May 1991

Timed Atomic Commitment

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

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

Victor Wolfe
University of Pennsylvania

Timed Atomic Commitment

Abstract

In a large class of hard-real-time control applications, components execute concurrently on distributed nodes and must coordinate, under timing constraints, to perform the control task. As such, they perform a type of atomic commitment. Traditional atomic commitment differs, however, because there are no timing constraints; agreement is eventual. We therefore define *timed atomic commitment* (TAC) which requires the processes to be functionally consistent, but allows the outcome to include an exceptional state, indicating that timing constraints have been violated. We then present centralized and decentralized protocols to implement TAC and a high-level language construct that facilitates its use in distributed real-time programming.

Keywords

atomic commitment, distributed protocols, distributed real-time systems, fault tolerance, language constructs

Comments

Timed Atomic Commitment

Susan B. Davidson, *Member, IEEE,* Insup Lee, *Member, IEEE,* and Victor Wolfe, *Student Member, IEEE*

***Abstract*— In a large class of hard-real-time control applications, components execute concurrently on distributed nodes and must coordinate, under timing constraints, to perform the control task. As such, they perform a type of atomic commitment. Traditional atomic commitment differs, however, because there are no timing constraints; agreement is eventual. We therefore define *timed atomic commitment* (TAC) which requires the processes to be functionally consistent, but allows the outcome to include an exceptional state, indicating that timing constraints have been violated. We then present centralized and decentralized protocols to implement TAC and a high-level language construct that facilitates its use in distributed real-time progra;:ming.**

***Index Terms*—Atomic commitment, distributed protocols, distributed real-time systems, fault tolerance, language constructs.**

I. Introduction

In a large class of hard-real-time control applications, components execute concurrently on distributed nodes and must coordinate, under timing constraints, to perform the control task. The application is often such that *all* or *none* of the components must perform correctly within timing constraints for the system to be consistent. If only some of the components perform correctly, then the system will be left in an inconsistent state that could violate functional requirements. The problem of coordinating all or nothing behavior under timing constraints is called *timed atomic commitment.*

As a simple example, consider a plant where containers of chemicals are processed on a conveyer belt. Occasionally, a defective container is detected which has to be carefully removed and discarded, preferably without stopping the belt. To do this, two robot arms, which are also servicing the belt in other capacities, must coordinate to perform the task within ten seconds of detecting the defective container. Before a container is lifted, each arm must have grasped the container and must know that operating conditions will allow it to lift the container within the deadline; if these conditions cannot be met, then the conveyer belt can be safely stopped, the container removed without timing restrictions, and the belt reset. Using the terminology of atomic commitment: if both arms complete the lift by the deadline, then the system has *committed;* if neither arm lifts and the belt stops, then the system has *aborted.* In either case, both arms have performed the same actions, and functional consistency has been maintained. However, if one or both arms have only partially lifted within 10 s (perhaps due to electrical or mechanical failure), a hazardous situation may occur, such as a spill or collision with the next container on the belt; the system is in an *exception* state calling for emergency actions.

In this application, the robot arm processes must perform a type of atomic commitment. However, traditional atomic commitment only requires that all processes *eventually* either commit or abort. There is no deadline by which the decision and action must be completed. We therefore introduce a new notion for distributed real-time computing called *timed atomic commitment* which enforces a deadline on the decision and performance of commitment actions. Similar notions have been called for in [1]–[3] and many discussions allude to the benefits of being able to time constrain traditional atomic commitment [4], [5], but timed atomic commitment remains without a clear definition or implementation.

Unfortunately, it is impossible to place a deadline on traditional atomic commitment if processor failure or message loss can occur. If a processor fails before a decision has been reached and remains down until after the deadline, it may be impossible for any processor to reach a decision. Furthermore, if a processor fails before completing the decided upon action, it may be down until after the deadline and obviously cannot complete the action. Even if processors do not fail, message loss alone causes timed atomic commitment to be impossible. This fact follows easily from the "Two General's Paradox" [4], which states that there can be no fixed length protocol for nontrivial agreement between two or more processes if messages can be lost. Since reasonable distributed operating environments include message loss and processor failure, traditional atomic commitment cannot be extended to observe a deadline. We therefore allow the global outcome of timed atomic commitment, which is a function of the local outcomes of the participant processes, to be either 1) all participant processes performed commitment actions within the deadline (COMMIT), 2) all participant processes performed no commitment actions (ABORT), or 3) the system is in an exceptional state indicating that a fault may have caused timing constraints to be violated (EXCEPTION).

The distinction between ABORT and EXCEPTION is important. In the coordinating robots example, if the outcome is ABORT, then neither arm has lifted; nothing "wrong" has happened, and the belt can merely be stopped for long enough for the container to be successfully lifted. However, if the outcome is EXCEPTION, then the container may be only partially lifted which may cause it to spill or to interfere with the next container on the belt. In general, EXCEPTION indicates that the system may be in an undesirable state, requiring recovery actions. However, regardless of the number of faults, we still require that the processes are functionally consistent, i.e., no process commits if some process aborts. Note that since it is provably impossible for any atomic

Manuscript received December 21, 1988; revised March 12, 1990. This work was supported in part by ARO DAA6-29-84-k-0061, ONR N000014-89-J-1131, and NSF CCR87-16975.

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

IEEE Log Number 9143278.

commitment to solve the problem of ensuring an "all-abort" or "all-commit" outcome within a deadline in the presence of faults, timed atomic commitment is defined to *detect* inconsistencies through the exceptional outcome and provide the opportunity for recovery.

Our goal is to define timed atomic commitment, devise protocols to implement it in a realistic operating environment, and show its usefulness though an example. The rest of this paper is organized as follows. Section II defines timed atomic commitment. In Section III, necessary requirements for the operating environment are discussed and centralized and decentralized protocols for timed atomic commitment are presented. Section IV introduces programming constructs for timed atomic commitment and illustrates their use in the coordinating robots example. Section V draws conclusions on the effectiveness of timed atomic commitment and when it should be used.

II. Definition of Timed Atomic Commitment

Atomic commitment is a problem that has been extensively studied, has a clean definition, and has a range of provably correct protocols for its implementation [5]. An especially clean statement of the problem can be found in [5], and it is this definition that we adapt to include a deadline.

There are N processes, called *participants,* that are to perform timed atomic commitment (TAC). When the TAC commences, a global clock is initiated to measure the deadline for completion, D. Each participant goes through three phases, as shown in Fig. 1: a *vote* phase, at the end of which it produces a vote of YES or NO; a *decision* phase, at the end of which it produces the decision, *COMMIT* or *ABORT;* and a *performance* phase, during which it performs the decided-upon action and records the outcome in its local state. The vote indicates the participant's perception of its ability to commit: a YES vote is a promise to commit if the decision is made to commit; a NO vote means it cannot promise to commit. The local state of a participant is initially EXCEPTION, and cannot be altered after the TAC ends at D.

Informally, in a "perfect" operating environment, the goal of TAC is to guarantee that, at D, either all participants have local states of COMMIT, or all participants have local states of ABORT. Furthermore, a COMMIT outcome is preferable to an ABORT outcome. To reach a COMMIT outcome, every participant must vote YES and decide to COMMIT; additionally, the commit actions must be successfully performed by D. To reach an ABORT outcome, some participant must vote NO, and thus all participants decide to ABORT; aborting (which may include performing restoring actions) must also be successfully performed by D.

Unfortunately, actual operating environments are not perfect and include faults. For example, local clocks may be skewed, messages may be delayed or even lost, processes may not be able to execute when they need to, and execution may take longer than expected. Any of these factors may cause some participant to have a local state of EXCEPTION after the TAC, i.e., be unable to vote, decide, or perform the decided-upon action by D. However, most operating environments offer "guarantees": for example, local clocks are synchronized to within a constant, and delivery time of messages has an upper bound. If the operating environment does not maintain a stated guarantee, we say that a *fault* has occurred. When faults occur we allow the TAC to indicate an EXCEPTION outcome.

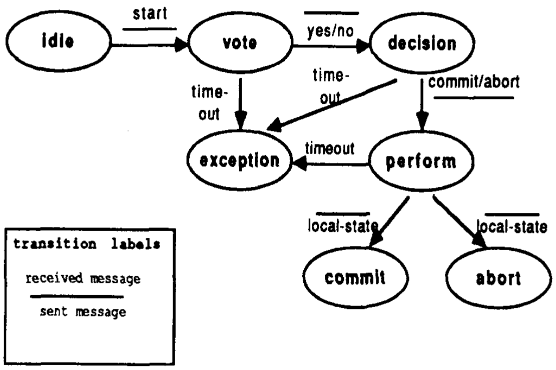


Fig. 1. FSM model of a participant in TAC.

A. TAC Correctness Criteria

We now specify what it means to perform *correct* timed atomic commitment.

TAC1 All participants that reach a decision reach the same one.

TAC2 The decision is to commit only if all participants vote YES.

TAC3 At D, a participant's local state either reflects the participant's completed action or is EXCEPTION.

TAC4 If there are no faults, then

a) all participants reach a decision;

b) if all participants vote YES, then the decision is to commit;

c) all participants complete the decided-upon action by D; and

d) at D, a participant's local state reflects the participant's completed action.

Criteria TAC1 and TAC2 define the functional consistency of TAC, while TAC3 requires the local state to be determined at D. TAC4 defines minimal "success" requirements: TAC4b requires the decision to be COMMIT if there are no faults and all participants vote YES; this invalidates trivial protocols that arbitrarily force the decision to be ABORT. TAC2 and TAC4a together imply that a decision must be made to ABORT rather than remaining EXCEPTION if there are no faults and some participant votes No; this eliminates trivial protocols that allow a process to remain undecided. TAC4c and TAC4d require that, in the absence of faults, the decided-upon action must be successfully completed and recorded in the local state by D.

Note that in addition to the "all-commit" or "all-abort" outcomes of traditional atomic commitment, there are three other combinations of local states in a TAC: 1) all exceptional; 2) some committed, some exceptional; and 3) some aborted, some exceptional. This increased number of outcomes is due to the distinction between the EXCEPTION state and the ABORT state. In an ABORT state, the participant returns to its original state. In the example, an ABORT state implies that neither robot arm lifted and the container is in the position it was before the TAC. In an EXCEPTION state, the participant may have *partially* performed commit or abort actions; e.g., one arm may have only partially lifted by the deadline while the other one has completely lifted. The EXCEPTION state indicates that the system may be inconsistent, and that recovery should be performed.

To see the difference between TAC and traditional atomic commitment, consider the case where there is *no* deadline, i.e., $D = \infty$. In the absence of faults, the correctness criteria require that all participants *eventually* reach a decision and perform the decided-upon action. Therefore, the result of TAC with $D = \infty$ will be either "all-abort" or "all-commit." No participant will ever terminate in the EXCEPTION local state, and this definition agrees with that of traditional atomic commitment in [5]. However, if faults occur, the correctness criteria pose no requirements on whether a decision will ever be reached. This contrasts with the traditional definition which states that if faults do not occur for *sufficiently long,* a decision will *eventually* be reached. The reason for this discrepancy is that "sufficiently long" and "eventually" are temporal statements that must be quantified in the presence of a deadline. However, they are impossible to quantify unless further assumptions about the operating environment are made (such as the number, time of occurrence, and frequency of faults). Therefore, we replace the requirement of eventually reaching a decision with the requirement that the outcome is EXCEPTION if a decision is not reached by the deadline.

B. Calling Process Extension

In practice, it is not enough that the participants establish their own local states by D; some other process must know all of the local states by D so that it can determine what action to take. Furthermore, it is natural to assume that this process initiates the TAC by sending start messages, and "embodies" the global clock by measuring D. In the coordinating robots example, if the outcome is ABORT, the belt should be stopped and the lift retried. If the outcome is EXCEPTION, some form of recovery should be taken. We therefore extend the definition of timed atomic commitment with a *calling process* that initiates the TAC by sending out the start messages, measures D on its clock, and establishes the outcome of the TAC by D. The outcome of the TAC is represented by a *global state vector.* The global state vector entry for each participant is initially EXCEPTION and is changed when the caller determines each participant's local state. To ensure that the caller correctly establishes the outcome of the TAC by D, we replace TAC3 and TAC4d in the timed correctness criteria with:

> TAC3′ At D, a participant's local state either reflects the participant's completed action or is EXCEPTION. Furthermore, the participant's global state vector entry is either its local state or is EXCEPTION.
>
> TAC4d′ at D, a participant's local state reflects the participant's completed action. Furthermore, the participant's global state vector entry is the same as its local state.

The protocols and language constructs we present for TAC are based on this extended definition.

III. Protocols for Timed Atomic Commitment

One's initial reaction in building a timed atomic commit protocol is to merely add a deadline to the end of the performance phase of a "favorite" traditional (untimed) atomic commit protocol. If D expires at any phase of the participant's execution, the participant merely makes a transition to the EXCEPTION local state (see Fig. 1 in the previous section). However, this simple solution violates the correctness criterion TAC4 since an EXCEPTION state may be reached with *no faults* occurring. For example, at some point in any atomic commitment protocol, the participant must reach a decision; this decision can be made just before D, not leaving enough time for the decided-upon action to be completed. Furthermore, the participant may not reach a decision at all before D expires; no faults have occurred, but again the participant enters an EXCEPTION local state. In light of these types of anomalies, we must develop slightly more complex protocols and carefully state what we require of the operating environment.

A. Operating Environment

In devising a correct TAC protocol, the guarantees made by the operating environment must be carefully considered. For example, if the operating environment makes no guarantees about message delivery, then message loss is not a fault. As argued in the Introduction, there can be no correct TAC protocol for this environment. Since the definition of TAC relies on the definition of faults, any protocol must describe what its assumed operating environment is, including what guarantees it makes and what faults can occur. Our assumed operating environment makes guarantees about processors, schedulers, clocks, and communication.

The assumed computation system is a collection of distributed processors that communicate with each other via messages over a network. A *processor fault* occurs when a processor goes down. While the processor is down, no process that is assigned to the processor performs any computation. Each processor has its own local clock. A *clock fault* occurs if two clocks drift too far apart, i.e., there is an assumed upper bound on clock drift, called ϵ. We assume that no malicious faults occur.

Communication is asynchronous. The time from executing **send** to arrival of the message at the recipient process's message queue is guaranteed not to exceed Δ. There are two forms of *communication faults:* lost messages, where a message is never delivered from the sender to the receiver, and late messages, where messages take longer than the guaranteed upper bound on delivery. We assume that messages never arrive out of order.

Finally, each processor has a collection of time-shared processes that are subject to preemption. We assume that scheduling is *fair:* each process is guaranteed to execute for at least τ_r time units within τ_P time units of becoming ready to execute. Processors use a *resource manager* to allocate and schedule resources such as the CPU and devices. The resource manager is assumed to be capable of guaranteeing resources for a duration of time within a given time interval [6]–[8]. A *scheduling fault* occurs either when the fairness assumption is violated, or the resource manager promises resources but fails to deliver them within the promised time. We assume that the execution time bounds are accurate, i.e., a process never requests too little time from a resource manager, and that the resource manager responds to guarantee requests within a fixed amount of time.

B. Notation

To facilitate the description of the protocols, we introduce the following notation. First, we express time dependent behavior using the *temporal scope* language construct. We outline only the aspects of temporal scopes used in this paper; further details can be found in [9]. A temporal scope consists of (optionally) a start time and a deadline, statements that are to be performed in the interval defined by the start time and deadline, and an exception handler. If the start time is not specified, it is assumed to be immediate; if the deadline is missing, it is assumed to be infinite. The structure of a temporal scope is as follows:

```
before ⟨start-time⟩ by ⟨deadline⟩ do
  ⟨statements_1⟩
except
  when E_START do ⟨statements_2⟩ end when
  when E_DEADLINE do ⟨statements_3⟩ end when
end before.
```

If ⟨statements_1⟩ are not started by the specified ⟨start-time⟩, then ⟨statements_2⟩ are executed. If the ⟨statements_1⟩ are not completed by ⟨deadline⟩, then execution of ⟨statements_1⟩ is terminated ⟨statements_3⟩ are executed.

Second, we describe how processes reserve resources. A process must be able to reserve resources to be able to complete the decided-upon action by the deadline. For simplicity, we assume that the only required resource is the CPU, although in general it could include other resources such as memory or devices. A system call, *Reserve*(e, [*low*, *high*]), returns true if e execution time units within the interval [*low*, *high*] are guaranteed by the resource manager to the invoking process; otherwise, false is returned.

Third, we describe communication. The **send** primitive, *send*(*process*, *message*), takes τ_s units of local processing time (included in the assumed bound Δ). We also assume a noninterruptible broadcast version of *send*(*process*, *message*) called *send-all*(*process-list*, *message*). By *noninterruptible* we mean that it is not possible to interrupt a *send-all* for a temporal scope deadline violation. The *send-all* primitive has a bound of Δ^*, of which τ_b is local processing time. The **receive** primitive, *receive*(*process-list*, *message*), blocks until a message arrives from any of the specified processes.

C. Centralized TAC Protocol

This section adapts a centralized two-phase commit protocol[1] to TAC by incorporating intermediate deadlines; the result is the centralized timed two-phase commit protocol (CT2PC). In CT2PC, an extra "coordinator" process is added to collect votes from the participants, and make and distribute the decision. For simplicity, we assume that the calling process is the coordinator, i.e., the caller sends out the start messages, acts as coordinator during the TAC, and establishes the global state vector at the end of the TAC.

In the TAC, let S be the absolute start time and D be the absolute deadline. For a participant P_i, let t_i be the maximum execution time needed to receive a pending decision message, carry out the commit or abort action, and send a completion message, measured on its clock. The largest of all the t_i's is called $\tau_{\max}$. For the coordinator, let τ_d be the maximum execution time needed to receive N waiting vote messages, process them, and make a decision; and τ_f be the maximum execution time needed to receive N pending completion messages and compute the result of a TAC. Recall that ϵ is the maximum clock drift, Δ is the bound on execution of *send*, τ_s is the local processing time for *send*, Δ^* is the bound on execution of *send-all*, and τ_b is the local processing time for *send-all*.

Intermediate Deadlines: Each phase of the CT2PC consists of a message exchange between the coordinator and the participants as shown in Fig. 2. The following intermediate deadlines are added to the phases:

- $D_p = D - \Delta - \tau_f - \epsilon$: deadline for sending a completion message by a participant. In the absence of faults, each participant must complete the decided-upon action and send the completion message (at most Δ time units) so that the coordinator has time to process it (at most τ_f time units) before D on the coordinator's clock (skewed by at most ϵ).
- DEC $= D_p - \tau_{\max} - \Delta^* - \epsilon$: deadline for sending a decision by the coordinator. For a participant with $\tau_{\max}$ execution time to guarantee completion of the decided-upon action by D_p in the absence of faults, it must start executing the action by $D_p - \tau_{\max}$ on its clock. The coordinator must then interpret this time on its own clock using the worst case assumption on clock skew, and allowing maximum message delay for the broadcast decision to arrive at the participant.
- $V =$ DEC $- \Delta - \tau_d - \epsilon$: deadline for a participant to vote. The participant must vote in time for the vote message to arrive at the coordinator and be processed before DEC expires on the coordinator's clock.
- [LST$_i$, D_p]: the interval of time during which P_i requests a guarantee of t_i time units of resources needed to perform the decided-upon action. There are several choices for LST$_i$, ranging from LST$_i$ = DEC $+ \Delta^* + \epsilon$ to LST$_i$ = $D_p - t_i$. Choosing an earlier LST$_i$ allows P_i to vote YES more frequently since the guarantee is more likely

[1] For an overview of centralized two-phase commit protocols see [5] and [4].

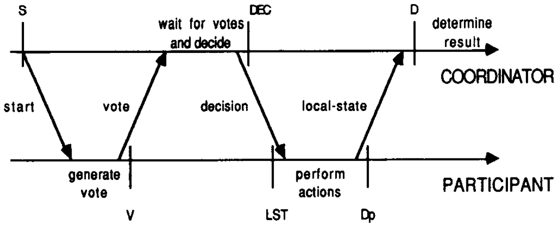


Fig. 2. Messages in a CT2PC protocol.

to be granted. Choosing the later LST_i can better tolerate a tardy decision message.

To understand why the assumption of fair scheduling has been imposed, consider the following scenario. Suppose that the coordinator sends START messages to the participants, and that the messages are delivered within Δ^* time units. If no assumption is made about scheduling, some participant could be ready to receive the message, but not be scheduled to execute until after the deadline, D. This will cause the coordinator to conclude that the outcome is EXCEPTION in the absence of any faults, violating TAC4c. However, if participants are guaranteed to execute for long enough to send a COMPLETION message to the coordinator before D, indicating that they have automatically aborted, this problem is avoided. Thus, τ_r must at least be long enough for the participant to null-abort, that is, allow enough time for the participant to receive a waiting START message, query the resource manager, and send a COMPLETION message to the coordinator. Furthermore, τ_r must be given after the start message is delivered and before D_p. This can be guaranteed if the participant is given τ_r units within τ_P time units of being ready, in which $\tau_P < D_p - S - \Delta^*$.

CT2PC Protocol: Fig. 3 outlines the coordinator process. Before starting a TAC, the coordinator ensures that D is sufficiently long to allow each participant to receive a START message and return a COMPLETION message in time for the coordinator to determine the result. The coordinator also reserves τ_d and τ_f units of execution so that it can send a decision message by DEC and determine the result by D. If the reservations are denied, the TAC is not started. Otherwise, the coordinator commences the TAC by sending START messages. The coordinator then waits to receive vote messages from the participants. When it receives all votes, or any NO vote, it decides and sends the decision to the participants. However, if DEC expires before it decides, it decides to abort and sends the ABORT decision to the participants. After sending the decision, it receives COMPLETION messages and updates the corresponding global state vector entries. If D expires before all COMPLETION messages have been received, the result is EXCEPTION.

Fig. 4 outlines a participant P_i. When a START message is received, the participant attempts to reserve t_i units of execution within $[\mathrm{LST}_i, D_p]$. If the reservation succeeds, it determines its vote and tries to send the vote by V. When the participant receives a decision from the coordinator, it performs the decided-upon action and sends a COMPLETION message by D_p.

```
process Caller($S$,$D$) /* $S$= start time, $D$= deadline */
begin
  $D_p := D - \Delta - \tau_f - \epsilon$
  $DEC := D_p - \Delta^* - \tau_{max} - \epsilon$
  $V := DEC - \Delta - \epsilon - \tau_d$
  if $(D_p - S \geq \Delta^* + \tau_r)$ and $(D_p - S - \Delta^* > \tau_P)$
    and Reserve $(\tau_d + \tau_b, [DEC - \tau_d, DEC + \tau_b])$
    and Reserve $(\tau_f, [D - \tau_f, D])$ then
    Initialize global state vector entries to EXCEPTION.
    decision := ABORT
    by $DEC$ do
      send-all ($[P_1, \ldots, P_N]$, START, $D_p$, $DEC$, $V$)
      while (not received all $N$ votes) and (no NO votes received) do
        receive ($[P_1, \ldots, P_N]$, vote)
      end while
      if all YES votes then decision := COMMIT end if
      send-all ($[P_1, \ldots, P_N]$, decision)
    except
      when E_DEADLINE do
        send-all ($[P_1, \ldots, P_N]$, decision)
      end when
    end by /* $DEC$ */
    by $D$ do
      while not received all COMPLETION messages do
        receive ($[P_1, \ldots, P_N]$, COMPLETION)
        Update global state vector entry.
      end while
    end by
  end if
end process
```

Fig. 3. Coordinator (caller) process for CT2PC.

```
process $P_i$ /* $i^{th}$ Participant Process */
begin
  receive (Caller, START/ABORT, $D_p$, $DEC$, $V$)
  by $D_p$ do
    if received ABORT then
      send (Caller, COMPLETION) /* null abort */
    else /* received START message */
      $LST_i := DEC + \Delta^* + \epsilon$
      if Reserve ($t_i$, $[LST_i, D_p]$) then
        by $V$ do
          compute vote (YES/NO)
          send (Caller, vote)
        end by /* $V$ */
        receive (Caller, decision)
        case decision of
          COMMIT: user-specified commit statements
          ABORT: user-specified abort statements
        end case
      end if
      send (Caller, COMPLETION)
    end if
  except
    when E_DEADLINE do exception statements end when
  end by /* $D_p$ */
end process
```

Fig. 4. Participant process for CT2PC.

Note that steps taken for vote determination are application dependent. For the coordinating robots example described in the Introduction, a robot must grasp the container before voting YES to ensure that it can lift it correctly. Thus, if the robot votes YES, but the decision is ABORT, the robot must release the container in its ABORT action.

If the participant cannot receive a reservation, or receives an ABORT message without a prior START message, the participant null-aborts and sends a COMPLETION message. A *null-abort* indicates that the participant has taken no steps in determining its vote that need to be undone during an ABORT.

D. Correctness of CT2PC

To show that CT2PC is correct, we now prove a series of lemmas corresponding to the correctness criteria of Section II-A. We assume that the TAC was initiated, i.e., the coordinator has received its requested guarantees, the deadline was far enough away to initiate the protocol, and start messages were sent to the participants.

Lemma 1 (TAC2): The decision is COMMIT only if all participants vote YES.

Proof: Follows immediately from the fact that a participant decides to commit only if the coordinator sends a COMMIT message, which is done only if all the votes are YES. □

Lemma 2 (TAC1): All participants that reach a decision reach the same one.

Proof: First, recall that *send-all* is noninterruptible, so the coordinator sends out the same decision message to every participant. The only case in which a participant makes a decision without explicitly receiving it from the coordinator is if the participant aborts. In this case, the coordinator cannot decide to commit since the aborting participant will not send a YES vote. It follows from Lemma 1 that the decision in this case cannot be COMMIT. □

In the following two lemmas, we assume that there are no faults. They are used to show that CT2PC satisfies the minimum goodness requirements, TAC4.

Lemma 3: If there are no faults, any message that process P_j sends to process P_i at time t on P_j's clock is guaranteed to arrive by $t + \Delta + \epsilon$ on P_i's clock. Furthermore, if process P_j broadcasts a message at time t, then it will arrive by $t + \Delta^* + \epsilon$ on any recipient P_i's clock.

Proof: Follows from the definitions of Δ, Δ^*, and ϵ. □

Lemma 4: If there are no faults and the participant P_i is not guaranteed its execution times, then it meets TAC4.

Proof: The fair scheduling assumption and definitions of τ_r and τ_P ensure that P_i will send a COMPLETION message by D_p (TAC4a, c). Using Lemma 3 and the fact that $D - D_p$ includes τ_f time to receive and process all COMPLETION messages, TAC4d′ holds. TAC4b is trivially satisfied because P_i does not vote YES. □

We now complete the proof of TAC4 by restricting our attention to participants who have received a guarantee of their execution times.

Lemma 5: If there are no faults, then the decision message arrives at each participant P_i by LST_i, measured on P_i's clock.

Proof: It is enough to show that in the absence of faults the decision message is broadcast by DEC, because Lemma 3 ensures that it arrives at P_i by $\mathrm{DEC} + \Delta^* + \epsilon = \mathrm{LST}_i$ on P_i's clock. Suppose that the decision message has not been broadcast before DEC. Since the coordinator has reserved $\tau_d + \tau_b$ execution time during $[\mathrm{DEC} - \tau_d, \mathrm{DEC} + \tau_b]$, the coordinator is guaranteed to start executing the exception handler at DEC and have enough local processing time for a *send-all* (τ_b); hence, the decision message is sent at DEC according to the coordinator clock in the worst case. □

Lemma 6 (TAC4a): If there are no faults, then all participants reach a decision.

Proof: By Lemma 5, the decision message arrives at P_i by LST_i. Since P_i has received a guarantee of t_i during $[\mathrm{LST}_i, D_p]$, and t_i includes execution time to receive the decision, P_i is guaranteed to reach a decision.

Lemma 7 (TAC4b): If there are no faults and all participants vote YES, then the decision is to commit.

Proof: Since there are no faults and each participant votes YES, each participant must have sent its vote message by V measured on its clock. Due to Lemma 3, every vote message must arrive at the coordinator by $V + \Delta + \epsilon = \mathrm{DEC} - \tau_d$, measured on the coordinator's clock. Since the coordinator has reserved τ_d units of execution during $[\mathrm{DEC} - \tau_d, \mathrm{DEC}]$, it is guaranteed to be able to receive all vote messages and decide to commit by DEC. By Lemma 6, all participants must also decide to commit. □

Lemma 8 (TAC4c): If there are no faults, then all participants complete their decided-upon action by D.

Proof: By Lemma 5, the decision message arrives at P_i by LST_i. Since P_i has reserved t_i execution time during $[\mathrm{LST}_i, D_p]$, then by the definition of t_i P_i completes the decided-upon action and sends a COMPLETION message by D_p. Note that we have proved something stronger than required, namely that the COMPLETION message is also sent by D_p. □

Lemma 9 (TAC4d′): If there are no faults, then at D, each participant's local state and global state vector entry reflect the participant's completed action.

Proof: As noted in the proofs of Lemmas 4 and 8, each participant sends a COMPLETION message by D_p. By Lemma 3, the COMPLETION messages must arrive at the caller by $D_p + \Delta + \epsilon = D - \tau_f$. Since the coordinator has reserved τ_f execution time in $[D - \tau_f, D]$, it must receive all COMPLETION messages and update the global state vector by D. □

Lemma 10 (TAC3′): At D, each participant either has its local state and global state vector entry reflect its completed action or its global state vector entry is EXCEPTION.

Proof: The global state vector is initially EXCEPTION for each participant, and is changed only when a COMPLETION message is received from a participant. A COMPLETION message is only sent if the participant has completed the decided-upon action and (implicitly) changed its local state to reflect completion of the decided-upon action. □

Using the above lemmas, we conclude that CT2PC is correct:

Theorem 1: CT2PC shown in Figs. 3 and 4 is correct with respect to the TAC Correctness Criteria.

E. A Decentralized TAC Protocol

This section adapts a decentralized two-phase commit protocol that requires each participant to receive a vote from every other participant, make its own decision, and perform the appropriate action in time to let the caller know its local state by D.

For a participant P_i, let τ_d be the maximum execution time needed to receive N vote messages, process them, and make a decision; let t_i be the maximum execution time needed carry out its commit or abort action and send its local state message; and let $\tau_{\max}$ be the largest of all the t_i's. As in CT2PC, let τ_f be the maximum execution time needed for the caller to receive N completion messages and compute the result of the TAC. Recall that ϵ is the maximum clock drift, Δ is the bound

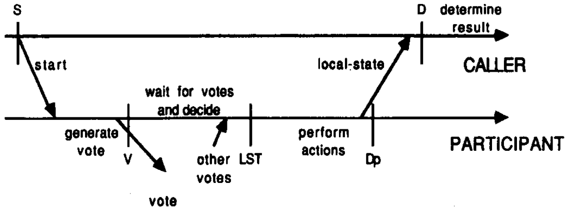


Fig. 5. Messages in a DT2PC protocol.

on execution of *send*, τ_s is the local processing time for *send*, Δ^* is the bound on execution of *send-all*, and τ_b is the local processing time for *send-all*.

Intermediate Deadlines: Participants execute as shown in Fig. 5. The intermediate deadlines are

- $D_p = D - \Delta - \tau_f - \epsilon$: deadline for sending a completion message by a participant.
- $V = D_p - \Delta^* - \tau_{\max} - \tau_d - \epsilon$: deadline for a participant to vote. Let P be a participant with $\tau_{\max}$ expected execution time. To guarantee that P can meet D_p, each participant must broadcast its vote by V to ensure that its vote arrives at P by $D_p - \tau_{\max} - \tau_d$ on P's clock.
- $[\mathrm{LST}_i, D_p]$: the interval of time during which P_i requests a guarantee of t_i time units of resources needed to perform the decided-upon action. LST_i can range from $\mathrm{LST}_i = D_p - \tau_{\max}$ to $\mathrm{LST}_i = D_p - t_i$. The former is the latest time that P_i receives all votes if no fault occurs, whereas the latter is the latest time that P_i must start executing its decided-upon action to complete by a pessimistic interpretation of D_p on its clock. The tradeoffs are similar to those discussed in the CT2PC protocol.

We now reiterate what is required of the fair scheduling assumption: τ_r must be long enough to null-abort, which in this case involves receiving a waiting START message, querying the resource manager, broadcasting a NO vote, and sending an ABORT message to the caller. Furthermore, all votes must arrive at each participant before LST_i, forcing $\tau_P < V - S - \Delta^*$.

DT2PC Protocol: Fig. 6 outlines the caller in DT2PC. It first checks that D is sufficiently long to allow each participant to receive a START message, send NO votes to other participants, and send ABORT to the caller. It then attempts to guarantee that it can receive τ_f execution time in order to receive the local-state messages (COMMIT/ABORT). If it receives a guarantee, start messages are sent using a **send-all** primitive. The caller then waits to receive local-state messages.

Fig. 7 outlines a participant P_i in DT2PC. Upon receiving a start message from the caller, P_i attempts to receive guarantees from its resource manager that it can vote by V, process other votes by LST_i, and perform the commit or abort actions in the interval $[\mathrm{LST}_i, D_p]$. If P_i does not receive these guarantees, it null-aborts by voting NO and sending a local state message (ABORT) to the caller. Otherwise, P_i attempts to determine its vote. If V expires before P_i sends its vote, the temporal scope handler generates a NO vote. Whenever P_i votes NO, it aborts and sends an ABORT message to the caller. Whenever P_i votes YES, it waits to receive *all* votes from the other participants. It then decides, performs the appropriate action, and communicates its local state to the calling process upon completion. If D_p expires, then P_i terminates by executing exception statements.

```
process $Caller(S, D)$
begin
    $D_p := D - \Delta - \tau_f - \epsilon$
    $V := D_p - \Delta^* - \tau_{max} - \tau_d - \epsilon$
    if ($V - S - \Delta^* > \tau_P$ and Reserve ($\tau_f$, $[D - \tau_f, D]$)) then
        Initialize global state vector entries to EXCEPTION.
        by $D$ do
            send-all ($[P_1, \ldots, P_N]$, START, $\tau_{max}$, $D_p$, $V$)
            while (not received all $N$ local-state messages) do
                receive ($[P_1, \ldots, P_N]$, ABORT/COMMIT)
                Update global state vector entry.
            end while
        end by /* D */
    end if
end process
```

Fig. 6. Caller process for DT2PC.

```
process $P_i$
begin
    receive (Caller, START, $\tau_{max}$, $D_p$, $V$)
    $LST_i := D_p - \tau_{max}$
    if not (Reserve($\tau_b, [V, V + \tau_b]$) and
        Reserve($\tau_d, [LST_i - \tau_d, LST_i]$) and
        Reserve($t_i, [LST_i, D_p]$)) then
            send-all ($[P_1, \ldots, P_N]$, NO)
            send (Caller, ABORT)
    else /* guarantee received */
        vote:= NO
        by $V$ do
            compute vote (YES/NO)
            send-all ($[P_1, \ldots, P_N]$, vote)
        except /* $V$ */
            when E_DEADLINE do send-all ($[P_1, \ldots, P_N]$,vote) end when
        end by /* $V$ */
        by $D_p$ do
            if vote= NO then temp:= ABORT else temp:= COMMIT
            while (not received all other votes) and (temp = COMMIT) do
                receive ($[P_1, \ldots, P_N]$, their_vote)
                if their_vote = NO then temp := ABORT end if
            end while
            decision:= temp
            case decision of
                COMMIT: user-specified commit statements
                ABORT: user-specified abort statements
            end case
            send (Caller, decision) /* local state message */
        except
            when E_DEADLINE do exception statements end when
        end by /* $D_p$ */
    end if
end process
```

Fig. 7. Participant process P_i in DT2PC.

F. Correctness of DT2PC

We now show that DT2PC is correct by proving a series of lemmas corresponding to the correctness criteria of Section II-A. We use Lemma 3 from Section III-D and again assume that the TAC is initiated, i.e., that the caller received its requested guarantees, the deadline was far enough away to initiate the protocol, and that start messages were sent to the participants.

Lemma 11 (TAC2): The decision is COMMIT only if all participants vote YES.

Proof: Obvious, since the only way a participant can decide to commit is to receive all votes with none of them being NO. □

Lemma 12 (TAC1): All participants that reach a decision reach the same one.

Proof: If some participant decides COMMIT, then any other participant that reaches a decision must decide COMMIT since all votes must be YES. If some participant decides ABORT, then some vote (possibly its own) must be NO; hence, by Lemma 11 no other participant can decide COMMIT. □

Lemma 13: If there are no faults and participant P_i is not guaranteed its execution times, then it meets TAC4.

Proof: Note that the fair scheduling assumption and definitions of τ_r and τ_P ensure that P_i will broadcast NO votes to all other participants and send an ABORT message to the caller by V (TAC4a, c). Using Lemma 3 and the facts that $V < D_p$ and that $D - D_p$ includes τ_f time for the caller to receive all ABORT/COMMIT messages, TAC4d′ holds. TAC4b is trivially satisfied because P_i does not vote YES. □

We now complete the proof of TAC4 by restricting our attention to participants who have received a guarantee of their execution times.

Lemma 14: If there are no faults, then each participant P_i sends its vote by V as measured on its own clock.

Proof: Follows since P_i is guaranteed τ_b time needed to broadcast its vote in the exception handler at V. □

Lemma 15: If there are no faults, then each participant P_i reaches a decision by LST_i, measured on its own clock.

Proof: Lemmas 14, 3, and the proof of Lemma 13 ensure that all vote messages arrive at P_i by $V + \Delta^* + \epsilon$ on its clock, which is $\mathrm{LST}_i - \tau_d$. Since P_i reserved τ_d time in $[\mathrm{LST}_i - \tau_d, \mathrm{LST}_i]$, it receives the votes and decides by LST_i. □

Lemma 16 (TAC4a): If there are no faults, then all participants reach a decision.

Proof: Follows directly from Lemmas 13 and 15. □

Lemma 17 (TAC4b): If there are no faults and all participants vote YES, then the decision is to commit.

Proof: By Lemma 15, each participant receives all votes and has time to reach a decision by LST_i. Since the votes are all YES, the decision must be to COMMIT. □

Lemma 18 (TAC4c): If there are no faults, then all participants complete their decided-upon action by D.

Proof: This follows from the fact that the decision is made by LST_i (Lemma 15), and t_i units of execution are guaranteed within $[\mathrm{LST}_i, D_p]$ which is sufficient both to complete the decided-upon action and to send the completion message by D_p. Note that for *any* participant, the completion message is sent by D_p. □

Lemma 19 (TAC4d′): If there are no faults, then at D, each participant's local state and global state vector entry reflect the participant's completed action.

Proof: The local state message is sent by D_p (proof of Lemma 18) and arrives at the caller by $D_p + \Delta + \epsilon$ (Lemma 3), which is $D - \tau_f$ on the caller's clock. τ_f allows the caller time to receive the message and update the global state vector. □

Lemma 20 (TAC3′): At D, each participant either has its local state and global state vector reflect its completed action or its global state vector entry is EXCEPTION.

Proof: The global state vector is initially EXCEPTION for each participant, and is changed only when a local state message is received from a participant. This message is only sent if the participant has completed the decided-upon action and (implicitly) changed its local state to reflect completion of the decided-upon action. □

Using the above lemmas, we conclude that DT2PC is correct.

Theorem 2: DT2PC shown in Figs. 6 and 7 is correct with respect to the TAC Correctness Criteria.

IV. Coordinating Robots Example

We now illustrate the usefulness of TAC using the coordinating robots example described in the introduction. To facilitate the description, we first introduce some language constructs.

A. Language Constructs

The language constructs include a *TAC block* for the calling process, and *timed actions* for the participants.

TAC Block: To invoke a TAC, the caller starts a set of concurrent participant timed actions, and waits for the participants' local states. The structure of the TAC block is

```
tac_begin $[V_1, \cdots, V_n]$ /* Global state vector. */
  $V_1$ := action $P_1$ (⟨args⟩)
  ⋮
  $V_n$ := action $P_n$ (⟨args⟩)
end tac;
```

The global state vector $[V_1, \cdots, V_n]$ is initialized to EXCEPTION for each entry; V_i is updated when P_i completes and returns its local state. When each entry in the global state vector has been updated, the TAC completes. To establish a deadline for TAC, the TAC block is enclosed within a temporal scope (see Section III-B and [9]). If the deadline is reached and TAC block has not completed (some V_i is still EXCEPTION), then the temporal scope exception handler starts recovery.

Timed Actions: TAC participants are *timed actions* which execute as remote procedures called from a TAC block. The structure of a timed action is

```
timed action ⟨action-name⟩ ( ⟨parameters⟩ )
    for ⟨time⟩ { resource ⟨resource-id⟩ }
begin
    ⟨statements$_1$⟩ /* decide vote: YES or NO */
    vote (YES or NO)
    await
        when COMMIT do ⟨statements$_2$⟩ end when
        when ABORT do ⟨statements$_3$⟩ end when
    except
        when E_DEADLINE do ⟨statements$_4$⟩ end
        when
end action.
```

The parameters allow data to be exchanged between the TAC block and the timed action; the explicit declaration of resources allows the underlying protocol to request reservations for the COMMIT/ABORT actions. When the timed action is invoked, it computes its vote; the decision is made based on the votes of all timed actions in the TAC block. If the decision is COMMIT, ⟨statements$_2$⟩ are executed; if the decision is ABORT, ⟨statements$_3$⟩ are executed. Note that the deadline (E_DEADLINE) is not explicitly specified, but is determined by the underlying protocol using the caller's deadline.

Another difference between timed atomic commitment and traditional atomic commitment should be discussed here. In traditional atomic commitment programmer-provided abort statements (such as ⟨statements$_3$⟩), are not used because only *automatically recoverable* actions are performed before the decision is known. However, in timed atomic commitment, state altering actions may be performed in the voting phase that can only be restored by the programmer. For instance, in the robot example of Section V, a robot bases its vote on whether or not it has grasped the container; if the decision is to abort, the programmer must provide explicit *compensating actions* [10], [11] in the abort clause to release the container. However, *unrecoverable actions* should be performed only during the commit phase so that they can be assured of completing (barring faults).

B. Coordinating Robots Example

The coordinating robots example described in the Introduction requires that a defective chemical container be picked up by two robot arms and discarded within 10 s of detection. The example consists of a caller process, *Belt_Controller* (see Fig. 8), and two participants, *Robot_1* and *Robot_2,* which control the arms needed to pick up a container from the conveyer belt. (See Fig. 9.)

Belt_Controller waits 5 s after a sensor detects a defective container before initiating a TAC with a 10 s deadline. It then waits until it knows both arms have completed the decided-upon action, or until the 10 s deadline expires. If the result is COMMIT, the belt continues without interruption; if it is ABORT, the belt is stopped and reset. Otherwise, *Belt_Controller* does not know whether or not *Robot_1* and *Robot_2* have successfully completed by the deadline; it stops the entire system and alerts the operator so that the unlifted container can be removed.

Upon invocation, *Robot_1* determines its vote by trying to grasp the container; this may fail since the arm is shared among several processes and only one process may control the arm at a time. If it is successful, the vote is YES; otherwise, the vote is NO. Note that the underlying protocol may also force the vote to be NO if intermediate deadlines cannot be met or the required reservations are not guaranteed; in this example, the arm is needed for 4 s during the COMMIT/ABORT phase. After voting, *Robot_1* awaits the decision; ABORT results in the container being released; otherwise, it is lifted. If the participant's deadline expires before the completion of the decided-upon action, then the arm is stopped and *Belt_Controller* handles the exception.

```
process Belt_Controller
    ⋮
    Wait for sensor to detect a defective-container.
    after 5 seconds within 10 seconds do
        tac_begin [$V_1, V_2$]
            $V_1$ := action Robot_1 ()
            $V_2$ := action Robot_2 ()
        end tac
    except
        when E_DEADLINE do
            stop entire system
            alert operator to clear container from arms
        end when
    end after
    if $V_1$ = ABORT and $V_2$ = ABORT
        then stop belt and reset
    ⋮
```

Fig. 8. Caller process *Belt_Controller.*

```
timed action Robot_1 ()
    for 4 sec resource arm1
begin
    lower arm and grasp container
    if grasped correctly then vote (YES) else vote (NO)
    await
        when COMMIT do raise arm end when
        when ABORT do
            if container is grasped then release container
        end when
    except
        when E_DEADLINE do stop arm end when
end action
```

Fig. 9. Participant timed action *Robot_1.*

V. Conclusion

In a large class of hard-real-time control applications, components of a control task must perform a type of atomic commitment under timing constraints. However, if the assumed operating environment includes the possibility of processor and communication faults, it is impossible to devise a protocol which guarantees that all participants either commit or abort by a deadline. We therefore modify the traditional definition of atomic commitment to one for timed atomic commitment by introducing an EXCEPTION state, which indicates that a participant may not have completed the decided-upon action by the deadline. As in traditional atomic commitment, we insist that the decisions made by participants are consistent, i.e., no participant decides to commit if another decides to abort; however, EXCEPTION is defined to be consistent with COMMIT or ABORT.

To formalize this notion, we presented minimal requirements for a correct implementation of timed atomic commitment. These correctness criteria capture the intuitive notion that an exceptional outcome should only occur in the presence of faults, and an aborted outcome should only occur in the presence of faults or if some process votes NO. That is, a correct TAC should succeed in committing *whenever possible.* In order to achieve a correct implementation, we also noted that it is necessary to have an operating environment that provides bounds on message delays and clock synchronization, and guarantees resources.

Centralized and decentralized timed two-phase commit protocols were modified to meet the correctness criteria by introducing intermediate deadlines on the voting and performance phases of participants, and on the decision phase for the caller. The deadlines were derived from D using several assumptions, e.g., maximum message delay, clock drift, and execution time bounds. If any of these assumptions are violated, correctness is still assured but an exception outcome may occur; to reduce exceptions, these bounds should be pessimistic.

There are tradeoffs between using the centralized or decentralized implementation. In CT2PC, there are $4N$ messages; of these, $2N$ messages (the decision and completion messages) are "critical." By critical we mean that if the message is lost, the result will be EXCEPTION. Note that if a START or VOTE message is lost in CT2PC, the coordinator will timeout and decide ABORT. In DT2PC there are $N^2 + N$ messages, all of which are critical. In either implementation, loss of any process, participant, or coordinator, may result in an EXCEPTION outcome.

If the caller wishes to know that there is a *possibility* of committing, using worst case assumptions, there is a minimum overall elapsed deadline, $D - S$. For the centralized protocol, $D - S$ must be greater than or equal to the sums of the time to send the start message (Δ^*), compute the vote ($(\tau_r - \tau_s) + \epsilon$), send the vote ($\Delta$), decide ($\tau_d + \epsilon$), send the decision ($\Delta^*$), perform the decided-upon action ($\tau_{\max} + \epsilon$), send the completion message (Δ), and update the global state vector (τ_f):

$$D - S \geq 2\Delta + 2\Delta^* + (\tau_r + \tau_s) + \tau_d + \tau_{\max} + \tau_f + 3\epsilon. \tag{1}$$

For the decentralized protocol, $D - S$ must be greater than or equal to the sums of the time to send the start message (Δ^*), compute the vote ($(\tau_r - \tau_b) + \epsilon$), send the vote ($\Delta^*$), decide and perform the decided-upon action ($\tau_d + \tau_{\max} + \epsilon$), send the completion message (Δ), and update the global state vector (τ_f):

$$D - S \geq \Delta + 2\Delta^* + (\tau_r - \tau_b) + \tau_d + \tau_{\max} + \tau_f + 2\epsilon. \tag{2}$$

A shorter deadline would not be incorrect nor necessarily cause exceptional outcomes. However, since the intermediate deadlines are derived from D, a shorter D may cause an increased ABORT rate. For example, there may not be enough time for guarantees to be made, or (in CT2PC) the coordinator may timeout while waiting for votes. Thus, these protocols are most useful for real-time applications in which the deadline is long compared to message delays and clock skew.

Note that a virtue of the TAC protocols is that the timed behavior of the caller is *predictable;* at the deadline, the caller either knows that all participants have performed the decided-upon action, or decides that some participant is exceptional and performs explicit recovery. It is our belief [1], [3], [8] that consistency and predictable performance are often more important than speed in real-time computing, thus the overhead of using the TAC protocols is justified.

To support the use of timed atomic commitment, we also introduced a temporal scope, TAC block, and timed action constructs. A timed action defines a participant with explicit voting, decision, and performance phases. The caller uses a TAC block to initiate the atomic commitment, and expresses the deadline by enclosing it in a temporal scope. These constructs were demonstrated in the coordinating robots example. Although it is possible to implement the example without these constructs, an equivalent implementation would require explicit synchronization, fault detection, and enforcement of timing constraints. In addition, these constructs support extensible and modifiable programs: Programs are extensible since adding another robot arm merely entails adding another participant in the TAC. Programs are modifiable since changing the deadline in the caller does not necessitate changing the participant code. Above all, TAC language constructs simplify program development and modification by hiding implementation details.

The language constructs and underlying protocols are currently being implemented using a real-time kernel [8] developed at the University of Pennsylvania for distributed real-time control applications.

Acknowledgment

We thank the referees for their constructive input into earlier versions of this paper.

References

[1] J. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *IEEE Comput. Mag.,* vol. 21, pp. 10–19, Oct. 1988.

[2] K. Schwan, T. Bihari, and B. Blake, "Adaptable, reliable software for distributed and parallel, real-time systems," in *Proc. Sixth Symp. Reliability in Distributed Software,* Mar. 1987, pp. 32–44.

[3] I. Lee, S. Davidson, and V. Wolfe, "Motivating time as a first class entity," Tech. Rep. MS-CIS-87-54, Dep. Comput. and Inform. Sci. Univ. of Pennsylvania, July 1987. Presented at IEEE Fourth Workshop on Real-Time Operating Systems.

[4] J. Gray, "Notes on database operating systems," in *Operating Systems.* Berlin, Germany: Springer-Verlag, 1979, pp. 394–481.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1986.

[6] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. Software Eng.,* vol. SE-13, pp. 564–577, May 1987.

[7] ——, "Preemptive scheduling under time and resource constraints," *IEEE Trans. Comput.,* vol. C-36, pp. 949–960, Aug. 1987.

[8] I. Lee, R. B. King, and R. P. Paul, "A predictable real-time kernel for distributed multisensor systems," *IEEE Comput. Mag.,* vol. 22, pp. 78–83, June 1989.

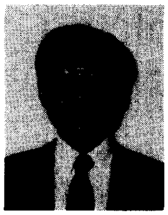
[9] I. Lee and V. Gehlot, "Language constructs for distributed real-time programming," in *Proc. IEEE Real-Time Syst. Symp.,* Dec. 1985.

[10] H. Tokuda, "Compensatable atomic objects in object-oriented operating systems," in *Proc. Pacific Comput. Commun. Symp.,* Oct. 1985, pp. 45–53.

[11] A. Gheith and K. Schwan, "CHAOSart: Support for real-time atomic transactions," in *Proc. 19th Int. Symp. Fault Tolerant Comput.,* IEEE, 1989, pp. 462–469.

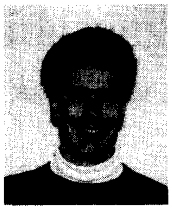
Susan B. Davidson (M'83) received the B.A. degree in mathematics from Cornell University, Ithaca, NY, in 1978, and the M.A. and Ph.D. degrees in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1980 and 1982.

She is currently an Associate 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.



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

He is currently an Associate Professor in the Department of Computer and Information Science at the University of Pennsylvania, Philadelphia, where he has been since 1983. His research interests include the specification and analysis of time dependent systems, real-time programming languages and semantics, and distributed real-time operating systems.



Victor Wolfe (S'90) received the B.S. degree in electrical engineering and computer science from Tufts University, Medford, MA, in 1983, the M.S.E. degree in computer and information science from the University of Pennsylvania, Philadelphia, in 1985, and is currently a Ph.D. degree candidate in computer and information science at The University of Pennsylvania.

He was a Computational Design Engineer for General Electric's Space Systems Division from 1983 to 1986. His research interests include concurrency control in real-time systems and distributed real-time programming languages.