August 1987

Adding Time to Synchronous Process Communications

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Recommended Citation

Insup Lee and Susan B. Davidson, "Adding Time to Synchronous Process Communications", . August 1987.

Adding Time to Synchronous Process Communications

Abstract

In distributed real-time systems, communicating processes cannot be delayed for arbitrary amounts of time while waiting for messages. Thus, communication primitives used for real-time programming usually allow the inclusion of a deadline or timeout to limit potential delays due to synchronization. This paper interprets timed synchronous communication as having absolute deadlines. Various ways of implementing deadlines are discussed, and two useful timed synchronous communication problems are identified which differ in the number of of participating senders and receivers and type of synchronous communication. For each problem, a simple algorithm is presented and shown to be correct. The algorithms are shown to guarantee maximal success and to require the smallest delay intervals during which processes wait for synchronous communication. We also evaluate the number of messages used to reach agreement.

Keywords

Ada, CPU, deadline, distributed system, real-time system, synchronous communication, timed synchronous communication

Adding Time to Synchronous Process Communications

INSUP LEE, MEMBER, IEEE, AND SUSAN B. DAVIDSON, MEMBER, IEEE

Abstract—**In distributed real-time systems, communicating processes cannot be delayed for arbitrary amounts of time while waiting for messages. Thus, communication primitives used for real-time programming usually allow the inclusion of a deadline or timeout to limit potential delays due to synchronization. This paper interprets timed synchronous communication as having absolute deadlines. Various ways of implementing deadlines are discussed, and two useful timed synchronous communication problems are identified which differ in the number of participating senders and receivers and type of synchronous communication. For each problem, a simple algorithm is presented and shown to be correct. The algorithms are shown to guarantee maximal success and to require the smallest delay intervals during which processes wait for synchronous communication. We also evaluate the number of messages used to reach agreement.**

Index Terms—**Ada, CSP, deadline, distributed system, real-time system, synchronous communication, timed synchronous communication.**

I. Introduction

A distributed program consists of a set of processes interacting with the environment and communicating among themselves. Reasons for communication are to send data or signal to another process, to synchronize with another process, or to request an action from another process. There has been considerable research in developing different communication primitives depending on the degree of fault tolerance and the nature of synchronization supported [1]. One common primitive is *synchronous communication*, which is provided in languages like CSP [2] and Ada[1] [3]. Synchronous communication means that a receiving process waits to receive a message, and a sending process waits for the message to be received. Thus, the sender and receiver synchronize to exchange a message. The correct implementation of synchronous communication requires the processes to reach agreement on when communication completes.

Consider the example of a multisensor robot system implemented using synchronous communication. In such a system, a coordinator process receives data from several sensor processes, and integrates the sensory data to determine the next task for each process [4]. Examples of a next task could be a new position for an arm with proximity sensors, or a new region for image processing. The coordinator then sends the next task to the sensor processes. Sensor processes wait for the coordinator to inform them of their next task, and then preprocess new sensory data to send to the coordinator.

Processes communicating synchronously may be delayed for arbitrary time periods while waiting for each other. Although unbounded delays may be (marginally) acceptable for processes without timing constraints, real-time processes simply cannot be delayed for arbitrary time periods. In the previous example, if one sensor process is slow to provide its sensory data, the coordinator will be delayed in its decision making, which in turn will cause other sensor processes to wait. This could cause large discontinuities in input sensory data, or discontinuities in next tasks for the sensor processes; for example, an arm with proximity sensors may have to come to a stop rather than complete its motion. In many cases, it would be preferable for the coordinator to make a decision based on partial sensory data received within an acceptable time frame than to allow the system to be stalled due to one late process. Furthermore, it may be preferable for individual sensor processes to continue their current operation rather than stopping completely waiting for the coordinator's decision. Thus, it is important to guard processes from indefinite delays due to synchronization.

Languages such as Ada and Occam[2] [5], which are designed for real-time programming therefore support the notion of a deadline with their synchronous communication constructs. Real-time processes written in these languages can then attach deadlines to their communication requests specifying how long they are willing to wait for successful communication. However, there are several interpretations of synchronous communication using deadlines depending on how message delays are handled and whether the deadlines are interpreted with respect to the sender's or receiver's clock. Our interpretation is that each process involved in the communication can specify a deadline; when the deadline of a process on its own clock is reached, the process must decide whether or not the communication was successful; that is, deadlines are absolute and are measured on the local clock of each process (see [6] for other interpretations). This interpretation makes it possible to statically determine the timing behavior of each process without having to consider run-time communication delays and clock discrepancies of a particular implementation.

This paper reviews three different implementations for

Manuscript received March 20, 1987. This work was supported in part by NSF DCR 8501482, NSF DMC 8512838, NSF MCS 8219196-CER, ARO DAA6-29-84-k-0061, and a grant from AT&T's Telecommunications Program at the University of Pennsylvania.

The authors are with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

IEEE Log Number 8715063.

[1] Ada is a registered trademark of the U.S. Government.

[2] Occam is a trademark of the INMOS Group of Companies.

deadlines associated with synchronous communication. Two variations of timed synchronous communication, illustrated in the example, are then identified: communication between multiple senders and a single receiver (as between several sensor processes and the coordinator for current sensory data), and N-way communication between multiple processes (an important special case being $N = 2$, as between the coordinator and a sensor process for a new task). Simple algorithms for these forms of timed synchronous communication are presented, shown to be correct, and to have good performance. The tradeoffs involved in using each implementation of a deadline are discussed along with the algorithms.

The rest of the paper is organized as follows. In the next section, we present our assumptions and correctness criteria, discuss the implementation of deadlines, and review related research. Section III presents the case of multiple senders and a single receiver; Section IV extends this to N-way synchronous communication. The last section summarizes the pros and cons of the chosen interpretation, discusses how to improve the fault tolerance of the algorithms, and closes with directions for future research.

II. Problem Definition

The timed synchronous communication problem can be defined as follows. Processes execute at their own speed and wish at various times to synchronously communicate with other processes. Sending and receiving processes both specify deadlines, which indicate the time by which communication must be achieved. The deadline D_p of a sending process P, measured according to P's local clock, means that P must know by D_p at the latest that the message has either been accepted by Q or that communication has failed. The deadline D_q of a receiving process Q, measured according to Q's local clock, indicates the deadline by which the message must be received. Furthermore, P and Q must agree on whether or not the message was accepted. Thus, the sending process can start executing its subsequent statement by time D_p, and the receiving process can start executing its next statement by time D_q.

This paper considers implementation issues for two variations of the timed synchronous communication problem defined above. The first problem assumes that there are many senders and one receiver; the receiver accepts a message from at most one sender [2]. Timed synchronous communication is achieved if the chosen sender and the receiver both decide within their respective deadlines that the message has been received, and all other senders know that their communication failed. In the second problem there are N processes, and each process sends a message to the other $N - 1$ processes [7]. N-way synchronous communication succeeds if each process receives the $N - 1$ other messages before its deadline, and knows that the other processes will also receive all their messages before their deadlines. The fundamental difference between these problems is that the first involves the element of choice while the second does not. Thus, in the first problem, since a sender does not necessarily know about the existence of other senders or of the decision process of the receiver, the chosen sender must be explicitly notified by the receiver. Communication between a single sender and a single receiver is therefore considered to be a special case of the second problem.

A. Assumptions and Definitions

The environment assumed is a system of distributed processes which exchange information using the primitive operations *send*(M), which places message M on the transmission queue of the sending process, and *receive*(M), which takes a waiting message off the receiving queue of the receiving process. We say that a message *is delivered to* or *has arrived at* a target process if the message has been transmitted and is queued to be received by the target process. A message is *received* by a target process when the process executes a *receive* command for the message. For simplicity, we make the following assumptions about the network and processes:

Assumption 1: The system is a completely connected network of perfect processors.

This is a very strong assumption, and is made to simplify presentation of the basic algorithm. Failures are considered in the conclusion.

We make two common assumptions about messages and their delivery. The first is that messages are sequenced correctly with respect to their sender, and the second that the delivery time d is bounded.

Assumption 2: Messages from the same process are received in the order in which they are sent.

Assumption 3: If process P sends a message to process Q at time t (measured according to process P's clock), then the message will arrive at process Q by time $t + d$ (measured according to process P's clock).

The next two assumptions are that messages can be sent and delivered asynchronously; that is, if a message is sent at time t, it will be delivered to Q by time $t + d$, although Q may not actually receive the message until a later time. Furthermore, Q does not have foreknowledge of when delivery will occur. Q also does not know what P's deadline is unless P explicitly communicates the information.

Assumption 4: If a message arrives at a process Q before Q is ready to receive the message, the message is queued.

Assumption 5: It is not known *a priori* when a process is ready to communicate and when its deadline will be. Only when a process is ready to communicate does its deadline become known.

When presenting the algorithms we need to express timing constraints within a program, and will therefore use the notion of temporal scopes presented in [8]. The syntax of a temporal scope is

```
within D do
    ⟨statement list⟩
    when a message arrives do
        ⟨statement list⟩
    end when
    ⟨statement list⟩
exception
    ⟨exception handling statements⟩
end within.
```

If the current time reaches deadline D while the process is waiting for message arrival, the deadline exception is raised and handled within the exception handler part of the temporal scope. We assume that deadlines cannot expire while executing other parts of its program since the execution times of statement lists are negligible compared to the transmission delay and the granularity of a clock.

B. Timing and Deadlines

It is well known that local clocks are not perfectly accurate and that maintaining some approximation of time is a difficult problem in a network of distributed processes [9]–[12]. Since deadlines build on the notion of time, it seems strange that no assumptions were made about local clocks in the previous section. Assumption 3, however, is an implicit assumption about local clocks: the bound on message delivery d is an *elapsed* time and must be guaranteed to be accurately measured by each process on its local clock. As will become clear later in this section, Assumption 3 is the weakest assumption about local clocks that we can make. Whether or not further assumptions must be made depends on the way deadlines are implemented.

There are basically three ways to implement deadlines. The correctness of each implementation follows trivially from Assumption 3 and whatever additional assumptions are made; proofs are therefore omitted. In the remainder of this section, P represents a sending process with deadline D_p, and Q represents a receiving process. We assume that an initial message from P contains both data and control information and an accept message from Q contains only control information.

The first approach makes an additional assumption about the synchronization of local clocks [13]: if the maximum drift between any two clocks in the network can be bounded by e and processes advertise their deadlines[3], then d, e, and D_p can be used by process Q to determine the latest time an accept message can be sent and guaranteed to be delivered to P by D_p.

Protocol 1 (Clock Drift): If Q sends an accept message to P before $D_p - d - e$ according to Q's clock, then it will be delivered to P before D_p according to process P's clock. However, if Q sends an accept message to P after $D_p - d - e$, delivery before D_p cannot be guaranteed.

Therefore, if process Q wants to make sure that process P receives a message before deadline D_p (P's clock), then Q must send the message before $D_p - d - e$ (Q's clock).

A variation on advertising deadlines is to advertise $\Delta_p = D_p - R_p$, and make an assumption about clock rates to translate this time period for P to that for the receiver Q [14]. Assume that P and Q are *tame* during the time that they attempt communication: process P is said to be tame if the rate at which its clock ticks with respect to some imaginary perfect clock can be bound by two fixed constants $r_{\min}$, $r_{\max}$. If P sends out an initial message advertising the time duration that it is willing to wait, Δ_p, then Q can interpret this with respect to its clock as $\lfloor (r_{\min}/r_{\max})\Delta_p \rfloor$ ticks. Note that P and Q are assumed to be tame only during the interval that they attempt communication.

Protocol 2 (Clock Rate): Let $A_{p,q}$ be the time at which P's initial message is delivered to Q according to Q's clock. If Q sends an accept message to P before $A_{p,q} - 2d + (r_{\min}/r_{\max})\Delta_p$ according to Q's clock, then it will be delivered to P before D_p according to P's clock. However, if Q sends an accept message to P after $A_{p,q} - 2d + (r_{\min}/r_{\max})\Delta_p$, delivery before D_p cannot be guaranteed.

The third approach uses only d, and does not assume that deadlines are advertised in any way. However, process P must send a last call message to define the latest time by which an accept message must be sent by Q to guarantee that it is delivered by D_p.

Protocol 3 (Last Call): If P sends a last call message to Q at $D_p - 2d$ according to process P's clock, then an accept message sent by Q before the last call message is delivered to Q will be delivered to P before D_p according to process P's clock.

All of the above implementations of deadlines use the assumption of a maximum communication delay. Since this is a strong assumption, and since d is large compared to an execution step of a program or even the actual delivery time of a message, it would be nice if there were an implementation that did not need d. Unfortunately, we cannot remove this assumption, as the next lemma shows.

Lemma 1: If maximum communication delay is not bound, there is no nontrivial algorithm guaranteeing that two asynchronous processes will reach agreement within their respective deadlines.

Proof: If maximum communication delay is not bound, then message delivery can be infinite, i.e, it can fail. However, it is well known that if communication is unreliable, it is not possible for two processes to reach agreement [15]. This is true even if there is no deadline involved, i.e., deadlines are infinite. □

In some environments, d is much smaller than e due to relatively large clock resolution compared to message transmit time. In [16], it is noted that 60 Hz "line" clocks commonly used on current work stations are only accurate to 16 ms. On the other hand, 4–8 ms intersite message transit times are common and 1–2 ms are reported increasingly often. Thus, it is impossible to synchronize clocks to better than 32–48 ms, enough time for a pair of sites to exchange between 4 and 50 messages. However, it is reasonable to assume that elapsed-time clocks are accurate, that is, $r_{\min}/r_{\max}$ is generally close to 1. For example, a VAX 750 is accurate to 1 μs ± 0.01 percent, so $r_{\min}/r_{\max} \approx 0.9998$.

In other environments, d may change as a function of the load on the network and thus become quite large. For example, when contention is high on a shared bus, d can grow. Note that Protocol 1 only requires that the receiver knows the *current* value of d before sending an accept message, and can therefore be used when d is not a fixed constant or when a message is broadcast without underlying system support for the capability. On the other hand, Protocol 2 requires that the receiver knows d for both the initial and accept messages; whereas, Protocol 3 requires the sender to know d for both its

[3] Note that e is also an elapsed time which needs to be accurately measured by processes when necessary.

last call message and an accept message. Having said this, we will use d loosely throughout the rest of the paper as if it were a constant.

C. Evaluation Criteria

Algorithms for solving timed synchronous communication problems should not only be correct but have good performance. Correctness implies two conditions.

1) Processes must agree on the success or failure of the communication.

2) Processes must decide by their respective deadlines.

Criteria for good performance should, in our opinion, include the following:

1) How often communication succeeds. Trivial algorithms can easily be thought up that are "correct" but never allow successful communication.

2) The number of messages exchanged to reach agreement about the success or failure of synchronous communication.

3) The delay interval during which each process waits for successful synchronous communication. This includes the overhead necessary to achieve a consensus among participating processes.

D. Related Work

A recent paper has considered various implementations of the timed entry call for distributed Ada programs [6]. The motivation for their work is the rendezvous construct, in which the sender is blocked until the remote entry code is executed; to avoid delay, Ada allows the sender process to specify a deadline D_p. The authors choose to interpret this deadline as the latest time by which the receiver must be ready to execute the entry code. This means that the sender may be delayed beyond D_p if the rendezvous fails since knowledge that the receiver cannot meet the deadline has to be relayed back to the sender. In this context, the inability of the sender to specify an absolute deadline by which time it must know of the failure of the rendezvous makes sense since even if the timed entry call succeeds the sender is blocked until the code is completed, which may be beyond D_p. This is an inherent problem with Ada [17]. In the setting of synchronous communication where the sender is not blocked if communication is successful, we feel that senders and receivers should be treated uniformly with absolute deadlines.

Reif and Spirakis also propose a probabilistic implementation for synchronous interprocess communication [14]. The method guarantees the following "real-time" response: if a pair of processes are mutually willing to communicate during some global time interval Δ and are tame during Δ, then they establish communication within Δ with *high likelihood* in the worst case. Note that our definition of real-time differs from theirs in two respects: 1) deadlines are defined the minute a process is willing to communicate, not when two processes are mutually willing to communicate, and 2) deadlines are guaranteed, not probable.

III. Timed Synchronous Communication between Multiple Senders and One Receiver

The problem of timed synchronous communication with multiple senders and one receiver is as follows. The receiver accepts at most one message within its deadline. Each sender must be able to determine by its deadline whether its message is accepted. Communication *succeeds* if some message is accepted by the receiver.

We assume one-way naming for communication as in Ada; that is, only the senders identify the receiver. We also assume that the senders do not know the identities of other sending processes. Since the senders, therefore, cannot communicate with each other, the receiver must choose a sender and notify it of successful communication. Obviously, the receiver cannot decide which sender will be successful until it accepts a message from the sender. In order to guarantee that the chosen sender and the receiver agree on the success of communication, the deadline of the chosen sender must be late enough to be able to receive the accept message from the receiver. Finally, the receiver cannot conclude with certainty that communication has failed until it has unsuccessfully received messages from all senders, or timed out, whichever is earlier.

Let $P_1, \cdots, P_n$ be senders and Q be the receiver. Let $[R_i, D_i]$ be the time interval during which process P_i is willing to communicate. If there are no timing constraints, then the D_i's are assumed to be ∞. Let A_i be the time at which process Q receives a message from process P_i (according to Q's clock). Fig. 1 describes our algorithm using deadlines implemented with clock drifts. The main idea is that the receiver waits for a message from senders and accepts the message if the deadline of the sender is late enough to receive the accept message.

Theorem 1: Algorithm 1 is correct.

Proof: To be correct, the algorithm must guarantee that

i) all processes decide by their respective deadlines;

ii) at most one sender succeeds in communication; and

iii) decisions are consistent.

The fact that i) and ii) are true is trivial due to the "within" construct and the return statement in the loop of the send function. To see that iii) is true, note that an accept message is only sent to a sender if it is successful. Therefore, unsuccessful senders will timeout and fail. Now let P_k be the sending process whose message is accepted. Since $D_k >$ *current time* $+ d + e$, the accept message must arrive at P_k in time, and P_k will return SUCCESS. □

We now show that Algorithm 1 guarantees maximal success in communication among all algorithms based on clock drifts for reaching agreement.

Lemma 2: For every algorithm using clock drifts to solve the multiple senders and one receiver problem, if communication succeeds, then there is a process P_i such that i) $A_i \leq D_q$ and ii) $D_i \geq A_i + d + e$.

Proof: Since the receiver does not know the identities of senders until it receives messages from them, it must initially wait for messages. To show that i) is necessary, suppose that there is no P_i such that $A_i \leq D_q$. Then no messages arrive by D_q, and communication cannot possibly succeed. To show that ii) is also necessary, assume that a message arrives at $A_i \leq D_q$, containing deadline $D_i < A_i + d + e$. Q cannot accept this message since it cannot be sure whether its message will be delivered before D_i. □

Theorem 2: The algorithm guarantees maximal success in communication.

Proof: It suffices to show that if there is process P_i such

Algorithm 1

```
function send (Q: process_id; msg: msg_type; $D_i$: time)
begin
    transmit ⟨msg,$D_i$⟩ to process Q
    within $D_i$ do
        when ("accepted") from Q arrives do
            return (SUCCESS)
        end when
    exception
        return (TIMEOUT)
    end within
end send

function receive (var P : process_id; var buf: msg_type; $D_q$: time)
begin
    within $D_q$ do
        loop
            when ⟨msg,D⟩ from process P arrives do
                if $D > current\ time + d + e$
                  then transmit ("accepted") to process P
                      buf := msg
                      return (SUCCESS)
            end when
        forever
    exception
        return (TIMEOUT)
    end within
end receive
```

Fig. 1. Timed communication between many senders and one receiver based on clock drifts.

that $A_i \leq D_q$ and $D_i \geq A_i + d + e$, then communication succeeds. Assume that there is such a P_i. If Q has not decided by A_i, then Q can accept the message from P_i and send an acknowledgment to P_i before D_i (P_i's clock). Otherwise, communication has already been successful. □

Algorithm 1 can be modified to use clock rates or last call instead of clock drifts. To use clock rates, process P_i sends Δ_i with the message to the receiver; the receiver accepts the message only if it has not accepted another message and $\Delta_i > (r_{\max}/r_{\min}) * 2d$. Similarly, to use last calls instead of clock drifts, if $D_i - R_i < 2d$, a sender P_i does not send a message and decides that communication fails. Otherwise, it sends a message at time R_i; if it does not receive an accept message from the receiver by $D_i - 2d$, it sends a last call message. P_i concludes that communication is successful only if it receives an accept message before its deadline. Furthermore, the receiver accepts a messsage from P_j only if it has not received P_j's last call message. Both of these modifications can easily be shown to be correct and to guarantee maximal success in communication among all algorithms based on their interpretation of deadlines.

One factor to consider in choosing between the various implementations of deadlines for the multiple senders and one receiver problem is the *minimum deadline* that can be specified. That is, suppose a sender P wishes to have a nontrivial solution in the best case of synchronous communication, where a receiver Q is willing to communicate at the moment that P's initial message is received and has a deadline that is long enough to process the request. What is the minimum time interval P can specify? This obviously depends on the implementation method and values of d, e, and $(r_{\min}/r_{\max})$. Let c be the actual communication delay for P's message to Q, and recall that $A_{p,q}$ is the time with respect to Q's clock that the message is received. That is, $A_{p,q}$ according to P's clock equals $R_p + c$. Then Protocol 1 (Clock Drifts) requires that $D_p - R_p \geq c + d + e$ since Q must use worst case return message delivery assumptions to be able to accept the synchronization; Protocol 2 (Clock Rates) requires that $D_p - R_p = \Delta_p \geq (r_{\max}/r_{\min}) * 2d$ since Q must use worst case assumptions for interpreting Δ_p and for delivery of the initial and return messages; and Protocol 3 (Last Call) requires that $D_p - R_p \geq 2d$ since P must use worst case assumptions in sending out its last call message at $D_p - 2d$. Thus, if $d > e + c$ then clock drifts allows a smaller $D_p - R_p$ than either clock rates or last call for successful communication. If $d < e + c$ then last call allows the smaller $D_p - R_p$; however if $r_{\max}/r_{\min}$ is close to one, then clock rates is preferable due to the message overhead of a last call. In general, if e is larger than d, or if c is close to d, then clock rates and last call allow smaller deadlines than clock drifts.

Another factor is the number of messages exchanged. Since senders do not know about each other and only senders know the receiver, the minimal number of messages is $n + 1$, which is achieved by our algorithms based on clock drifts or rates. For unsuccessful senders, there is a tradeoff between messages and delay. In our algorithms, each unsuccessful sender must timeout.[4] If the unsuccessful sender's message reaches the receiver in time, but not in time for a guaranteed response, a "reject" message could be sent. However, since it is not guaranteed to reach the sender before its deadline and the action is the same in either case, we do not feel the overhead is worthwhile.

IV. N-WAY SYNCHRONOUS COMMUNICATION WITH DEADLINES

The problem of timed N-way synchronous communication can be stated as follows. Let $P_1, \cdots, P_n$ be a set of processes. Each process executes independently and wishes at various times to synchronize with the other processes. For each process P_i, there is a deadline D_i by which the process must decide whether the N-way communication was successful. Furthermore, processes $P_1, \cdots, P_n$ must agree on the result. For $N = 2$, this problem is synchronous communication between two processes, where each process knows the identity of the other.

Fig. 2 describes an algorithm for N-way synchronous communication using deadlines implemented as clock drifts. We use EA_i for the expected arrival time of a message sent by P_i measured according to P_i's clock (i.e., $EA_i = R_i + d$). For this problem, we extend Assumption 1 so that d denotes the maximum communication delay for broadcasting a message from P_i to $P_1, \cdots, P_n$. The main idea behind the algorithm is as follows: each process broadcasts a message with an expected arrival time and deadline. Processes that receive messages from all other processes use this information to decide whether the other processes also accept all other messages before their deadlines.

Lemma 3: Let EA and EA' be the two largest EA_i's ($EA > EA'$) and D and D' be the two smallest D_i's ($D < D'$). If either i) EA and D belong to two different processes and $EA \leq D - e$, or ii) EA and D belong to the same process and $EA \leq D' - e$ and $EA' \leq D - e$, then every process receives all messages before its deadline.

[4] In the algorithm based on last call, a sender with not enough slack (i.e., less than $2d$) in deadline immediately decides that communication must fail without sending a message.

```
Algorithm 2

function CheckTiming ($EA_1, \ldots, EA_n, D_1, \ldots, D_n$ : time)
begin
    Let $EA$ and $EA'$ be the two largest $EA_i$'s in that order
    Let $D$ and $D'$ be the two smallest $D_i$'s in that order
    if $EA$ and $D$ are from the same process
      then if $EA' \leq D - e$ and $EA \leq D' - e$
              then return (SUCCESS)
              else return (TIMEOUT)
      else if $EA \leq D - e$
              then return (SUCCESS)
              else return (TIMEOUT)
end CheckTiming


function n_synchronization ([$P_1, \ldots, P_n$]: set of process_id; $D_i$: time)
var count : $0..n-1 = 0$
begin
    $EA_i$ := current time + $d$
    broadcast ($msg, P_i, EA_i, D_i$) to $P_1, \ldots, P_n$
    within $D_i$ do
        while count $< n-1$ do
            when ($EA_j, D_j$) from process $P_j$ arrives do
                count := count + 1
            end when
        end while
    exception
        return (TIMEOUT)
    end within
    if CheckTiming ($EA_1, \ldots, EA_n, D_1, \ldots, D_n$) = SUCCESS
      then return (SUCCESS)
      else return (TIMEOUT)
end n_synchronization
```

Fig. 2. Timed N-way synchronous communication based on clock drifts.

Proof: i) Since for every P_i and P_j, $EA_i \leq EA$ and $D_j \geq D$, if $EA \leq D - e$ then $EA_i \leq D_j - e$. Therefore, every P_j receives a message from every P_i by its deadline. This condition is stronger than necessary if EA and D belong to the same process, and is weakened in ii).

ii) Let EA and D belong to the same process P_i. If $EA \leq D' - e$, then $EA_i \leq D_j - e$ for every other P_j, hence P_i's message will arrive at every other P_j in time. Similarly, if $EA' \leq D - e$, then $EA_j \leq D_i - e$ for every other process P_j, hence P_i will receive a message from every other P_j in time. Now, since $EA' \leq D' - e$, we can use the same reasoning as in i) to show that for every $j, k \neq i$, P_j and P_k receive messages from each other by their deadlines. □

Theorem 3: Algorithm 2 is correct.

Proof: To be correct, the algorithm must guarantee that

i) all processes decide by their respective deadlines, and

ii) decisions are consistent.

The fact that i) is true is trivial since it is guaranteed by the "within" construct. To prove ii), suppose that for some $i \neq j$, P_i returns SUCCESS, but P_j returns TIMEOUT. Then P_i must have received messages from every other process and executed *CheckTiming*. Furthermore, since *CheckTiming* evaluates the conditions in Lemma 3 and returns SUCCESS only if they are true, every other process must have received all messages in time. Therefore, P_j must also have executed *CheckTiming*. But this is a contradiction since if P_j had executed *CheckTiming*, it also would have had to return SUCCESS. □

We now show that Algorithm 2 guarantees maximal success in communication among all algorithms based on clock drifts for reaching agreement. We first show the necessary condition for successful communication and then prove that the necessary condition is also a sufficient condition for our algorithm.

Lemma 4: For every algorithm based on clock drifts, if for some $j \neq i$

$$R_i + d + e > D_j$$

then synchronous communication within deadlines among $P_1, \cdots, P_n$ must fail.

Proof: Let P_i and P_j, $i \neq j$, be such processes. If $R_i + d > D_j - e$, then the message from P_i cannot be *guaranteed* to be delivered before D_j. If $R_i \leq D_j$ and the actual delay and clock drift are 0, the message may in fact be delivered at P_j by D_j. Since P_i cannot know for sure that the message has been delivered, some form of acknowledgments would have to be used. However, even if acknowledgments are used to verify this fact, there is no *finite* algorithm to guarantee that both processes will know that the message is accepted. This is due to the "unreliability" of the acknowledgments, which can be thought of as lost if it arrives after the deadline of the receiving process. Therefore, P_j must decide not to accept the message since P_i cannot be sure of the outcome.

To see that $R_j + d > D_i - e$ implies that communication must fail, note that P_i cannot conclude with certainty that communication was successful unless it receives some message from P_j. However, as argued above, even if $R_j + d > D_i - e$, P_i may receive a response from P_j if $R_j \leq D_i$ and the actual clock drift and communication are 0. P_j cannot count on this response being received by P_i. Some form of acknowledgment is therefore necessary, which does not work as shown above. □

Theorem 4: The algorithm guarantees maximal success in N-way synchronization.

Proof: Because of Lemma 4, it suffices to show that if $R_i + d_i \leq D_j - e$ for all $i \neq j$ then communication succeeds. However, this is implicitly tested in *CheckTiming* (see proof of Lemma 3). □

Algorithm 2 can also be modified to use clock rates or last call instead of clock drifts. To use clock rates, a process P_i broadcasts a message including Δ_i when it becomes ready to synchronize. Communication succeeds if P_i receives messages from all other processes and, for all j, k,

$$\Delta_j + A_{j,i} \geq A_{k,i} + (r_{\max}/r_{\min}) * 2d,$$

where $A_{i,i} = R_i$, and $A_{j,i}$, $j \neq i$, is the time at which P_j's message is arrived at P_i according to P_i's clock. To use last call, a process P_i does not send any messages if $D_i - R_i < 2d$ or it received a last call message from some P_j. Otherwise, it broadcasts a message at time R_i and waits for messages from others. If it does not receive messages from all the others by $D_i - R_i - 2d$, then it broadcasts the last call message. It then eventually times out or receives messages from all other processes, in which case communication succeeds. The modified algorithms can easily be shown to be correct and to guarantee maximal success among all algorithms based on their interpretation of deadlines as in Theorems 3 and 4.

Ideally, N-way synchronous communication should succeed whenever the synchronizing periods overlap in some global time frame. While this is a necessary condition as shown in the following lemma, it cannot always be achieved due to clock synchronization and message delays. Let $[R_i, D_i]_r$ be the time interval with respect to a global time frame r (such as EST) corresponding to $[R_i, D_i]$ according to P_i's clock.

Lemma 5: If P_1 with $[R_1, D_1]$, $\cdots$, and P_n with $[R_n, D_n]$ successfully synchronize, then

$$\bigcap_{i=1}^{n} [R_i, D_i]_r \neq \phi.$$

Proof: Let P_i and P_j, $i \neq j$, be any two processes. If D_i occurs before R_j in r, then even a message with no delay cannot arrive in time to be accepted by P_i. Similarly, D_j must be greater than or equal to R_i. Thus, $[R_i, D_i]_r$ and $[R_j, D_j]_r$ must overlap. The lemma follows from the fact that the intersection of all closed intervals on real numbers that are pairwise nondisjoint is not empty. □

This lemma confirms our intuitive notion of what it means to synchronize N processes. That is, under any protocol, if N processes succeed in synchronization within their respective deadlines, there is a common time (interval) during which all N processes are willing to synchronize. In our protocol, each process decides on the success of communication as soon as it possibly can, that is, when it receives all the messages from the other processes, or (in the absence of any message) at its deadline.

As was the case for the multiple senders and one receiver problem, the minimal time intervals needed by the P_i's for successful communication depend on the protocol used and values of d, e, and $(r_{\min}/r_{\max})$. Suppose all P_i's are ready to communicate at the same time with respect to some global clock. Then, for successful communication, clock drifts requires that $D_i - R_i \geq d + e$ for all P_i; clock rates requires that $D_i - R_i = \Delta_i \geq (r_{\max}/r_{\min}) * 2d$ for all P_i; and last call requires that $D_i - R_i \geq 2d$ for all P_i. Thus, if $d > e$ then clock drifts requires a smaller $D_i - R_i$ than either clock rates or last call for successful communication. If $d < e$ then last call requires the smaller $D_i - R_i$; however, if $(r_{\max}/r_{\min})$ is close to one, then clock rates is preferable due to the message overhead of last calls. If the minimum communication delay, say $d_{\min}$, is known, the algorithm based on clock rates can be improved to require $\Delta_i \geq (r_{\max}/r_{\min}) * 2d - d_{\min}$ for successful communication.

Each process sends a message to $N - 1$ other processes. Thus, the algorithms based on clock drifts and rates generate $N(N - 1)$ messages and the algorithm based on last calls generates $2N(N - 1)$ messages in the worst case. If the underlying communication medium supports broadcasting, the cost may be close to linear in the number of processes involved (rather than quadratic). To reduce the number of messages to $O(N)$, a coordinator could be used. However, this would also reduce the interval during which successful communication could take place since the coordinator would have to broadcast the result to the participants. Note that for $N = 2$, the algorithms based on clock drifts and rates use the minimum number of messages since at least two messages are needed in any algorithm.

V. Conclusions and Directions for Future Work

Timed synchronous communication is an important concept for real-time systems. In this paper, we first talked about the general problem of implementing deadlines, and then presented two timed synchronous communication primitives. We described simple algorithms for implementing these primitives in a distributed system, and discussed how the various implementations of deadlines would change the basic algorithm. Each algorithm was shown to be correct and to guarantee maximal success; the number of messages exchanged to reach agreement on the success or failure of synchronous communication was also evaluated.

Both of the primitives presented in this paper require a bound on communication delay d. This adversely affects the minimum deadline that can be specified since it must be larger than d under any of the proposed implementations. However, d is commonly quite large compared to the actual communication delay, and in some cases simply cannot be measured, i.e., is infinite. Unfortunately, if deadlines are absolute and processes must always agree on the success or failure of communication, this bound is necessary as shown in Lemma 1. That is, either timed synchronous communication is interpreted with absolute deadlines for both the sender and the receiver and minimal deadlines must be longer than d, or minimum deadlines can be shorter than d, in which case not all processes can be guaranteed to know by their deadlines as to the success of communication (the approach taken in [6]) or processes will only have some "probability" of success by their deadlines.

To counter this objection about minimum deadlines, one should remember that the deadline specified by a process under the proposed implementation is a "worst case" statement of how long the process will have to wait to know the outcome of the desired communication. Frequently, processes will know much sooner than their deadlines, especially in the case that processes are well matched and wish to communicate at approximately the same time. Absolute deadlines for all processes also makes it possible to statically reason about the temporal behavior of a program. We feel that relaxing the deadline of some process not only makes it difficult to determine the temporal behavior of a set of communicating processes, but elevates implementation dependent considerations to the programmer level. That is, the brunt of the problem of bounding communication has been shifted from the system to the programmer.

A more serious consequence of needing an upper bound on d is that the algorithms cannot tolerate message failures and guarantee consistent decisions. In the multiple senders case, if the accept message from the receiver is lost or takes longer than d, the successful sender will timeout and decide that communication failed, while the receiver will assume that it was successful. In fact, under this failure assumption, no finite protocol can guarantee agreement between the sender and the receiver [15]. That is, if decisions are to be consistent, the guarantee that the sender will know by its deadline should be relaxed. One technique for doing this is to assume that messages will be delivered within d' with some high probability $p < 1$. The use of d' requires slight changes to the algorithms. For the case of multiple sends and one receiver, the receiver sends a reject message to a sender if communication is not successful. Each sender sends a message only if its deadline is greater than $2 * d' + \epsilon$, where ϵ takes into account discrepancies between local clocks, and waits for an accept or reject message from the receiver. Senders will wait beyond their deadline with very low probability, much less than $2 * (1 - p)$ if deadlines are greater than $2 * d'$. Exception handling techniques could also be used if messages were not received within an acceptable time [6]. For N-way synchronization, a process broadcasts to others a reject message if it is not ready to synchronize by the earliest deadline of messages received so far, and broadcasts an accept message otherwise. Here, a process succeeds if it has not sent a reject message and received accept messages from all others. In this scheme, a reject message may have to be sent even if a process is not ready to communicate when a message arrives.

The algorithms can, however, handle "clean" processor failures in which the state prior to failure is remembered and messages delivered during failure are remembered. In the multiple senders case, if the receiver fails before sending its decision, all senders will assume that communication is not successful; if it fails after sending the decision, the successful sender will receive the accept message, and the receiver will remember its decision upon recovery. If a sender fails, it will receive outstanding messages upon recovery and make the correct decision. If faulty processes can be identified, the multiple senders case can be changed so the receiver attempts to synchronize with a *nonfaulty* process before its deadline, at the expense of longer delays. The N-way communication could also be reinterpreted to avoid failing whenever some process fails, and extended to identify the set of processes that succeed in synchronization.

It is probably not reasonable to consider more drastic or bizarre types of failures in this environment. Agreement problems such as two-phase commit have been found to have fairly severe limitations in the face of failures which partition the network (see [18]), making it unlikely that N-way synchronization will succeed under this failure assumption. The problem of reaching agreement with Byzantine failures has also been extensively studied (see [19], [20] for overviews of the problem and solutions). The solutions necessarily allow the agreement to take time proportional to the number of processes that can fail. This is not acceptable in a real-time environment.

References

[1] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *ACM Comput. Surv.*, vol. 15, pp. 3–43, Mar. 1986.

[2] C. A. R. Hoare, "Communicating sequential processes," *Commun. Ass. Comput. Mach.*, pp. 75–83, Feb. 1981.

[3] U.S. Department of Defense, "Ada programming language," 1983. ANSI/MIL-STD-1815A-1983.

[4] R. P. Paul, H. F. Durrant-Whyte, and M. Mintz, "A robust, distributed sensor and actuation robot control system," in *Proc. Third Int. Symp. Robotics Res.* Cambridge, MA: MIT Press, 1986, pp. 93–100.

[5] INMOS Limited, *Occam Programming Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[6] R. A. Volz and R. N. Mudge, "Timing issues in the distributed execution of ada programs," *IEEE Trans. Comput.*, vol. C-36, pp. 449–459, Apr. 1987.

[7] K. G. Shin, "Intertask communications in an integrated multi-robot system," Tech. Rep. RSD-TR-4-85, Robot Syst. Div., Center for Research on Integrated Manufacturing, College of Eng., Univ. Michigan, 1985.

[8] I. Lee and V. Gehlot, "Lanugage constructs for distributed real-time programming," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1985.

[9] D. Dolev, J. Halpern, B. Simons, and R. Strong, "Fault-tolerant clock synchronization," in *Proc. ACM Symp. Principles Distributed Comput.*, Aug. 1984.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558–565, July 1978.

[11] L. Lamport and P. M. Melliar-Smith, "Byzantine clock synchronization," in *Proc. ACM Symp. Principles Distributed Comput.*, Aug. 1984.

[12] J. Lundelius and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," in *Proc. ACM Symp. Principles Distributed Comput.*, Aug. 1984.

[13] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Trans. Programming Languages Syst.*, vol. 6, pp. 254–280, Apr. 1984.

[14] J. H. Reif and P. G. Spirakis, "Real-time synchronization of interprocess communications," *ACM Trans. Programming Languages Syst.*, pp. 215–238, Apr. 1984.

[15] J. L. Peterson and A. Silberschatz, *Operating System Concepts*, 2nd ed. Reading, MA: Addison-Wesley, 1985, pp. 497–498.

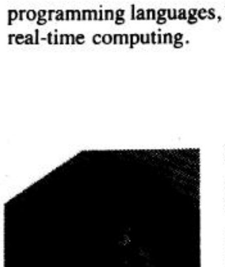
[16] K. Birman and T. Joseph, "Communication support for reliable distributed computing," Tech. Rep., Dep. Comput. Sci., Cornell Univ., 1986.

[17] P. N. Hilfinger, *Abstraction Mechanisms and Language Design*. Cambridge, MA: MIT Press, 1983.

[18] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed database system," in *Proc. Fifth Int. Workshop Distributed Data Management Comput. Networks*, 1981.

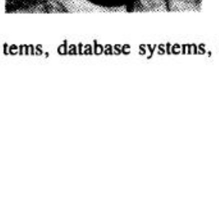
[19] M. J. Fisher, "The concensus problem in unreliable distributed systems (a brief survey)," Tech. Rep. YALEU/DCS/RR-273, Dep. Comput. Sci., Yale Univ., June 1983.

[20] H. Garcia, F. Pittelli, and S. Davidson, "Applications of byzantine agreement in database systems," *ACM Trans. Database Syst.*, vol. 11, pp. 27–47, Mar. 1986.



Insup Lee (S'80–M'82–S'82–M'83) received the B.S. degree in mathematics from the University of North Carolina, Chapel Hill, in 1977, and the M.S. and Ph.D. degrees in computer science from the University of Wisconsin, Madison, in 1978 and 1983.

Since 1983, he has been an Assistant Professor in the Department of Computer and Information Science, University of Pennsylvania, Philadelphia. His research interests include interconnection network synthesis algorithms, process mapping algorithms, programming languages, formal models, and operating systems for distributed real-time computing.



Susan B. Davidson (M'83) received the B.A. degree in mathematics from Cornell University, Ithaca, NY, in 1978, and the M.A. and Ph.D. degrees in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1980 and 1982.

She is currently an Assistant Professor in the Department of Computer and Information Science at the University of Pennsylvania, Philadelphia, where she has been since 1982. Her research interests include fault tolerance, distributed systems, database systems, and real-time systems.