Model-based Clock Synchronization in Networks with Drifting Clocks

Michael D. Lemmon, Joydeep Ganguly, and Lucia Xia
Dept. of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
lemmon@maddog.ee.nd.edu

Abstract

This paper examines a clock synchronization algorithm in which processors identify dynamic models of neighboring processor clocks. These models are then used as signature functions to develop a clock synchronization algorithm that functions in the presence of drifting clocks.

1. Introduction

Clock synchronization is a well-studied problem in distributed systems. The synchronization problem is concerned with methods that help processors agree upon *when* to send messages. Over the past twenty years, a variety of papers [1] [2] [3] [4] [5] [6] [7] have addressed various aspects of this problem. Early algorithms [2] [3] were based on averaging, assumed drift-free clocks, and required a minimum number of correct processors for fault-tolerant operation. In [4], the restriction on the number of correct processors [8] was relaxed by using an authenticated algorithm in which time stamps were distributed through the network using a diffusion process [9]. Restrictions on drift-free clocks were relaxed in [5]. Implementation specific issues for clock synchronization were handled in the Network Timing Protocol (NTP) [6] and probabilistic synchronization algorithms [7].

This paper presents a synchronization algorithm for drifting clocks that is similar to that found in [4] but handles drift in a different manner than was done in [5]. In our proposed method, each processor *identifies* a *set* of dynamical models for its neighbor's clocks. This model allows processor P_i to time-stamp messages for processor P_j with P_j's local time. Clock synchronization is then achieved by using a diffusion type algorithm [9] in which messages are stamped with these transformed times. These transformed time-stamps represent a form of authentication. The proposed algorithm, therefore preserves the fault-tolerant properties of the method in [4]; namely that no minimum number of correct processors is required for fault-tolerant operation in the presence of drifting clocks.

2. Model-based Clock Identification

System identification uses statistical estimation methods to estimate the parameters of a dynamical system. When the parameter estimates are compact sets (rather than points), then we have a *set-valued* identification method [10]. Set-valued identification is useful when modeling uncertainty is not directly captured by a priori models. This section introduces a set-valued algorithm that a processor can use to estimate the drift and offset of its neighbor's clocks.

Assume that the distributed system consists of processors P_i for $i = 1, \ldots, N$. Let t_i denote the local time on the clock for processor P_i. We assume that the local times, t_i and t_j, on processors P_i and P_j can be related by the linear equation

$$t_i = a_{ij} t_j + b_{ij} \tag{1}$$

where a_{ij} and b_{ij} are real-valued constants representing the *relative drift* and *relative offset*, respectively, between the two hardware clocks.

Let $\underline{t_{ik}}$ be the kth local time ($k = 1, \ldots, N$) when processor P_i sends a message $M_k^{(\text{init})}$ to processor P_j. The local time, $\underline{t_{ik}}$ is measured with respect to P_i's clock. The message $M_k^{(\text{init})}$ is the kth *initiating* message and we assume that it is time stamped with the local time $\underline{t_{ik}}$. Processor P_j receives message $M_k^{(\text{init})}$ at local time t_{jk} (measured with respect to P_j's clock). Upon receipt of the initiating message, P_j appends its local time, t_{jk}, to the message and replies to processor P_i with the message $M_k^{(\text{reply})}$. The kth *reply message*, $M_k^{(\text{reply})}$ therefore contains two time-stamps, $\underline{t_{ik}}$ and t_{jk}. Processor P_i receives the kth reply

0-7695-0975-4/00 $10.00 © 2000 IEEE

message at local time $\overline{t_{ik}}$ (measured with respect to P_i's clock). As soon as processor P_i receives the kth reply message, it has the following set of times $\underline{t_{ik}}$, t_{jk}, and $\overline{t_{ik}}$. Moreover, it knows that these times are related as

$$a_{ij}t_{jk} + b_{ij} \in [\underline{t_{ik}}, \overline{t_{ik}}] \tag{2}$$

where a_{ij} and b_{ij} are the relative drift and offsets and where $\overline{t_{ik}} - \underline{t_{ik}} = \Delta_k$ is the round trip delay for the kth round of messages.

We assume that processor P_i sends out N messages at local times $\underline{t_{ik}}$ for $k = 1, \ldots, N$. Processor P_i receives replies to these messages from processor P_j at times $\overline{t_{ik}}$ and each received message is stamped with $\underline{t_{ik}}$ and the local time, t_{jk}, when processor P_j received the kth message. When the last reply is received from processor P_j, the originating processor P_i has a set of time-stamped triples,

$$\left\{(\underline{t_{ik}}, t_{jk}, \overline{t_{ik}})\right\}_{k=1,\ldots,N} \tag{3}$$

that satisfy the relation in equation 2. The true relative drift a_{ij} and offset b_{ij} are not known, but processor P_i can use this time-stamped data to estimate the relative drift and offset.

Let's assume processor P_i has some (as yet unspecified algorithm) for computing the estimates $\hat{a}_{ij}$ and $\hat{b}_{ij}$ of the true relative drift and offset. This estimate is computed from the N data points (see equation 3). We say that the estimate is *consistent* if and only if the predicted time $\hat{t}_{ik}$ satisfies

$$\underline{t_{ik}} \leq \hat{t}_{ik} = \hat{a}_{ij} + \hat{b}_{ij} \leq \overline{t_{ik}} \tag{4}$$

for all $k = 1, \ldots, N$.

For a given data set, there are many possible estimates $(\hat{a}_{ij}, \hat{b}_{ij})$ that can be consistent with the data. So rather than picking a specific estimate, we will identify a *set* $\hat{\Omega} \subset \Re^2$ that contains all estimates $(\hat{a}_{ij}, \hat{b}_{ij})$ that are consistent with the data. Note that $\hat{\Omega}$ may be an overbound for the set of consistent estimates in that if $(\hat{a}_{ij}, \hat{b}_{ij}) \in \hat{\Omega}$ then this estimate may not be consistent. However, if the estimate is consistent then we know it lies in $\hat{\Omega}$. A set that contains all consistent estimates of the data set will be said to be *consistent*. Our primary concern, therefore, involves finding a set $\hat{\Omega}$ that is consistent with the given data in equation 3.

There are, of course, an infinite number of sets $\hat{\Omega}$ that are consistent with the data. So we'll confine our attention to sets $\hat{\Omega}$ that are formed from the Cartesian product of two intervals. In other words, we require that $\hat{\Omega}$ be a rectangle of the form $[\underline{a_{ij}}, \overline{a_{ij}}] \times [\underline{b_{ij}}, \overline{b_{ij}}]$. The objective will be to find a consistent estimate of minimal area. In other words, we look for parameters $\overline{a_{ij}}$, $\underline{a_{ij}}$, $\overline{b_{ij}}$, and $\underline{b_{ij}}$ such that $\hat{\Omega} = [\underline{a_{ij}}, \overline{a_{ij}}] \times [\underline{b_{ij}}, \overline{b_{ij}}]$ is consistent and such that $(\overline{a_{ij}} - \underline{a_{ij}})(\overline{b_{ij}} - \underline{b_{ij}})$ is minimized subject to $\overline{a_{ij}} \geq \underline{a_{ij}}$ and $\overline{b_{ij}} \geq \underline{b_{ij}}$.

We now introduce a simple algorithm for computing the optimal set-valued estimate $\hat{\Omega}$. Let $\overline{a_{ij}}$ and $\hat{b}_1$ be solutions to the linear program

$$\begin{array}{ll} \text{maximize} & \begin{bmatrix} a_{ij} & b_{ij} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ \text{with respect to} & a_{ij}, b_{ij} \\ \text{subject to} & \underline{t_{ik}} \leq a_{ij}t_{jk} + b_{ij} \leq \overline{t_{ik}} \\ & (k = 1, \ldots, N) \end{array} \tag{5}$$

Let $\underline{a_{ij}}$ and $\hat{b}_2$ be solutions to the linear program,

$$\begin{array}{ll} \text{minimize} & \begin{bmatrix} a_{ij} & b_{ij} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ \text{with respect to} & a_{ij}, b_{ij} \\ \text{subject to} & \underline{t_{ik}} \leq a_{ij}t_{jk} + b_{ij} \leq \overline{t_{ik}} \\ & (k = 1, \ldots, N) \end{array} \tag{6}$$

Now let $\overline{b_{ij}} = \max(\hat{b}_1, \hat{b}_2)$ and $\underline{b_{ij}} = \min(\hat{b}_1, \hat{b}_2)$. The parameters $\overline{a_{ij}}$, $\underline{a_{ij}}$, $\overline{b_{ij}}$, and $\underline{b_{ij}}$ obtained by solving the preceding linear programs form an optimal estimate (in the sense mentioned above) to $\hat{\Omega}$. The proof of this assertion is given below.

Lemma 1: The estimates $(\overline{a_{ij}}, \hat{b}_1)$ and $(\underline{a_{ij}}, \hat{b}_2)$ are consistent estimates.

Proof: This fact follows from the fact that these estimates solve the aforementioned linear programs and that the feasibility of these solutions is sufficient for the consistency of the estimates. •

Lemma 2: If $a_{ij} > \overline{a_{ij}}$ or $a_{ij} < \underline{a_{ij}}$ then the estimate (a_{ij}, b_{ij}) is not consistent.

Proof: Assume that $a_{ij} > \overline{a_{ij}}$ and (a_{ij}, b_{ij}) is consistent. From Lemma 1, we know that $(\overline{a_{ij}}, \hat{b}_1)$ is consistent and we know that it is the consistent estimate with largest relative drift. So if (a_{ij}, b_{ij}) is consistent then there is an estimate that is larger than the point determined by the linear program. This is clearly a contradiction and we can conclude that (a_{ij}, b_{ij}) is not consistent for any b_{ij}. A similar argument will also apply to the other inequality in the lemma. •

Lemma 3: The estimates $(\overline{a_{ij}}, \hat{b}_1 + \delta)$ and $(\underline{a_{ij}}, \hat{b}_2 + \delta)$ are consistent if and only if $\delta = 0$.

Proof: We consider the estimate $(\overline{a_{ij}}, \hat{b}_1)$. By Lemma 1 this estimate is consistent and we know it is an extreme point of the linear program in equation 5, therefore there exists a k' with data triple $(\underline{t_{ik'}}, t_{jk'}, \overline{t_{ik'}})$ such that either

$$\overline{a_{ij}}t_{jk'} + \hat{b}_1 = \underline{t_{ik'}} \tag{7}$$

or

$$\overline{a_{ij}} t_{jk'} + \hat{b}_1 = \overline{t_{ik'}} \tag{8}$$

Now consider a $\delta > 0$ such that $(\overline{a_{ij}}, \hat{b}_1 + \delta)$ is a consistent estimate. This means that for all k that the consistency relation (eq. 4) is satisfied. In particular it must be satisfied for k' so that

$$\underline{t_{ik'}} \leq \overline{a_{ij}} t_{jk'} + \hat{b}_1 + \delta \leq \overline{t_{ik'}} \tag{9}$$

We can rewrite this, again, as

$$\left(\overline{a_{ij}} + \frac{\epsilon}{t'_{jk}}\right) t'_{jk} + \hat{b}_1 + \delta - \epsilon < \overline{t'_{ik}}$$

where $|\epsilon| < |\delta|$ and $\epsilon > 0$. In other words, $(\overline{a_{ij}} + \epsilon/t'_{jk}, \hat{b}_1 + \delta - \epsilon)$ is a consistent estimate. But since $\overline{a_{ij}} + \epsilon/t'_{jk} > \overline{a_{ij}}$ we again violate lemma 2 and have a contradiction and the proof is complete. •

Proposition 1: If we let $\overline{b_{ij}} = \max(\hat{b}_1, \hat{b}_2)$ and $\underline{b_{ij}} = \min(\hat{b}_1, \hat{b}_2)$, then $\hat{\Omega} = [\underline{a_{ij}}, \overline{a_{ij}}] \times [\underline{b_{ij}}, \overline{b_{ij}}]$ is a consistent set-valued estimate that minimizes the product $(\overline{a_{ij}} - \underline{a_{ij}})(\overline{b_{ij}} - \underline{b_{ij}})$.

Proof: This theorem requires that we show two things. First we must show that $\hat{\Omega}$ is a consistent set-valued estimate. Second we must show that no other consistent set-valued estimate has a smaller area.

For the first part, we can establish consistency by a contradiction. So let (a_{ij}, b_{ij}) be a consistent point estimate and assume that it does not lie in $\hat{\Omega}$, then this would mean that either $a_{ij} > \overline{a_{ij}}$, $a_{ij} < \underline{a_{ij}}$, $b_{ij} > \overline{b_{ij}}$ or $b_{ij} < \underline{b_{ij}}$. If either of the first two inequalities hold then we contradict lemma 2. If either of the second two inequalities hold we contradict lemma 3. So all consistent point estimates must lie in $\hat{\Omega}$.

For the second part, let's assume that there is another consistent and rectangular set-valued estimate

$$\hat{\Omega}' = \left[\underline{a'_{ij}}, \overline{a'_{ij}}\right] \times \left[\underline{b'_{ij}}, \overline{b'_{ij}}\right]$$

such that

$$(\overline{a'_{ij}} - \underline{a'_{ij}})(\overline{b'_{ij}} - \underline{b'_{ij}}) < (\overline{a_{ij}} - \underline{a_{ij}})(\overline{b_{ij}} - \underline{b_{ij}})$$

Without loss of generality, assume that $\hat{b}_1 = \underline{b_{ij}}$ and $\hat{b}_2 = \overline{b_{ij}}$. By definition, $(\overline{a_{ij}}, \underline{b_{ij}})$ and $(\underline{a_{ij}}, \overline{b_{ij}})$ are consistent, so we know that

$$\underline{a'_{ij}} \leq \underline{a_{ij}} \leq \overline{a_{ij}} \leq \overline{a'_{ij}}$$
$$\underline{b'_{ij}} \leq \underline{b_{ij}} \leq \overline{b_{ij}} \leq \overline{b'_{ij}}$$

These inequalities imply that

$$\overline{a'_{ij}} - \underline{a'_{ij}} \geq \overline{a_{ij}} - \underline{a_{ij}}$$
$$\overline{b'_{ij}} - \underline{b'_{ij}} \geq \overline{b_{ij}} - \underline{b_{ij}}$$

Which directly contradicts our earlier assumption that the $\hat{\Omega}'$ has an area smaller than $\hat{\Omega}$, so the second part of the theorem is proven. •

3. Clustered Implementation of Model-based Identification

The model-based identification algorithm generates an estimate of neighboring clock offsets and drifts. This is comparable to the first stage of traditional synchronization algorithms in which averaging is used to estimate average clock offsets. This section compares the number of messages passed using the proposed model-based approach against the traditional averaging method in [1].

Let n be the number of processors in the network. Traditional synchronization [1] requires a few rounds of message passing to compute the average network delay. First a designated processor sends a message to the $n-1$ remaining processors and receives $n-1$ replies. The time-stamps obtained from this round of message passing are used to compute an average message delay and this average delay is then broadcast throughout the network. This approach, therefore, requires that $3(n-1)$ messages be passed through the network.

The model-based identification algorithm requires each processor to broadcast N initiating messages to every processor in the network. So the total number of messages passed would be $\sum_{k=1}^{n-1} Nk$. If we consider a network with 100 processors, then an averaging algorithm would require 297 messages. The model based approach (assuming $N = 3$) would require 14,850 messages. Clearly, this is too expensive.

It is possible, however, to significantly reduce the number of messages passed in our approach by adopting a clustering approach. This is accomplished by exploiting the transitive nature of the timing relationships. Assume that processors P_i and P_j know their timing relationship and assume that processors P_j and P_k know their timing relationship. Then for P_i to establish its timing relationship with P_k, it need not send messages, it only has to obtain the timing relationships that processor P_j has with P_k. In other words, if these timing relations are $t_i = a_{ij} t_j + b_{ij}$ and $t_j = a_{jk} t_k + b_{jk}$, then processor P_i can determine its relationship to P_k's clock if it knows the constants a_{jk} and b_{jk}. This relation is simply $t_i = a_{ik} t_k + b_{ik}$ where $a_{ik} = a_{ij} a_{jk}$ and $b_{ik} = a_{ij} b_{jk} + b_{ij}$. Therefore any two processors can infer their timing relationships provided they know an existing timing relationship with a common third processor.

We now illustrate how to use this transitivity property to reduce the number of messages in our model-based method. Consider a system of six processors, depicted in figure 1. This network is partitioned into 2 clusters, each cluster containing three processors. We identify a special control processor for each cluster. The control processor is responsible for communicating with other control processors outside of its cluster. The first step in this approach involves communication between neighbors within each cluster. The three processors in each cluster establish models between themselves. The second step lies in the control processors communicating with neighboring control processors to identify the timing relations between control processors. In this way, message passing either occurs within clusters or occurs through a distinguished set of control processors in each cluster.

Consider a clustered implementation of a model-based synchronization algorithm in which each cluster contains three processors. Let M be the number of clusters, n be the number of processors in the network, and N be the number of messages passed between two processors to establish their timing model. We assume that the entire network is partitioned into clusters that contain 3 processors. If the number of processors is not a multiple of three, there will exist an "isolated" cluster that will consist of either two or one processors. Since we are making use of the transitive property, in each cluster, only two timing models need to be established in each cluster, and the third timing model can be inferred by the two processors which have not communicated by using the transitive property. This means there will be $M(2N+1)$ messages passed within each cluster to form timing models of processors within each cluster. The $2N$ term accounts for the overhead in forming the two timing models between the processors, and the additional message accounts for the inference of the model via the transitive property. Since there are M clusters, we multiply the expression $2N+1$ by M. There will be $(M-1)(N+1)$ messages transmitted in forming models between the control processors in the network. Finally $N(n-3M)$ messages will be passed for an isolated cluster. isolated clusters. Hence the total number of messages required in this algorithm is $M(2N+1)+(M-1)(N+1)+2M+N(n-3M)$, where M is the number of clusters, n is the number of processors, and N is the number of initiating messages.

We can now compare the message count of the clustered model-based synchronization algorithm against the averaging algorithm's message count. For $n = 99$ processors, we obtain a total of 425 messages that would need to be passed which is comparable to the 294 messages passed in the averaging algorithm. Figure 1 gives a graphical comparison of the number of messages to be passed in each approach as a function of the number of processors. In this figure, approach 1 is our base model-based algorithm and approach 2 is the averaging algorithm. As can be seen, our clustering approach to model-based identification incurs a slightly higher cost, but the number of messages is still comparable to that of the averaging algorithm.

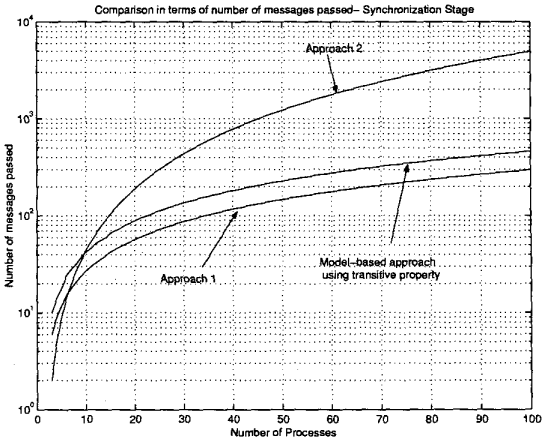

Figure 1. Number of Messages

4. Model-based Synchronization

Consider a set-valued estimate $\hat{\Omega}$ that is consistent with the data in equation 3. This estimate resides on processor P_i. If processor P_i's clock reads t_i, then we know that the local time, t_j, on processor P_j must satisfy

$$\begin{aligned} \underline{M}_j(t_i) &= \min_{(a,b)\in\hat{\Omega}_{ij}} \frac{t_i - b}{a} \leq t_j \\ &\leq \max_{(a,b)\in\hat{\Omega}_{ij}} \frac{t_i - b}{a} = \overline{M}_j(t_i) \end{aligned} \tag{10}$$

We now use this information to develop a clock synchronization algorithm similar to that in [4].

We characterize our algorithm by two tasks. The first task is the *time-monitor* task and the second task is a *message-handler* task. The tasks control the updating of the local variables *sync count* and *period. Sync count* is the number of messages the processor has sent since the algorithm started. The *period* is the period (measured with respect to the processor's local clock) at which synchronization messages are sent. The local variables in these tasks are

$$\begin{aligned} t &= \text{local processor time} \\ t_{\text{snd}} &= \text{time sent in message} \\ t_{\text{rec}} &= \text{time received in a message} \\ c &= \text{sync count of next message to be sent} \\ c_{\text{rec}} &= \text{sync count of the message just received} \end{aligned}$$

p = period of synchronization messages
N = number of neighboring processors
$\hat{\Omega}_j$ = set-valued estimate of the jth neighboring clock

The description of the following algorithm uses three functions. These functions are

- `send(c,t,j)` is a non-blocking send that sends the sync count c, and time t to processor j.
- `receive(c,t)` is a non-queued and non-blocking receive that returns true if a message is received at time t and returns zero otherwise.
- `MAX(t,Omega(j)` computes the largest possible time on neighboring processor P_j that is associated with the local time t. `Omega(j)` is assumed to be a structure with members $\underline{a}$, $\overline{a}$, $\underline{b}$, and $\overline{b}$ that form the set-valued estimate $\hat{\Omega}_j = [\underline{a}, \overline{a}] \times [\underline{b}, \overline{b}]$. The actual time computed by this function is

$$\overline{M}_j(t) = \max_{(a,b)\in\hat{\Omega}_j} \frac{t-b}{a}$$

The pseudo C-code for the time-monitor task is:

```
time_monitor(){
  if(t==p)
    for(j=1,N,j++){
      t_snd=MAX(p,Omega(j));
      send(c,t_snd,j);
    }
    c=c+1;
    t=0;
  }
}
```

This task waits until the local time t equals the period p. At that instant, the processor computes the estimated time t_{snd} on the neighboring processors' clock and then sends this time along with the current sync count to those processors. After the messages are sent, then the sync count is incremented and the local duration time t is reset to zero.

The pseudo C-code for the message handler task is:

```
msg_handler(){
  if( receive(c_rec,t_rec) &&
    (c_rec==c) && (t_rec < p)){
    for(j=1,N,j++){
      p=t_rec;
      t_snd=MAX(t_rec,Omega(j));
      send(c_rec,t_snd,j);
    }
    c=c+1;
    t=0;
  }
}
```

This task waits until a message is received from a neighboring processor. The received message delivers a sync count c_{rec} and time t_{rec} associated with the sending processor's message. Upon receipt of the message, the receiving processor first checks to see if it has already sent a message with the received sync count (i.e., $c = c_{\text{rec}}$) and if not, then it checks to see if the latest time when the message was sent is prior to the expiration of its period, (i.e., $t_{rec} < p$). If both of these tests are passed, then we know that the receiving processor is late in sending out a message. The task therefore updates the period p, and sends a time-stamped message to all of its neighbors. After this is done, the task resets the time to zero and increments the sync count.

The preceding tasks work together in much the same way as is seen in the diffusion algorithm of [4]. Essentially, both algorithms synchronize to the fastest clock in the network. The obvious difference between these two algorithms is that our algorithm does not explicitly use a signature function to authenticate messages and the correctness proof places few restrictions on the relative drift of processor clocks. The first difference is actually superficial, for the time-stamps computed by our algorithm are actually signature functions in the same spirit as defined in [4]. Our algorithm, therefore, sends authenticated messages and should therefore enjoy similar fault-tolerance properties as the diffusion algorithm of [4]. The other difference, however, is not superficial. As will be seen in the following correctness proof, we actually do not need drift-free clocks and so the proposed algorithm provides a practical algorithm for clock synchronization in the presence of drifting clocks.

Before stating and proving the main propositions of this section, we need to define a formal model for the algorithm and introduce some convenient notation. The fact that both processors have different clock rates suggests that a *hybrid* model for the algorithm will be needed in which we account for both the logical state of the program and the state of the continuous duration timers in each processor. In particular, we intend to use a *hybrid automaton* [11] to model the algorithm and then to establish the correctness of the algorithm using a forward reachability analysis.

A *hybrid automaton* is the ordered tuple (V, A, X, L) where V is a set of vertices, $A \subset V \times V$ is a set of arcs, and X is a linear space of continuous-state trajectories. If $x \in X$, then it satisfies the differential equation $dx/dt = a$ almost everywhere where a is a constant. The hybrid states are *continuous* states in which $dx/dt = a \neq 0$ and *discrete* states where $dx/dt = 0$. Finally, L is a map from $V \cup A$ onto a set of logical predicates whose truth values are defined with respect to the system state x at global time t. We identify four different predicates. An *invariant pred-*

icate labels a vertex. This vertex is *marked* only if the state in x satisfies the invariant formula. One important type of invariant predicate is a *rate constraint* that can be used to reset the rate constant a. A *guard condition* is a conditional statement labeling arcs. The truth of the guard predicate is required before an arc (whose preset is marked) can fire. *Reset equations* are assignments associated with system arcs. These equations are executed when the arc is traversed. A *sync label* is a label that synchronizes the execution of other arcs in the graph. A full discussion of hybrid automata is beyond the scope of this paper. See [11] for a more detailed discussion of the model as well as algorithmic verification tools.

Figure 2 shows the hybrid automaton for a model-based synchronization algorithm used to synchronize the actions of two processors, P_1 and P_2. The figure shows two concurrent state machines with two states labeled `waiting` and `update`. The invariant on the ith processor's *waiting* state ($i = 1$ or 2) is $dt_i = 1$, which is the rate equation for the processor's local time t_i. The arcs are labeled with synchronization labels (shown in italics) along with the guard (enclosed by square brackets) and reset formulae (enclosed by curly brackets). Therefore arc a_{12} in the first automaton has synchronization label *snd2*, guard condition $(\overline{M}_1(t_2) < p_2) \wedge (c == 0)$, and reset condition $p_2 := \overline{M}_1(t_2)$. The function $\overline{M}_1(t_2)$ is the max function defined earlier in equation 10. Note that both automata have initial conditions defined by arcs pointing to the *waiting* states. These arcs are labeled with the system's initial conditions. The state variables in this graph are t_1 (the local time on processor P_1), t_2 (processor P_2's local time), p_1 (processor P_1's sync period), p_2 (processor P_2's sync period), and c (the sync count). Note that the sync count is a global variable representing the difference between the local sync counts on both processors.

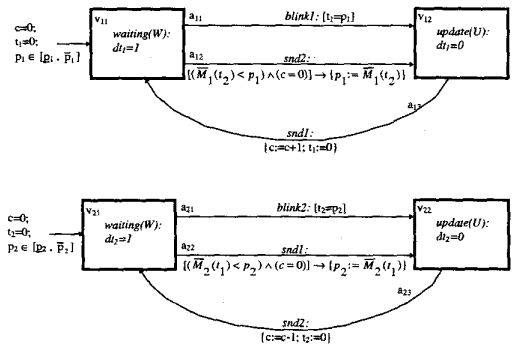

Figure 2. Hybrid Automaton

The initial state of the automaton is given by both the values of the hybrid states and the marking of the graph. In particular, the automaton's state will have the format

$$V_1.V_2 : [\ t_1 \quad p_1 \quad t_2 \quad p_2 \quad c\ ]$$

where V_1 and V_2 are the currently marked states in the two automata. This means, therefore, that the initial state for the system is

$$W.W : [\ 0 \quad [\underline{p}_1, \overline{p}_1] \quad [\underline{t}_2, \overline{t}_2] \quad [\underline{p}_2, \overline{p}_2] \quad 0\ ]$$

where W means that the state *waiting* is currently marked. The automaton will be said to be in a *synchronized state* if at a specified instant in time, the automaton's state has the form

$$W.W : [\ 0 \quad [\underline{p}_1, \overline{p}_1] \quad 0 \quad [\underline{p}_2, \overline{p}_2] \quad 0\ ]$$

The objectives of the following analysis are two-fold. First to show that the system always reaches a synchronized state and secondly to show that the set-valued bounds for the periods converge to a bounded steady state value.

Proposition 3: Consider the hybrid automaton shown in figure 2 assume that $\hat{\Omega}_1$ and $\hat{\Omega}_2$ are consistent clock models such that

$$2\underline{M}_2(\underline{p}_1) < \overline{M}_2(\overline{p}_1) \tag{11}$$

$$2\underline{M}_1(\underline{p}_2) < \overline{M}_1(\overline{p}_2)\ , \tag{12}$$

where $\underline{M}_2$ and $\overline{M}_2$ are defined in equation 10. Then the hybrid automaton always reaches a synchronized state after a finite number of transitions.

Proof: Proving this proposition requires a forward reachability analysis of the automaton. We begin by building the reachability tree generated by the directed graph of the hybrid automaton. This tree is shown in figure 3. In this figure, the logical state of the system is represented as $V_1.V_2$ where $V_i \in \{W, U\}$ for $i = 1, 2$. The W and U represent the `waiting` and `update` states, respectively. The root node of the tree is the furthest vertex to the left and this node represents the initial logical state of the system. Each arc in the reachability tree is labeled with the arc name. In some cases, where the firing of arcs is synchronized, the arc label has the form $a_{ij}||a_{kl}$. This means that arc a_{ij} and a_{kl} are evaluated and fired in a synchronized (i.e., at the same real-time) manner. Note that some of the arcs in this graph are traversed immediately after the enabling state is marked. The logical state $U.W$ is such a state, for while processor P_2 must wait for the local time t_2 to equal the period p_2, the arc a_{13} is immediately traversed since the rate for the local time t_2 is set to zero in this state. We therefore expect arc a_{13} to always be fired before the arc a_{21} can be traversed and so the arcs leaving the state $U.W$ are only a_{13} and/or $a_{13}||a_{22}$.

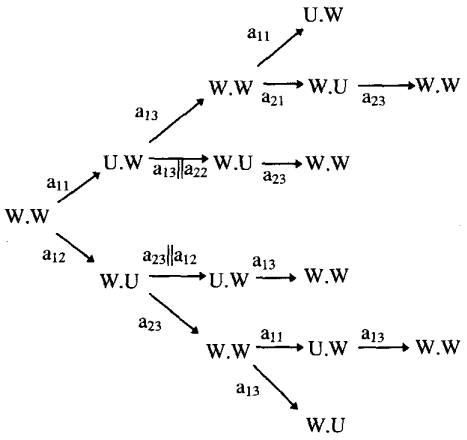

Figure 3. Reachability Tree

From the reachability tree in figure 3, we see that there are six different *occurrence sequences* (sequences of arcs) that can be executed from the initial state. Without loss of generality, we'll assume that the first arc fired is the a_{11} arc. A dual set of arguments can be used to determine what happens if a_{12} fires first. Under the assumption that a_{11} fires first, we have three different occurrence sequences to consider;

$$\begin{aligned} \sigma_{1a} &= a_{11}, a_{13}, a_{11} \\ \sigma_{1b} &= a_{11}, a_{13}, a_{21}, a_{23} \\ \sigma_{1c} &= a_{11}, a_{13}||a_{22}, a_{23} \end{aligned}$$

We now analyze each of these occurrence sequences separately.

We first assert that σ_{1a} cannot occur. This is proven by contradiction. The system state after firing a_{11} is

$$U.W : \left[[\underline{p}_1, \overline{p}_1], [\underline{p}_1, \overline{p}_1], [\underline{q}, \overline{q}], [\underline{p}_2, \overline{p}_2], 0\right] \tag{13}$$

where $\underline{q} = \underline{t}_2 + \underline{M}_2(\underline{p}_1)$ and $\overline{q} = \overline{t}_2 + \overline{M}_2(\overline{p}_1)$. The firing of a_{13} yields the state

$$W.W : \left[0, [\underline{p}_1, \overline{p}_1], [\underline{q}, \overline{q}], [\underline{p}_2, \overline{p}_2], 1\right] \tag{14}$$

If we were then to fire a_{11} again, then the state becomes

$$U.W : \left[[\underline{p}_1, \overline{p}_1], [\underline{p}_1, \overline{p}_1], [\underline{q}_2, \overline{q}_2], [\underline{p}_2, \overline{p}_2], 1\right] \tag{15}$$

where $\underline{q}_2 = \underline{t}_2 + 2\underline{M}_2(\underline{p}_1)$ and $\overline{q}_2 = \overline{t}_2 + 2\overline{M}_2(\overline{p}_1)$. Now there are some consequences of this sequence being taken. In the first place, the fact that a_{11} fired twice means that the condition allowing the firing of a_{21} was not satisfied. In particular, we infer that for the actual $p_2 \in [\underline{p}_2, \overline{p}_2]$ that

$$\underline{t}_2 + 2\underline{M}_2(\underline{p}_1) \quad < \quad p_2 \tag{16}$$

Moreover, the fact that we fired a_{13} instead of $a_{13}||a_{22}$ means that the condition on a_{22} was not satisfied, so that

$$\overline{M}_2(\overline{p}_1) > p_2 \tag{17}$$

Combining equations 16 and 17 implies that

$$2\underline{M}_2(\underline{p}_1) < \overline{M}_2(\overline{p}_1)$$

This equation, however, contradicts the assumption in equation 11 so the occurrence sequence σ_{1a} cannot in fact occur.

The other occurrence sequence of interest, σ_{1c}, has the state in equation 14 after the sequential firing of a_{11} and a_{13}. At this point, however, since we've shown that we cannot fire a_{11}, then we must fire a_{21}. The system state after firing a_{21} is

$$W.U : \left[0, [\underline{p}_1, \overline{p}_1], [\underline{p}_2, \overline{p}_2], [\underline{p}_2, \overline{p}_2], 1\right] \tag{18}$$

Since $c = 1$, we know that we cannot fire $a_{23}||a_{12}$ in a synchronized manner, so the arc leaving this state is labeled with a_{23}. After firing this arc, the state is

$$W.W : \left[0, [\underline{p}_1, \overline{p}_1], 0, [\underline{p}_1, \overline{p}_2], 0\right] \tag{19}$$

which is a synchronized state.

The other possible occurrence sequence is σ_{1b} where the system fires a_{11} to reach the state in equation 13. In this case, however, the condition on the arc labeled $a_{13}||a_{22}$ is satisfied so that arcs a_{13} and a_{22} are fired in a synchronized manner. The system state after this arc's traversal is

$$W.U : \left[0, [\underline{p}_1, \overline{p}_1], [\underline{M}_2(\underline{p}_1), \overline{M}_2(\overline{p}_1)], [\underline{p}_2, \overline{p}_2], 1\right] \tag{20}$$

Note that since $c = 1$,that processor P_2 will not fire a_{23} in a synchronized manner with a_{12}. Therefore, only arc a_{23} is traversed and the final state becomes,

$$W.W : \left[0, [\underline{p}_1, \overline{p}_1], 0, [\underline{M}_2(\underline{p}_1), \overline{M}_2(\overline{p}_1)], 0\right] \tag{21}$$

This, once again, is a synchronized state.

Using similar arguments if a_{21} fires first, we can show that the occurrence sequence $\sigma_{2a} = a_{21}, a_{23}, a_{21}$ cannot occur. We can show that the sequence $\sigma_{2c} = a_{21}, a_{23}, a_{11}, a_{13}$ terminates in a synchronized state of equation 19. Finally, we can show that the occurrence sequence $\sigma_{2b} = a_{21}, a_{23}||a_{12}, a_{13}$ terminates in the synchronized state,

$$W.W : \left[0, [\underline{M}_1(\underline{p}_2), \overline{M}_1(\overline{p}_2)], 0, [\underline{p}_2, \overline{p}_2], 0\right] \tag{22}$$

Since these are the only possibilities from the reachability graph, the proposition's conclusion is proven. •

Remark: The assumption in equation 11 essentially requires that the synchronization periods are sufficiently separated and that our clock uncertainty is small enough so that no overlap occurs between successive synchronization instants. If this condition is violated, then it may be possible for σ_{1a} to fire which could cause the algorithm to fail.

Since the occurrence sequences generated by this automaton are a concatenation of several fundamental sequences, there is a distinct possibility that the periods computed by this algorithm may not always be bounded. The following proposition addresses this issue.

Proposition 4: Under the assumptions of proposition 3, the periods computed by all runs of the synchronization algorithm will generate a sequence of synchronization periods p_i (for $i = 1, 2$) that converge to a bounded value.

Proof: There are essentially three occurrence sequences to consider, σ_{1c}^*, σ_{2c}^*, and $(\sigma_{1c}\sigma_{2c})^*$. In the first two cases, however, the final states are given in equations 21 and 22, respectively. Therefore, we only need to worry about $(\sigma_{1c}\sigma_{2c})^*$. This particular sequence, however, always replaces p_2 with $\overline{M}_2(p_1)$ which can only happen if $p_2 > \overline{M}_2(p_1)$. This means, therefore, that $\overline{M}_1(\overline{M}_2(p_1)) < p_1$ and that

$$\overline{M}_2(\overline{M}_1(\overline{M}_2(p_1))) < \overline{M}_2(p_1) < p_2$$

So this particular occurrence sequence generates a sequence of monotone decreasing periods p_1 and p_2. Since both periods are bounded below by zero, we can immediately conclude that these sequences converge to a bounded value. •

We now briefly discuss the *fault-tolerance* of this algorithm. We restrict our attention to *transient clock* faults that result in incorrect time-stamps. Incorrect time-stamps may also arise due to malicious clock faults, but we do not consider that case in this paper. A given processor, P_i detects a fault when it obtains a set of time-stamps that fail to satisfy the consistency relation in equation 4. Let's assume that processor P_i receives a set of faulty time-stamps, $(\underline{t}, t, \overline{t})$ from neighbor P_j. At this point the processor knows that one of two events has occurred. Either processor P_i or P_j has suffered a transient fault. Note that this isolates the fault to one of two processors. We don't really care which procesor faulted. All that is important is that the clock models are incorrect and this means a new set of clock models need to be identified. In other words, upon detection of a fault, processor P_i should begin checkings its clock models for its neighbors. For all neighbors in which a fault is detected, processor P_i sends a message to recheck the validity of its clock models. Unlike the results in [3], there is no minimum number of non-faulty processors before a global reboot is needed. Our algorithm gets around the theoretical bound in [8] because it is an authenticated algorithm. In our case, the signature function is the function generating the time-stamped messages.

References

[1] Jennifer Lundelius and Nancy Lynch, "An Upper and Lower Bound for Clock Synchronization", In *Information and Control*,62, pp. 190-204, 1984

[2] L. Lamport and P.J. Melliar-Smith, "Synchronizing clocks in the presence of faults", *Journal of the ACM*, 32:1, 1985, pp. 52-78.

[3] J. Lundelius Welch and N. Lynch, " A new fault-tolerant algorithm for clock synchronization", *Information and Computation*, Vol 77:1, 1988, pp. 1-36.

[4] D. Dolev, J.Y. Halpern, B. Simmons, H.R.Strong, "Dynamic Fault-Tolerant Clock Synchronization", *Journal of the ACM*, 42(1):143-185, 1995.

[5] R. Ostrovsky and B. Patt-Shamir, "Optimal and Efficient Clock Synchronization Under Drifting Clocks", *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing(PODC'99)*, 1999.

[6] D.L. Mills, "Internet time synchronization: the Network Time Protocol", *IEEE Trans. Communications*, COM-39,10 (October 1991), 1482-1493.

[7] F. Cristian, "Probabilistic Clock Synchronization", *Distributed Computing*, 3, 1989, pp. 146-158

[8] D. Dolev, J.Y. Halpern, and H.R. Strong, "On the possibility and impossibility of achieving clock synchronization", *Journal of Computer and Systems Science*, 32:2, 1986, pp. 230-250.

[9] F. Cristian, H.R. Strong, D. Dolev, and H. Aghili, "Atomic Broadcast: from simple message diffusion to Byzantine agreement", *Information and Control*, 118:1, 1995.

[10] Eli Fogel and Yih-Fang Huang, "On the value of information in system identification - bounded noise case", *Automatica* , Vol. 18, No. 2, pp. 229-238, March 1982.

[11] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P-H Ho, X. Nicollin, A.Olivero, J. Sifakis, and S. Yovine, The algorithmic analysis of hybrid systems, *Theoretical Computer Science*, 138:3-34, 1995.