PROCESS SYNCHRONIZATION: DESIGN AND PERFORMANCE EVALUATION OF DISTRIBUTED ALGORITHMS

Rajive Bagrodia

November 1987
CSD-870062

PROCESS SYNCHRONIZATION: DESIGN AND PERFORMANCE EVALUATION OF DISTRIBUTED ALGORITHMS[1]

Rajive Bagrodia

October, 1987

Computer Science Department
3531 Boelter Hall
University of California at Los Angeles,
Los Angeles, CA 90024

Arpanet address: rajive@cs.ucla.edu.arpa

[1]This research was partially supported by the Defence Advanced Projects Agency under contract No. F29601-87-K-0072.

ABSTRACT

The concept of n-party rendezvous has been proposed to implement synchronous communication among an arbitrary number of concurrent, asynchronous processes. The synchronization and exclusion problems associated with implementing n-party rendezvous are expressed succintly in the context of the committee coordination problem. This paper presents a simple solution for the problem and shows how it can be implemented in a variety of ways. The paper also compares the performance of the implementations suggested in this paper with other algorithms for this problem.

1. Introduction

Some recent research efforts [Charlesworth 87, Forman 87, Milne 85, Back 84, Francez 86] have focused on the utility of communication primitives that model synchronous communication between an arbitrary number of asynchronous processes; this form of communication is referred to as an **n-party rendezvous**. The synchronization and exclusion problems associated with implementing n-party rendezvous for a set of concurrent processes are illustrated by Chandy & Misra [Chandy ed] in the following anthropomorphic description of this problem:

> ... Professors in a certain university have organized themselves into committees. Each committee has an unchanging membership roster of one or more professors; each professor is in zero or more committees. From time to time, a professor ... starts waiting to attend some committee meeting and remains waiting until a meeting of a committe, of which he is a member, is convened. All meetings terminate in finite time. The restrictions on convening a meeting are as follows: (1) a meeting of a committee is convened only if all its members are waiting, and (2) no two committees convene meetings simultaneously if they have a common member. The problem is to devise a protocol that ensures that ... if all members of a committee are waiting, then a meeting involving some member of this committee is convened.

We consider two recent proposals for distributed programming, Script [Francez 86] and Raddle [Forman 87], that incorporate communication mechanisms that can be represented by the committee-coordination problem. The *script* notation was proposed as an abstraction mechanism for communication patterns. A *script* defines a communication pattern among a set of *roles*, where a *role* is a formal process parameter of a *script*. In order to participate in a particular type of communication, processes *enroll* themselves in an instance of the corresponding *script*. If script enrollment is allowed within the guard of an alternative command, the problem of deciding which script-instance may be executed, reduces to the committee-coordination problem as follows: a script-instance represents a committee and an *enrolling* process a professor. Since script enrollment is allowed within a guard, a process may be ready to enroll in any one of many different script-instances. A script-instance (committee) may be executed (convened) if the required set of processes (professors) are waiting to *enroll*.

Raddle proposes the use of *teams* as an encapsulation mechanism to define the behavior of a set of *roles*, some of which may be formal process parameters of a *team*. The *roles* within a *team* communicate via *interactions*; an *interaction* specifies synchronous communication among a set of *roles*. Raddle provides a guarded construct, called a *rule*, which allows an *interaction* in the guard. An *interaction* may be executed only when all *roles* named within it are *ready*. Once again, the problem of selecting *interactions* for execution reduces to the committee-coordination problem as follows: an *interaction* is a committee and *roles* are professors. A particular *role* (professor) may be ready to execute (attend) any one of many *interactions* (meetings). An *interaction* (committee) may be executed (convened), when all the *roles* (professors) named in the *interaction* (committee) are waiting.

The n-party rendezvous is an extension of the binary rendezvous that has been suggested for CSP [Hoare 78] and Ada [Ada 82]. In a binary rendezvous, a communication involves the synchronization of exactly two processes. A number of algorithms have been suggested to implement binary rendezvous [Bernstein 80, Buckley 83, Van De Snepscheut 81, Natarajan 86, Schneider 82, Bagrodia 86]. Chandy & Misra [Chandy 87] have proposed an elegant distributed algorithm, referred to as the committee-coordination algorithm to implement n-party rendezvous in the context of the committee coordination problem. In this paper, we present two simple algorithms to implement n-party rendezvous and compare their performance with the committee coordination algorithm.

Section 2 presents a detailed description of the problem. Section 3 introduces our solution. Sections 4 and 5 present centralized and distributed implementations of the solution. Section 6 presents a brief description of the committee coordination algorithm. Section 7 describes a modified implementation of the solution, which uses an algorithm for the dining philosophers problem to implement exclusion. Section 8 compares the performance of the three algorithms discussed in this paper. Section 9 is the conclusion.

2. The Problem

Let $\mathbf{P} = \{p_1, p_2, \ldots, p_n\}$ be a set of **n** *processes* and $\mathbf{E} = \{e_1, e_2, \ldots, e_m\}$ be a set of **m** *events*. Each process participates in zero or more events. The set of events in which a process p_i participates is called its *event-set* and is referred to as E_i; each E_i is a subset of **E**. Each event, say e_k is associated with a set of one or more processes; this set is called the *process-set* of the event and is referred to as P_k; each P_k is a subset of **P**. Two events e_k and e_j are said to be in *conflict* if $P_k \cap P_j \neq \{\ \}$. The process-set of an event and the event-set of a process are both *static* sets.

A process is either *idle* or *active*. An *active* process autonomously makes the transition to become *idle*. An *idle* process p_i is waiting to **commit** to any one of the events in E_i. A process commits to some event e_k, only when it determines that all other processes that belong to P_k will also do so. An *idle* process may commit to at most one event at any time. Every process in the system satisfies the following two conditions:

- An *idle* process remains *idle* until it commits to some event C1

• An *idle* process becomes *active* if it commits to some event C2

An event is either *enabled* or *disabled*. An event e_k is *enabled* iff all processes that belong to P_k are *idle*; e_k is *disabled* if there exists some process $p_i \in P_k$, such that p_i is *active*. We use the phrase 'an event e_k is executed' to mean that each process that belongs to P_k has committed to e_k. We are required to devise an algorithm which allows an *idle* process to commit to an *enabled* event such that the following safety and liveness properties are satisfied:

1. Safety(Exclusion):
   a. If a process p_i commits to an event e_k, then $\forall p_j \in P_k$, process p_j cannot commit to another event. In other words, conflicting events cannot be executed simultaneously.

   b. An *active* process cannot commit to any event.
2. Liveness:

   a. Synchronization: If process p_i commits to event e_k, then all processes that belong to P_k will eventually commit to event e_k.

   b. Progress: If all processes that belong to the process-set P_k of some event e_k are *idle*, then eventually some p_j that belongs to P_k must become *active*. This property ensures that if an event e_k is *enabled*, then eventually, e_k is *disabled*.

In the above discussion, we have assumed that when a process is *idle* it is waiting to commit to any event from its event-set. In general, a process may be waiting to commit to any one of only a subset of the events from its event-set. As indicated subsequently, the algorithm presented in this paper can be easily extended to the general problem.

This paper does not impose any semantic association with events or their execution. We consider the communication primitives associated with CSP and Ada to indicate the possible semantics that may be associated with events and their execution. In CSP, an event represents a pair of *matched* CSP communication statements. Execution of an event corresponds to a message being sent by one process and received by another. For instance, consider an event e_k that represents synchronous communication between two processes, say p_1 and p_2; the process-set of e_k is $\{p_1,p_2\}$. In this context, the term process p_1 commits to event e_k, may imply that p_1 *synchronously* sends a message to process p_2. Similarly, in Ada, an event represents a remote procedure call; execution of an event corresponds to the execution of a *procedure call* statement in one task (process) and a corresponding *accept* statement in another task. In the context of Ada, the term process p_i commits to an event may imply that a particular **accept** statement was executed in task p_i.

3. A Solution

For every process p_i in the system, we introduce two variables as follows:

- w_i: total number of times process p_i has become *idle*; also referred to as the *idle-count* for p_i.
- n_i: total number of times process p_i has committed to any event; also referred to as the *active-count* for p_i.

We temporarily ignore the problem of overflow of variables w_i and n_i. We assume that initially, every process is *active*, and variables n_i and w_i are initialized to 0 for every p_i. In order to satisfy the safety conditions, the following invariant must be satisfied by every process in the system:

$$n_i \leq w_i \leq n_i+1 \qquad \text{I1}$$

We further claim that an event e_k can be executed if the following condition is satisfied for all processes that belong to its process-set:

$$\forall p_i \in P_k,\ w_i = n_i+1 \qquad \text{L1}$$

The condition L1 is henceforth referred to as the progress condition, and is used to ensure that all processes in the process-set of a given event are *idle*.

In the following sections, we present various implementations that maintain the invariant and ensure that eventually an event that satisfies the progress condition is selected for execution.

4. Centralized Algorithm

The simplest solution to the problem of n-party rendezvous is to designate a single process as a centralized manager for the system; this manager is referred to as **M**. We assume that **M** has access to the process-set of each event in the system and stores the *idle-count* and *active-count* for each process. All counters are initialized to zero. When any process p_i in the system becomes *idle*, it sends a *ready* message to **M**. On receiving a *ready* message from some p_i, the manager increments the idle-count of p_i by 1 and finds an event, if any, that satisfies the progress condition L1 defined above. If such an event exists, the manager increments the active-count n_i, for every p_i that belongs to P_k by 1, and informs each p_i to commit to the event[2]. It can be shown that the above algorithm does not violate invariant I1 and the exclusion, synchronization and progress properties are trivially satisfied. Although this algorithm may be suitable for certain architectures, in general, the concerns of performance and reliability argue for a distributed algorithm.

[2]The centralized algorithm may be implemented more simply by using a single bit in the manager to represent the current state of each process. The above implementation is suggested as it can be generalized in a simple manner to a distributed implementation.

5. Distributed Algorithm

5.1. Informal Description

The previous algorithm may be distributed by using multiple managers located at different processes. The distributed implementation is referred to as the event-manager algorithm. The set of managers is represented by M; m_j represents an element of M. Each m_j is assigned a set of events referred to as *e-set*$_j$. A manager may only select events for execution from its e-set. Each manager is also associated with a set of processes referred to as its *p-set*. The *p-set* of a manager is the union of the process-sets of all events in its e-set. The managers cooperate with one another to select events for execution without violating the safety property. We note that no restrictions are imposed on the composition of an e-set for a manager. In particular, a given event e_k may be managed by more than one manager, and two events e_j and e_k in the e-set of a manager may be in conflict with each other. Conflicts between two events in the e-set of a manager may be resolved locally by the manager. Conflicts between different managers are resolved by using a circulating token. A manager may select an event for execution only when it possesses the token. Various strategies may be defined whereby a manager that holds the token may determine the current state of processes in its p-set. One possibility may be to assign a maximal set of non-conflicting events to each manager. On receiving the token, the manager may poll processes in parallel for their *idle* and *active* counts, and choose events for execution that satisfy the progress condition. The algorithm discussed here extends the idea of the centralized algorithm in a simple manner. Each manager stores the idle-count for every process in its p-set. The idle-count for process p_i stored by manager m_j is represented as $nr_j(p_i)$ and is referred to as its *ready-count*. Due to the asynchronous nature of the system, the ready-count of a process with a manager may be less than its idle-count. We prove subsequently that the following is an invariant for every manager m_j and process p_i:

$$nr_j(p_i) \leq w_i \qquad \text{I2}$$

The active-count for all processes in the system is carried by the token, with the count for process p_i referred to as $nx(p_i)$. On receiving the token, a manager selects events for execution that satisfy the progress condition. This condition may be redefined in terms of the local ready-count of a manager and the active-count carried by the token as follows:

$$\forall p_i \in P_k,\ nr_j(p_i) = nx(p_i) + 1 \qquad \text{L2}$$

Once a manager selects an event e_k for execution, $\forall p_i \in P_k$, it increments the active-count for p_i carried in the token by 1, informs p_i to commit to event e_k, and sends the token to the next manager. For brevity, we say that manager m_j executes event e_k to imply that the manager has determined that e_k can be executed and has informed each process in P_k accordingly. Finally, in the distributed implementation, the invariant I1 for every process may be redefined in terms of the active-count carried by the token as follows:

$nx(p_i) \leq w_i \leq nx(p_i)+1$ I3

The synchronization property is guaranteed by the algorithm, since a manager either informs all or none of the processes in the process-set of an event, to commit to the event. The ready-count of a process with a manager is used to satisfy the exclusion property as follows: after an event e_k is executed by a manager m_j, for every p_i in P_k, $nr_j(p_i)$ is exactly equal to the active-count $nx(p_i)$ carried in the token. This implies that when the token reaches another manager m_l, for every p_i in P_k, the ready-count $nr_l(p_i)$ can be at most *equal* to the active-count $nx(p_i)$ for the process. Thus the progress condition L2 cannot be satisfied for any event that conflicts with e_k and manager m_l cannot execute any event that conflicts with e_k. The proof for the exclusion property is presented in the next section.

Correctness of this algorithm only requires that each event be present in the e-set of *at least* one manager. As long as the above property is satisfied, the allocation of events among managers does not affect the correctness of the algorithm. In fact, the algorithm will work correctly even if there is only one manager, in which case the implementation of the algorithm will reduce to a centralized implementation. On the other hand, the number of managers may be equal to the number of processes, with the e-set of each manager containing the entire list of events in the system. This corresponds to having a fully replicated event-list. In the distributed implementations with more than one manager, the performance of the algorithm improves as the number of managers is increased. However, the improved performance is achieved at the cost of heavier message traffic, since a larger number of *ready* messages need to be sent to the various managers. Any arbitrary allocation of events to the managers can be considered in the distributed implementation, in order to optimize the response-time/message-count trade off for a given network configuration. However, in general, for a given number of managers, performance will be enhanced if the events are allocated among the managers in a manner such that the e-set of each manager includes a maximal set of non-conflicting events. The performance issues are examined in greater detail subsequently.

5.2. Algorithm

We now give a precise description of the algorithm described above. The algorithm uses the following three types of messages:

- **ready**(p_i): message sent by a process p_i to a manager to indicate that the process is *idle*.
- **execut**(e_k) : message sent by a manager to a process to inform the process to commit to event e_k.
- **token**(*nx* : array [1..*p*] of **Integer**) : This message represents the circulating token. *p* represents the total number of processes in the system; $nx(p_i)$ is the active-count for process p_i.

Each process continuously alternates between being *idle* or *active*. When a process p_i is *idle*, it sends a *ready* message to each manager in its manager-list. Process p_i then waits to receive an *execut* message from some manager at which point it becomes *active*. In addition to variables w_i and n_i introduced earlier, we introduce variables $idle_i$ and array variable com_i.

- w_i: integer variable that counts the number of times process p_i has been *idle*.
- n_i: integer variable that counts the number of times process p_i has been *active*.
- $idle_i$: boolean variable set to *true* when the process becomes idle; *false* otherwise.
- com_i : $com_i(e_k)$ is set to *true* when the process commits to event e_k; *false* otherwise.

The code executed by a process is expressed in the form of rules P1 and P2 given below. The two rules constitute a single guarded command. We reiterate that a process p_i autonomously sets $idle_i$ to *true* when it becomes idle. Initially each process is assumed to be *active* ($idle_i$ is set to *false*), variables w_i and n_i are initialized to 0, and array com_i is initialized to *false*. The subscripts on variables have been dropped for ease of readability.

P1: Sending a *ready* message:

```
    $n=w \wedge idle \Rightarrow$
        $\forall e_k \in E_i,\ com(e_k) := false;$
        send ready message to each manager in manager_list of $p_i$;
        $w := w+1;$
```

P2: Receiving an *execut* message:

```
    Upon receiving an execut($e_k$) message $\Rightarrow$
        $com(e_k) := true;$
        $idle := false;$
        $n := n+1;$
```

The code executed by a manager is expressed in the form of the rules R1 and R2 given below. Each manager m_j has an integer array nr_j, where $nr_j(p_i)$ counts the number of *ready* messages received by manager m_j from process p_i. Once again the subscripts on local variables have been dropped for ease of readability. For every manager m_j, the array nr_j is initialized to *0*.

R1: Receiving a *ready* message:

```
Upon receiving a ready($p_i$) message ==>
                    $nr(p_i) := nr(p_i)+1$;
```

R2: Executing an event:

```
Upon receiving the token =>
       for every event $e_k$ in the manager's e-set
            if ($\forall p_i \in P_k$, $nr(p_i) = nx(p_i)+1$) then
                  ($\forall p_i \in P_k$, send execut($e_k$) message to $p_i$);
                  ($\forall p_i \in P_k$, $nx(p_i) := nr(p_i)$);
             end_if;
        end_for;
        send token to next manager;
```

5.3. Correctness

5.3.1. Safety

The two safety properties that must be satisfied by the algorithm were stated in section 2. The properties are restated and proven as corollaries to theorem 1 and theorem 2 respectively.

Lemma 1: For every manager m_j and process p_i, $0 \leq nr_j(p_i) \leq w_i$

Proof: From R1, $nr_j(p_i)$ counts the number of times a *ready* message is received by manager m_j from process p_i. From P1, w_i counts the number of times process p_i has sent a *ready* message to any of its managers. Since we assume that the message-transmission is error-free, the number of *ready* messages received by a manager from some process p_i may be at most equal to the number of *ready* messages sent to that manager by p_i.

Lemma 2: If a manager m_j executes an event e_k, then $\forall p_i \in P_k$, $nx(p_i) = nr_j(p_i)$.

Proof: Follows directly from rule R2 for a manager.

Lemma 3:For any process p_i, if $w_i = k_i$, and a manager m_j executes an event e_k such that $p_i \in P_k$, then $nr_j(p_i) = k_i$.

Proof: Consider a process p_i. From P1, we note that w_i is incremented only when p_i becomes *idle* and sends a *ready* message. We use induction on the number of *ready* messages sent by p_i to prove the above lemma.

From the initial value for *nx* and rule R2 (the only rule that modifies the value of this variable), we conclude that

$nx(p_l) \geq 0$ and increases monotonically. A1

<u>Base Case</u>: Let $w_l = 1$. Assume that some manager m_j executes an event e_k, such that $p_l \in P_k$. From the rule for executing an event (R2), it follows that in order for manager m_j to execute event e_k, the following condition must be satisfied:

$nr_j(p_l) = nx(p_l) + 1$

Since w_l is assumed to be 1, due to lemma 1, $nr_j(p_l)$ is either 0 or 1. If $nr_j(p_l)$ is 0, the above condition is satisfied only if $nx(p_l)$ is -1, which is impossible due to A1 above. It follows that $nr_j(p_l)$ must be 1, and thus equal to w_l.

<u>Induction Hypothesis</u>: Let $w_l = k_l$ (i.e. p_l has sent exactly k_l *ready* messages) and assume that some manager m_j executes an event e_k, such that $p_l \in P_k$. Then $nr_j(p_l) = k_l$.

From the above hypothesis and lemma 2, it also follows that

$nx(p_l) = k_l$ A2

<u>Induction Step</u>: We show that if $w_l = k_l + 1$ (i.e. process p_l has sent $k_l + 1$ *ready* messages), and some manager m_j executes event e_k, such that $p_l \in P_k$, then $nr_j(p_l)$ is also equal to $k_l + 1$.

Since w_l is assumed to be $k_l + 1$, it follows from lemma 1 that $nr_j(p_l) \leq k_l + 1$. In order for event e_k to be executed by manager m_j, due to R2, $nr_j(p_l) = nx(p_l) + 1$. Due to A1 and A2, $nx(p_l) \geq k_l$. Thus $nr_j(p_l) \geq k_l + 1$. Earlier, we showed that $nr_j(p_l) \leq k_l + 1$. The two conditions can simultaneously be satisfied only if $nr_j(p_l) = k_l + 1$.

Theorem 1: Conflicting events cannot be executed simultaneously.

<u>Proof</u> Assume that a manager m_j executes an event e_k. We show that no manager may simultaneously execute another event that conflicts with e_k.

When manager m_j executes event e_k, due to lemmas 2 and 3 the following relation must hold:

$\forall p_i \in P_k,\ nr_j(p_i) = nx(p_i) = w_i = k_i$ (say) A3

Further, from lemma 1 and A3 above, it follows that

$(\forall m_l, \forall p_i \in P_k),\ nr_l(p_i) \leq k_i$ A4

Thus, if an event e_k is executed by a manager, due to A3 and A4, the following relation must hold for any manager that subsequently receives the token:

$(\forall m_l, \forall p_i \in P_k)\ nr_l(p_i) \leq nx(p_i)$

Due to the above relation, the progress condition P2 specified in rule R2 for the execution of an event, cannot be satisfied for any event that conflicts with event e_k. We conclude that if a manager executes an event, no other manager in the system can simultaneously execute a conflicting event.

Corollary:If a process p_i commits to an event e_k, no process in P_k may commit to another event. In other words,

$com_i(e_k) \Rightarrow (\forall e_l \neq e_k, \forall p_j \in P_k), \neg com_j(e_l))$

Proof From rule P2, $com_i(e_k)$ is set to *true*, only if process p_i receives an *execut*(e_k) message from some manager. From rule R2, a manager m_j sends an *execut*(e_k) message to a process p_i, only if $p_i \in P_k$, and m_j executes e_k. If $\exists(p_i, p_j \in P_k)$, such that $com_i(e_k) \wedge com_j(e_l)$, where $e_k \neq e_l$, then due to rules R2 and P2, events e_k and e_l must have been executed simultaneously. Since $p_j \in P_k \wedge p_j \in P_l$, events e_k and e_l are in conflict with each other and cannot be executed simultaneously due to the result of the theorem. The result of the corollary follows directly.

Theorem 2: For any process, say p_i, in the system, $n_i \leq nx(p_i) \leq n_i+1$

Proof: From rule R2, $nx(p_i)$ counts the number of *execut* messages sent to p_i and from P2, n_i counts the number of *execut* messages received by p_i. Since message-transmission is assumed to be error-free, the first part of the inequality is trivially satisfied. The exclusion condition in theorem 1 guarantees that conflicting events are not executed simultaneously, which implies that there may be at most one *execut* message in transit to any process. The result of the lemma follows directly.

Corollary: An *active* process cannot commit to any event.

Proof: Due to P2, a process commits to an event, only on the receipt of an *execut* message. We show that an *active* process may never receive an *execut* message, by proving that if p_i is *active*, $n_i = nx(p_i)$.

From the theorem, either $nx(p_i) = n_i$, or $nx(p_i) = n_i+1$. If $nx(p_i) = n_i+1$, due to P2 and R2, it must be an *execut* message is in transit to process p_i. This implies that a manager has executed some event e_k, such that $p_i \in P_k$. When a manager executes an event e_k, due to lemmas 2 and 3, $\forall p_i \in P_k, nx(p_i) = w_i$. Subsequently, due to the exclusion condition in theorem 1, $nx(p_i)$ could not have been incremented. From $nx(p_i) = w_i$ and $nx(p_i) = n_i+1$, it follows that $w_i = n_i+1$. Due to P1, this implies that process p_i is *idle*, which contradicts the statement of the corollary.

5.3.2. Liveness

Lemma 5:The *idle-count* of a process is equal to or at most one more than its *active-count* carried by the token. Stated otherwise: $\forall p_i,\ nx(p_i) \le w_i \le nx(p_i)+1$

Proof: We consider two cases: process p_i is *idle* or p_i is *active.*

Case i: p_i is *active*. The corollary to theorem 2 implies that if p_i is *active*, $nx(p_i)=n_i$, and no *execut* message is in transit to the process. From rules P1, P2 and the corollary to theorem 2, it also follows that if p_i is *active*, $n_i=w_i$. The two relations imply that $w_i=nx(p_i)$ and the invariant is satisfied.

Case ii: p_i is *idle*. From P1, when p_i is *idle*, $w_i=n_i+1$. Further, from theorem 2, either $n_i=nx(p_i)$ or $n_i=nx(p_i)-1$. In either case, $w_i=nx(p_i)+1$, or $w_i=nx(p_i)$ and the invariant is satisfied.

Lemma 6: For any process p_i, if $w_i=nx(p_i)$, then p_i is *active* or eventually becomes *active*.

Proof: Assume that p_i is *idle* and $w_i=nx(p_i)$. If process p_i is idle, due to rule P1, it must eventually send a *ready* message and increment w_i, such that $w_i=n_i+1$. From the assumption of the lemma, this implies that $nx(p_i)=n_i+1$. Due to R2 and P2, $nx(p_i)=n_i+1$ implies that an *execut* message has been sent to process p_i, but has not been received. Since message-transmission is assumed to be error-free, the *execut* message must eventually be received by p_i, causing it to become *active* due to rule P2.

Theorem 3: Progress: If an event, say e_k is *enabled*, then eventually some p_i that belongs to P_k must become *active.*

Proof: Consider an event e_k, such that $\forall p_i \in P_k$, p_i is *idle*. Due to P1, each p_i must have sent a *ready* message to its manager(s) and due to P2, no process in P_k has received an *execut* message.

For every p_i that belongs to P_k, let $w_i = k_i$. Since message delivery can take only finite time, eventually for every manager m_j, of a process p_i that belongs to P_k, we must have

$$nr_j(p_i)=k_i \qquad \text{A5}$$

Further, due to lemma 5, for every p_i that belongs to P_k, we also have $nx(p_i) = k_i$ or $nx(p_i) + 1 = k_i$. If for any p_i that belongs to P_k, $nx(p_i)=k_i$, then due to the lemma 6, p_i must eventually become *active* and the liveness property will be satisfied. We assume, for every p_i that belongs to P_k:

$$nx(p_i) + 1 = k_i \qquad \text{A6}$$

If the processes remain *idle*, then due to A5 and A6 eventually, we must have some manager m_l, such that when m_l receives the token, for every p_i that belongs to P_k:

$nx(p_l) + 1 = nr_l(p_l)$

The above represents the progress condition included in rule R2 for the execution of an event by a manager. It follows that manager m_l must execute event e_k causing processes that belong to P_k to become *active*.

5.4. Extension

In the general case of the n-party rendezvous problem, on becoming *idle*, a process may be waiting to commit to only a subset of the events in its event-set. The algorithm described above can be extended to apply to the general case. The condition for the execution of an event e_k by a manager is modified as follows: the ready-count of every process in the process-set of event e_k must be one more that its active-count; in addition, each process in P_k must be waiting to commit to event e_k. Recall that when a process is *idle*, it sends a *ready* message to a manager. On receiving the message, the manager implicitly assumes that the process is waiting to commit to any one of the events from its event-set. The *ready* message is modified to include explicit information that indicates the specific events, to any one of which the process is waiting to commit. In particular, every *ready* message carries a boolean array *elist*; *elist*(k) is *true* if the process sending the message is waiting to commit to event e_k and is *false* otherwise. In addition, we define a boolean array *status* for every manager. On receiving a *ready* message from process p_i, *status*(i,k) is set to *true* if process p_i is waiting to commit to event e_k and is set to *false* otherwise. With an appropriate modification to the manager's data-structures, the condition specified above can be easily incorporated into the rule for execution of an event by a manager.

5.5. Overflow

In the description of the event-manager's algorithm, we have ignored the problem of overflow. We present a simple solution to ensure that no variable exceeds the maximum permissible value for an integer. The algorithm uses the following counters for each process:

- w_i: represents the idle-count for process p_i.
- n_i: represents the active-count for process p_i.
- $nr_j(p_i)$: counts the number of times manager m_j has received *ready* messages from process p_i.
- $nx(p_i)$: represents the active-count carried by the token for process p_i.

The values of the counters for each process are tightly coupled. If $nx(p_i)$ is monitored for overflow, then we can guarantee that neither of w_i, n_i or $nr_j(p_i)$ will overflow as follows: let **maxint** be the maximum permissible value for an integer. Define a variable **max** such that **max** < **maxInt**. As long as $nx(p_i) <$ **max**, due to lemma 4, $n_i <$ **max**, and due to lemma 5, $w_i <$ **maxInt**; this in turn implies $\forall(m_j)$, $nr_j(p_i) <$ **maxInt**, due to lemma 1. We describe a technique that monitors the value of $nx(p_i)$ to prevent overflow errors. We define two new types of messages:

- *reset*: sent to a manager/process to reset its counters.
- *ack*: sent by a manager to indicate that its counters have been reset.

From the algorithm, we note that $nx(p_i)$ is incremented only when a manager possesses the token and executes some event e_k, such that $p_i \in P_k$ (Rule R2). On executing an event e_k, if a manager m_j determines that $\exists p_i : p_i \in P_k$ and $nx(p_i) =$ **max**-1, m_j initiates the scheme to reset all counters; henceforth, m_j is referred to as the **initiator**. The **initiator** resets array *nx* to zero and sends a *reset* message to *all processes in the system*. The initiator does not transmit the token until it has received an *ack* message from all other managers in the system. This ensures that no events in the system are executed until all counters have been reset. On receiving a *reset* message, each process sets its active-count to zero; it resets its idle-count to 1 if it is *idle* or to 0 if it is *active*. A process then sends a reset message containing its idle-count to each manager. On receiving a reset message from p_i, a manager m_j sets $nr_j(p_i)$ to the idle-count carried in the message. After receiving reset messages from all processes in its p-set, the manager sends an ack message to the initiator. The algorithm proceeds normally after the initiator has received ack messages from all managers in the system. We note that invariant I3 is temporarily violated from the time a manager resets array *nx* until the processes receive the *reset* messages. However, since no events can be executed while the reset procedure is underway, and the invariant is restored on the completion of the reset procedure, correctness of the algorithm is not affected.

6. Committee-Coordination Algorithm

In this section, we briefly describe the algorithm suggested by Chandy & Misra [Chandy 87]. This algorithm, referred to as the committee-coordination algorithm splits the problem of n-party rendezvous into two sub-problems: exclusive selection of one among many conflicting events, and the determination of the state of each process in the process-set of the selected event. The exclusion problem is solved by appropriately mapping the n-party rendezvous problem onto the dining philosopher's problem. The synchronization problem is solved by a judicious use of tokens. The committee-coordination algorithm postulates a special process, called a coordinator for every event in the system. Two coordinators c_i and c_j are neighbors if the corresponding events e_i and e_j conflict with each other. Every pair of neighbors share a unique fork. Each process in the system is initially assigned a fixed number of tokens, whose count is equal to the number of events in the event-set of that process. When a process becomes *idle*, it sends a *token* to the coordinator for every event in its event-set. On receiving tokens from all processes in the process-set of its event, a coordinator requests forks from all its neighbors. The algorithm guarantees that every coordinator that requests forks will eventually receive all the forks from its neighbors. Once a coordinator has received all its forks, it determines if all processes in its process-set are still *idle*. (This is required since a neighboring coordinator may have executed a conflicting event.) The algorithm defines a scheme based on shuffling tokens among the coordinators to allow a coordinator to determine whether a conflicting event has been executed. The algorithm thus works in two steps: in the first step, a coordinator obtains the exclusive right to execute its event by requesting forks from its neighbors. In the second step, a coordinator that has obtained all the forks determines the current state of processes by requesting tokens from its neighbors. In the next section, we show how the event manager algorithm may be modified to use the selection technique suggested by the committee coordination algorithm.

7. Modified Event Manager Algorithm

This section presents a modified version of the event manager algorithm, referred to as the *M*odifed *E*vent *M*anager (MEM) algorithm. The MEM algorithm combines the synchronization technique used in the event-manager (EM) algorithm (counting the number of times each process has been *idle* and *active*) with the selection technique of the committee-coordination (CC) algorithm (using auxiliary resources to arbitrate between conflicting events). The exclusion problem is solved in the MEM algorithm (as in the CC algorithm) by mapping the n-party rendezvous problem onto the dining philosophers problem. The dining philosopher's problem [Chandy 84] may be stated as follows: Let G be a finite, undirected graph. A philosopher is placed at each vertex of G. Two philosophers t_j and t_k are neighbors if an edge exists between them. Each edge has a fork associated with it. A philosopher is in one of three states:*thinking*, *hungry*, or *eating*. A philosopher autonomously goes from thinking to hungry. When a philosopher is hungry, he tries to obtain the forks on all edges that are incident on that philosopher. When the hungry philosopher obtains all its forks, he starts to eat. The eating period of every philosopher is assumed to be of finite duration. After a philosopher finishes eating, he returns to the thinking state. It is required to devise an algorithm that enables every philosopher to eat within a finite time of his becoming hungry.

The algorithm to solve the dining philosopher's problem as presented in [Chandy 84] is as follows: Let the fork shared between philosophers t_j and t_k be referred to as $f_{j,k}$. A fork is either *dirty* or *clean.* Initially all forks are dirty. Subsequently, a fork is *dirty,* if it is used by a philosopher to eat and remains *dirty* until it is cleaned. A *dirty* fork becomes *clean* when it is sent by one philosopher to its neighbor. On becoming hungry, if a philosopher t_j does not possess fork $f_{j,k}$, he requests the fork from t_k. A philosopher t_k responds to a request for a fork according to the following rule: if t_k is not *eating* and the fork is *dirty,* then t_k cleans the fork and sends it to t_j; otherwise t_k complies with the request after he finishes eating. The algorithm satisfies the exclusion property by ensuring that *at most* one of a group of neighboring philosophers will possess all the forks it shares with his neighbors at any given time. Further, the algorithm guarantees that every philosopher's request for a fork will eventually be satisfied (and hence every hungry philosopher will eventually eat).

We now present the MEM algorithm which solves the exclusion problem using the above algorithm and the synchronization problem using message-counts. As in the EM algorithm, the MEM algorithm postulates a set of managers. Each manager is associated with a set of events, referred to as its e-set[3], and a set of processes, called its p-set[4]. Each manager stores the *idle-count* and *active_count* for every process that belongs to its p-set. Variables $nr_j(p_i)$ and $na_j(p_i)$ respectively represent the idle-count and active-count stored by manager m_j for process p_i. Two managers m_i and m_j are neighbors, if p-set$_i$ and p-set$_j$ have a common member. A unique fork is defined for every pair of neighboring managers, with $f_{i,j}$ representing the fork shared by managers m_i and m_j. Each fork, say $f_{i,j}$, is associated with a set of processes represented by *f-set*$_{i,j}$. This set consists of processes that are members of both p-set$_i$ and p-set$_j$. Each fork carries the active-count for processes in its f-set. A manager is either *thinking, hungry,* or *eating.* A manager is *thinking* if every event in its e-set is *disabled.* A *thinking* manager becomes *hungry* when it determines that an event e_i in its e-set is *enabled* (that is all processes in the process-set of e_i are *idle*). On becoming *hungry,* a manager requests the forks it does not possess from its neighbors. The rules for requesting and releasing forks are identical to those described above for the dining philosophers algorithm. When sending a fork to a neighbor, the manager also sends the active-count for all processes in the fork's f-set. On receiving a fork *f,* for every p_i in *f*.f-set, a manager updates its local active-count for p_i to that carried by the fork, if the count carried by the fork is greater. A *hungry* manager starts to *eat* when it holds all its forks that it shares with its neighbors. Events may be executed by a

[3]The e-set of a manager is the set of events managed by a manager

[4]The p-set of a process was defined as the union of process-sets of all events managed by the manager.

manager only when it is eating. An eating manager can deduce that an event e_k from its e-set is enabled if e_k satisfies the progress condition. For a manager m_j and an event e_k, this condition is restated as follows:

$\forall p_i \in P_k,\ nr_j(p_i) = na_j(p_i)+1$ L3

If the manager finds an event that satisfies the above condition, the event is executed by the manager. If event e_k is executed by manager m_j, $\forall p_i \in P_k$, the manager increments $na_j(p_i)$ by 1, and sends an *execut* message to every p_i in P_k. We now specify the conditions under which a manager makes the transition from the thinking to hungry and eating to thinking states. (The transition from the hungry to eating states is determined by the dining philosophers algorithm.)

Thinking to Hungry

A manager goes from thinking to hungry, when the following condition is satisfied for some event e_k in its e-set:

$\exists e_k: \forall p_i \in P_k,\ nr_j(p_i) > na_j(p_i)$

Note that due to arbitrary delays in message-transmission, a manager may become hungry based on obsolete values of the idle and active-counts for a process.

Eating to Thinking

An eating manager goes to thinking after it has scanned its e-set for enabled events. We introduce a boolean variable called *test*. The variable is set to *false*, when a manager is *thinking*. It is set to *true* by a manager after it has scanned its e-set for enabled events. An eating manager, say m_j, goes to thinking, within a finite time of $test_j$ being set to *true*.

In the MEM algorithm, a hungry manager that holds all its forks, uses the active-count carried by the fork to deduce the current state of a process. This obviates the need to use separate tokens to guarantee process synchronization, thus significantly improving the performance of this algorithm with respect to the CC algorithm. Further, unlike the EM algorithm, only managers that manage conflicting events exchange information in the MEM algorithm, thus potentially improving its performance with respect to the EM algorithm, in architectures where the synchronization patterns among processes are localized. Finally, as in the EM algorithm, the number of events in the e-set of each manager in the MEM algorithm does not affect the correctness of the algorithm, as long as each event is present in the e-set of at least one manager. In the MEM algorithm, as the number of events in the e-set of a manager is increased, the number of neighbors for that manager also increases, which increases the selection time for the algorithm. As the e-set of each manager is expanded, the manager is more likely to become hungry and request forks, leading to increased fork transmissions in the network which might further increase the average selection time for an event. This implies that the algorithm should perform optimally when each manager has exactly one event in its e-set. A comparative study of the performance of the three algorithms is presented in the next section.

We now present the MEM algorithm. The rules and data structures for a process are identical to those for the event manager algorithm and have been described in section 4.2.2 . The actions of each manager are described in terms of the following rules. The rules for requesting and relinquishing forks have been omitted and the reader is referred to the dining philosopher's algorithm as described in [Chandy 84]. For the purpose of this paper, we assume that on becoming hungry, a manager requests the forks it does not possess from its neighbors and every hungry philosopher eventually obtains all its forks at which point it goes to eating.

Each manager m_j has integer arrays nr_j and na_j, where $nr_j(p_i)$ and $na_j(p_i)$ respectively refer to the idle and active count for process p_i with manager m_j. Arrays nr_j and na_i are initialized to *0*. In the following rules, subscripts on local variables have been dropped for ease of readability. Note that variable *test* is set to *false*, when a manager is *hungry*, and variable *eating* is set to *true* when the manager is eating.

R1: Receiving a *ready* message:

```
    On receiving a $ready(p_i)$ message $\Rightarrow$
                              $nr(p_i) := nr(p_i)+1$;
```

R2: Receiving a fork:

```
    On receiving a fork $f$ $\Rightarrow$
          $(\forall p_i \in f.\text{f-set}$, if $f.na(p_i) > na(p_i)$ then $na(p_i) := f.na(p_i))$
```

R3: Sending a fork:

```
    On sending a fork $f$ $\Rightarrow$
          $(\forall p_i \in f.\text{f-set}$, $f.na(p_i) := na(p_i))$
```

R4: Executing an event:

```
    $eating \wedge \sim test \Rightarrow$
          $test := true$;
          for every event $e_k$ in the manager's e-set
               if $(\forall p_i \in P_k$, $nr(p_i) = na(p_i)+1)$ then
                    $(\forall p_i \in P_k$, send $execut(e_k)$ message to $p_i)$;
                    $(\forall p_i \in P_k$, $na(p_i) := nr(p_i))$;
               end_if;
           end_for;
```

7.1. Correctness

In this section, we use some of the results proven for the EM algorithm in section 5 to prove safety and liveness of the MEM algorithm.

7.1.1. Exclusion

Lemmas 1 and 3 in section 5 relate the idle-count of a process with its ready-count with a manager. These lemmas are directly applicable for the MEM algorithm and are reproduced below.

Lemma 1: For every manager m_j and process p_i, $0 \leq nr_j(p_i) \leq w_i$

Lemma 3:For any process p_i, if manager m_j executes an event e_k such that $p_i \in P_k$, then $nr_j(p_i) = w_i$.

We mow use the above results to prove the exclusion property for the MEM algorithm.

Theorem 4: Conflicting events cannot be executed simultaneously.

Proof: When a manager m_j executes an event e_k, from lemma 3, we have

$\forall p_i \in P_k,\ nr_j(p_i) = w_i = k_i$ (say) A7

Assume that a conflicting event e_s is simultaneously executed by some manager, say m_l. We show that this is impossible. Managers m_j and m_l must share a fork $f_{j,l}$, which due to rule R4 must be held by m_j when event e_k was executed. Let a process p_c be a member of both P_k and P_s. Due to A7, when event e_k is executed by m_j, it must be that

$nr_j(p_c) = w_c = k_c$

Further, due to R4 and the above relation, $na_j(p_c) = nr_j(p_c) = k_c$. In order to execute an event, manager m_l must obtain all its forks (including $f_{j,l}$). On receiving $f_{j,l}$ from m_j, due to R2 and the above relation, it must be that

$na_l(p_c) \geq k_c$ A8

Due to lemma 1, $nr_l(p_c) \leq w_c$. Due to A7, this implies that $nr_l(p_c) \leq k_c$, which in turn, due to A8 above implies that $na_l(p_c) \geq nr_l(p_c)$. This relation violates the progress condition L3 for any event (including event e_s) that includes process p_c in its process-set. It follows that event e_s could not have been executed simultaneously by manager m_l as assumed.

7.1.2. Liveness

Informal Idea The circulating token of the event manager algorithm has been implemented in this algorithm by a set of forks, each of which is shared between two managers. In the event manager algorithm, an event was executed only by a manager that possessed the token. Similarly, in the modified event manager algorithm, an event may be executed only by a manager who possesses all his forks. In order to establish a direct relationship between the active-count carried by the token in the EM algorithm, and the active-counts distributed among managers in the MEM algorithm, we define an auxiliary variable *namax*, for the MEM algorithm, as follows:

$$namax(p_i)=\mathbf{max}(na_1(p_i),na_2(p_i),\ldots na_m(p_i))$$

where m represents the total number of managers in the system. In lemma 7, we prove that if a manager m_j is eating, then $\forall p_i \in p\text{-}set_j, na_j(p_i)=namax(p_i)$. Further, due to rule R4, $na_j(p_i)$ is incremented by 1, only if manager m_j is eating and it executes an event e_k, such that $p_i \in P_k$. The above statement, together with lemma 7 implies that $namax(p_i)$ is incremented only when a manager executes an event e_k, such that $p_i \in P_k$ and hence $namax(p_i)$ must count the number of *execut* messages sent to process p_i. Thus the auxiliary variable, $namax(p_i)$ has exactly the semantics associated with the active-count *nx* carried by the token in the EM algorithm. It follows that lemmas 5 and 6 proven for the EM algorithm also apply for the MEM algorithm with the active-count $nx(p_i)$ being replaced by the auxiliary variable $namax(p_i)$. The proof of the liveness property uses lemmas 5 and 6, which are restated here as lemmas 5' and 6' respectively, with the variables substituted appropriately.

Lemma 5': $\forall p_i,\ namax(p_i) \le w_i \le namax(p_i)+1$

Lemma 6': For any process p_i, if $w_i=namax(p_i)$, then p_i is *active* or eventually becomes *active*.

Lemma 7: If a manager m_j is eating, then $\forall p_i \in p\text{-}set_j,\ na_j(p_i)=namax(p_i)$

Proof Let M represent the set of neighbors of m_j (excluding m_j). If m_j is eating, due to the dining philosophers algorithm, it must possess all forks which it shares with its neighbors. Due to rule R2, it must be that $na_j(p_i) \ge na_l(p_i)$, for every $m_l \in M$. The result of the lemma follows from the above relation and the definition of the variable *namax*.

Theorem 5: If an event e_k is *enabled*, then eventually some p_i that belongs to P_k must become *active*.

Proof: Consider an event e_k, such that $\forall p_i \in P_k$, p_i is *idle*. Since the processes are *idle*, due to P1, each

p_i must have sent a *ready* message to its manager(s) and due to P2, no process in P_k has received an *execut* message.

For every p_i that belongs to P_k, let $w_i = k_i$. Since message delivery can take only finite time, eventually for every manager, say m_j, of a process p_i that belongs to P_k, we must have

$nr_j(p_i) = k_i$ A9

Further, due to lemma 5', and the assumption that $w_i = k_i$, we have $namax(p_i) = k_i$ or $namax(p_i) + 1 = k_i$. If $namax(p_i) = k_i$, then due to lemma 6', p_i must eventually become *active* and the liveness property will be satisfied. We assume,

$\forall p_i \in P_k, namax(p_i) + 1 = k_i$ A10

From the definition of the auxiliary variable $namax(p_i)$ and A10 above, it follows that for every manager, say m_j, of a process p_i that belongs to P_k, we must have

$\forall m_j, na_j(p_i) < k_i$

The above relation and A9 together imply that a manager m_j that contains event e_k in its e-set must become *hungry* and set $test_j$ to *false*. The dining philosophers algorithm guarantees that m_j will eventually eat. When m_j eats, due to lemma 7, $\forall p_i \in P_k$, $na_j(p_i) = namax(p_i)$. Due to A10, the above relation yields $\forall p_i \in P_k, na_j(p_i) + 1 = k_i$, which, due to A9 yields the following result

$\forall p_i \in P_k, na_j(p_i) + 1 = nr_j(p_i)$

The above is precisely the condition in rule R4 for the execution of an event by a manager that is eating. It follows that m_j will execute event e_k, causing all processes in p_k to become *active*.

Overflow

As in the EM algorithm, the problem of counter overflow can be prevented by tracking the value of the active-count for a process, and ensuring that it does not exceed **max**, a suitably defined number for a particular architecture. We define a pseudo-event e_p. The process-set of this event is defined such that event e_p is in conflict with every other event in the system. This ensures that if event e_p is executed, no other event in the system may be executed (simultaneously). A manager m_j that executes event e_p initiates the procedure to reset the counters in the system, as described for the EM algorithm. It remains to ensure that event e_p will be executed within a finite time of the active-count for a process becoming equal to **max**. For this purpose, we introduce a set of pseudo-processes, one for each event in the system (except e_p). The pseudo-process for event e_k is henceforth referred to as p'_k. Each pseudo-process, say p'_k, belongs to the process-set of events e_p and e_k. The process-set of event e_p, thus consists entirely of the pseudo-processes defined in the system. On becoming *active*, a pseudo-process immediately becomes *idle*. In general, an *idle* pseudo-process waits to commit to either e_k or e_p. However, if the counter reset procedure needs to be initiated, on next becoming idle, a pseudo-process waits to execute only e_p, the pseudo-event. (Recall that in the generalized version of this problem, a process may wait to commit to any subset of events in its event-set) This implies that eventually, the pseudo-event e_p is the only event in the system that is *enabled*. The liveness condition guarantees that e_p will eventually be executed by some manager, who then initiates the procedure to reset counters as described in the EM algorithm.

8. Performance

The committee coordination problem captures two important issues in the design of distributed systems: exclusion and synchronization. A variety of techniques exist to solve each of these problems. Different combinations of these techniques result in a variety of algorithms, and the use of a specific technique has a significant impact on performance. Two metrics are used in this paper to measure the performance of algorithms: message-count and response time. The response time for each algorithm is measured from the instant that an event becomes *enabled* (as viewed by a global observer) to the instant that it is selected for execution by a specific algorithm. The two components that determine the total response time for an algorithm are:

1. Synchronization time: The time taken by the algorithm to ascertain that a given event is *enabled* (i.e. all processes in the process-set of the event are *idle*).
2. Selection time(T_s): The time taken by an algorithm to select an event (possibly out of many *conflicting* events) for execution.

The three algorithms discussed in this paper use different combinations of techniques to implement exclusion and synchronization. The event manager (EM) algorithm uses a message-counting technique to determine when an event is enabled. The synchronization time for this algorithm is determined entirely by the *average* time required for a message to travel from a process to a manager. Selection among conflicting events is achieved by using a single token that circulates among managers. Hence the synchronization time is determined primarily by the *average* time required by the token to travel between two managers whose e-sets contain an event e_i. The committee-coordination (CC) algorithm uses auxiliary resources (forks and tokens) to implement both selection and synchronization. Furthermore, these resources are transmitted only between the coordinators for conflicting events. The coordinators for events that do not conflict with each other do not exchange any messages. The selection time for this algorithm is determined by the *average* time required by a coordinator to obtain forks from its neighbors. This in turn depends on the number of neighbors for every coordinator and on the average distance between neighboring coordinators. The synchronization time is determined by the time required for a message to travel from a process to a coordinator, and also by the time required by a coordinator to obtain tokens from its neighbors. As a result, the response time for this algorithm is heavily influenced by the average message transmission time between coordinators for *conflicting* events. Finally, the modified event manager algorithm(MEM), uses message-counting to implement synchronization and auxiliary resources (forks) to implement selection. Thus, the synchronization time for this algorithm is determined entirely by the *average* time required for a message to travel from a process to a manager. The selection time for this algorithm is determined by the *average* time required by a manager to obtain forks from its neighbors.

A performance study was undertaken to compare and evaluate the response time and message-count for each of the three algorithms discussed above. Simulation models were constructed for each algorithm using the MAY [Bagrodia 87a] simulation language. Approximate analytical models were constructed for a restricted version of each algorithm to validate the results of the simulation. The detailed performance study has been described in [Bagrodia 87b]. In this section, we present a summary of some significant results.

In order to simplify the models, the following assumptions were made about message-transmissions: the simulation models assumed that *all* processes are connected to each other via point-to-point channels. The models also assume that the time taken to perform local computations by a process is insignificant as compared to the message-transmission time and ignores the former in the computation of the response time. Finally, to simplify the allotment of the manager (coordinator) processes, we assume that, in the EM and MEM algorithms, the number of managers is equal to the number of processes, and in the CC algorithm, the number of coordinators is equal to the number of processes[5]. Due to the above assumption, a single manager/coordiantor can be assigned uniformly to each process. None of the above assumptions are an inherent feature of the simulation model. They were made primarily to simplify the code for message routing among processes. In addition to the above assumptions, the analytical models also assume that queuing delays are negligible and can be ignored for message-transmission.

[5]Since the CC algorithm uses one coordinator for each event, this also implies that the number of events is equal to the number of processes

The main factors that may potentially affect the performance of the algorithms considered in this paper are the total number of processes(*n*), the total number of events in the system(*e*), the average time to transmit a message between processes(*t*), the level of activity *a*, and the synchronization pattern among processes in the system. The level of activity in the system is determined by the frequency with which events in the system are *enabled* for execution. The synchronization pattern is specified by the cardinality of an event *c*, the degree of conflict for each event *d*, and the level of activity *a*, in the system. We define the *cardinality* of an event as the number of processes in the process-set of that event, and the *degree of conflict* as the maximum number of events that may be in conflict with a given event. In addition to the above factors, the performance of the event-manager algorithm also depends on the average distance separating two managers for any event. This factor in turn depends on the number of managers in the system, the number of events managed by each manager and the allocation of events among the managers. As mentioned earlier, we assume that, the number of managers is equal to the number of processes. Under this assumption, for the EM algorithm, it was determined that as the number of events in the e-set of each manager is increased, the response time for the algorithm decreased and was lowest for the configuration where each manager contained the entire list of events in its e-set. In contrast, for the MEM algorithm, as the number of events in the e-set of each manager was increased, the response time for the algorithm also increased and was lowest for the configuration where each manager contained exactly one event in its e-set. Subsequent discussions in this section assume that the e-set of each manager for the EM and MEM algorithms is set up for the minimum response time.

Experiments conducted for individual algorithms indiacted that *other things being equal*, the response time for each of the three algorithms were independent of *n* and *e*. Furthermore, under the simplified assumptions for message-transmission delays, the response time varied almost linearly with *t*. With respect to the level of activity, it was seen that for the CC and MEM algorithms, as the level of activity in the system was increased, the response time for the algorithm also increased. This was due primarily to the fact that as more events are simultaneously *enabled* for execution, there is greater contention for forks among the coordinators for conflicting events, which increases the selection times for these algorithms. However, with respect to the EM algorithm, as the level of activity was increased, the response time for the algorithm actually decreased by a small amount. For this algorithm, as the level of activity in the system increases, in general, the probability that a manager which currently holds the token will contain an enabled event in its e-set also increases, thus reducing the selection time for the algorithm.

We now compare the response time and message-counts for the three algorithms. The experiments were performed for two different types of process configurations: uniform configurations, in which all processes are equidistant from each other, and non-uniform configurations in which processes are organized into clusters, with the distance between processes in the same cluster being much less than that between processes in different clusters. For simplicity, in uniform process configurations, the experiments were restricted to synchronization patterns, where each event has the same cardinality and degree of conflict. Each experiment was run for a duration required to schedule 500 events for execution. The results reported in this paper represent the average response time and number of messages required to schedule *one* event for execution for each of the three algorithms.

The response time and message-counts for the uniform process configurations are presented in figures 9-1 and 9-2. This set of experiments studied the effect of varying the cardinality (*c*) (and degree of conflict *d*) of an event on the message-count and response time for the algorithms. The response time for the EM algorithm was essentially independent of the degree of conflict, whereas the response time for the CC and MEM algorithm increased as *d* increased. Furthermore, the response-time for the EM algorithm was significantly lower than that of the CC and MEM algorithms; the response time of the MEM algorithm was lower than that for CC by exactly the amount of time required by a coordinator to obtain tokens from its neighbors. With respect to message-counts, the EM algorithm required the maximum number of messages, whereas the MEM algorithm had the lowest message-count. Note that the message-counts reported in figure 9-2 also include the messages that were generated due to events that were *enabled* but were subsequently not executed (due to the execution of a conflicting event). For each algorithm, the number of messages increased, as the cardinality and degree of conflict was increased. In case of the EM algorithm, as the cardinality of events is increased, each manager must receive an increasing number of *ready* messages for each event in its e-set, thus increasing overall message-traffic in the network. However, under the assumption of point-to-point message transmissions, the increased message-traffic may occur in parallel, and does not significantly affect the response time for the algorithms. In case of the MEM and CC algorithms, if total number of processes and events are held constant, increasing the cardinality of an event increases the degree of conflict in the system. As each event is in conflict with an increasing number of events, the number of neighbors for each coordinator also increases. This gives rise to an increase in the number of forks requested and transmitted, thus increasing both the response time and the overall message-traffic in the network.

In the case of non-uniform process configurations, two types of synchronization patterns were identified: *localized synchronization*(LS), where processes interact primarily with other processes in the same cluster; and *uniform synchronization*(US), where processes interact uniformly with other processes, regardless of their location. Figure 9-3 presents a non-uniform process configuration comprising of two clusters. The transmission time for a message from one process to another in the same cluster is assumed to be τ_1 units, and the transmission time for messages from one cluster to another is τ_2, where $\tau_2 = 5\tau_1$. The results for this experiment are presented in tables 9-1 and 9-2, which compares the performance of the two algorithms for both LS and US configurations. For the EM algorithm, the response time was the same in both configurations, since the number of managers, and the e-set of each manager remains the same. For the MEM and CC algorithms, in the LS configuration, the coordinators for *conflicting* events are relatively close to each other, and hence the response time was small. However, in the US configuration, the distances between coordinators for *conflicting* events included the large separation between different clusters causing the response time of the algorithm to increase significantly. As seen from the results in tables 9-1 and 9-2, in the LS configuration, the response time for the MEM and CC algorithm was much superior to that of the EM algorithm, whereas in the US configuration, the EM algorithm performed better.

9. Conclusion

The committee coordination problem is a non-trivial problem that captures the two central issues in the design of distributed systems: synchronization and exclusion. This paper proposed a basic solution to this problem in the form of two invariant properties. Two different implementations of the solution were described. In general, a variety of centralized and distributed techniques exist to solve each of the two problems. For instance, synchronization may be solved by means of polling, message-counts, or using auxiliary resources like tokens; the exclusion problem may be solved by using timestamps, unique process-ids, auxiliary resources(forks), or probability-based techniques. The event manager algorithm presented in this paper used message-counts to implement synchronization and a circulating token to implement exclusion. The committee-coordination algorithm used auxiliary resources (tokens and forks) to implement synchronization and exclusion. The modified event manager algorithm combined the technique of message-counts to implement synchronization, with the use of auxiliary resources to implement exclusion. Other known techniques to solve synchronization and exclusion problems may be combined with each other to suggest different ways to solve this problem. The variety of possible

Number of processes(*n*)=number of events(*e*)= 10.
Average message transmission time between two processes(*t*) = 5000.

c	*d*	Response Time		
		CC	MEM	EM
2	2	18444	8444	6915
3	4	20574	10570	6962
4	6	21516	11516	7060
5	8	22059	12364	6986

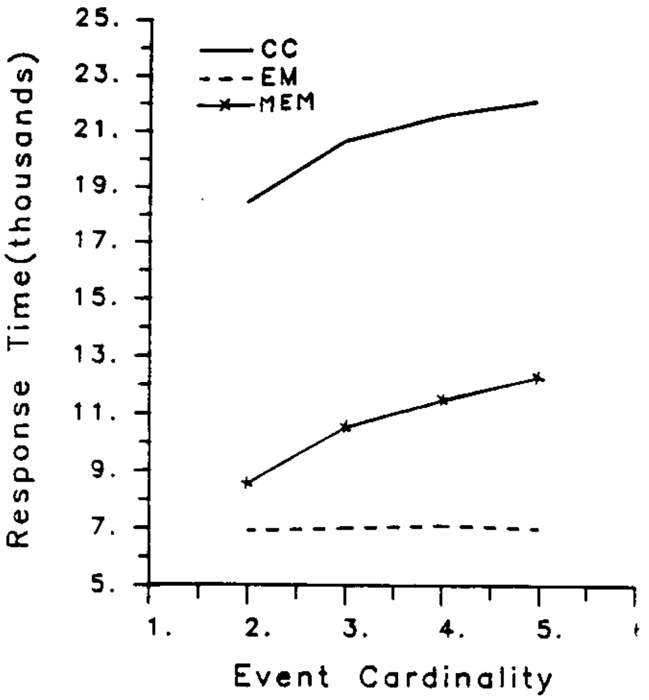

Figure 9-1: Comparison of CC and EM Algorithms: Response Time

solutions indicates the strong need for some performance metrics that can be used to identify the suitability of a specific algorithm for a specific type of network configuration. The paper presented a brief performance study of the three algorithms. Results from the study were used to identify the basic trade-offs among the different algorithms. Under the assumptions of this study, in uniform process configurations, whereas the event manager algorithm had the lowest response time, it also had the highest message-count. In non-uniform configurations, where processes are not equidistant from each other, the performance of the algorithms depended on the pattern of interaction among the processes.

Acknowledgements

This work was initiated at MCC. I am thankful to Dr. Ira Forman of MCC for initial discussions on the problem. I am indebted to Professors K.M.Chandy and J.Misra, of the University of Texas at Austin for their invaluable input in the formulation and presentation of the algorithm.

Number of processes(*n*)=number of events(*e*)= 10.
Average message transmission time between two processes(*t*) = 5000.

c	*d*	Message Count		
		CC	MEM	EM
2	2	10.2	6.2	27.2
3	4	21.2	13.2	41.2
4	6	34.2	22.1	55.6
5	8	49.5	33.8	65.9

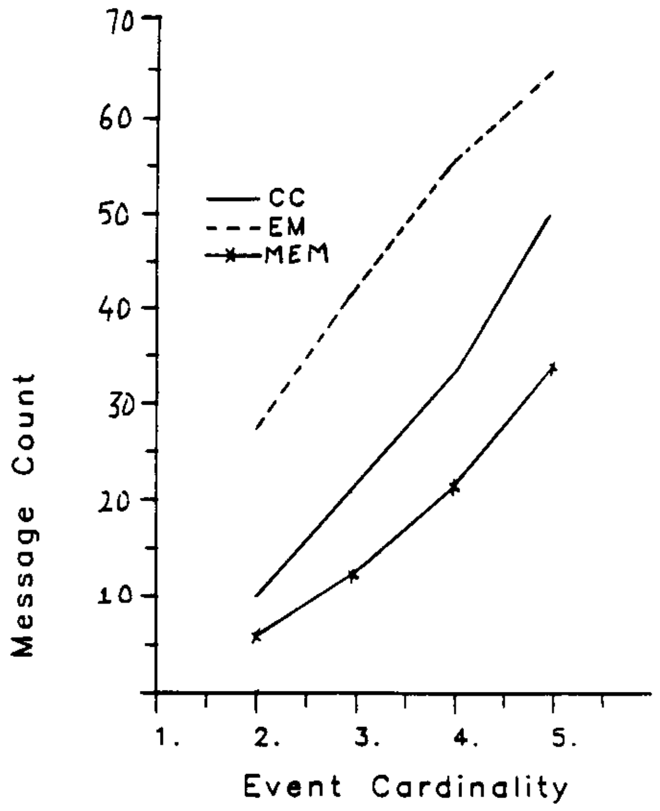

Figure 9-2: Comparison of CC and EM Algorithms: Message Count

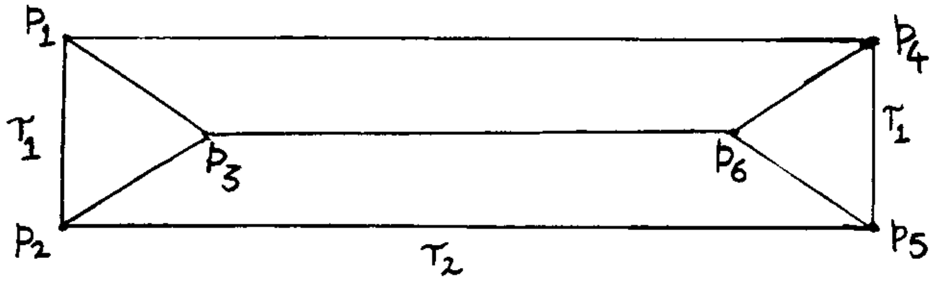

Figure 9-3: Network of Two Clusters

Number of processes(n) = 6.
Number of events(e) = 6; e_1 = {p_1,p_2}; e_2 = {p_2,p_3}; e_3 = {p_1,p_3};
e_4 = {p_4,p_5}; e_5 = {p_5,p_6}; e_6 = {p_4,p_6}.
Message transmission times : τ_1 = 5000; τ_2 = 25000.

Algorithm	Message Count	Response Time
CC	10.0	19188
MEM	6.0	9188
EM	18.4	25489

Table 9-1: Comparison of Algorithms : LS Configuration

Number of processes(n) = 6.
Number of events(e) = 6; e_1={p_1,p_4}; e_2={p_1,p_2}; e_3={p_3,p_5};
e_4={p_4,p_6}; e_5={p_2,p_5}; e_6={p_3,p_6};
Message transmission times : τ_1 = 5000; τ_2 = 25000.

Algorithm	Message Count	Response Time
CC	10.1	87633
MEM	6.1	37633
EM	17.3	25282

Table 9-2: Comparison of Algorithms : US Configuration

References

[Ada 82] *Reference Manual for the Ada Programming Language*
United States Department Of Defense, 1982.

[Back 84] Back, R. and Kurki-Suonio, R.
Cooperation in Distributed Systems Using Symmetric Multi-Process Handshaking.
Technical Report No. Ser. A, No. 34, Department of Information Processing and Mathematics, Swedish University of Abo, Finland, 1984.

[Bagrodia 86] Bagrodia, R.
A Distributed Algorithm To Implement The Generalized Alternative Command of CSP.
In *Proccedings of 6th International Conference on Distributed Systems.* Cambridge, May, 1986.

[Bagrodia 87a] Bagrodia, R. and Chandy, K.M. and Misra, J.
A Message-Based Approach To Discrete-Event Simulation.
IEEE Transactions on Software Engineering , June, 1987.

[Bagrodia 87b] Bagrodia, R.
An Environment For the Design and Performance Analysis of Distributed Systems.
PhD thesis, Dept. of Computer Sciences, University of Texas, Austin, Tx 78712., May, 1987.

[Bernstein 80] Bernstein, A.J.
Output guards And Non-determinism in Communicating Sequential Processes.
ACM TOPLAS 2(2):234-238, April, 1980.

[Buckley 83] Buckley, G. and Silberschatz, A.
An Effective Implementation Of The Generalized Input-Output Construct of CSP.
ACM TOPLAS 5(2):223-235, April, 1983.

[Chandy 84] Chandy, K.M. and Misra, J.
The Drinking Philosophers Problem.
ACM TOPLAS 6(4):632-646, October, 1984.

[Chandy 87] Chandy, K.M. and Misra,J.
Synchronizing Asynchronous Processes:The Committee Coordination Problem.
Technical Report, Dept. of Computer Sciences, University of Texas, Austin, Tx 78712., 1987.

[Chandy ed] Chandy, K.M. and Misra, J.
A Foundation of Parallel Program Design.
Addison-Wesley, To be published.

[Charlesworth 87] A. Charlesworth.
The Multiway Rendezvous.
ACM Trans. on Programming Languages and Systems 9(3):350-366, 1987.

[Forman 87] Forman, I.R.
On the Design of Large Distributed Systems.
Technical Report No. STP-098-86, Microelectronics and Computer Technology Corp, Austin, Texas, January, 1987.
Preliminary version in Proc. First Int'l Conf. on Computer Languages, Miami, Florida, October 25-27, 1986.

[Francez 86] Francez, N., Hailpern, B. and Taubenfeld, G.
Script:A Communication Abstraction Mechanism.
Science of Computer Programming 6(1), January, 1986.

[Hoare 78] Hoare, C.A.R.
Communicating Sequential Processes.
CACM 21(8):666-677, August, 1978.

[Milne 85] Milne, George.
CIRCAL and the Representation Of Communication, Concurrency and Time.
ACM TOPLAS 7(2), April, 1985.

[Natarajan 86] Natarajan,N.
A Distributed Synchronization Scheme for Communicating Processes.
The Computer Journal 29(2):109-117, 1986.

[Schneider 82] Schneider,F.
Synchronization In Distributed Programs.
ACM TOPLAS 4(2):125-148, April, 1982.

[Van De Snepscheut 81]
Van De Snepscheut, J.L.A.
Synchronous Communication Between Asynchronous Components.
IPL 13(3):127-130, December, 1981.