The first algorithm works by having each process at every round read all the other processes' clocks and set its clock to the average of those values that aren't too different from its own. The size of the adjustment is no more than the amount by which the clocks differ plus the uncertainty in obtaining the other processes' clock values. However, the closeness of the synchronization achieved depends on the total number of processes, n. The message complexity is n^2 at each round, if getting another process' clock value is equated with sending a message.

In the other two algorithms in [7], each process sets its clock to the median of the values obtained by receiving messages from the other processes. To make sure each nonfaulty process has the same set of values, the processes execute a Byzantine Agreement protocol on the values. The two algorithms use different Byzantine Agreement protocols. One of the protocols doesn't require digital signatures, whereas the other one does. As a result, the clock synchronization algorithm derived from the latter will work even if almost one half of the processes are faulty, while the other two algorithms in [7] can only handle less than one third faulty processes. For both of the Byzantine clock synchronization algorithms, the closeness of synchronization and the size of the adjustment depend on the number of faulty processes, and the number of messages per round is exponential in the number of faults.

The algorithm of Halpern, Simons and Strong [5] works in the presence of any number of process and link failures as long as the nonfaulty processes can still communicate. It requires digital signatures. When a process' clock reaches a certain value (decided on in advance), it broadcasts that time. If it receives a message containing the value not too long before it reaches the value, it updates its clock to the value and relays the message. The closeness of synchronization depends only on the drift rate, the round length, the message delivery time, and the diameter of the communication graph after the faulty elements are removed. The message complexity per round is n^2. However, the size of the adjustment depends on the number of faulty processes.

The framework and error model used by Marzullo in [9] make a direct comparison of his results with ours difficult. He considers intervals of time and analyzes the error probabilistically.

The problem addressed in these papers is only that of maintaining synchronization of local times once it has been established. None of them explicitly discusses any sort of validity condition, quantifying how clock time increases in relation to real time. Only [5] includes a reintegration procedure for repaired processes.

Chapter Two

Formal Model

2.1 Introduction

We present a formal model for describing a system of distributed processes, each of which has its own clock. The processes communicate by sending messages to each other, and they can set timers to cause themselves to take steps at some specified future times. The model is designed to handle arbitrary clock rates, Byzantine process failures, and a variety of assumptions about the behavior of the message system.

The advantages of a formal model are that lower bound proofs can be seen to be rigorous, and the effects of an algorithm, once it is stated in a language that maps to the model, can be discerned unambiguously.

This model will be used in subsequent chapters to describe our particular versions of the clock synchronization problem.

2.2 Informal Description

We model a distributed system consisting of a set of processes that communicate by sending messages to each other. Each process has a physical clock that is not under its control.

A typical message consists of text and the sending process' name. There are also two special messages, START, which comes from an external source and indicates that the recipient should begin the algorithm, and TIMER, which a process receives when its physical clock has reached a designated time.

A process is modelled as an automaton with a set of states and a transition function. The transition function describes the new state the process enters, the messages it sends out, and the timers it sets for itself, all as a function of the process' current state, received message and physical clock time. An application of the transition function constitutes a process step, the only kind of event in our model.

The system is interrupt-driven in that a process only takes a step when a message arrives. The message may come from another process, or it may be a TIMER message that was sent by the process itself. Thus, by using a TIMER message, a process can ensure that an interrupt will occur at a specified time in the future. We neglect local processing time by assuming that the processing of an arriving message is instantaneous.

We assume that the communication network is fully connected, so that every process can send a message directly to every other process. Processes possess the capability of broadcasting a message to all the processes at one step. The message system is described as a buffer that holds messages until they are delivered.

System histories consist of sequences of "actions", each of which is a process event surrounded by a description of the state of the system, one sequence for each real time of interest. The sequences must satisfy certain natural consistency and correctness conditions. We introduce the notion of "shifting" the real times at which a particular process' steps occur in a history and note the resulting changes to the message delivery times. Finally, we define an execution to be a history in which the message system behaves as desired.

2.3 Systems of Processes

Let P be a fixed set of *process names*. Let X be a fixed set of *message values*. Then M, the set of *messages*, is {START, TIMER} $\cup$ (X x P). A process receives a START message as an external indication of the beginning of an algorithm. A process receives a TIMER message when a specified time has been reached on its physical clock. All other messages consist of a message value and a process name, indicating the sender of the message.

Let $\mathcal{F}(S)$ denote the finite subsets of the set S.

A *process* p is modelled as an automaton. It has a set Q of states, with a distinguished subset I of initial states, and a distinguished subset F of final states. It has a *transition function*, τ, where τ: Q x $\mathbb{R}$ x M $\rightarrow$ Q x $\mathcal{F}$(X x P) x $\mathcal{F}(\mathbb{R})$. The transition function maps p's state, a real number indicating its physical clock time, and an incoming message, all to a new state for p, a finite set of (message value, destination) pairs, and a finite set of times at which to set timers. For any r in $\mathbb{R}$, m in M, Y in $\mathcal{F}$(X x P), and Z in $\mathcal{F}(\mathbb{R})$, if q is in F and if τ(q,r,m) = (q',Y,Z), we require that q' also be in F. That is, once a process is in a final state, it can never change to non-final state.

We assume that, in the absence of non-TIMER messages, a process does not set an infinite sequence of timers for itself within a finite amount of time. To state this condition formally, we choose any time r_1 and state q_1 for p, and consider the following sequence of applications of τ_p:

$\tau_p(q_1,r_1,\text{TIMER}) = (q_2,Y_2,Z_2)$

$\tau_p(q_2,r_2,\text{TIMER}) = (q_3,Y_3,Z_3)$, where $r_2 = \min\{r \in Z_2\colon r > r_1\}$

.

.

.

$\tau_p(q_i,r_i,\text{TIMER}) = (q_{i+1},Y_{i+1},Z_{i+1})$, where $r_i = \min\{r \in \cup_{j=2..i} Z_j\colon r > r_{i-1}\}$

.

.

.

Then as i approaches ∞, it must be that r_i approaches ∞.

We define a *step* of p to be a tuple (q,r,m,q',Y,Z) such that $\tau(q,r,m) = (q',Y,Z)$.

A *clock* is a monotonically increasing, everywhere differentiable function from $\mathbb{R}$ (real time) to $\mathbb{R}$ (clock time). We will employ the convention that clock names are capitalized and that the inverse of a clock has the same name but is not capitalized. Also, real times are denoted by small letters and clock times by capital letters.

A *system of processes*, denoted (P,N,S), consists of a set of processes, one for each name in P, a nonempty subset N of P called the *nonfaulty* processes, and a nonempty subset S of P called the *self-starting* processes. (We will use P to denote both the set of names and the set of processes, relying on context to distinguish the two.) The nonfaulty processes represent those processes that are required to follow the algorithm. The self-starting processes are intended to model those that will begin executing the algorithm on their own, without first receiving a message. A *system of processes with clocks*, denoted (P,N,S,PH), is a system of processes (P,N,S) together with a set of clocks $PH = \{Ph_p\}$, one for each p in P. Clock Ph_p is called p's *physical clock*. The transition function for p is denoted by τ_p. Throughout this thesis we assume $|P| = n$.

2.4 Message System

We assume that every process can communicate directly with every process, (including itself, for uniformity) at each step. The message system is modelled by a message buffer, which stores each message, together with the real times at which it is sent and delivered. For technical convenience, we do not require that messages be sent before being received. This correctness condition is imposed later.

A state of the message buffer consists of a multiset of tuples, each of the form (p,x,q) or (TIMER,T,p) or (START,p), with associated real times of sending and delivery. The message (x,p) with recipient q is represented by (p,x,q). (TIMER,T,p) indicates a timer set for time T on p's physical clock. (START,p) represents a START message with p as the recipient.

An initial state of the message buffer is a state consisting of some set of START messages. The sending and delivery times are all initialized as ∞.

The behavior of the message buffer is captured as a set of sequences of SEND and RECEIVE operations, each operation with its associated real time. Each operation involves a message tuple. The result of performing each operation is described below.

SEND(u,t): the tuple u is placed in the message buffer with sending time t and delivery time ∞ as long as there is no u entry already in the message buffer with sending time ∞. If there is, then t is made the new sending time of the u entry with the earliest delivery time and sending time ∞.

RECEIVE(u,t): the tuple u is placed in the message buffer with delivery time t and sending time ∞, as long as there is no u entry already in the message buffer with delivery time ∞. If there is, then t is made the new delivery time of the u entry with the earliest sending time and delivery time ∞.

The *message delay* of a non-START message is the delivery time minus the sending time. A positive message delay means the message was sent before it was delivered. A negative message delay means the message was delivered before it was sent. A message delay of $+\infty$ means the message was sent but never delivered, and a message delay of $-\infty$ means the message was delivered, but never sent. (The message delay is not defined for START messages that are never delivered.)

2.5 Histories

In this section we define a history, a construct that models a computation in which nonfaulty processes follow their state-transition functions. Constraints to ensure that the message system behaves correctly will be added in Section 2.8.

Fix a system of processes and clocks $\mathcal{S}$ = (P,N,S,PH).

An *event* for P is of the form receive(m,p), the receipt of message m by process p, where p is in P. A *schedule* for P is a mapping from $\mathbb{R}$ (real times) to finite sequences of events for P such that only a finite number of events occur before any finite time, and for each real time t and process p, all TIMER events for p are ordered after all non-TIMER events for p. The first condition rules out a process taking an infinite number of steps in a finite amount of time, and the second condition allows messages that arrive at the same time as a timer goes off to get in "just under the wire".

In order to discuss how an event affects the system as a whole, we define a *configuration* for P to consist of a state for each process in P and a state for the message buffer. An *initial configuration* for (P,N,S) consists of an initial state for each process and an initial state for the message buffer.

An *action* for P is a triple (F,e,F'), consisting of an event for P and two configurations F and F' for P. F is the *preceding* and F' the *succeeding* configuration for the action.

A *history* for $\mathcal{S}$ is a mapping from real times to sequences of actions for (P,N,S) with the following properties:

- the projection onto the events is a schedule;
- if the sequence of actions is nonempty, then the preceding configuration of the first action is an initial configuration, and the succeeding configuration of each action is the same as the preceding configuration of the following action;
- if an action (F,receive(m,p),F') occurs at real time t, then F = F' except for p's state and the state of the message buffer; moreover, there exist Y in $\mathcal{F}(X \times P)$ and Z in $\mathcal{F}(\mathbb{R})$ such that the buffer in F' is obtained from the buffer in F by executing the following operations:
  - if m = START, then RECEIVE((START,p),t);
    if m = TIMER, then RECEIVE((TIMER,$Ph_p(t)$,p),t);
    if m = (x,p') for some p', then RECEIVE((p',x,p),t);
  - SEND((p,x,p'),t) for all messages of the form (x,p') in Y;
  - SEND((TIMER,T,p),t) for all T in Z such that T > r (that is, as long as the timer is set for a future time); if T $\leq$ r, then no operation is performed.